॥ श्री गणेशाय नमः ॥

Optimal Control

Varad Vaidya

November 29, 2025

Abstract

Contents

Lecture 1: Introduction to State-Space Dynamics

1.1 Continuous-Time Dynamics

Welcome to the first lecture on Optimal Control! Today we will begin our journey by formalizing the concept of a dynamical system. The core idea is to represent the evolution of a system over time using a mathematical model.

Definition 1.1 (State-Space Representation). A general continuous-time smooth dynamical system can be described by a first-order ordinary differential equation (ODE) of the form:

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t), \mathbf{u}(t), t) \tag{1.1}$$

where:

- $\mathbf{x}(t) \in \mathbb{R}^n$ is the **state vector** of the system at time t. It is a complete summary of the system's history, meaning that the state at time t is sufficient to predict the future evolution of the system given the future inputs.
- $\mathbf{u}(t) \in \mathbb{R}^m$ is the **input vector** (or control vector) at time t. These are the external signals we can manipulate to influence the system's behavior.
- $f : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}^n$ is a smooth function called the **dynamics function** or vector field. It maps the current state, input, and time to the time derivative of the state.

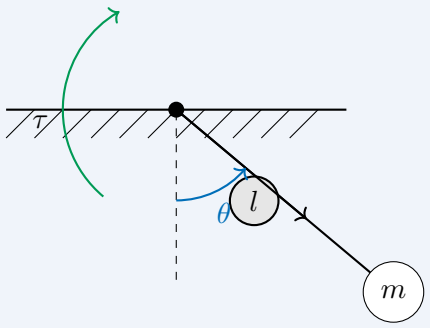
For many systems, especially mechanical ones, the state vector $\mathbf{x}$ has a specific structure. It's often composed of the system's configuration and its velocity.

Notation. For a mechanical system, we typically define the state as:

$$\mathbf{x} = \begin{bmatrix} \mathbf{q} \\ \mathbf{v} \end{bmatrix}$$

where $\mathbf{q}$ is the **configuration** or "pose" of the system, and $\mathbf{v}$ is its corresponding **velocity**. Note that the configuration space is not always a simple vector space like $\mathbb{R}^k$; it can be a more complex manifold, like a circle S^1 or the special orthogonal group $SO(3)$ for rotations.

Example 1.1 (The Simple Pendulum). Let's consider a simple pendulum, which consists of a point mass m attached to a massless rod of length l, pivoting frictionlessly. A torque τ can be applied at the pivot.



The equation of motion for this system can be derived from Newton's second law for rotation, $\sum \tau = I\alpha$, where $I = ml^2$ is the moment of inertia and $\alpha = \ddot{\theta}$ is the angular

acceleration. The torques acting on the mass are the applied torque τ and the torque due to gravity, $-mgl\sin(\theta)$. This gives us:

$$ml^2\ddot{\theta} + mgl\sin(\theta) = \tau \tag{1.2}$$

To put this into our state-space form, we define our state and input:

- Configuration: $q = \theta \in S^1$ (the space of angles, a circle)
- Velocity: $v = \dot{\theta} \in \mathbb{R}$
- State: $\mathbf{x} = \begin{bmatrix}\theta \\ \dot{\theta}\end{bmatrix} \in S^1 \times \mathbb{R}$ (a cylinder)
- Input: $u = \tau \in \mathbb{R}$

Now we can write the dynamics $\dot{\mathbf{x}} = f(\mathbf{x}, u)$:

$$\dot{\mathbf{x}} = \frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix}\theta \\ \dot{\theta}\end{bmatrix} = \begin{bmatrix}\dot{\theta} \\ \ddot{\theta}\end{bmatrix}$$

From Equation 1.2, we can solve for $\ddot{\theta}$:

$$\ddot{\theta} = \frac{1}{ml^2}(\tau - mgl\sin(\theta))$$

Substituting this back, we get the state-space representation:

$$\dot{\mathbf{x}} = f(\mathbf{x}, u) = \begin{bmatrix} x_2 \\ \frac{1}{ml^2}(u - mgl\sin(x_1)) \end{bmatrix} \tag{1.3}$$

where we have used $x_1 = \theta$ and $x_2 = \dot{\theta}$.

1.2 Control-Affine Systems

A very common and important class of nonlinear systems are those that are "affine" in the control input.

Definition 1.2 (Control-Affine System). A system is called **control-affine** if its dynamics can be written in the form:

$$\dot{\mathbf{x}} = f_0(\mathbf{x}) + B(\mathbf{x})\mathbf{u} \tag{1.4}$$

Here, $f_0(\mathbf{x})$ is called the **drift vector field**, representing the system's natural dynamics when no control is applied. The matrix $B(\mathbf{x})$ is the **input Jacobian**, which describes how the control input $\mathbf{u}$ affects the state's velocity at a given state $\mathbf{x}$.

Note. Most mechanical systems can be expressed in this form. For our pendulum example Equation 1.3, we can separate the terms to see its control-affine structure:

$$\dot{\mathbf{x}} = \underbrace{\begin{bmatrix} x_2 \\ -\frac{g}{l}\sin(x_1) \end{bmatrix}}_{f_0(\mathbf{x})} + \underbrace{\begin{bmatrix} 0 \\ \frac{1}{ml^2} \end{bmatrix}}_{B(\mathbf{x})} u$$

Notice that in this case, the input Jacobian B is constant.

1.3 Manipulator Dynamics and Euler-Lagrange

The dynamics of robotic manipulators (and mechanical systems in general) have a well-defined structure.

Theorem 1.1 (Manipulator Equation). The dynamics of a fully-actuated mechanical system can be written as:

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{g}(\mathbf{q}) = \mathbf{B}(\mathbf{q})\mathbf{u} + \tau_{ext} \tag{1.5}$$

where:

- $\mathbf{M}(\mathbf{q})$ is the symmetric, positive-definite **mass matrix**.
- $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}}$ represents Coriolis and centrifugal forces.
- $\mathbf{g}(\mathbf{q})$ is the vector of gravitational forces.
- $\mathbf{B}(\mathbf{q})$ is the input mapping, and τ_{ext} are other external forces.

This is a second-order ODE. To convert it to a first-order state-space form, we can again let $\mathbf{x} = [\mathbf{q}^\top, \dot{\mathbf{q}}^\top]^\top$. Then:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{\mathbf{q}} \\ \mathbf{M}(\mathbf{q})^{-1}(\mathbf{B}(\mathbf{q})\mathbf{u} + \tau_{ext} - \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} - \mathbf{g}(\mathbf{q})) \end{bmatrix}$$

This is also in the control-affine form.

Intuition. This structured form is not arbitrary; it is a direct consequence of the Euler-Lagrange equations from classical mechanics. The Lagrangian $\mathcal{L}$ of a system is defined as the difference between its kinetic energy T and potential energy U:

$$\mathcal{L}(\mathbf{q}, \dot{\mathbf{q}}) = T(\mathbf{q}, \dot{\mathbf{q}}) - U(\mathbf{q})$$

For mechanical systems, the kinetic energy is $T = \frac{1}{2}\dot{\mathbf{q}}^\top \mathbf{M}(\mathbf{q})\dot{\mathbf{q}}$. The Euler-Lagrange equation then gives the dynamics:

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial \mathcal{L}}{\partial \dot{\mathbf{q}}}\right) - \frac{\partial \mathcal{L}}{\partial \mathbf{q}} = \tau_{\text{gen}}$$

where τ_{gen} are the generalized forces acting on the system (like control inputs and external forces). Working through this derivation yields the manipulator Equation 1.5.

1.4 Linear Systems

A particularly simple yet powerful class of systems are linear systems.

Definition 1.3 (Linear Time-Varying System). A system is linear if its dynamics can be written as:

$$\dot{\mathbf{x}}(t) = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{B}(t)\mathbf{u}(t) \tag{1.6}$$

If the matrices $\mathbf{A}$ and $\mathbf{B}$ are constant, the system is called **Linear Time-Invariant (LTI)**.

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$$

Linear systems are fundamental in control theory, not just because they are easy to analyze, but because they can serve as local approximations of nonlinear systems.

As previously seen. We can linearize a nonlinear system $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$ around a nominal trajectory $(\bar{\mathbf{x}}(t), \bar{\mathbf{u}}(t))$. Let $\delta\mathbf{x} = \mathbf{x} - \bar{\mathbf{x}}$ and $\delta\mathbf{u} = \mathbf{u} - \bar{\mathbf{u}}$. A first-order Taylor expansion gives:

$$\dot{\bar{\mathbf{x}}} + \delta\dot{\mathbf{x}} \approx f(\bar{\mathbf{x}}, \bar{\mathbf{u}}) + \underbrace{\left.\frac{\partial f}{\partial \mathbf{x}}\right|_{\bar{\mathbf{x}},\bar{\mathbf{u}}}}_{\mathbf{A}(t)} \delta\mathbf{x} + \underbrace{\left.\frac{\partial f}{\partial \mathbf{u}}\right|_{\bar{\mathbf{x}},\bar{\mathbf{u}}}}_{\mathbf{B}(t)} \delta\mathbf{u}$$

Since $\dot{\bar{\mathbf{x}}} = f(\bar{\mathbf{x}}, \bar{\mathbf{u}})$ (it's a valid trajectory), the dynamics of the error $\delta\mathbf{x}$ are approximately linear:

$$\delta\dot{\mathbf{x}} \approx \mathbf{A}(t)\delta\mathbf{x} + \mathbf{B}(t)\delta\mathbf{u}$$

This process of linearization is a cornerstone of control design for nonlinear systems, and we will revisit it many times in this course.

Lecture 2: Equilibria, Stability, and Simulation

As previously seen. In our last lecture, we introduced the state-space representation for continuous-time dynamical systems, $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$. We examined the structure of control-affine systems and general manipulator dynamics derived from Euler-Lagrange principles. We concluded by defining linear systems and showing how they arise from the linearization of nonlinear systems.

2.1 Equilibrium Points

A fundamental concept in analyzing dynamical systems is the notion of an equilibrium point, a state where the system can remain indefinitely if undisturbed.

Definition 2.1 (Equilibrium Point). An equilibrium point (or fixed point) $\mathbf{x}^\star$ of a continuous-time system $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$ for a constant control input $\mathbf{u}^\star$ is a state that satisfies:

$$f(\mathbf{x}^\star, \mathbf{u}^\star) = 0 \tag{2.1}$$

This means that if the system starts at $\mathbf{x}^\star$ with control $\mathbf{u}^\star$, its state derivative is zero, and it will not move.

Example 2.1 (Pendulum Equilibria). Consider the unforced pendulum ($u = 0$). Its dynamics are:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{\theta} \\ -\frac{g}{l}\sin(\theta) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

This system of equations requires $\dot{\theta} = 0$ and $\sin(\theta) = 0$. This occurs when $\theta = k\pi$ for any integer k. The two distinct physical equilibria are:

- $\mathbf{x}_1^\star = [0, 0]^\top$: The pendulum is hanging straight down, at rest.
- $\mathbf{x}_2^\star = [\pi, 0]^\top$: The pendulum is balanced perfectly upright, at rest.

A basic control problem is to create a new equilibrium point. For instance, can we make the pendulum hang at $\theta = \pi/2$? We need to find a constant control $u^\star$ such that $f([\pi/2, 0]^\top, u^\star) = 0$.

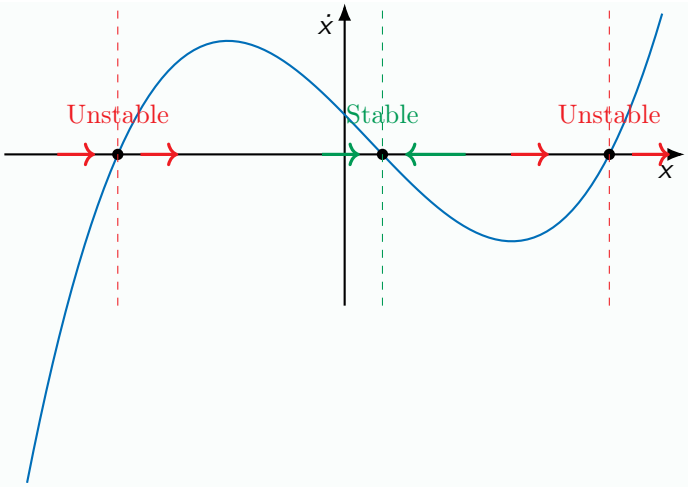
$$\dot{\mathbf{x}} = \begin{bmatrix} 0 \\ \frac{1}{ml^2}(u^\star - mgl\sin(\pi/2)) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

This requires $u^\star - mgl(1) = 0$, so $u^\star = mgl$. A constant torque can indeed hold the pendulum at this new equilibrium.

2.2 Stability of Equilibria

It's not enough to know where the equilibria are; we need to know if they are stable. If the system is perturbed slightly from an equilibrium, does it return, or does it move away?

Intuition (1D Systems). For a 1D system $\dot{x} = f(x)$, we can visualize stability by plotting $\dot{x}$ vs. x.



An equilibrium $x^\star$ is **locally stable** if, for points x near $x^\star$, the dynamics push the state back towards $x^\star$. This happens when the slope of $f(x)$ at $x^\star$ is negative.

- If $\frac{\partial f}{\partial x}|_{x^\star} < 0$, the equilibrium is stable.
- If $\frac{\partial f}{\partial x}|_{x^\star} > 0$, the equilibrium is unstable.
- If $\frac{\partial f}{\partial x}|_{x^\star} = 0$, the analysis is inconclusive (marginally stable).

This concept generalizes to higher dimensions. For a system $\dot{\mathbf{x}} = f(\mathbf{x})$, we linearize the dynamics around an equilibrium $\mathbf{x}^\star$. Let $\delta\mathbf{x} = \mathbf{x} - \mathbf{x}^\star$.

$$\delta\dot{\mathbf{x}} \approx \left.\frac{\partial f}{\partial \mathbf{x}}\right|_{\mathbf{x}^\star} \delta\mathbf{x} = \mathbf{A}\delta\mathbf{x}$$

The stability of the nonlinear system near $\mathbf{x}^\star$ is determined by the stability of this linear system.

Theorem 2.1 (Stability by Linearization). An equilibrium point $\mathbf{x}^\star$ of $\dot{\mathbf{x}} = f(\mathbf{x})$ is:

- **Asymptotically Stable** if all eigenvalues of the Jacobian $\mathbf{A} = \frac{\partial f}{\partial \mathbf{x}}|_{\mathbf{x}^\star}$ have strictly negative real parts ($\mathrm{Re}(\lambda_i) < 0$).
- **Unstable** if at least one eigenvalue of $\mathbf{A}$ has a strictly positive real part ($\mathrm{Re}(\lambda_i) > 0$).
- **Marginally Stable** if all eigenvalues have non-positive real parts ($\mathrm{Re}(\lambda_i) \leq 0$) and at least one has a zero real part.

Example 2.2 (Pendulum Stability Analysis). The Jacobian of the unforced pendulum dynamics is:

$$\mathbf{A} = \frac{\partial f}{\partial \mathbf{x}} = \begin{bmatrix} 0 & 1 \\ -\frac{g}{l}\cos(\theta) & 0 \end{bmatrix}$$

1. **At $\mathbf{x}_1^\star = [0, 0]^\top$ (hanging down)**:

$$\mathbf{A}|_{\theta=0} = \begin{bmatrix} 0 & 1 \\ -\frac{g}{l} & 0 \end{bmatrix}$$

The eigenvalues are $\lambda = \pm i\sqrt{g/l}$. Since the real parts are zero, this equilibrium is **marginally stable**. A small push will cause it to oscillate forever (in this frictionless

model). If we added any damping (e.g., $u = -K_d\dot{\theta}$), the eigenvalues would move into the left-half plane, making it asymptotically stable.

2. **At $\mathbf{x}_2^\star = [\pi, 0]^\top$ (upright)**:

$$\mathbf{A}|_{\theta=\pi} = \begin{bmatrix} 0 & 1 \\ \frac{g}{l} & 0 \end{bmatrix}$$

The eigenvalues are $\lambda = \pm\sqrt{g/l}$. Since there is one positive real eigenvalue, this equilibrium is **unstable**. Any small perturbation will cause the pendulum to fall.

2.3 Discrete-Time Dynamics and Simulation

Analytically solving nonlinear ODEs is rarely possible. We rely on numerical simulation to understand their behavior. This requires converting the continuous-time model into a discrete-time one.

Definition 2.2 (Discrete-Time System). An explicit discrete-time system has the form:

$$\mathbf{x}_{k+1} = f_d(\mathbf{x}_k, \mathbf{u}_k) \tag{2.2}$$

where $\mathbf{x}_k$ is the state at time step k, and f_d is the discrete-time dynamics function that maps the current state and input to the next state.

The process of converting a continuous-time model f to a discrete-time model f_d is called **discretization** or **integration**.

2.3.1 The Forward Euler Method

The simplest way to discretize is to approximate the derivative $\dot{\mathbf{x}}$ with a finite difference: $\dot{\mathbf{x}} \approx \frac{\mathbf{x}_{k+1}-\mathbf{x}_k}{h}$, where h is the time step.

$$\frac{\mathbf{x}_{k+1} - \mathbf{x}_k}{h} = f(\mathbf{x}_k, \mathbf{u}_k) \Rightarrow \mathbf{x}_{k+1} = \mathbf{x}_k + hf(\mathbf{x}_k, \mathbf{u}_k)$$

This is the **Forward Euler** integration scheme.

Code 2.1 (Julia Notebook: Forward Euler Simulation).
The provided Julia notebook first simulates the pendulum using this method. When you run this code, you will observe that the amplitude of the pendulum's oscillation grows with each swing. Eventually, it "blows up" with the angle going to infinity. This is because the Forward Euler method is not energy-preserving. At each step, it adds a small amount of energy to this conservative system, leading to catastrophic instability. This is a critical lesson: **never use Forward Euler for simulating mechanical systems**.

2.3.2 Stability of Discrete-Time Systems

Stability for discrete-time systems is defined by how the system behaves under repeated application of the map f_d. For a linear discrete-time system $\mathbf{x}_{k+1} = \mathbf{A}_d\mathbf{x}_k$, stability requires that $\lim_{k\to\infty} \mathbf{A}_d^k\mathbf{x}_0 = 0$. This condition is met if and only if all eigenvalues of $\mathbf{A}_d$ are strictly inside the unit circle in the complex plane.

Theorem 2.2 (Discrete-Time Stability). A discrete-time linear system $\mathbf{x}_{k+1} = \mathbf{A}_d\mathbf{x}_k$ is stable if and only if $|\lambda_i| < 1$ for all eigenvalues λ_i of $\mathbf{A}_d$.

For the Forward Euler method, the discrete-time Jacobian is $\mathbf{A}_d = \frac{\partial f_d}{\partial \mathbf{x}} = \mathbf{I} + h\mathbf{A}$.

Code 2.2 (Julia Notebook: Stability of Forward Euler).
The notebook analyzes the stability of the Forward Euler discretization of the pendulum around its stable equilibrium ($\theta = 0$). It computes the eigenvalues of $\mathbf{A}_d = \mathbf{I} + h\mathbf{A}|_{\theta=0}$. The result shows that the magnitude of these eigenvalues is always slightly greater than 1 for any $h > 0$. This analytically confirms why the simulation is unstable: the discretization method itself is unstable for this type of system (systems with purely imaginary eigenvalues).

2.3.3 The Runge-Kutta 4 (RK4) Method

A much better explicit integrator is the 4th-order Runge-Kutta (RK4) method. Instead of taking a single step in the direction of the derivative at the start of the interval, it evaluates the derivative at several points within the interval to get a much more accurate estimate of the next state.

Algorithm 1: RK4 Step

Input: Current state $\mathbf{x}_k$, time step h
Output: Next state $\mathbf{x}_{k+1}$

$\mathbf{k}_1 = f(\mathbf{x}_k)$
$\mathbf{k}_2 = f(\mathbf{x}_k + \frac{h}{2}\mathbf{k}_1)$
$\mathbf{k}_3 = f(\mathbf{x}_k + \frac{h}{2}\mathbf{k}_2)$
$\mathbf{k}_4 = f(\mathbf{x}_k + h\mathbf{k}_3)$
$\mathbf{x}_{k+1} = \mathbf{x}_k + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)$

Code 2.3 (Julia Notebook: RK4 Simulation and Stability).
The notebook also simulates the pendulum with RK4. The result is a stable oscillation. The energy is much better conserved, and the amplitude remains constant. Analyzing the eigenvalues of the RK4 discrete Jacobian $\mathbf{A}_d$ shows their magnitudes are extremely close to 1. This indicates that RK4 is a much more stable and accurate choice for this type of simulation. The slight deviation from 1 explains the very slow energy drift that still occurs over very long simulations.

Note. The notebook also includes a **Backward Euler** integrator. This is an *implicit* method, meaning it solves an equation $\mathbf{x}_{k+1} = \mathbf{x}_k + hf(\mathbf{x}_{k+1})$ at each step. Implicit methods are generally very stable, but often introduce "numerical damping," causing energy to decrease even in a conservative system. This can be useful for stiff systems or when stability is paramount, but it may not accurately reflect the physics.

2.4 Key Takeaways on Simulation

- Choosing the right numerical integrator is crucial for obtaining meaningful simulation results.
- Explicit methods like Forward Euler can be unstable and add energy to conservative systems. Avoid them.
- Higher-order methods like RK4 provide a much better balance of accuracy, stability, and computational cost for many problems in robotics.

- Always sanity-check your simulations by monitoring physical quantities that should be conserved, such as total energy or momentum.

2.5 Supplementary Concepts

2.5.1 Lyapunov Stability Theory

Consider an autonomous system $\dot{\mathbf{x}} = f(\mathbf{x})$ with an equilibrium $\mathbf{x}^\star$ (i.e. $f(\mathbf{x}^\star) = 0$). Define the error $\mathbf{e} = \mathbf{x} - \mathbf{x}^\star$ and rewrite about the origin (w.l.o.g. take $\mathbf{x}^\star = 0$ by translation). We recall several nested stability notions.

Definition 2.3 (Stability Notions). Let $\dot{\mathbf{x}} = f(\mathbf{x})$ with equilibrium at 0.

- (Lyapunov or *stable in the sense of Lyapunov*): For every $\varepsilon > 0$ there exists $\delta(\varepsilon) > 0$ s.t. $\|\mathbf{x}(0)\| < \delta \Rightarrow \|\mathbf{x}(t)\| < \varepsilon$ for all $t \geq 0$.
- (Asymptotically Stable): Stable, and $\exists\, \delta' > 0$ s.t. $\|\mathbf{x}(0)\| < \delta' \Rightarrow \lim_{t\to\infty} \mathbf{x}(t) = 0$.
- (Exponentially Stable): $\exists\, c > 0, \alpha > 0, \delta'' > 0$ s.t. $\|\mathbf{x}(0)\| < \delta'' \Rightarrow \|\mathbf{x}(t)\| \leq ce^{-\alpha t}\|\mathbf{x}(0)\|$ for all $t \geq 0$.
- (Globally Asymptotically / Exponentially Stable): The above properties hold for all initial conditions (no radius restriction).

Definition 2.4 (Positive (Semi)Definite, Radially Unbounded). A continuous scalar function $V : \mathbb{R}^n \to \mathbb{R}$ is

- Positive definite if $V(0) = 0$ and $V(\mathbf{x}) > 0$ for all $\mathbf{x} \neq 0$.
- Positive semidefinite if $V(\mathbf{x}) \geq 0$ and $V(0) = 0$.
- Radially unbounded (proper) if $\|\mathbf{x}\| \to \infty \Rightarrow V(\mathbf{x}) \to \infty$.

Definition 2.5 (Lyapunov Function). A continuously differentiable V is a (strict) Lyapunov function for the equilibrium if it is positive definite and its orbital derivative $\dot{V}(\mathbf{x}) := \nabla V(\mathbf{x})^\top f(\mathbf{x})$ is negative definite (or negative semidefinite for weaker conclusions) in a neighborhood $\mathcal{D}$ of the origin.

Theorem 2.3 (Lyapunov Stability Theorem). If there exists a positive definite function V with $\dot{V}(\mathbf{x}) \leq 0$ (negative semidefinite) in a neighborhood $\mathcal{D}$ of the origin, then the equilibrium is Lyapunov stable. If additionally $\dot{V}(\mathbf{x}) < 0$ (negative definite) for all $\mathbf{x} \in \mathcal{D} \setminus \{0\}$, the equilibrium is asymptotically stable. If V and $-\dot{V}$ satisfy quadratic bounds $\alpha_1\|\mathbf{x}\|^2 \leq V(\mathbf{x}) \leq \alpha_2\|\mathbf{x}\|^2$ and $\dot{V}(\mathbf{x}) \leq -\alpha_3\|\mathbf{x}\|^2$, exponential stability follows.

Theorem 2.4 (LaSalle Invariance Principle). Let $\Omega \subset \mathbb{R}^n$ be compact, positively invariant, and suppose $V : \Omega \to \mathbb{R}$ is continuous, C^1 on interior, with $\dot{V} \leq 0$ on Ω. Let $E = \{\mathbf{x} \in \Omega : \dot{V}(\mathbf{x}) = 0\}$ and let $\mathcal{M}$ be the largest invariant subset of E. Then every trajectory starting in Ω approaches $\mathcal{M}$ as $t \to \infty$. If $\mathcal{M} = \{0\}$, the origin is asymptotically stable. If $\Omega = \mathbb{R}^n$ and V is proper, global asymptotic stability can be concluded.

Linear Systems and Lyapunov Equation. For $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}$ with $\mathbf{A}$ Hurwitz (all eigenvalues with negative real parts), for any symmetric positive definite $\mathbf{Q}$ there exists a unique symmetric positive definite $\mathbf{P}$ solving the Lyapunov equation

$$\mathbf{A}^\top \mathbf{P} + \mathbf{P}\mathbf{A} = -\mathbf{Q}.$$

Then $V(\mathbf{x}) = \mathbf{x}^\top \mathbf{P}\mathbf{x}$ is a quadratic Lyapunov function certifying exponential stability.

2.5.2 Basin / Region of Attraction

Definition 2.6 (Basin (Region) of Attraction). For an asymptotically stable equilibrium $\mathbf{x}^\star$, its (open) basin of attraction $\mathcal{B}(\mathbf{x}^\star)$ is the set of initial conditions whose trajectories converge to $\mathbf{x}^\star$:

$$\mathcal{B}(\mathbf{x}^\star) := \{\mathbf{x}_0 \in \mathbb{R}^n : \lim_{t\to\infty} \phi_t(\mathbf{x}_0) = \mathbf{x}^\star\},$$

where ϕ_t denotes the flow map. If $\mathcal{B}(\mathbf{x}^\star) = \mathbb{R}^n$, the equilibrium is globally asymptotically stable.

Lyapunov Level-Set Inner Approximations. Suppose V is a Lyapunov function certifying asymptotic stability on domain $\mathcal{D}$. Any sublevel set $\mathcal{L}_c := \{\mathbf{x} : V(\mathbf{x}) \le c\}$ contained in $\mathcal{D}$ with $\dot{V} < 0$ for nonzero points inside provides an inner approximation of $\mathcal{B}(0)$. One may enlarge c until tangency with $\dot{V} = 0$ occurs. For polynomial systems, sum-of-squares (SOS) optimization can automate this search.

Nonlinear Pendulum Example. Including viscous damping $\dot{\theta} = \omega$, $\dot{\omega} = -\frac{g}{l}\sin\theta - k\omega$ ($k > 0$) makes $\theta = 0$ asymptotically stable. The (mechanical) energy $E = \frac{1}{2}ml^2\omega^2 + mgl(1 - \cos\theta)$ decreases monotonically ($\dot{E} = -kml^2\omega^2 \le 0$). The basin of attraction is all of $\mathbb{R}^2$ (global) when friction prevents escape; without damping it was only (marginally) stable, not attracting.

2.5.3 Poincaré–Bendixson Theorem

In planar continuous-time autonomous systems ($n = 2$), the long-term behaviors are heavily constrained. The Poincaré–Bendixson Theorem rules out chaos (which requires dimension ≥ 3).

Theorem 2.5 (Poincaré–Bendixson). Let $\dot{\mathbf{x}} = f(\mathbf{x})$ with $f \in C^1$ on an open set $\mathcal{U} \subset \mathbb{R}^2$. Suppose a trajectory $\phi_t(\mathbf{x}_0)$ remains in a compact set $\mathcal{K} \subset \mathcal{U}$ for all $t \ge 0$ and contains no equilibrium points. Then the ω-limit set of $\mathbf{x}_0$ is a periodic orbit (limit cycle). More generally, any nonempty compact ω-limit set that contains only finitely many equilibria is either (i) an equilibrium, (ii) a periodic orbit, or (iii) a finite union of equilibria and connecting heteroclinic orbits.

Consequences. In 2D one cannot have strange attractors. For physical robot subsystems reduced to two states (e.g. simplified balancing planes), observed sustained oscillations typically imply a limit cycle certified by the theorem.

Application Template. To use Poincaré–Bendixson:

1. Find a forward-invariant compact set $\mathcal{K}$ (e.g. by trapping region, energy bounds).
2. Show the trajectory stays in $\mathcal{K}$ and does not converge to an equilibrium (e.g. by Dulac's criterion or sign of divergence arguments).
3. Conclude existence of a periodic orbit (limit cycle).

Example (Van der Pol). $\ddot{x} - \mu(1 - x^2)\dot{x} + x = 0$ with $\mu > 0$ can be written as a planar system possessing a unique stable limit cycle; trajectories enter a compact trapping region and cannot settle at the sole equilibrium (the origin is unstable for $\mu > 0$).

2.5.4 Controllability and Observability

We restrict to linear time-invariant (LTI) systems $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}+\mathbf{B}\mathbf{u}$, $\mathbf{y} = \mathbf{C}\mathbf{x}+\mathbf{D}\mathbf{u}$, with $\mathbf{A} \in \mathbb{R}^{n\times n}$, $\mathbf{B} \in \mathbb{R}^{n\times m}$, $\mathbf{C} \in \mathbb{R}^{p\times n}$.

Definition 2.7 (Reachable / Controllable). The system is (state) controllable if for any $\mathbf{x}_0, \mathbf{x}_f$ there exists a finite time T and input $\mathbf{u}(t)$ steering $\mathbf{x}(0) = \mathbf{x}_0$ to $\mathbf{x}(T) = \mathbf{x}_f$. It is (origin) reachable if each state can be reached from the origin. For LTI systems these notions coincide.

Definition 2.8 (Observable). The pair $(\mathbf{A}, \mathbf{C})$ is observable if any two distinct initial states $\mathbf{x}_1(0) \neq \mathbf{x}_2(0)$ produce different output trajectories on some finite interval; equivalently the initial state is uniquely determined from knowledge of $\mathbf{y}(t)$ and $\mathbf{u}(t)$ over a finite horizon.

Kalman Rank Conditions. Define the controllability matrix

$$\mathcal{C} = \begin{bmatrix} \mathbf{B} & \mathbf{AB} & \mathbf{A}^2\mathbf{B} & \cdots & \mathbf{A}^{n-1}\mathbf{B} \end{bmatrix} \in \mathbb{R}^{n\times nm}$$

and the observability matrix

$$\mathcal{O} = \begin{bmatrix} \mathbf{C} \\ \mathbf{CA} \\ \mathbf{CA}^2 \\ \vdots \\ \mathbf{CA}^{n-1} \end{bmatrix} \in \mathbb{R}^{np\times n}.$$

Then $(\mathbf{A}, \mathbf{B})$ is controllable iff $\operatorname{rank}(\mathcal{C}) = n$; $(\mathbf{A}, \mathbf{C})$ is observable iff $\operatorname{rank}(\mathcal{O}) = n$.

PBH (Popov–Belevitch–Hautus) Tests. An equivalent spectral test: $(\mathbf{A}, \mathbf{B})$ is controllable iff $\operatorname{rank}\begin{bmatrix} \lambda\mathbf{I} - \mathbf{A} & \mathbf{B} \end{bmatrix} = n$ for all $\lambda \in \mathbb{C}$. Likewise $(\mathbf{A}, \mathbf{C})$ is observable iff $\operatorname{rank}\begin{bmatrix} \lambda\mathbf{I} - \mathbf{A} \\ \mathbf{C} \end{bmatrix} = n$ for all $\lambda \in \mathbb{C}$.

Stabilizability and Detectability. If uncontrollable modes (eigenvalues) all lie in the open left-half plane, the system is stabilizable (we can design feedback to stabilize). Dually, if unobservable modes all decay, the system is detectable (an observer can be designed with asymptotic error convergence). These relaxed properties are sufficient for many optimal control designs (e.g. LQR requires stabilizability / detectability, not full controllability / observability).

Energy / Gramian Characterization. Over horizon $[0, T]$, the controllability Gramian is

$$\mathbf{W}_c(T) = \int_0^T e^{\mathbf{A}t}\mathbf{B}\mathbf{B}^\top e^{\mathbf{A}^\top t}\, dt.$$

If $\mathbf{W}_c(T)$ is nonsingular for some (hence all sufficiently large) $T > 0$, the system is controllable. Minimum input energy to reach $\mathbf{x}_f$ from 0 in time T is $\mathbf{x}_f^\top \mathbf{W}_c(T)^{-1}\mathbf{x}_f$. Similarly, the observability Gramian $\mathbf{W}_o(T) = \int_0^T e^{\mathbf{A}^\top t}\mathbf{C}^\top\mathbf{C}e^{\mathbf{A}t}\, dt$ is nonsingular iff $(\mathbf{A}, \mathbf{C})$ is observable.

Connection to Optimal Control. Existence of a unique positive semidefinite solution to the Algebraic Riccati Equation (ARE) for LQR hinges on stabilizability and detectability. The resulting optimal feedback $\mathbf{u} = -\mathbf{K}\mathbf{x}$ ensures closed-loop Lyapunov stability via the ARE as a Lyapunov equation for the cost-to-go.

Summary. Lyapunov functions certify (local/global, asymptotic/exponential) stability and approximate basins via level sets; planar systems admit only equilibria and limit cycles as nontrivial recurrent sets (Poincaré–Bendixson), and linear controllability/observability (with their relaxed forms) determine if we can stabilize or reconstruct system states—all foundational for designing and analyzing optimal controllers.

Lecture 3: Derivatives, Root Finding, and Minimization

3.1 A Note on Notation: Derivatives as Linear Operators

To proceed, we must first establish a clear and consistent notation for derivatives, treating them as linear operators that describe the local change of a function.

Notation. Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a differentiable function. The **Jacobian** of f at a point $\mathbf{x}$ is the unique $m \times n$ matrix, denoted $\frac{\partial f}{\partial \mathbf{x}}$, that satisfies the first-order Taylor approximation:

$$f(\mathbf{x} + \delta\mathbf{x}) \approx f(\mathbf{x}) + \left.\frac{\partial f}{\partial \mathbf{x}}\right|_{\mathbf{x}} \delta\mathbf{x} \tag{3.1}$$

This convention ensures that the chain rule for Jacobians works as expected with standard matrix multiplication. If we have $h(\mathbf{z}) = f(g(\mathbf{z}))$, then:

$$\frac{\partial h}{\partial \mathbf{z}} = \left.\frac{\partial f}{\partial \mathbf{y}}\right|_{g(\mathbf{z})} \frac{\partial g}{\partial \mathbf{z}}$$

For a scalar-valued function $f : \mathbb{R}^n \to \mathbb{R}$, its Jacobian $\frac{\partial f}{\partial \mathbf{x}}$ is a $1 \times n$ row vector. For convenience, we define the **gradient** vector, $\nabla f(\mathbf{x})$, as the transpose of the Jacobian.

$$\nabla f(\mathbf{x}) := \left(\frac{\partial f}{\partial \mathbf{x}}\right)^\top \quad \text{(an } n \times 1 \text{ column vector)}$$

The **Hessian** matrix is the Jacobian of the gradient, resulting in an $n \times n$ matrix of second partial derivatives.

$$\nabla^2 f(\mathbf{x}) := \frac{\partial}{\partial \mathbf{x}} = \frac{\partial^2 f}{\partial \mathbf{x}^2} \quad \text{(an } n \times n \text{ matrix)}$$

With this, the second-order Taylor expansion of a scalar function is:

$$f(\mathbf{x} + \delta\mathbf{x}) \approx f(\mathbf{x}) + (\nabla f(\mathbf{x}))^\top \delta\mathbf{x} + \frac{1}{2}\delta\mathbf{x}^\top (\nabla^2 f(\mathbf{x}))\delta\mathbf{x} \tag{3.2}$$

3.2 Root Finding

A fundamental problem in numerical methods is root finding: for a given function $g : \mathbb{R}^n \to \mathbb{R}^n$, find a point $\mathbf{x}^\star$ such that $g(\mathbf{x}^\star) = 0$.

Example 3.1. Finding the equilibrium point of a dynamical system $\dot{\mathbf{x}} = f(\mathbf{x})$ is a root-finding problem. Similarly, as we saw last lecture, solving the implicit equation in a Backward Euler step, $\mathbf{x}_{k+1} - \mathbf{x}_k - hf(\mathbf{x}_{k+1}) = 0$, is also a root-finding problem for $\mathbf{x}_{k+1}$.

3.2.1 Fixed-Point Iteration

A closely related problem is finding a **fixed point**, i.e., an $\mathbf{x}^\star$ such that $g(\mathbf{x}^\star) = \mathbf{x}^\star$. We can always convert a root-finding problem $r(\mathbf{x}) = 0$ to a fixed-point problem by defining $g(\mathbf{x}) = \mathbf{x} - r(\mathbf{x})$.

The simplest method for finding a fixed point is **fixed-point iteration**:

$$\mathbf{x}_{k+1} = g(\mathbf{x}_k)$$

This method is only guaranteed to converge if the fixed point is stable (i.e., $|\text{eig}(\frac{\partial g}{\partial \mathbf{x}})| < 1$) and the initial guess is within the basin of attraction. The convergence rate is typically linear, which can be quite slow.

3.2.2 Newton's Method

A much more powerful and faster method is **Newton's method**. It works by iteratively solving a linearized version of the root-finding problem. Given a guess $\mathbf{x}_k$, we seek a correction $\delta\mathbf{x}$ such that $g(\mathbf{x}_k + \delta\mathbf{x}) = 0$. We linearize g around $\mathbf{x}_k$:

$$g(\mathbf{x}_k + \delta\mathbf{x}) \approx g(\mathbf{x}_k) + \left.\frac{\partial g}{\partial \mathbf{x}}\right|_{\mathbf{x}_k} \delta\mathbf{x} = 0$$

Solving for the correction $\delta\mathbf{x}$ gives the Newton step:

$$\delta\mathbf{x} = -\left(\frac{\partial g}{\partial \mathbf{x}}\right)^{-1} g(\mathbf{x}_k) \tag{3.3}$$

The next guess is then $\mathbf{x}_{k+1} = \mathbf{x}_k + \delta\mathbf{x}$. This process is repeated until convergence.

Algorithm 3: Newton's Method

Input: Function $g(\mathbf{x})$, initial guess $\mathbf{x}_0$, tolerance ϵ
Output: Root $\mathbf{x}^\star$

$\mathbf{x}_k \leftarrow \mathbf{x}_0$
while $\|\mathbf{g}(\mathbf{x}_k)\| > \epsilon$ **do**
  Compute Jacobian $\mathbf{J} = \frac{\partial g}{\partial \mathbf{x}}|_{\mathbf{x}_k}$
  Solve the linear system $\mathbf{J}\delta\mathbf{x} = -g(\mathbf{x}_k)$ for $\delta\mathbf{x}$
  $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \delta\mathbf{x}$
return $\mathbf{x}_k$

Code 3.1 (Julia Notebook: Root Finding for Backward Euler).
The 'root-finding.ipynb' notebook provides a perfect comparison of fixed-point iteration and Newton's method. Both are used to solve the Backward Euler equation for the pendulum. The notebook plots the error (norm of the residual) at each iteration for both methods.

- **Fixed-point iteration** shows a slow, steady, linear decrease in error on a semi-log plot. It takes many iterations to reach high precision.
- **Newton's method** exhibits its characteristic **quadratic convergence**. The error decreases extremely rapidly, with the number of correct digits roughly doubling at each step. It typically converges to machine precision in just a few iterations.

This demonstrates the power of Newton's method. While each step is more expensive (requiring a Jacobian and a linear solve, is of the order $O(n^3)$), the drastically fewer iterations required make it far more efficient overall.

3.3 Unconstrained Minimization

We now turn to the problem of finding a local minimum of a smooth scalar function $f : \mathbb{R}^n \to \mathbb{R}$. A **first-order necessary condition** for a point $\mathbf{x}^\star$ to be a local minimum is that the gradient vanishes:

$$\nabla f(\mathbf{x}^\star) = 0 \tag{3.4}$$

This turns the minimization problem into a root-finding problem on the gradient! We can directly apply Newton's method.

Definition 3.1 (Newton's Method for Minimization). To find a point where $\nabla f(\mathbf{x}) = 0$, we apply the Newton's method recipe. The "function" we are finding the root of is now $\nabla f(\mathbf{x})$. The "Jacobian" of this function is the Hessian, $\nabla^2 f(\mathbf{x})$. The Newton step is therefore:

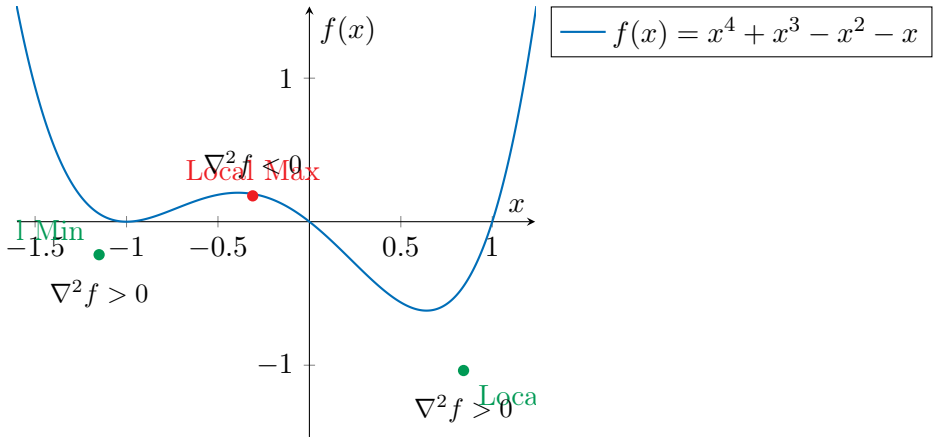
$$\delta\mathbf{x} = -(\nabla^2 f(\mathbf{x}))^{-1}\nabla f(\mathbf{x}) \tag{3.5}$$

The intuition here is that we are fitting a quadratic model to the function at the current point (based on the second-order Taylor expansion) and then jumping directly to the minimum of that quadratic model.

However, there are pitfalls. The condition $\nabla f(\mathbf{x}^\star) = 0$ applies to local maxima and saddle points as well as minima. A standard Newton step can easily converge to the wrong type of point.

A **second-order sufficient condition** for $\mathbf{x}^\star$ to be a strict local minimum is that $\nabla f(\mathbf{x}^\star) = 0$ and the Hessian $\nabla^2 f(\mathbf{x}^\star)$ is **positive definite** (all its eigenvalues are positive). A Newton step is a *descent direction* (i.e., a step that decreases the function value) if and only if the Hessian is positive definite. If the Hessian is negative definite (at a maximum), Newton's method becomes an *ascent* method.

Code 3.2 (Julia Notebook: The Pitfall of Standard Newton's Method).
The 'minimization.ipynb' notebook clearly illustrates this problem. It attempts to minimize the function $f(x) = x^4 + x^3 - x^2 - x$.



When an initial guess of $x_0 = 0$ is used, the Hessian $\nabla^2 f(0)$ is negative. The notebook shows that the standard Newton step moves from $x = 0$ towards the local maximum, not the minimum.

3.3.1 Regularization (Damped Newton's Method)

To fix this, we can **regularize** the Hessian. The goal is to modify the Hessian $\mathbf{H} = \nabla^2 f(\mathbf{x})$ to ensure it is positive definite, thus guaranteeing a descent direction. A simple and effective method is Levenberg-Marquardt style damping:

$$\text{If } \mathbf{H} \text{ is not positive definite, replace it with } \mathbf{H}' = \mathbf{H} + \beta\mathbf{I}$$

where $\beta > 0$ is a damping parameter. We can increase β until $\mathbf{H}'$ becomes positive definite. This has two effects:

1. It guarantees the resulting step $\delta\mathbf{x} = -(\mathbf{H}')^{-1}\nabla f(\mathbf{x})$ is a descent direction.

2. As $\beta \to \infty$, the step becomes $\delta \mathbf{x} \approx -\frac{1}{\beta}\nabla f(\mathbf{x})$, which is a small step in the steepest descent direction.

This method adaptively blends between the fast Newton step (when the function is locally convex) and the robust but slow gradient descent step (when it is not).

Code 3.3 (Julia Notebook: Regularized Newton's Method).
The notebook implements this regularization. When starting from $x_0 = 0$, the algorithm detects the negative Hessian, adds damping, and computes a new step. The resulting step is now correctly pointed towards the local minimum. This demonstrates how regularization makes Newton's method a robust tool for nonlinear optimization.

3.4 Supplementary Concepts

We adopt the setting of unconstrained smooth optimization $\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x})$ with $f \in C^2$ unless otherwise specified.

3.4.1 Convexity and Strong Convexity

Definition 3.2 (Convex Function). Let $\mathcal{D} \subseteq \mathbb{R}^n$ be convex. A function $f : \mathcal{D} \to \mathbb{R}$ is convex if for all $\mathbf{x}, \mathbf{y} \in \mathcal{D}$ and $\theta \in [0, 1]$:

$$f(\theta\mathbf{x} + (1-\theta)\mathbf{y}) \leq \theta f(\mathbf{x}) + (1-\theta) f(\mathbf{y}).$$

It is *strictly convex* if the inequality is strict whenever $\mathbf{x} \neq \mathbf{y}$ and $\theta \in (0, 1)$.

Definition 3.3 (Strong Convexity). For $m > 0$, f is m-strongly convex if

$$f(\mathbf{y}) \geq f(\mathbf{x}) + \nabla f(\mathbf{x})^\top(\mathbf{y} - \mathbf{x}) + \frac{m}{2}\|\mathbf{y} - \mathbf{x}\|^2, \quad \forall \mathbf{x}, \mathbf{y}.$$

Theorem 3.1 (Second-Order Characterizations). Assume $f \in C^2$ on an open convex domain.

- f convex $\Leftrightarrow \nabla^2 f(\mathbf{x})$ is positive semidefinite (PSD) for all $\mathbf{x}$.
- f m-strongly convex $\Leftrightarrow \nabla^2 f(\mathbf{x}) \succeq m\mathbf{I}$ for all $\mathbf{x}$ (i.e., all eigenvalues $\geq m$).

Theorem 3.2 (Alternative Strong Convexity Characterizations). Let f be differentiable and $m > 0$. The following are equivalent:

1. f is m-strongly convex.
2. For all $\mathbf{x}, \mathbf{y}$, $(\nabla f(\mathbf{x}) - \nabla f(\mathbf{y}))^\top(\mathbf{x} - \mathbf{y}) \geq m\|\mathbf{x} - \mathbf{y}\|^2$.
3. For all $\mathbf{x}, \mathbf{y}$, $f(\mathbf{y}) \geq f(\mathbf{x}) + \nabla f(\mathbf{x})^\top(\mathbf{y} - \mathbf{x}) + \frac{m}{2}\|\mathbf{y} - \mathbf{x}\|^2$.

Consequences. Strong convexity implies uniqueness of the global minimizer. Plain convexity only guarantees that any local minimum is global (but possibly non-unique). For L-smooth functions (gradient Lipschitz with constant L), gradient descent with step size $\alpha \in (0, 2/L)$ converges; if additionally m-strongly convex, the convergence is linear with rate $1 - m/L$.

3.4.2 Optimality Conditions (Unconstrained)

Let $f \in C^2$. A point $\mathbf{x}^\star$ is a (local) minimizer only if the following hold.

Theorem 3.3 (First-Order Necessary Condition). If $\mathbf{x}^\star$ is a local minimizer, then $\nabla f(\mathbf{x}^\star) = 0$.

Theorem 3.4 (Second-Order Necessary Condition). If $\mathbf{x}^\star$ is a local minimizer and $f \in C^2$, then $\nabla f(\mathbf{x}^\star) = 0$ and $\nabla^2 f(\mathbf{x}^\star)$ is PSD.

Theorem 3.5 (Second-Order Sufficient Condition). If $\nabla f(\mathbf{x}^\star) = 0$ and $\nabla^2 f(\mathbf{x}^\star)$ is positive definite (PD), then $\mathbf{x}^\star$ is a strict local minimizer. If $\nabla^2 f(\mathbf{x}) \succeq m\mathbf{I}$ globally ($m > 0$), then $\mathbf{x}^\star$ is the unique global minimizer.

Saddle Points. If the Hessian has both positive and negative eigenvalues at a stationary point, Newton's method may be attracted there; regularization or negative curvature exploitation is needed to escape.

3.4.3 Positive Definiteness, Cholesky Factorization, and On-the-Spot Padding

Definition 3.4 (Positive (Semi)Definiteness). A symmetric matrix $\mathbf{H} \in \mathbb{R}^{n\times n}$ is PSD if $\mathbf{v}^\top \mathbf{H}\mathbf{v} \geq 0$ for all $\mathbf{v}$; PD if the inequality is strict for all nonzero $\mathbf{v}$.

Cholesky Factorization. A symmetric matrix $\mathbf{H}$ is PD $\Leftrightarrow$ it admits a (unique) Cholesky factorization $\mathbf{H} = \mathbf{R}^\top \mathbf{R}$ with $\mathbf{R}$ upper triangular with positive diagonal. Attempting Cholesky is therefore a practical test: breakdown (negative pivot) indicates loss of definiteness.

Lecture 4: Constrained Minimization

4.1 Globalization Strategy I: Line Search

While regularization ensures we take steps in the right direction, it doesn't control how far we step. The full Newton step $\delta\mathbf{x} = -(\nabla^2 f)^{-1}\nabla f$ is derived from minimizing a quadratic approximation of the function. If the function is not well-approximated by a quadratic, this step can overshoot the actual minimum.

A **line search** is a procedure to find an appropriate step size $\alpha \in (0, 1]$ to scale the Newton direction $\delta\mathbf{x}$, such that the update $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha\delta\mathbf{x}$ makes sufficient progress.

Definition 4.1 (Armijo Backtracking Line Search). A simple and effective line search strategy is the Armijo rule. It ensures that the actual reduction in the function value is at least a fraction of the reduction predicted by the linear approximation.

Algorithm 4: Armijo Backtracking Line Search

Input: Current point $\mathbf{x}$, search direction $\delta\mathbf{x}$, parameters $b \in (0, 1), c \in (0, 1)$
Output: Step size α

$\alpha \leftarrow 1.0$
while $f(\mathbf{x} + \alpha\delta\mathbf{x}) > f(\mathbf{x}) + b\alpha(\nabla f(\mathbf{x}))^\top \delta\mathbf{x}$ **do**
  $\alpha \leftarrow c\alpha$
return α

Note. The term $(\nabla f(\mathbf{x}))^\top \delta\mathbf{x}$ is the directional derivative, which represents the slope of the function along the search direction. The condition checks if our actual progress is better than a certain fraction (b) of the progress we would expect from a linear model. Typical values are $b \approx 10^{-4}$ and $c = 0.5$.

Code 4.1 (Julia Notebook: Backtracking Newton's Method).
The 'minimization.ipynb' notebook implements this backtracking line search. When applied, it prevents the large, unproductive steps that the pure Newton method might take, especially when far from the optimum. This combination of **regularization** (to pick a good direction) and **line search** (to pick a good distance) makes Newton's method a robust and powerful "globalized" algorithm for finding local minima.

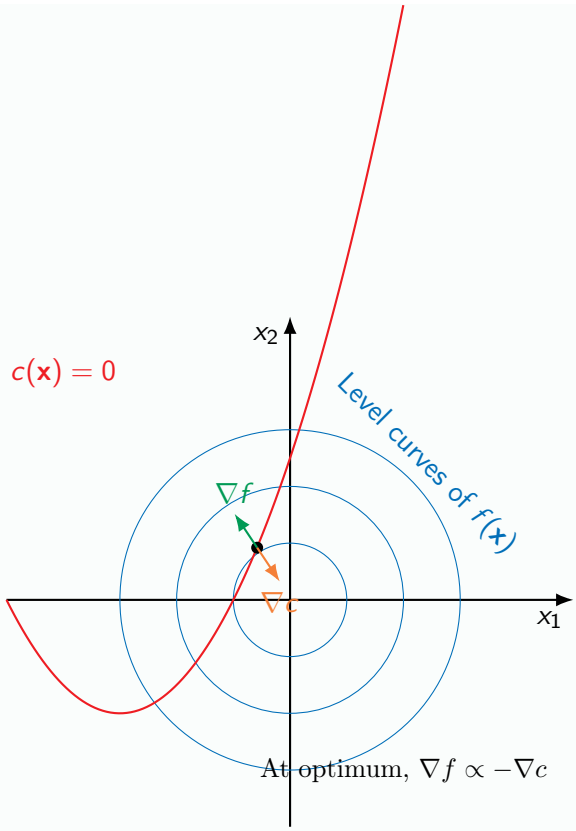
4.2 Equality-Constrained Minimization

We now consider problems of the form:

$$\begin{aligned} \min_{\mathbf{x}\in\mathbb{R}^n} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{c}(\mathbf{x}) = 0 \end{aligned}$$

where $\mathbf{c} : \mathbb{R}^n \to \mathbb{R}^m$ is a set of m equality constraints.

Intuition. At a constrained optimum $\mathbf{x}^\star$, you cannot make further progress by moving along the constraint surface. This means the gradient of the objective function, $\nabla f(\mathbf{x}^\star)$, must be orthogonal to the constraint surface. The gradient of the constraint function, $\nabla c(\mathbf{x}^\star)$, is also orthogonal to the constraint surface. Therefore, the two gradients must be parallel.



This geometric condition implies that at the optimum, $\nabla f(\mathbf{x}^\star)$ must be a linear combination of the constraint gradients.

Definition 4.2 (The Lagrangian). This leads to the first-order necessary conditions for optimality. There must exist a vector of **Lagrange multipliers** $\lambda \in \mathbb{R}^m$ such that:

$$\nabla f(\mathbf{x}^\star) + (\frac{\partial \mathbf{c}}{\partial \mathbf{x}})^\top \lambda = 0 \tag{4.1}$$

We can elegantly combine this condition with the original feasibility condition, $\mathbf{c}(\mathbf{x}) = 0$, by defining the **Lagrangian** function, $\mathcal{L} : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$:

$$\mathcal{L}(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda^\top \mathbf{c}(\mathbf{x}) \tag{4.2}$$

The optimality conditions are now equivalent to finding a stationary point of the Lagrangian:

$$\begin{aligned} \nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \lambda) &= \nabla f(\mathbf{x}) + (\frac{\partial \mathbf{c}}{\partial \mathbf{x}})^\top \lambda = 0 \\ \nabla_\lambda \mathcal{L}(\mathbf{x}, \lambda) &= \mathbf{c}(\mathbf{x}) = 0 \end{aligned}$$

This is a root-finding problem for the concatenated variable vector $(\mathbf{x}, \lambda)$. We can solve it using Newton's method! Applying Newton's method yields the following linear system, known as the Karush-Kuhn-Tucker (KKT) system:

$$\begin{bmatrix} \nabla^2_{\mathbf{xx}} \mathcal{L} & (\frac{\partial \mathbf{c}}{\partial \mathbf{x}})^\top \\ \frac{\partial \mathbf{c}}{\partial \mathbf{x}} & 0 \end{bmatrix} \begin{bmatrix} \delta \mathbf{x} \\ \delta \lambda \end{bmatrix} = - \begin{bmatrix} \nabla_{\mathbf{x}} \mathcal{L} \\ \mathbf{c}(\mathbf{x}) \end{bmatrix} \tag{4.3}$$

4.2.1 The Gauss-Newton Approximation

The full Hessian of the Lagrangian is $\nabla^2_{\mathbf{xx}}\mathcal{L} = \nabla^2 f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i \nabla^2 c_i(\mathbf{x})$. Another way to write this, that allows for using jacobian-vector product tricks from autodifferentiation, is:

$$\nabla^2_{\mathbf{xx}}\mathcal{L} = \nabla^2 f(\mathbf{x}) + \frac{\partial}{\partial x}\left[\left(\frac{\partial c}{\partial x}\right)^\top \lambda\right]$$

Computing the second derivatives of the constraints (the "constraint curvature" term) can be expensive. The **Gauss-Newton method** (or Sequential Quadratic Programming, SQP) approximates this Hessian by simply dropping the constraint curvature term:

$$\nabla^2_{\mathbf{xx}}\mathcal{L} \approx \nabla^2 f(\mathbf{x})$$

This approximation often works very well, leading to cheaper iterations that still make good progress.

Code 4.2 (Julia Notebook: Equality Constrained Optimization).
The 'equality-constraints.ipynb' notebook solves a simple quadratic minimization problem subject to a nonlinear equality constraint. It compares the full Newton method with the Gauss-Newton method.

- **Full Newton**: Converges very quickly (quadratically) to the solution from a good initial guess.
- **Gauss-Newton**: Also converges, but may take a few more iterations since its model of the problem is less accurate. However, each iteration is computationally cheaper. In many large-scale robotics problems, this trade-off makes Gauss-Newton the preferred method.

The notebook also shows a case where the full Newton method can get "stuck" if the second-order constraint curvature term is problematic, while the simpler Gauss-Newton approximation avoids this issue and successfully finds the solution.

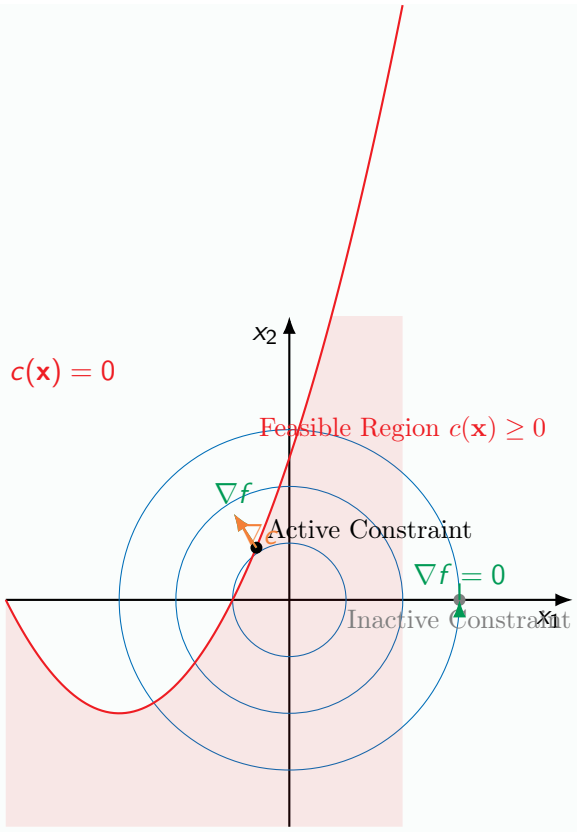
4.3 Inequality-Constrained Minimization

Finally, we consider the general case:

$$\begin{aligned} \min_{\mathbf{x}\in\mathbb{R}^n} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{c}(\mathbf{x}) \geq 0 \end{aligned}$$

At the solution $\mathbf{x}^\star$, some constraints may be **active** ($c_i(\mathbf{x}^\star) = 0$) while others are **inactive** ($c_i(\mathbf{x}^\star) > 0$).

Intuition. Inactive constraints don't affect the solution locally, so we can ignore them. Active constraints behave like equality constraints. However, there is a key difference: for an inequality constraint, the gradient ∇f must point "away" from the feasible region. This means ∇f must be a *non-negative* linear combination of the active constraint gradients.



Theorem 4.1 (Karush-Kuhn-Tucker (KKT) Conditions). The first-order necessary conditions for a point $\mathbf{x}^\star$ to be a local minimum are that there exists a vector of Lagrange multipliers $\lambda^\star$ satisfying:

1. **Stationarity**: $\nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}^\star, \lambda^\star) = \nabla f(\mathbf{x}^\star) + (\frac{\partial \mathbf{c}}{\partial \mathbf{x}})^\top \lambda^\star = 0$
2. **Primal Feasibility**: $\mathbf{c}(\mathbf{x}^\star) \geq 0$
3. **Dual Feasibility**: $\lambda^\star \geq 0$
4. **Complementary Slackness**: $\lambda_i^\star c_i(\mathbf{x}^\star) = 0$ for all $i = 1, \ldots, m$.

The complementary slackness condition is a clever way of stating that if a constraint is inactive ($c_i > 0$), its corresponding multiplier must be zero ($\lambda_i = 0$). This mathematically enforces our intuition. Modern algorithms for solving such problems, like interior-point methods, are designed to find points that satisfy these KKT conditions.

4.4 Supplementary Material: Advanced Line Search Concepts

The backtracking line search using the Armijo rule is simple and effective. However, to fully appreciate the theory of optimization, it's crucial to understand the landscape of line search conditions and the convergence guarantees they provide.

4.4.1 The Challenge of Exact Line Search

At each iteration of a descent method, we have a current point $\mathbf{x}_k$ and a descent direction $\mathbf{p}_k$. The ideal step size α would be the one that solves the one-dimensional optimization problem:

$$\alpha_k = \arg\min_{\alpha > 0} \phi(\alpha) = \arg\min_{\alpha > 0} f(\mathbf{x}_k + \alpha \mathbf{p}_k) \tag{4.4}$$

Solving this subproblem exactly is often computationally expensive, requiring many function and gradient evaluations. The cost of finding the exact minimum might be comparable to the cost of several iterations of the main algorithm. Therefore, in practice, we use **inexact line searches** that aim to find a step size that is "good enough" with minimal effort. The following conditions define what "good enough" means.

4.4.2 A Tour of Inexact Line Search Conditions

The Armijo (Sufficient Decrease) Condition This is the condition we have already seen. It ensures that the step size α leads to a decrease in the objective function that is at least a fraction of the decrease predicted by the linear approximation at $\mathbf{x}_k$.

$$f(\mathbf{x}_k + \alpha \mathbf{p}_k) \leq f(\mathbf{x}_k) + c_1 \alpha \nabla f(\mathbf{x}_k)^\top \mathbf{p}_k \tag{4.5}$$

with $c_1 \in (0, 1)$. A typical value for c_1 is 10^{-4}. While this condition prevents steps that are too long, it does not prevent steps that are excessively short. A backtracking algorithm based solely on this rule might produce a sequence of very small steps and converge slowly.

The Wolfe Conditions To rule out unacceptably short steps, the Armijo condition is often paired with a **curvature condition**. The combination is known as the **Wolfe conditions**.

1. **Sufficient Decrease (Armijo):** Same as (4.5).
2. **Curvature Condition:**

$$\nabla f(\mathbf{x}_k + \alpha \mathbf{p}_k)^\top \mathbf{p}_k \geq c_2 \nabla f(\mathbf{x}_k)^\top \mathbf{p}_k \tag{4.6}$$

   with $c_2 \in (c_1, 1)$. This condition ensures that the slope of $\phi(\alpha)$ at the new point is less negative than the initial slope, which means we have made reasonable progress and are not at a point where the function is still decreasing rapidly. It forces the step length α to be in a region where the function is flatter.

A step length satisfying both conditions is guaranteed to exist for many common functions, provided $\mathbf{p}_k$ is a descent direction.

The Strong Wolfe Conditions A slight modification leads to the **Strong Wolfe conditions**, which constrain the slope to be closer to zero, forcing α to be closer to a stationary point of $\phi(\alpha)$.

1. **Sufficient Decrease (Armijo):** Same as (4.5).
2. **Strong Curvature Condition:**

$$|\nabla f(\mathbf{x}_k + \alpha \mathbf{p}_k)^\top \mathbf{p}_k| \leq c_2 |\nabla f(\mathbf{x}_k)^\top \mathbf{p}_k| \tag{4.7}$$

This is particularly useful in quasi-Newton methods (like BFGS), where it helps ensure that the Hessian approximation matrix remains positive definite.

The Goldstein Conditions The Goldstein conditions are another pair of inequalities that ensure both sufficient decrease and that the step is not too short.

$$f(\mathbf{x}_k) + (1 - c)\alpha \nabla f(\mathbf{x}_k)^\top \mathbf{p}_k \leq f(\mathbf{x}_k + \alpha \mathbf{p}_k) \tag{4.8}$$

$$f(\mathbf{x}_k + \alpha \mathbf{p}_k) \leq f(\mathbf{x}_k) + c\alpha \nabla f(\mathbf{x}_k)^\top \mathbf{p}_k \tag{4.9}$$

with $0 < c < 1/2$. The first inequality ensures α is not too large, while the second (the Armijo condition) ensures it's not too small. A disadvantage is that the first inequality might exclude the actual minimizer of $\phi(\alpha)$, which is why the Wolfe conditions are generally preferred in modern algorithms.

4.4.3 Convergence of Backtracking Line Search

We can prove that a simple backtracking algorithm using the Armijo rule will terminate and that the overall optimization algorithm will converge. This analysis relies on the smoothness of the objective function, specifically that its gradient is Lipschitz continuous.

Definition 4.3 (Lipschitz Continuity of the Gradient). A function f has a Lipschitz continuous gradient if there exists a constant $L > 0$ (the Lipschitz constant) such that for all $\mathbf{x}, \mathbf{y}$ in the domain:

$$\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\| \leq L\|\mathbf{x} - \mathbf{y}\| \tag{4.10}$$

This is a measure of the smoothness of the gradient. For twice-differentiable functions, it is related to the maximum eigenvalue of the Hessian. A key consequence, derived from the Mean Value Theorem, is the following quadratic upper bound on the function:

$$f(\mathbf{x} + \mathbf{p}) \leq f(\mathbf{x}) + \nabla f(\mathbf{x})^\top \mathbf{p} + \frac{L}{2}\|\mathbf{p}\|^2 \tag{4.11}$$

Proof of Termination for Backtracking Line Search Here, we prove that the backtracking line search algorithm (using the Armijo rule) finds a step size α that satisfies the condition in a finite number of iterations, assuming the gradient ∇f is Lipschitz continuous.

The Armijo condition is given by:

$$f(\mathbf{x}_k + \alpha \mathbf{p}_k) \leq f(\mathbf{x}_k) + c_1 \alpha \nabla f_k^\top \mathbf{p}_k$$

From the definition of Lipschitz continuity, we have the quadratic upper bound on f shown in (4.11). Applying this with $\mathbf{p} = \alpha \mathbf{p}_k$, we get:

$$f(\mathbf{x}_k + \alpha \mathbf{p}_k) \leq f(\mathbf{x}_k) + \alpha \nabla f_k^\top \mathbf{p}_k + \frac{L}{2}\alpha^2 \|\mathbf{p}_k\|^2$$

For the Armijo condition to hold, it is sufficient that the upper bound on $f(\mathbf{x}_k + \alpha \mathbf{p}_k)$ satisfies the condition. We therefore seek an α such that:

$$f(\mathbf{x}_k) + \alpha \nabla f_k^\top \mathbf{p}_k + \frac{L}{2}\alpha^2 \|\mathbf{p}_k\|^2 \leq f(\mathbf{x}_k) + c_1 \alpha \nabla f_k^\top \mathbf{p}_k$$

Subtracting $f(\mathbf{x}_k)$ from both sides and rearranging gives:

$$(1 - c_1)\alpha \nabla f_k^\top \mathbf{p}_k + \frac{L}{2}\alpha^2 \|\mathbf{p}_k\|^2 \leq 0$$

Since $\alpha > 0$, we can divide by it:

$$(1 - c_1)\nabla f_k^\top \mathbf{p}_k + \frac{L}{2}\alpha \|\mathbf{p}_k\|^2 \leq 0$$

Now, we solve for α. Since $\mathbf{p}_k$ is a descent direction, we know $\nabla f_k^\top \mathbf{p}_k < 0$. Also, $c_1 \in (0, 1)$ means $1 - c_1 > 0$.

$$\alpha \leq \frac{2(1 - c_1)(-\nabla f_k^\top \mathbf{p}_k)}{L\|\mathbf{p}_k\|^2} \tag{4.12}$$

This inequality shows that any α that is sufficiently small (i.e., below the positive threshold on the right-hand side) will satisfy the Armijo condition. The backtracking algorithm starts with an initial α (e.g., $\alpha = 1$) and multiplies it by a contraction factor $c \in (0, 1)$ until the condition is met. Since each step reduces α, it is guaranteed to eventually fall below this threshold. Therefore, the backtracking line search terminates in a finite number of steps. This proves that the backtracking line search is a well-posed and practical compromise, avoiding the expense of exact search while still guaranteeing sufficient progress towards the minimum.

Lecture 5: Solving Inequality-Constrained Problems

5.1 Challenges with Inequality Constraints

Recall the KKT conditions for a problem $\min f(\mathbf{x})$ s.t. $\mathbf{c}(\mathbf{x}) \geq 0$:

1. **Stationarity**: $\nabla f(\mathbf{x}^*) - (\frac{\partial \mathbf{c}}{\partial \mathbf{x}})^\top \lambda^* = 0$ (note the sign change on λ is a common convention for $\geq$ constraints)
2. **Primal Feasibility**: $\mathbf{c}(\mathbf{x}^*) \geq 0$
3. **Dual Feasibility**: $\lambda^* \geq 0$
4. **Complementary Slackness**: $\lambda \odot c(\mathbf{x}^*) = 0$

Role of Complementary Slackness ("Switching"). For each constraint component $c_i(\mathbf{x}) \geq 0$ with multiplier $\lambda_i \geq 0$, the complementarity condition $\lambda_i c_i(\mathbf{x}^*) = 0$ enforces an automatic switch:

- (Inactive case) If at the solution $c_i(\mathbf{x}^*) > 0$, then necessarily $\lambda_i = 0$. The stationarity condition then contains no contribution from this constraint, so locally this constraint behaves as if it were absent; i.e., the problem reduces (locally) to an unconstrained one in that direction.
- (Active case) If $c_i(\mathbf{x}^*) = 0$, the multiplier can be strictly positive $\lambda_i > 0$. In that case the gradient contribution $-\lambda_i \nabla c_i(\mathbf{x}^*)$ appears in stationarity, and c_i effectively acts like an equality constraint: motion that would violate $c_i \geq 0$ is opposed by a restoring first-order term. The boundary is therefore "sticky": any feasible descent direction must be tangent to $c_i(\mathbf{x}) = 0$.

Thus, complementary slackness encodes a discrete (active/inactive) combinatorial choice inside a continuous system of equations. This logical structure is the main obstacle to naively applying Newton's method directly to all KKT conditions: the feasible manifold changes dimension depending on which constraints are active.

The presence of the inequality conditions (2 and 3) and the complementarity constraint (4) prevents us from directly applying Newton's method to solve this system as a standard root-finding problem. We need more sophisticated approaches.

5.2 Methods for Inequality-Constrained Optimization

5.2.1 Active-Set Methods

These methods work by maintaining a "guess" of which constraints will be active (equal to zero) at the solution.

1. Solve the equality-constrained problem assuming the current active set is correct.
2. Check the KKT conditions. If any Lagrange multipliers for the active set are negative, it means the objective could be improved by making that constraint inactive. Remove it from the active set.
3. If any inactive constraints are violated, add them to the active set.
4. Repeat until all conditions are satisfied.

Active-set methods work well for problems where the active set doesn't change much between iterations, such as Quadratic Programs (QPs).

Consequences of a Wrong Active Set Guess. Suppose the true optimal active set is $\mathcal{A}^* = \{i : c_i(\mathbf{x}^*) = 0\}$.

- (Missing an active constraint) If we temporarily exclude some $j \in \mathcal{A}^*$ from the working set, we solve a relaxation. The intermediate solution $\tilde{\mathbf{x}}$ will typically violate feasibility with $c_j(\tilde{\mathbf{x}}) < 0$ (if constraints were encoded as $c_i(\mathbf{x}) \geq 0$), flagging the need to add constraint j.
- (Including an inactive constraint) If we include some $k \notin \mathcal{A}^*$, we may obtain a Lagrange multiplier $\lambda_k < 0$. Violated dual feasibility indicates the constraint should be removed, since maintaining it would artificially restrict descent directions.

These tests (primal violation for missing constraints, negative multipliers for superfluous constraints) drive the update logic.

Combinatorial Nature and Complexity. In the worst case with m inequality constraints there are 2^m possible active sets. Enumerating them is exponential. Active-set methods avoid this by iteratively refining a single guess, but they can still require many iterations if the active set changes frequently or if the problem is ill-conditioned. They are most effective when the optimal active set is small or changes little between similar problems (e.g., in sequential QPs for trajectory optimization).

5.2.2 Penalty and Augmented Lagrangian Methods

These methods transform the constrained problem into an unconstrained one by adding a penalty term to the objective that discourages constraint violation. For a problem $\min f(\mathbf{x})$ s.t. $c(x) \geq 0$, the penalty formulation is:

$$\min_{\mathbf{x}} f(\mathbf{x}) + \frac{\rho}{2}(\min(0, c(x)))^2$$

The solver gradually increases the penalty weight $\rho \to \infty$. A major drawback is that large ρ values lead to an ill-conditioned Hessian, making the problem difficult to solve accurately. Additionally, at the constraint boundary, the Hessian is techinicaly not defined, so second order methods like Newton's method can struggle, even though they might work in practice. One of the recent modifications to

The **Augmented Lagrangian** method improves upon this by introducing an estimate of the Lagrange multipliers, avoiding the need for $\rho \to \infty$. This is a very powerful and popular technique.

5.2.3 The Barrier Problem

Consider the problem $\min f(\mathbf{x})$ s.t. $\mathbf{c}(\mathbf{x}) \geq 0$. We first introduce **slack variables** $\mathbf{s} \geq 0$ to convert the general inequality into an equality:

$$\begin{aligned} \min_{\mathbf{x},\mathbf{s}} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & \mathbf{c}(\mathbf{x}) - \mathbf{s} = 0 \\ & \mathbf{s} \geq 0 \end{aligned}$$

Now, we enforce the non-negativity constraint $\mathbf{s} \geq 0$ by adding a logarithmic barrier term to the objective. This creates a new equality-constrained subproblem, parameterized by the barrier parameter $\rho > 0$:

$$\min_{\mathbf{x},\mathbf{s}} \quad f(\mathbf{x}) - \rho \sum_i \log(s_i) \quad \text{s.t.} \quad \mathbf{c}(\mathbf{x}) - \mathbf{s} = 0 \tag{5.1}$$

The $-\log(s_i)$ term acts as a barrier: as $s_i \to 0^+$, its value goes to infinity, preventing the solver from ever making a slack variable non-positive. This keeps the iterates strictly "interior" to the feasible set.

5.3 Interior-Point Methods

Code 5.1 (Julia Notebook: Interior-Point Method).
The 'interior-point.ipynb' notebook implements a simplified interior-point method to solve a quadratic program with a linear inequality constraint.

- **Formulation**: Instead of slack variables, it uses a clever change of variables, $s = \sqrt{\rho}e^{\sigma}$ and $\lambda = \sqrt{\rho}e^{-\sigma}$, to implicitly enforce positivity and the relaxed complementarity condition. This transforms the problem into finding the roots of a system of two equations in $(\mathbf{x}, \sigma)$.
- **Central Path Following**: The notebook demonstrates the concept of the central path. By starting with a large ρ and solving, and then using that solution as a warm start for a smaller ρ, you can trace the path of solutions. The plot shows the solver taking a step towards the constraint boundary with a large ρ, and then converging directly to the true solution as ρ is decreased to a small value (e.g., 10^{-8}).
- **Robustness**: The solver uses Newton's method with a line search to robustly solve the root-finding problem at each ρ value.

This method is the engine behind powerful, open-source solvers like IPOPT, which are widely used in robotics for problems like trajectory optimization.

5.4 Quadratic Programs (QPs)

A particularly important class of optimization problems is the Quadratic Program (QP), which involves minimizing a quadratic objective subject to linear constraints.

$$\begin{aligned} \min_{\mathbf{x}} \quad & \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{q}^\top \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & \mathbf{C}\mathbf{x} = \mathbf{d} \end{aligned}$$

If the matrix $\mathbf{Q}$ is positive definite, the problem is convex, meaning it has a single global minimum. QPs can be solved extremely quickly (often in kHz) by specialized solvers, making them a cornerstone of many real-time control and estimation algorithms, including Model Predictive Control (MPC) and contact-implicit optimal control.

Lecture 6: Duality, Regularization, and Merit Functions

6.1 A Dual Perspective on Constraints

Let's revisit the core idea of constrained optimization. For an equality constrained problem of the form:

$$\begin{aligned} \min_{\mathbf{x}\in\mathbb{R}^n} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{c}(\mathbf{x}) = 0 \end{aligned}$$

We can recast this problem into an unconstrained form by using an **indicator function**, which assigns an infinite penalty to any point that is not feasible.

$$\min_{\mathbf{x}} f(\mathbf{x}) + \mathcal{P}_\infty(\mathbf{c}(\mathbf{x})) \quad \text{where} \quad \mathcal{P}_\infty(\mathbf{v}) = \begin{cases} 0, & \text{if } \mathbf{v} = 0 \\ +\infty, & \text{if } \mathbf{v} \neq 0 \end{cases}$$

While this formulation is mathematically equivalent, it is computationally intractable due to the discontinuous nature of the indicator function. A more practical approach is to achieve the same effect through duality. The indicator function can be expressed as a maximum over a new set of variables, the Lagrange multipliers λ:

$$\mathcal{P}_\infty(\mathbf{c}(\mathbf{x})) = \max_{\lambda\in\mathbb{R}^m} \lambda^\top \mathbf{c}(\mathbf{x})$$

Substituting this into our minimization problem gives us the primal-dual formulation, where we seek a saddle point of the Lagrangian:

$$\min_{\mathbf{x}} \max_{\lambda} \mathcal{L}(\mathbf{x}, \lambda) = \min_{\mathbf{x}} \max_{\lambda} \Big(f(\mathbf{x}) + \lambda^\top \mathbf{c}(\mathbf{x}) \Big)$$

Whenever the constraint $\mathbf{c}(\mathbf{x}) \neq 0$, the inner maximization problem with respect to λ will blow up, effectively enforcing the constraint.

This concept can also be applied to inequality constraints. For an inequality constrained problem:

$$\begin{aligned} \min_{\mathbf{x}\in\mathbb{R}^n} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{c}(\mathbf{x}) \geq 0 \end{aligned}$$

We can recast this problem into an unconstrained form by using an **indicator function** for the inequality constraint, which assigns an infinite penalty to any point that violates feasibility.

$$\min_{\mathbf{x}} f(\mathbf{x}) + \mathcal{P}^-_\infty(\mathbf{c}(\mathbf{x})) \quad \text{where} \quad \mathcal{P}^-_\infty(\mathbf{v}) = \begin{cases} 0, & \text{if } \mathbf{v} \geq 0 \\ +\infty, & \text{if } \mathbf{v} \not\geq 0 \end{cases}$$

While this formulation is mathematically equivalent, it is computationally intractable due to the discontinuous nature of the indicator function. A more practical approach is to achieve the same effect through duality. The indicator function for inequality constraints can be expressed as a maximum over a restricted set of Lagrange multipliers $\lambda \geq 0$:

$$\mathcal{P}^-_\infty(\mathbf{c}(\mathbf{x})) = \max_{\lambda\geq 0} -\lambda^\top \mathbf{c}(\mathbf{x})$$

To see why this holds, note that if $\mathbf{c}(\mathbf{x}) \geq 0$ (feasible), then $-\lambda^\top \mathbf{c}(\mathbf{x}) \leq 0$ for all $\lambda \geq 0$, and the maximum is zero (achieved by setting $\lambda = 0$). Conversely, if any component $c_i(\mathbf{x}) < 0$ (infeasible), we can take the corresponding $\lambda_i \to +\infty$, which drives $-\lambda_i c_i(\mathbf{x}) \to +\infty$, making the maximum infinite.

Substituting this into our minimization problem gives us the primal-dual formulation for inequality constraints, where we seek a saddle point of the Lagrangian:

$$\min_{\mathbf{x}} \max_{\lambda \geq 0} \mathcal{L}(\mathbf{x}, \lambda) = \min_{\mathbf{x}} \max_{\lambda \geq 0} \Big(f(\mathbf{x}) - \lambda^\top \mathbf{c}(\mathbf{x}) \Big)$$

Whenever the constraint $\mathbf{c}(\mathbf{x}) \not\geq 0$, the inner maximization problem with respect to λ will blow up, effectively enforcing the constraint. Note the sign convention: for inequality constraints $\mathbf{c}(\mathbf{x}) \geq 0$, the Lagrangian uses $-\lambda^\top \mathbf{c}(\mathbf{x})$ with $\lambda \geq 0$, in contrast to the equality case where we had $+\lambda^\top \mathbf{c}(\mathbf{x})$ with λ unrestricted.

Intuition. This saddle-point view is fundamental. The KKT conditions we derived earlier are precisely the conditions for a point $(\mathbf{x}^\star, \lambda^\star)$ to be a stationary point (a saddle point) of the Lagrangian. At this point, the gradient with respect to both $\mathbf{x}$ and λ is zero. The KKT matrix from Equation 4.3 is the Hessian of the Lagrangian with respect to the concatenated vector $(\mathbf{x}, \lambda)$. For a true saddle point, this Hessian should not be positive definite, but rather **quasi-definite**: it must have a specific inertia, with a number of positive eigenvalues equal to the dimension of $\mathbf{x}$ and a number of negative eigenvalues equal to the dimension of λ.

6.2 Regularization of the KKT System

When we solve the KKT system (Equation 4.3) using Newton's method, we rely on the KKT matrix having the correct inertia. If it does not, for example, if the Hessian of the Lagrangian $\nabla^2_{\mathbf{xx}}\mathcal{L}$ is not positive definite on the tangent space of the constraints, the resulting Newton step may not be a descent direction. This can cause the optimization to diverge.

To robustify the solver, we can regularize the KKT matrix. Similar to the damped Newton's method for unconstrained optimization, we can add a multiple of the identity to ensure the correct matrix properties.

$$\begin{bmatrix} \nabla^2_{\mathbf{xx}}\mathcal{L} + \beta\mathbf{I} & (\frac{\partial \mathbf{c}}{\partial \mathbf{x}})^\top \\ \frac{\partial \mathbf{c}}{\partial \mathbf{x}} & -\beta\mathbf{I} \end{bmatrix} \begin{bmatrix} \delta\mathbf{x} \\ \delta\lambda \end{bmatrix} = - \begin{bmatrix} \nabla_{\mathbf{x}}\mathcal{L} \\ \mathbf{c}(\mathbf{x}) \end{bmatrix}$$

where $\beta > 0$ is a regularization parameter. By adding $\beta\mathbf{I}$ to the top-left block and $-\beta\mathbf{I}$ to the bottom-right, we can "push" the eigenvalues of the matrix to have the desired signs, guaranteeing that the computed step $(\delta\mathbf{x}, \delta\lambda)$ is a valid descent direction for our merit function.

Code 6.1 (Julia Notebook: Regularization).
The 'regularization.ipynb' notebook provides a clear example of this issue. It sets up an equality constrained quadratic program. At a particular guess for $(\mathbf{x}, \lambda)$, it computes the eigenvalues of the KKT matrix and finds that it does not have the correct inertia (i.e., it is not quasi-definite). A standard Newton step from this point would fail.

The notebook then implements a regularized Newton step. It includes a 'while' loop that checks the eigenvalues of the KKT matrix. If the inertia is incorrect, it adds a small damping term β and re-checks. This process continues until the matrix is guaranteed to be quasi-definite. The resulting step correctly moves towards the constrained minimum. This demonstrates how regularization is a crucial tool for creating robust, globalized solvers for constrained optimization.

6.3 Merit Functions and Line Search

Regularization gives us a good search direction, but it doesn't tell us how far to step along it. A full Newton step $\alpha = 1$ might overshoot the minimum or even increase the objective or

constraint violation. We need a scalar-valued **merit function** $\Phi(\mathbf{x}, \lambda)$ that allows us to perform a line search to find an appropriate step size α.

The goal of the merit function is to provide a single value that balances the competing goals of reducing the objective function and satisfying the constraints. There are several common choices:

1. l_2 **Merit Function**: Based on the squared norm of the KKT residual. This measures how close we are to satisfying the optimality conditions.

$$\Phi(\mathbf{x}, \lambda) = \frac{1}{2} \left\| \begin{bmatrix} \nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \lambda) \\ \min(0, \mathbf{c}(\mathbf{x})) \\ \mathbf{d}(\mathbf{x}) \end{bmatrix} \right\|_2^2$$

2. l_1 **Penalty Function (Augmented Lagrangian)**: This is a very popular and effective choice. It combines the objective function with an l_1 penalty on the constraint violation.

$$\Phi(\mathbf{x}) = f(\mathbf{x}) + \rho \left\| \begin{bmatrix} \min(0, \mathbf{c}(\mathbf{x})) \\ \mathbf{d}(\mathbf{x}) \end{bmatrix} \right\|_1$$

   Here, $\rho > 0$ is a penalty parameter that weights the importance of feasibility.

Once we have a merit function Φ and a search direction $\Delta \mathbf{z} = (\delta \mathbf{x}, \delta \lambda)$, we can use a standard backtracking line search (like the Armijo rule from Equation 4.5) to find a step length $\alpha \in (0, 1]$ that ensures sufficient decrease in the merit function. Starting with $\alpha = 1$, we iteratively reduce the step size:

$$\text{while } \Phi(\mathbf{z} + \alpha \Delta \mathbf{z}) > \Phi(\mathbf{z}) + c_1 \alpha \nabla \Phi(\mathbf{z})^\top \Delta \mathbf{z} \quad \text{do} \quad \alpha \leftarrow \tau \alpha$$

where $\tau \in (0, 1)$ is the backtracking factor and $c_1 \in (0, 1)$ is the Armijo parameter. The loop terminates when the the Armijo condition is satisfied, at which point we accept the step $\mathbf{z}^{k+1} = \mathbf{z}^k + \alpha \Delta \mathbf{z}$.

This combination of a robustly computed direction (via a regularized KKT solve) and a careful selection of step size (via a line search on a merit function) forms the core of modern Sequential Quadratic Programming (SQP) and Interior-Point solvers.

Code 6.2 (Julia Notebook: Merit Functions).
The 'merit-functions.ipynb' notebook illustrates the necessity of a line search. It first computes a Gauss-Newton step for a constrained problem. The plot shows that taking a full step ($\alpha = 1$) significantly overshoots the solution.

The notebook then defines an l_1 merit function. It implements an Armijo backtracking line search that iteratively reduces α by half until the sufficient decrease condition is met. The final step, scaled by the accepted α, makes clear progress towards the true minimum without overshooting. This powerfully demonstrates how a merit function and line search "globalize" the convergence of Newton-like methods, ensuring progress even when far from the optimal solution.

Lecture 7: Deterministic Optimal Control and LQR

7.1 The Optimal Control Problem

We now formally define the problem of optimal control. We begin with the continuous-time formulation, which is the most general.

Definition 7.1 (Continuous-Time Optimal Control). Find the state trajectory $\mathbf{x}(t) \in \mathbb{R}^n$ and control trajectory $\mathbf{u}(t) \in \mathbb{R}^m$ that solve the following minimization problem:

$$\begin{aligned} \min_{\mathbf{x}(t),\mathbf{u}(t)} \quad & J\left(\mathbf{x}(t), \mathbf{u}(t)\right) = \int_{t_0}^{t_f} l(\mathbf{x}(t), \mathbf{u}(t), t)\, dt + l_F(\mathbf{x}(t_f)) \\ \text{subject to} \quad & \dot{\mathbf{x}}(t) = f(\mathbf{x}(t), \mathbf{u}(t)) \quad \text{(Dynamics Constraint)} \\ & \mathbf{x}(t_0) = \mathbf{x}_0 \quad \text{(Initial Condition)} \\ & \mathbf{g}(\mathbf{x}(t), \mathbf{u}(t)) \leq 0 \quad \text{(Path Constraints, optional)} \\ & \mathbf{h}(\mathbf{x}(t_f)) = 0 \quad \text{(Terminal Constraints, optional)} \end{aligned}$$

Here, $l(\cdot)$ is the **stage cost** (or running cost) and $l_F(\cdot)$ is the **terminal cost**.

This is an **infinite-dimensional** optimization problem because we are optimizing over functions ($\mathbf{x}(t)$ and $\mathbf{u}(t)$), which live in infinite-dimensional spaces. The solutions that we get from solving the above optimisation problems are open loop trajectories, not feedback control policies that we expect in robotic control. Analytic solutions are extremely rare, existing only for a few special cases (like LQR, which we will see).

To make the problem computationally tractable, we discretize it in time.

Definition 7.2 (Discrete-Time Optimal Control). Find the sequence of states $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ and controls $\mathbf{U} = \{\mathbf{u}_1, \ldots, \mathbf{u}_{N-1}\}$ that solve:

$$\begin{aligned} \min_{\mathbf{X},\mathbf{U}} \quad & J = \sum_{k=1}^{N-1} l(\mathbf{x}_k, \mathbf{u}_k) + l_F(\mathbf{x}_N) \\ \text{subject to} \quad & \mathbf{x}_{k+1} = f_d(\mathbf{x}_k, \mathbf{u}_k) \quad \text{for } k = 1, \ldots, N-1 \\ & \mathbf{x}_1 = \mathbf{x}_0 \\ & \mathbf{c}(\mathbf{x}_k, \mathbf{u}_k) \geq 0 \quad \text{(e.g., torque limits, safety)} \end{aligned}$$

This is now a **finite-dimensional** (though very large) constrained optimization problem. The variables are the **knot points** $(\mathbf{x}_k, \mathbf{u}_k)$ of the trajectory. We can solve this using the KKT conditions we derived in previous lectures. To go from the continuous to discrete formulation, we need to choose a discretization scheme for the dynamics f_d (e.g., Euler, Runge-Kutta). While, to go from discrete to continuous, to upsample the trajectory for robot execution, we can use interpolation methods (e.g., cubic splines).

7.2 Pontryagin's Minimum Principle (Discrete-Time)

These provide the first-order necessary conditions for optimality for deterministic optimal control problems. In discrete time, they can be seen as an extension of the KKT conditions to account for the dynamics constraints.

Let's derive them. We have a minimization problem with only equality constraints ($\mathbf{x}_{k+1} - f_d(\mathbf{x}_k, \mathbf{u}_k) = 0$). We can form the Lagrangian:

Definition 7.3 (Lagrangian for Optimal Control).

$$\mathcal{L}(\mathbf{X}, \mathbf{U}, \mathbf{\Lambda}) = l_F(\mathbf{x}_N) + \sum_{k=1}^{N-1} \left(l(\mathbf{x}_k, \mathbf{u}_k) + \lambda_{k+1}^\top \left(f_d(\mathbf{x}_k, \mathbf{u}_k) - \mathbf{x}_{k+1} \right) \right) \tag{7.1}$$

where $\mathbf{\Lambda} = \{\lambda_2, \ldots, \lambda_N\}$ is the sequence of Lagrange multipliers, which we call the **costate**.

To simplify notation, we introduce the **Hamiltonian**, which groups the terms inside the summation:

$$H_k(\mathbf{x}_k, \mathbf{u}_k, \lambda_k) = l(\mathbf{x}_k, \mathbf{u}_k) + \lambda_k^\top f_d(\mathbf{x}_k, \mathbf{u}_k) \tag{7.2}$$

The Lagrangian can then be rewritten as:

$$\mathcal{L}(\mathbf{X}, \mathbf{U}, \mathbf{\Lambda}) = H_1(\mathbf{x}_1, \mathbf{u}_1, \lambda_2) + \sum_{k=2}^{N-1} H_k(\mathbf{x}_k, \mathbf{u}_k, \lambda_{k+1}) + l_F(\mathbf{x}_N) - \lambda_N^\top \mathbf{x}_N \tag{7.3}$$

Now, we can find the stationary point of $\mathcal{L}$ by taking its gradient with respect to all variables and setting them to zero.

Theorem 7.1 (Discrete-Time PMP (KKT Conditions)). A trajectory $(\mathbf{X}^\star, \mathbf{U}^\star)$ is a candidate for a local optimum only if there exists a costate trajectory $\mathbf{\Lambda}^\star$ such that the following conditions hold:

1. **State Equation (Primal Feasibility):** $\nabla_{\lambda_{k+1}} \mathcal{L} = 0 \Rightarrow \mathbf{x}_k^\star = f_d(\mathbf{x}_k^\star, \mathbf{u}_k^\star)$ (This just recovers the dynamics).
2. **Costate Equation (Backward):** $\nabla_{\mathbf{x}_k} \mathcal{L} = 0 \Rightarrow \lambda_k^\star = \nabla_{\mathbf{x}} H_k = \nabla_{\mathbf{x}} l_k + \left(\frac{\partial f_d}{\partial \mathbf{x}_k} \right)^\top \lambda_{k+1}^\star$ (This is a backward recurrence for λ).
3. **Terminal Costate Condition:** $\nabla_{\mathbf{x}_N} \mathcal{L} = 0 \Rightarrow \lambda_N^\star = \nabla_{\mathbf{x}} l_F(\mathbf{x}_N^\star)$ (This is the boundary condition for the costate).
4. **Stationarity of Hamiltonian:** $\nabla_{\mathbf{u}_k} \mathcal{L} = 0 \Rightarrow \nabla_{\mathbf{u}} H_k = \nabla_{\mathbf{u}} l_k + \left(\frac{\partial f_d}{\partial \mathbf{u}_k} \right)^\top \lambda_{k+1}^\star = 0$

If control constraints $\mathbf{u}_k \in \mathcal{U}$ exist, condition 4 is replaced by:

$$\mathbf{u}_k^\star = \arg\min_{\mathbf{u} \in \mathcal{U}} H_k(\mathbf{x}_k^\star, \mathbf{u}, \lambda_{k+1}^\star)$$

The discrete time PMP conditions can be converted into continuous time by taking the limit as the time step goes to zero. The resulting conditions are very similar, with the costate equation becoming a differential equation and the Hamiltonian stationarity condition remaining similar. These are as follows:

1. **State Equation:** $\dot{\mathbf{x}}^\star(t) = f(\mathbf{x}^\star(t), \mathbf{u}^\star(t))$
2. **Costate Equation:** $-\dot{\lambda}^\star(t) = \nabla_{\mathbf{x}} H(\mathbf{x}^\star(t), \mathbf{u}^\star(t), \lambda^\star(t))$
3. **Terminal Condition:** $\lambda^\star(t_f) = \nabla_{\mathbf{x}} l_F(\mathbf{x}^\star(t_f))$
4. **Hamiltonian Stationarity:** $\mathbf{u}^\star(t) = \arg\min_{\mathbf{u} \in \mathcal{U}} H(\mathbf{x}^\star(t), \mathbf{u}, \lambda^\star(t))$

7.3 The Linear Quadratic Regulator (LQR)

The LQR problem is a special case where the dynamics are linear and the cost is quadratic.

Definition 7.4 (Discrete-Time LQR).

$$\begin{aligned} \min_{\mathbf{X},\mathbf{U}} \quad & J = \frac{1}{2}\mathbf{x}_N^\top \mathbf{Q}_N \mathbf{x}_N + \sum_{k=1}^{N-1} \frac{1}{2}\left(\mathbf{x}_k^\top \mathbf{Q}_k \mathbf{x}_k + \mathbf{u}_k^\top \mathbf{R}_k \mathbf{u}_k\right) \\ \text{subject to} \quad & \mathbf{x}_{k+1} = \mathbf{A}_k \mathbf{x}_k + \mathbf{B}_k \mathbf{u}_k \end{aligned}$$

We assume $\mathbf{Q}_k \succeq 0$ (positive semidefinite) and $\mathbf{R}_k \succ 0$ (positive definite).

LQR is foundational because it can be solved efficiently and serves as a local approximation for nonlinear problems.

7.3.1 Solving LQR via PMP

Let's apply the PMP conditions to the LQR problem. The Hamiltonian is:

$$H_k = \frac{1}{2}(\mathbf{x}_k^\top \mathbf{Q}_k \mathbf{x}_k + \mathbf{u}_k^\top \mathbf{R}_k \mathbf{u}_k) + \lambda_{k+1}^\top (\mathbf{A}_k \mathbf{x}_k + \mathbf{B}_k \mathbf{u}_k)$$

Applying the PMP conditions:

1. **State Eq.:** $\mathbf{x}_{k+1} = \mathbf{A}_k \mathbf{x}_k + \mathbf{B}_k \mathbf{u}_k$
2. **Costate Eq.:** $\lambda_k = \nabla_{\mathbf{x}} H_k = \mathbf{Q}_k \mathbf{x}_k + \mathbf{A}_k^\top \lambda_{k+1}$
3. **Terminal Cond.:** $\lambda_N = \nabla_{\mathbf{x}} l_F = \mathbf{Q}_N \mathbf{x}_N$
4. **Stationarity:** $\nabla_{\mathbf{u}} H_k = \mathbf{R}_k \mathbf{u}_k + \mathbf{B}_k^\top \lambda_{k+1} = 0$

From (4), we can find the optimal control $\mathbf{u}_k^\star$ in terms of the costate:

$$\mathbf{u}_k^\star = -\mathbf{R}_k^{-1} \mathbf{B}_k^\top \lambda_{k+1} \tag{7.4}$$

7.4 Indirect Shooting Methods

The PMP conditions give us a **two-point boundary value problem**. The state evolves forward from $\mathbf{x}_1$, while the costate evolves backward from λ_N. These two are coupled: $\mathbf{x}$ depends on $\mathbf{u}$, which depends on λ, and λ depends on $\mathbf{x}$.

Indirect shooting methods (now less common) try to solve this by guessing the initial costate λ_1 and integrating both systems forward, then adjusting λ_1 until the terminal condition $\lambda_N = \mathbf{Q}_N \mathbf{x}_N$ is met.

A more modern approach, often called a "shooting method" in trajectory optimization, is to treat the problem as a large optimization on *only* the control sequence $\mathbf{U}$.

$$\min_{\mathbf{U}} J(\mathbf{U}) \quad \text{where} \quad J(\mathbf{U}) = J(\mathbf{X}(\mathbf{U}), \mathbf{U})$$

We can solve this with gradient descent. The PMP equations give us a highly efficient way to compute the gradient $\nabla_{\mathbf{U}} J$.

The gradient of the total cost J with respect to a single control $\mathbf{u}_k$ is exactly the gradient of the Hamiltonian:

$$\nabla_{\mathbf{u}_k} J = (\nabla_{\mathbf{u}_k} \mathcal{L})^\top = (\nabla_{\mathbf{u}} H_k)^\top = \mathbf{R}_k \mathbf{u}_k + \mathbf{B}_k^\top \lambda_{k+1}$$

This gives us a full algorithm.

Algorithm 5: Gradient Descent for LQR (Shooting Method)

Input: $\mathbf{A}_k, \mathbf{B}_k, \mathbf{Q}_k, \mathbf{R}_k$, $\mathbf{x}_1$, initial guess $\mathbf{U} = (\mathbf{u}_1, \ldots, \mathbf{u}_{N-1})$
Output: Optimal $\mathbf{U}^\star, \mathbf{X}^\star$

while *not converged* **do**
  `// 1. Forward Pass (Rollout)`
  $\mathbf{x}_{k+1} \leftarrow \mathbf{A}_k \mathbf{x}_k + \mathbf{B}_k \mathbf{u}_k$ for $k = 1..N-1$
  $\mathbf{X} \leftarrow (\mathbf{x}_1, \ldots, \mathbf{x}_N)$
  Compute $J_{old} = J(\mathbf{X}, \mathbf{U})$
  `// 2. Backward Pass (Compute Costate/Adjoint)`
  $\lambda_N \leftarrow \mathbf{Q}_N \mathbf{x}_N$
  $\lambda_k \leftarrow \mathbf{Q}_k \mathbf{x}_k + \mathbf{A}_k^\top \lambda_{k+1}$ for $k = N-1..1$
  `// 3. Compute Search Direction`
  $\mathbf{g}_k \leftarrow \mathbf{R}_k \mathbf{u}_k + \mathbf{B}_k^\top \lambda_{k+1}$ `// Gradient` $\nabla_{\mathbf{u}_k} J$
  $\Delta \mathbf{u}_k \leftarrow -\mathbf{R}_k^{-1} \mathbf{g}_k$ `// Newton step direction`
  Set $\Delta \mathbf{U} \leftarrow (\Delta \mathbf{u}_1, \ldots, \Delta \mathbf{u}_{N-1})$
  if $\|\Delta \mathbf{U}\|_\infty < \epsilon$ **then**
    return $\mathbf{U}, \mathbf{X}$
  `// 4. Backtracking Line Search`
  $\alpha \leftarrow 1.0$
  $\mathbf{p} \leftarrow (\nabla J)^\top \Delta \mathbf{U} = \sum_k \mathbf{g}_k^\top \Delta \mathbf{u}_k = -\sum_k \mathbf{g}_k^\top \mathbf{R}_k^{-1} \mathbf{g}_k$
  while $J(\mathbf{U} + \alpha \Delta \mathbf{U}) > J_{old} + c_1 \alpha \mathbf{p}$ **do**
    $\alpha \leftarrow 0.5\alpha$
  `// 5. Update Control`
  $\mathbf{U} \leftarrow \mathbf{U} + \alpha \Delta \mathbf{U}$

Code 7.1 (Julia Notebook: LQR Shooting).
FILL THIS OUT

Lecture 8: LQR Solutions: QP and the Riccati Recursion

8.1 LQR as a single Quadratic Program (QP)

Last time, we treated the LQR problem as a search over the control sequence $\mathbf{U}$ only, with the state $\mathbf{X}$ being an implicit function of $\mathbf{U}$. An alternative, "direct" approach is to make *both* the states and controls decision variables in one large optimization problem.

Recall the LQR problem:

$$\begin{aligned} \min_{\mathbf{X},\mathbf{U}} \quad & J = \frac{1}{2}\mathbf{x}_N^\top \mathbf{Q}_N \mathbf{x}_N + \sum_{k=1}^{N-1} \frac{1}{2}\left(\mathbf{x}_k^\top \mathbf{Q}_k \mathbf{x}_k + \mathbf{u}_k^\top \mathbf{R}_k \mathbf{u}_k\right) \\ \text{subject to} \quad & \mathbf{x}_{k+1} = \mathbf{A}_k \mathbf{x}_k + \mathbf{B}_k \mathbf{u}_k, \quad \mathbf{x}_1 = \mathbf{x}_0 \end{aligned}$$

(Note: For simplicity, we can omit the $\frac{1}{2}\mathbf{x}_1^\top \mathbf{Q}_1 \mathbf{x}_1$ term from the sum, as $\mathbf{x}_1 = \mathbf{x}_0$ is a fixed constant and doesn't affect the optimal solution.)

We can stack all the decision variables into a single, large vector $\mathbf{z}$. A convenient stacking is:

$$\mathbf{z} = \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{x}_2 \\ \mathbf{u}_2 \\ \mathbf{x}_3 \\ \vdots \\ \mathbf{u}_{N-1} \\ \mathbf{x}_N \end{bmatrix} \in \mathbb{R}^{(N-1)(m+n)}$$

The total cost J is a quadratic function of $\mathbf{z}$, which we can write as $\frac{1}{2}\mathbf{z}^\top \mathbf{H} \mathbf{z}$, where $\mathbf{H}$ is a large block-diagonal matrix:

$$\mathbf{H} = \text{blockdiag}(\mathbf{R}_1, \mathbf{Q}_2, \mathbf{R}_2, \mathbf{Q}_3, \ldots, \mathbf{R}_{N-1}, \mathbf{Q}_N)$$

The dynamics constraints are all linear in $\mathbf{z}$. We can express them in the standard QP form $\mathbf{C}\mathbf{z} = \mathbf{d}$:

$$\begin{bmatrix} \mathbf{B}_1 & -\mathbf{I} & 0 & 0 & \cdots & \\ 0 & \mathbf{A}_2 & \mathbf{B}_2 & -\mathbf{I} & \cdots & \\ \vdots & & \ddots & \ddots & & \\ 0 & \cdots & 0 & \mathbf{A}_{N-1} & \mathbf{B}_{N-1} & -\mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{x}_2 \\ \mathbf{u}_2 \\ \mathbf{x}_3 \\ \vdots \\ \mathbf{x}_N \end{bmatrix} = \begin{bmatrix} -\mathbf{A}_1 \mathbf{x}_1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Thus, the entire LQR problem is equivalent to the (convex) Quadratic Program:

$$\begin{aligned} \min_{\mathbf{z}} \quad & \frac{1}{2}\mathbf{z}^\top \mathbf{H} \mathbf{z} \\ \text{subject to} \quad & \mathbf{C}\mathbf{z} = \mathbf{d} \end{aligned}$$

8.2 Solving the QP: The KKT System

This is an equality-constrained QP. It can be solved directly by finding the stationary point of its Lagrangian:

$$\mathcal{L}(\mathbf{z}, \lambda) = \frac{1}{2}\mathbf{z}^\top \mathbf{H} \mathbf{z} + \lambda^\top (\mathbf{C}\mathbf{z} - \mathbf{d})$$

The first-order necessary (and sufficient, by convexity) conditions are the KKT equations:

$$\begin{aligned} \nabla_{\mathbf{z}} \mathcal{L} &= \mathbf{H}\mathbf{z} + \mathbf{C}^\top \lambda = 0 \\ \nabla_{\lambda} \mathcal{L} &= \mathbf{C}\mathbf{z} - \mathbf{d} = 0 \end{aligned}$$

This is a single, large, sparse linear system:

$$\begin{bmatrix} \mathbf{H} & \mathbf{C}^\top \\ \mathbf{C} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{z} \\ \lambda \end{bmatrix} = \begin{bmatrix} 0 \\ \mathbf{d} \end{bmatrix} \tag{8.1}$$

Solving this system yields $\mathbf{z}$ (and the multipliers λ) in a single linear solve. This yields a non-iterative, exact solution. The multipliers λ coincide with the costate variables from the Pontryagin Maximum Principle (PMP).

Code 8.1 (Julia Notebook: LQR as QP).
The 'lqr-qp.ipynb' notebook implements this exact approach.

- It uses 'blockdiag' and 'kron' (Kronecker product) to efficiently construct the large, sparse matrices $\mathbf{H}$ and $\mathbf{C}$.
- It forms the right-hand-side vector $\mathbf{d}$ using the initial state $\mathbf{x}_0$.
- It assembles and solves the full KKT system (8.1) using Julia's backslash operator ('\'), which is highly optimized for sparse linear systems.
- Finally, it extracts the state and control trajectories from the solution vector $\mathbf{z}$.
- The resulting plots show the double integrator being driven to the origin, exactly as in the shooting method, but this solution was found in one (large) linear solve, not via iteration.

Trade-off: This method is exact and non-iterative, but it requires building and solving a potentially massive linear system (size $\mathcal{O}(N(n+m)) \times \mathcal{O}(N(n+m))$). This is often less efficient than methods that exploit the time-structure.

8.3 Deriving the Riccati Recursion

The KKT system (8.1) is large, but also extremely sparse and structured. It is advantageous to exploit this structure rather than rely on a generic sparse solver. In what follows, we solve this system using block substitution, proceeding backward in time.

Assume a time-invariant system $(\mathbf{A}, \mathbf{B}, \mathbf{Q}, \mathbf{R})$ for simplicity. The KKT system's equations (from $\nabla_z \mathcal{L} = 0$) for the final steps are:

1. $\nabla_{\mathbf{x}_N} \mathcal{L} = \mathbf{Q}_N \mathbf{x}_N - \lambda_N = 0 \Rightarrow \lambda_N = \mathbf{Q}_N \mathbf{x}_N$
2. $\nabla_{\mathbf{u}_{N-1}} \mathcal{L} = \mathbf{R}\mathbf{u}_{N-1} + \mathbf{B}^\top \lambda_N = 0$
3. $\nabla_{\mathbf{x}_{N-1}} \mathcal{L} = \mathbf{Q}\mathbf{x}_{N-1} + \mathbf{A}^\top \lambda_N - \lambda_{N-1} = 0$

This structure repeats. We posit that the costate is a linear function of the state at every time step, namely $\lambda_k = \mathbf{P}_k \mathbf{x}_k$. From (1), this holds at time N with $\mathbf{P}_N = \mathbf{Q}_N$.

We now proceed backward in time.

Step 1: Find $\mathbf{u}_{N-1}$. Substitute the relation $\lambda_N = \mathbf{P}_N \mathbf{x}_N$ into (2):

$$\mathbf{R}\mathbf{u}_{N-1} + \mathbf{B}^\top \mathbf{P}_N \mathbf{x}_N = 0$$

Substitute the dynamics $\mathbf{x}_N = \mathbf{A}\mathbf{x}_{N-1} + \mathbf{B}\mathbf{u}_{N-1}$:

$$\mathbf{R}\mathbf{u}_{N-1} + \mathbf{B}^\top \mathbf{P}_N (\mathbf{A}\mathbf{x}_{N-1} + \mathbf{B}\mathbf{u}_{N-1}) = 0$$

Group terms by $\mathbf{u}_{N-1}$ and $\mathbf{x}_{N-1}$:

$$(\mathbf{R} + \mathbf{B}^\top \mathbf{P}_N \mathbf{B})\mathbf{u}_{N-1} = -(\mathbf{B}^\top \mathbf{P}_N \mathbf{A})\mathbf{x}_{N-1}$$

Solving for $\mathbf{u}_{N-1}$ yields the feedback law $\mathbf{u}_{N-1} = -\mathbf{K}_{N-1}\mathbf{x}_{N-1}$, where:

$$\mathbf{K}_{N-1} = (\mathbf{R} + \mathbf{B}^\top \mathbf{P}_N \mathbf{B})^{-1}(\mathbf{B}^\top \mathbf{P}_N \mathbf{A})$$

Here, $\mathbf{K}_{N-1}$ is the **feedback gain** at time $N-1$.

Step 2: Find $\mathbf{P}_{N-1}$. Next, use (3) to obtain the relationship for λ_{N-1}:

$$\lambda_{N-1} = \mathbf{Q}\mathbf{x}_{N-1} + \mathbf{A}^\top \lambda_N = \mathbf{Q}\mathbf{x}_{N-1} + \mathbf{A}^\top (\mathbf{P}_N \mathbf{x}_N)$$

Substitute $\mathbf{x}_N = \mathbf{A}\mathbf{x}_{N-1} + \mathbf{B}\mathbf{u}_{N-1}$ and $\mathbf{u}_{N-1} = -\mathbf{K}_{N-1}\mathbf{x}_{N-1}$:

$$\lambda_{N-1} = \mathbf{Q}\mathbf{x}_{N-1} + \mathbf{A}^\top \mathbf{P}_N (\mathbf{A}\mathbf{x}_{N-1} - \mathbf{B}\mathbf{K}_{N-1}\mathbf{x}_{N-1})$$

Factor out $\mathbf{x}_{N-1}$:

$$\lambda_{N-1} = \Big(\mathbf{Q} + \mathbf{A}^\top \mathbf{P}_N (\mathbf{A} - \mathbf{B}\mathbf{K}_{N-1})\Big)\,\mathbf{x}_{N-1}$$

This confirms the proposed form, namely $\lambda_{N-1} = \mathbf{P}_{N-1}\mathbf{x}_{N-1}$, where:

$$\mathbf{P}_{N-1} = \mathbf{Q} + \mathbf{A}^\top \mathbf{P}_N (\mathbf{A} - \mathbf{B}\mathbf{K}_{N-1})$$

The matrix $\mathbf{P}_k$ is the Hessian of the **cost-to-go** function, representing the remaining cost from time k to N.

The Recursion By repeating this process, we get the general **Discrete-Time Riccati Recursion**:

1. Initialize: $\mathbf{P}_N = \mathbf{Q}_N$
2. For $k = N-1$ down to 1:

$$\mathbf{K}_k = (\mathbf{R}_k + \mathbf{B}_k^\top \mathbf{P}_{k+1}\mathbf{B}_k)^{-1}(\mathbf{B}_k^\top \mathbf{P}_{k+1}\mathbf{A}_k) \tag{8.2}$$

$$\mathbf{P}_k = \mathbf{Q}_k + \mathbf{A}_k^\top \mathbf{P}_{k+1}(\mathbf{A}_k - \mathbf{B}_k\mathbf{K}_k) \tag{8.3}$$

8.4 The LQR Algorithm (via Dynamic Programming)

Algorithm 6: LQR via Riccati Recursion

Input: $\mathbf{A}_k, \mathbf{B}_k, \mathbf{Q}_k, \mathbf{R}_k, \mathbf{Q}_N, \mathbf{x}_0$
Output: Optimal $\mathbf{U}^\star, \mathbf{X}^\star$

// 1. Backward Pass (Compute Gains)
$\mathbf{P}_N \leftarrow \mathbf{Q}_N$
for $k = N-1$ **to** 1 **do**
  $\mathbf{K}_k \leftarrow (\mathbf{R}_k + \mathbf{B}_k^\top \mathbf{P}_{k+1}\mathbf{B}_k)^{-1}\mathbf{B}_k^\top \mathbf{P}_{k+1}\mathbf{A}_k$
  $\mathbf{P}_k \leftarrow \mathbf{Q}_k + \mathbf{A}_k^\top \mathbf{P}_{k+1}(\mathbf{A}_k - \mathbf{B}_k\mathbf{K}_k)$
// 2. Forward Pass (Rollout Trajectory)
$\mathbf{x}_1 \leftarrow \mathbf{x}_0$
for $k = 1$ **to** $N-1$ **do**
  $\mathbf{u}_k \leftarrow -\mathbf{K}_k\mathbf{x}_k$
  $\mathbf{x}_{k+1} \leftarrow \mathbf{A}_k\mathbf{x}_k + \mathbf{B}_k\mathbf{u}_k$
return $\mathbf{U} = (\mathbf{u}_1, \ldots, \mathbf{u}_{N-1})$, $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_N)$

This is the standard LQR solution. Its complexity is $\mathcal{O}(N(n+m)^3)$, which is drastically better than the $\mathcal{O}((N(n+m))^3)$ for the full QP solve. More importantly, it provides a **feedback policy** $\mathbf{u}_k = -\mathbf{K}_k\mathbf{x}_k$. The gains $\mathbf{K}_k$ are pre-computed offline and are independent of $\mathbf{x}_0$. If the initial state changes, or if noise perturbs the system, we don't need to re-solve for $\mathbf{K}_k$; we just apply the same feedback law. This makes it robust and computationally cheap at runtime.

Code 8.2 (Julia Notebook: LQR Riccati).
The 'lqr-riccati.ipynb' notebook implements this exact two-pass algorithm.

- The first loop runs backward from 'k = (N-1):-1:1' to compute and store the 'K' and 'P' matrices.
- The second loop runs forward from 'k = 1:(N-1)' to simulate the trajectory using the computed feedback gains $\mathbf{u}_k = -\mathbf{K}_k\mathbf{x}_k$.
- The resulting plots for $\mathbf{x}$ and $\mathbf{u}$ are identical to the previous methods.
- The notebook also plots the components of the gain $\mathbf{K}$ over time. It shows that for a long-horizon problem, the gains calculated backward from $k = N-1$ quickly converge to a steady value.

8.5 Infinite-Horizon LQR

For time-invariant systems ($\mathbf{A}, \mathbf{B}, \mathbf{Q}, \mathbf{R}$ are constant) and a long horizon $N \to \infty$, the gain $\mathbf{K}_k$ and cost-to-go $\mathbf{P}_k$ converge to steady-state values $\mathbf{K}_\infty$ and $\mathbf{P}_\infty$.

The Riccati recursion $\mathbf{P}_k = f(\mathbf{P}_{k+1})$ becomes a fixed-point equation:

$$\mathbf{P} = \mathbf{Q} + \mathbf{A}^\top\mathbf{P}(\mathbf{A} - \mathbf{B}(\mathbf{R} + \mathbf{B}^\top\mathbf{P}\mathbf{B})^{-1}\mathbf{B}^\top\mathbf{P}\mathbf{A})$$

This is the **Discrete Algebraic Riccati Equation (DARE)**. We solve this equation (e.g., using 'dare' in MATLAB/Python, or 'dlqr' in Julia's 'ControlSystems.jl') to find the single, constant gain $\mathbf{K}_\infty$.

This constant gain $\mathbf{u}_k = -\mathbf{K}_\infty\mathbf{x}_k$ is the optimal controller for the infinite-horizon LQR problem and is used to stabilize the system.

Code 8.3 (Julia Notebook: Infinite-Horizon).
The 'lqr-riccati.ipynb' notebook demonstrates this.

- It calls 'Kinf = dlqr(A,B,Q,R)' to solve the DARE.
- It shows that 'Kinf' is nearly identical to 'K[:,:,1]' (the gain at the start of the time-horizon), confirming the convergence.
- It computes the eigenvalues of the closed-loop system, 'eigvals(A - B*Kinf)'. All eigenvalues have a magnitude less than 1, analytically proving that the feedback controller $\mathbf{K}_\infty$ makes the system stable.

Lecture 9: Controllability and Dynamic Programming

9.1 Controllability

A fundamental question arises prior to solving the LQR problem: can the controls steer the system to a desired state? In the LQR context, this typically means: can the state be driven to the origin from an arbitrary initial condition? This property is termed **controllability**.

For a time-invariant LTI system, $\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k$, consider the state $\mathbf{x}_N$ after N steps, starting from $\mathbf{x}_0$:

$$\begin{aligned}
\mathbf{x}_1 &= \mathbf{A}\mathbf{x}_0 + \mathbf{B}\mathbf{u}_0 \\
\mathbf{x}_2 &= \mathbf{A}\mathbf{x}_1 + \mathbf{B}\mathbf{u}_1 = \mathbf{A}(\mathbf{A}\mathbf{x}_0 + \mathbf{B}\mathbf{u}_0) + \mathbf{B}\mathbf{u}_1 = \mathbf{A}^2\mathbf{x}_0 + \mathbf{A}\mathbf{B}\mathbf{u}_0 + \mathbf{B}\mathbf{u}_1 \\
&\vdots \\
\mathbf{x}_N &= \mathbf{A}^N\mathbf{x}_0 + \mathbf{A}^{N-1}\mathbf{B}\mathbf{u}_0 + \mathbf{A}^{N-2}\mathbf{B}\mathbf{u}_1 + \cdots + \mathbf{B}\mathbf{u}_{N-1}
\end{aligned}$$

The control-dependent terms can be grouped by factoring out a single matrix:

$$\mathbf{x}_N - \mathbf{A}^N\mathbf{x}_0 = \underbrace{\begin{bmatrix}\mathbf{B} & \mathbf{A}\mathbf{B} & \mathbf{A}^2\mathbf{B} & \cdots & \mathbf{A}^{N-1}\mathbf{B}\end{bmatrix}}_{\mathcal{C}_N} \begin{bmatrix}\mathbf{u}_{N-1} \\ \mathbf{u}_{N-2} \\ \vdots \\ \mathbf{u}_0\end{bmatrix}$$

The system is controllable if there exists a control sequence $\mathbf{U} = [\mathbf{u}_{N-1}, \ldots, \mathbf{u}_0]^\top$ that reaches *any* target state. For LQR stabilization, the specific objective is $\mathbf{x}_N = 0$. This requires solving the linear system $\mathcal{C}_N\mathbf{U} = -\mathbf{A}^N\mathbf{x}_0$ for $\mathbf{U}$.

A solution exists for any $\mathbf{x}_0$ if and only if the matrix $\mathcal{C}_N$ has full row rank n (the dimension of the state), i.e., its columns span $\mathbb{R}^n$.

Theorem 9.1 (Cayley-Hamilton Theorem). A fundamental result from linear algebra states that any matrix $\mathbf{A}$ satisfies its own characteristic polynomial. A consequence is that any power $\mathbf{A}^k$ for $k \geq n$ can be written as a linear combination of the lower powers:

$$\mathbf{A}^k = \sum_{i=0}^{n-1} \alpha_i \mathbf{A}^i$$

This theorem implies that any columns $\mathbf{A}^k\mathbf{B}$ in the matrix $\mathcal{C}_N$ for $k \geq n$ are linearly dependent on the preceding columns. Consequently, adding more time steps (and hence additional columns) beyond $k = n - 1$ does not increase the rank.

This yields the standard test for controllability.

Definition 9.1 (Controllability Matrix). The Controllability Matrix for a time-invariant LTI system is:

$$\mathcal{C} = \begin{bmatrix}\mathbf{B} & \mathbf{A}\mathbf{B} & \mathbf{A}^2\mathbf{B} & \cdots & \mathbf{A}^{n-1}\mathbf{B}\end{bmatrix} \tag{9.1}$$

The system is controllable if and only if $\text{rank}(\mathcal{C}) = n$.

Code 9.1 (Julia Notebook: Controllability).
The ‘lqr-dp.ipynb‘ notebook evaluates controllability for the double integrator example:

$$\mathbf{A} = \begin{bmatrix}1 & h \\ 0 & 1\end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix}h^2/2 \\ h\end{bmatrix}$$

The controllability matrix is $\mathcal{C} = [\mathbf{B}, \mathbf{AB}] = \begin{bmatrix} h^2/2 & 3h^2/2 \\ h & h \end{bmatrix}$. The code 'rank([B A*B])' computes the rank of this 2×2 matrix. Since $h \neq 0$, its determinant is $(h^3/2) - (3h^3/2) = -h^3 \neq 0$, hence the matrix has full rank 2 and the system is controllable.

9.2 Dynamic Programming

Dynamic Programming (DP) is a general framework for optimal control based on a single principle.

Definition 9.2 (Bellman's Principle of Optimality). An optimal policy has the property that, for any initial state and initial decision, the remaining decisions form an optimal policy for the resulting state. Equivalently, every segment of an optimal trajectory is itself optimal for its sub-problem; otherwise a strictly better replacement would contradict global optimality.

This principle enables construction of the solution by backward induction from the terminal time. This is precisely the mechanism underlying the Riccati recursion and the PMP costate backward pass.

To formalize this, we define the optimal cost-to-go, or **Value Function**.

Definition 9.3 (Value Function). The Value Function $V_k(x)$ is the optimal cost-to-go, defined as the minimal cost accumulated starting from state $\mathbf{x}$ at time step k while acting optimally until the final time N.

$$V_k(\mathbf{x}) = \min_{\mathbf{u}_{k\dots N-1}} \left(l_F(\mathbf{x}_N) + \sum_{j=k}^{N-1} l(\mathbf{x}_j, \mathbf{u}_j) \right) \quad \text{s.t.} \quad \mathbf{x}_k = \mathbf{x}$$

The Bellman Principle gives us a recursive relationship for the Value Function:

$$V_k(\mathbf{x}) = \min_{\mathbf{u}} \left[l(\mathbf{x}, \mathbf{u}) + V_{k+1}(f(\mathbf{x}, \mathbf{u})) \right] \tag{9.2}$$

The value of being at state $\mathbf{x}$ at time k is the minimal cost achievable by taking one step: incurring the immediate cost $l(\mathbf{x}, \mathbf{u})$ and transitioning to a new state $f(\mathbf{x}, \mathbf{u})$, from which the optimal future cost $V_{k+1}(f(\mathbf{x}, \mathbf{u}))$ must be paid.

9.3 DP Derivation of the Riccati Recursion

We apply the Bellman equation to the LQR problem and prove by induction that $V_k(\mathbf{x})$ is quadratic for all k.

Base Case (k=N): The recursion terminates at time N. The cost-to-go from N is the terminal cost.

$$V_N(\mathbf{x}) = l_F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}_N \mathbf{x}$$

This is a quadratic form, $V_N(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{P}_N \mathbf{x}$, where $\mathbf{P}_N = \mathbf{Q}_N$.

Inductive Step (k < N): Assume $V_{k+1}(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{P}_{k+1}\mathbf{x}$ for some matrix $\mathbf{P}_{k+1}$. We now compute $V_k(\mathbf{x})$ using Equation 9.2:

$$V_k(\mathbf{x}) = \min_{\mathbf{u}} \left[\frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \frac{1}{2}\mathbf{u}^\top \mathbf{R}\mathbf{u} + V_{k+1}(\mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}) \right]$$

Substitute the assumed form for V_{k+1}:

$$V_k(\mathbf{x}) = \min_{\mathbf{u}} \left[\frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \frac{1}{2}\mathbf{u}^\top \mathbf{R}\mathbf{u} + \frac{1}{2}(\mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u})^\top \mathbf{P}_{k+1}(\mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}) \right]$$

This is an unconstrained quadratic minimization problem in $\mathbf{u}$. The minimum is attained where the gradient with respect to $\mathbf{u}$ is zero:

$$\nabla_{\mathbf{u}}[\dots] = \mathbf{R}\mathbf{u} + \mathbf{B}^\top \mathbf{P}_{k+1}(\mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}) = 0$$

Solving for $\mathbf{u}$:

$$(\mathbf{R} + \mathbf{B}^\top \mathbf{P}_{k+1}\mathbf{B})\mathbf{u} = -(\mathbf{B}^\top \mathbf{P}_{k+1}\mathbf{A})\mathbf{x}$$

This yields the optimal policy $\mathbf{u}_k^\star(\mathbf{x})$:

$$\mathbf{u}_k^\star = -\underbrace{(\mathbf{R} + \mathbf{B}^\top \mathbf{P}_{k+1}\mathbf{B})^{-1}(\mathbf{B}^\top \mathbf{P}_{k+1}\mathbf{A})}_{\mathbf{K}_k}\mathbf{x} = -\mathbf{K}_k\mathbf{x}$$

This coincides with the Riccati gain $\mathbf{K}_k$ from Lecture 8.

To obtain $V_k(\mathbf{x})$, substitute the optimal control $\mathbf{u}_k^\star = -\mathbf{K}_k\mathbf{x}$ into the expression for $V_k(\mathbf{x})$. The algebra yields:

$$V_k(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \left[\mathbf{Q} + \mathbf{K}_k^\top \mathbf{R}\mathbf{K}_k + (\mathbf{A} - \mathbf{B}\mathbf{K}_k)^\top \mathbf{P}_{k+1}(\mathbf{A} - \mathbf{B}\mathbf{K}_k) \right] \mathbf{x}$$

This demonstrates that $V_k(\mathbf{x})$ is also a quadratic form, $V_k(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{P}_k \mathbf{x}$, where $\mathbf{P}_k$ is given by the recursive update:

$$\mathbf{P}_k = \mathbf{Q} + \mathbf{K}_k^\top \mathbf{R}\mathbf{K}_k + (\mathbf{A} - \mathbf{B}\mathbf{K}_k)^\top \mathbf{P}_{k+1}(\mathbf{A} - \mathbf{B}\mathbf{K}_k)$$

This is the **Riccati Recursion**. This DP derivation confirms that the backward Riccati pass is a special, analytically tractable instance of solving the Bellman equation, wherein the quadratic form of the value function is preserved at each step.

9.4 The Curse of Dimensionality

The DP algorithm is general:

1. Initialize $V_N(\mathbf{x}) = l_F(\mathbf{x})$.
2. For $k = N - 1 \dots 1$:
3. $V_k(\mathbf{x}) = \min_{\mathbf{u}}[l(\mathbf{x}, \mathbf{u}) + V_{k+1}(f(\mathbf{x}, \mathbf{u}))]$

This provides a sufficient condition for global optimality. Why is it not used universally?

- **Tractability:** It is tractable only for relatively simple problems.
- **Value Function Representation:** For LQR, $V_k(\mathbf{x})$ is represented exactly by a single $n \times n$ matrix $\mathbf{P}_k$. For a general nonlinear problem, $V_k(\mathbf{x})$ becomes a complex, non-analytic function that cannot be represented exactly in closed form.
- **Optimization Step:** For LQR, the step $\min_{\mathbf{u}}[\dots]$ is a convex QP, solvable analytically. For nonlinear problems, this step $\min_{\mathbf{u}}[l(\mathbf{x}, \mathbf{u}) + V_{k+1}(f(\mathbf{x}, \mathbf{u}))]$ is generally non-convex and computationally demanding, and must be solved at every $\mathbf{x}$.
- **The Curse:** The cost of representing the function $V_k(\mathbf{x})$ (e.g., on a grid) grows exponentially with the dimension n of the state $\mathbf{x}$. This phenomenon is the **Curse of Dimensionality**.

This motivates Reinforcement Learning (RL): RL uses function approximators (e.g., neural networks) to find an *approximate* solution to the Bellman equation, mitigating the curse in practice.

9.5 The Fundamental Connection

We now have two views of LQR:

1. **PMP (Lecture 7-8):** We get a state trajectory $\mathbf{x}_k$ and a costate trajectory λ_k. From the QP derivation, we found $\lambda_k = \mathbf{P}_k \mathbf{x}_k$.
2. **DP (Lecture 9):** We get a value function $V_k(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{P}_k \mathbf{x}$.

Let's compute the gradient of the value function from DP:

$$\nabla_{\mathbf{x}} V_k(\mathbf{x}) = \nabla_{\mathbf{x}} \left(\frac{1}{2}\mathbf{x}^\top \mathbf{P}_k \mathbf{x} \right) = \mathbf{P}_k \mathbf{x}$$

Comparing the two, we arrive at a fundamental insight:

$$\boxed{\lambda_k = \nabla_{\mathbf{x}} V_k(\mathbf{x}_k)} \tag{9.3}$$

The **costate** λ_k (the Lagrange multiplier of the dynamics constraint at time k) is exactly equal to the **gradient of the optimal cost-to-go (Value) function** $\nabla_{\mathbf{x}} V_k$ evaluated at the optimal state $\mathbf{x}_k$.

This connection is profound and holds true for nonlinear optimal control as well, linking the worlds of Pontryagin (PMP) and Bellman (DP).

Code 9.2 (Julia Notebook: Verifying the Connection).
The 'lqr-dp.ipynb' notebook numerically verifies all of these concepts:

- It solves the LQR problem using both the QP method and the DP (Riccati) method and plots the trajectories, showing they are identical.
- It numerically confirms Bellman's Principle by showing that the optimal subtrajectory from time k is identical to the corresponding slice of the full optimal trajectory.
- Most importantly, it compares the Lagrange multipliers from the QP solve (the array `lambda_hist_qp[:,k-1]`) to the gradient of the DP value function computed with ForwardDiff on `xhist`, and shows that they are the same, numerically proving $\lambda_k = \nabla_{\mathbf{x}} V_k(\mathbf{x}_k)$.

Lecture 10: Convexity and Model Predictive Control

As previously seen. In our last lecture, we formalized the backward-in-time reasoning of the Riccati recursion by introducing the general theory of **Dynamic Programming (DP)**. We showed that the LQR solution is a special case of DP where the Value Function $V_k(\mathbf{x})$ remains quadratic. We also established the fundamental connection between the costate (Lagrange multiplier) and the value function: $\lambda_k = \nabla_{\mathbf{x}} V_k(\mathbf{x})$.

Today, we address the primary weakness of LQR: its inability to handle constraints. This will lead us to **Model Predictive Control (MPC)**, a method that leverages our knowledge of optimization to control systems in real-time. First, we must build a foundation in **convexity**.

10.1 The Problem: Constraints in Optimal Control

The LQR problem $\min J$ s.t. $\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k$ is powerful, but its dynamics are *unconstrained*. In robotics, we must reason about real-world limitations, such as:

- **Actuator Limits:** A motor can only produce so much torque, and a thruster has minimum and maximum thrusts. $\mathbf{u}_{\min} \leq \mathbf{u}_k \leq \mathbf{u}_{\max}$.
- **State Constraints:** A joint angle must stay within its limits, or a robot must not enter a "keep-out" zone. $\mathbf{x}_{\min} \leq \mathbf{x}_k \leq \mathbf{x}_{\max}$.

These constraints **break the analytic Riccati solution**. The unconstrained LQR solution does not account for these limits. A common but significantly flawed heuristic is to "clip" or "saturate" the LQR output: $\mathbf{u}_k = \max(\mathbf{u}_{\min}, \min(\mathbf{u}_{\max}, \mathbf{u}_{\text{LQR}}))$. This approach is suboptimal and can even destabilize the system.

A more principled approach is to solve the full constrained optimization problem. While the LQR QP formulation (from Lecture 8) can be modified to include constraints, a superior method exists. This leads to MPC, which solves a constrained optimization problem *at every time step*. As computational power has increased, this "online" optimization has become a dominant strategy in robotics.

10.2 Background: Convex Optimization

The reason we can solve these problems in real-time is that they are often **convex**.

Definition 10.1 (Convex Set). A set $\mathcal{C}$ is **convex** if for any two points $\mathbf{x}, \mathbf{y} \in \mathcal{C}$, the line segment connecting them is also fully contained in $\mathcal{C}$.

$$\theta\mathbf{x} + (1-\theta)\mathbf{y} \in \mathcal{C} \quad \forall \mathbf{x}, \mathbf{y} \in \mathcal{C}, \theta \in [0,1]$$

Note (Examples of Convex Sets).

- **Linear Subspaces:** Solutions to $\mathbf{A}\mathbf{x} = \mathbf{b}$.
- **Polytopes (Boxes, Half-spaces):** Solutions to $\mathbf{A}\mathbf{x} \leq \mathbf{b}$ (e.g., actuator limits).
- **Ellipsoids:** Solutions to $\mathbf{x}^\top \mathbf{P}\mathbf{x} \leq 1$ for $\mathbf{P} \succ 0$.
- **Second-Order Cones:** $\{\mathbf{x} \mid x_1 \geq \|\mathbf{x}_{2:n}\|_2\}$.

Definition 10.2 (Convex Function). A function $f : \mathbb{R}^n \to \mathbb{R}$ is **convex** if its **epigraph** (the set of points $\{(\mathbf{x}, t) \mid t \geq f(\mathbf{x})\}$ lying on or above the function) is a convex set.

Note (Examples of Convex Functions).

- **Linear/Affine:** $f(\mathbf{x}) = \mathbf{c}^\top \mathbf{x} + d$.
- **Quadratic:** $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{q}^\top \mathbf{x}$, but only if the Hessian $\mathbf{Q}$ is positive semidefinite ($\mathbf{Q} \succeq 0$).
- **Norms:** $f(\mathbf{x}) = \|\mathbf{x}\|$ for any valid norm.

Definition 10.3 (Convex Optimization Problem). A convex optimization problem is one that minimizes a convex function over a convex set.

$$\begin{aligned} \min_{\mathbf{x}} \quad & f_0(\mathbf{x}) \quad \text{(convex)} \\ \text{s.t.} \quad & f_i(\mathbf{x}) \leq 0 \quad \text{(convex)} \\ & \mathbf{A}\mathbf{x} = \mathbf{b} \end{aligned}$$

Common examples include **Linear Programs (LP)**, **Quadratic Programs (QP)**, and **Second-Order Cone Programs (SOCP)**.

Intuition (Why Convexity is Valuable). Convex problems are considered effectively solved in the field of optimization. This is due to the following properties:

1. **No Spurious Local Minima:** Any local minimum is a global minimum. The KKT conditions are not just necessary, but also *sufficient* for a global optimum.
2. **Reliable Solvers:** Algorithms (such as interior-point methods) are guaranteed to converge to the global optimum.
3. **Computational Efficiency:** These solvers are remarkably efficient and reliable, often converging in a small number of iterations, regardless of the problem size.

This combination of reliability and computational efficiency allows us to bound the solution time and use optimization for real-time control.

10.3 Convex Model Predictive Control (MPC)

MPC, also known as **Receding Horizon Control**, is a strategy that turns the full, infinite-horizon optimal control problem into a series of small, solvable, finite-horizon problems.

Recall from Dynamic Programming, the optimal control $\mu_k(\mathbf{x})$ is the solution to the one-step problem:

$$\mu_k(\mathbf{x}) = \arg\min_{\mathbf{u}} \left[l(\mathbf{x}, \mathbf{u}) + V_{k+1}(f_d(\mathbf{x}, \mathbf{u})) \right]$$

For LQR, $V_{k+1}(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{P}_{k+1}\mathbf{x}$ (where $\mathbf{P}$ comes from the DARE). We could *try* to add constraints to this:

$$\begin{aligned} \min_{\mathbf{u}} \quad & \frac{1}{2}\mathbf{u}^\top \mathbf{R}\mathbf{u} + \frac{1}{2}(\mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u})^\top \mathbf{P}(\mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}) \\ \text{s.t.} \quad & \mathbf{u}_{\min} \leq \mathbf{u} \leq \mathbf{u}_{\max} \end{aligned}$$

This approach yields poor performance. The value function V_{k+1} (and its associated matrix $\mathbf{P}$) represents the anticipated future cost, but it was computed under the assumption of no future constraints. Therefore, it provides an inaccurate approximation of the true, constrained cost-to-go.

10.3.1 The MPC Formulation

Rather than solving a 1-step problem with an inaccurate terminal cost approximation, we solve an H-step problem where the approximation only affects the final step. We define a planning **horizon** H (e.g., $H = 20$ steps). At the current state $\mathbf{x}_t$, we solve the following QP:

$$\min_{\mathbf{x}_{t:t+H}, \mathbf{U}_{t:t+H-1}} \quad \sum_{k=t}^{t+H-1} \left(\frac{1}{2} \delta \mathbf{x}_k^\top \mathbf{Q} \delta \mathbf{x}_k + \frac{1}{2} \delta \mathbf{u}_k^\top \mathbf{R} \delta \mathbf{u}_k \right) + \frac{1}{2} \delta \mathbf{x}_{t+H}^\top \mathbf{P} \delta \mathbf{x}_{t+H}$$

subject to

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k && \text{(Linear Dynamics)} \\ \mathbf{u}_{\min} &\leq \mathbf{u}_k \leq \mathbf{u}_{\max} && \text{(Actuator Limits)} \\ \mathbf{x}_t &= \mathbf{x}_{\text{current}} && \text{(Initial Condition)} \end{aligned}$$

where $\delta \mathbf{x}_k = \mathbf{x}_k - \mathbf{x}_{\text{ref}}$, $\delta \mathbf{u}_k = \mathbf{u}_k - \mathbf{u}_{\text{ref}}$, and $\mathbf{P}$ is the terminal cost from the unconstrained DARE solution.

Algorithm 7: Model Predictive Control (MPC) Loop

Input: Current state $\mathbf{x}_{\text{current}}$, reference $\mathbf{x}_{\text{ref}}$, horizon H, cost matrices $\mathbf{Q}, \mathbf{R}, \mathbf{P}$, dynamics $\mathbf{A}, \mathbf{B}$, constraints $\mathbf{u}_{\min}, \mathbf{u}_{\max}$

Output: Control $\mathbf{u}_{\text{apply}}$

1. Formulate the H-step QP based on $\mathbf{x}_{\text{current}}$ and $\mathbf{x}_{\text{ref}}$
2. Solve the QP to find the optimal *sequence* $\mathbf{U}^\star = \{\mathbf{u}_t^\star, \mathbf{u}_{t+1}^\star, \dots, \mathbf{u}_{t+H-1}^\star\}$
3. **Apply only the first step:** $\mathbf{u}_{\text{apply}} \leftarrow \mathbf{u}_t^\star$
4. **Discard the rest of the plan:** $\{\mathbf{u}_{t+1}^\star, \dots\}$
5. At the next time step, get new state $\mathbf{x}_{\text{current}}$ and **repeat from step 1**.

10.3.2 MPC Trade-offs

- **The Terminal Cost P:** The LQR cost-to-go provides an accurate approximation for the cost of the remaining trajectory after the horizon H.
- **The Horizon H:**
  - A **longer horizon** H yields improved performance (it can plan around constraints more effectively) but requires more computation time. It relies less on the terminal cost $\mathbf{P}$.
  - A **shorter horizon** H is computationally faster, but depends more heavily on an accurate terminal cost approximation $\mathbf{P}$.
- **Key Insight:** By optimizing over the first H steps, the controller can plan to approach the unconstrained reference trajectory, such that by step H, the constraints are no longer active, and the LQR terminal cost becomes a valid and accurate approximation of the future cost.

10.4 Code Analysis: `mpc.ipynb` (Planar Quadrotor)

The `mpc.ipynb` notebook provides a complete implementation of the MPC framework described above.

Code 10.1 (Julia Notebook: MPC for Planar Quadrotor).
The notebook implements MPC for a planar quadrotor system with actuator constraints.

- **System:** The notebook defines the nonlinear dynamics of a planar quadrotor and its RK4 discretization.
- **Linearization:** The nonlinear dynamics are linearized around the hover equilibrium ($\mathbf{x} = 0$, $\mathbf{u} = [mg/2, mg/2]$) to obtain the $\mathbf{A}$ and $\mathbf{B}$ matrices for the MPC model.
- **LQR Baseline:** The unconstrained DARE is solved (`P = dare(...)`) and `dlqr` is computed to obtain the infinite-horizon terminal cost $\mathbf{P}$ and the LQR gain $\mathbf{K}$.
- **MPC QP Formulation:**
  - The notebook sets a horizon `Nh = 20` (1 second at 20 Hz).
  - The large, sparse QP matrices are constructed. The cost matrix `H` is constructed as `blockdiag(R, Q, ..., R, Q, P)`, correctly using the DARE solution `P` as the terminal cost.
  - The constraint matrix `D` includes both the dynamics `C` ($\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k$) and the actuator bounds `U` ($\mathbf{u}_{\min} \leq \mathbf{u}_k \leq \mathbf{u}_{\max}$).
- **MPC Controller Loop:** The function `mpc_controller` implements the receding horizon strategy:
  1. The function takes the current state $\mathbf{x}$ and reference $\mathbf{x}_{\text{ref}}$ as input.
  2. The QP is updated:
     - The initial state constraint is updated: `lb[1:6] .= -A*x` and `ub[1:6] .= -A*x`. This implements $\mathbf{x}_1 = \mathbf{A}\mathbf{x}_{\text{current}} + \mathbf{B}\mathbf{u}_0$.
     - The cost vector `b` is updated to track the reference `xref`.
  3. The function calls `OSQP.solve!` to find the optimal *sequence* of 20 control inputs.
  4. It returns **only the first** control action `results.x[1:Nu]` from that sequence.

Simulation: The `closed_loop` function simulates both the saturating LQR controller and the MPC controller against the *nonlinear* dynamics. The resulting plots and visualizations demonstrate that MPC achieves accurate tracking while explicitly respecting the actuator limits, whereas the saturated LQR controller exhibits different (and typically inferior) performance characteristics.

Lecture 11: Nonlinear Trajectory Optimization and DDP

11.1 What About Nonlinear Dynamics?

The MPC controller we built in the last lecture, while powerful, relied on a **linear** model of the dynamics, $\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k$. This model is an approximation, linearized around an equilibrium (e.g., hovering).

- **Regime of validity:** For tasks near the equilibrium, such as stabilization or tracking small motions, this linear model is often sufficiently accurate.
- **Limitations:** For aggressive maneuvers or tasks that move far from the equilibrium (e.g., the Acrobot swing-up), the linear model can be an inadequate representation of the underlying nonlinear dynamics.

Using nonlinear dynamics, $\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k)$, inside an MPC-like optimization makes the problem **non-convex**. The implications are:

1. Global optimality guarantees are lost.
2. Solvers may converge to suboptimal local minima.
3. Real-time guarantees are weakened, as solving non-convex problems can be slow and unpredictable.

We address this problem directly by formulating the full **nonlinear trajectory optimization** problem and introducing an algorithm to solve it: Differential Dynamic Programming (DDP).

Definition 11.1 (Nonlinear Trajectory Optimization). Find the sequences of states $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ and controls $\mathbf{U} = \{\mathbf{u}_1, \ldots, \mathbf{u}_{N-1}\}$ that solve:

$$\begin{aligned} \min_{\mathbf{X},\mathbf{U}} \quad & J = \sum_{k=1}^{N-1} l_k(\mathbf{x}_k, \mathbf{u}_k) + l_N(\mathbf{x}_N) \\ \text{subject to} \quad & \mathbf{x}_{k+1} = f_k(\mathbf{x}_k, \mathbf{u}_k) \quad \text{(Nonlinear Dynamics)} \\ & \mathbf{x}_k \in \mathcal{X}_k, \quad \mathbf{u}_k \in \mathcal{U}_k \quad \text{(Non-convex Constraints)} \end{aligned}$$

We will assume all cost functions l_k and dynamics functions f_k are C^2 (twice continuously differentiable).

11.2 Differential Dynamic Programming (DDP)

DDP is an effective algorithm for solving this problem. It is essentially **Newton's method applied to trajectory optimization**. Like Newton's method, it uses a second-order approximation to determine an improvement step.

DDP is based on **approximate Dynamic Programming**. It follows the same logic as the Riccati recursion:

1. It performs a **backward pass** from $k = N-1$ down to 1, but instead of computing the LQR gain $\mathbf{K}_k$, it computes a local *quadratic model* of the cost-to-go function, $V_k(\mathbf{x})$.
2. It then performs a **forward pass** (a "rollout") from $k = 1$ to $N-1$ to update the trajectory using the policy computed in the backward pass.
3. It iterates these two passes until convergence.

11.2.1 The DDP Backward Pass

Let's assume we have a nominal trajectory $(\bar{\mathbf{X}}, \bar{\mathbf{U}})$ and we want to find a better one.

Base Case (k=N): We approximate the terminal cost l_N as a quadratic function around the nominal terminal state $\bar{\mathbf{x}}_N$. This defines our initial cost-to-go model:

$$V_N(\bar{\mathbf{x}}_N + \delta\mathbf{x}) \approx V_N(\bar{\mathbf{x}}_N) + \mathbf{p}_N^\top \delta\mathbf{x} + \frac{1}{2}\delta\mathbf{x}^\top \mathbf{P}_N \delta\mathbf{x}$$

where

$$\mathbf{p}_N = \nabla l_N(\bar{\mathbf{x}}_N)$$
$$\mathbf{P}_N = \nabla^2 l_N(\bar{\mathbf{x}}_N)$$

Inductive Step (k < N): Now, assume we have a quadratic model for V_{k+1} (defined by $\mathbf{p}_{k+1}, \mathbf{P}_{k+1}$). We want to find the quadratic model for V_k (i.e., find $\mathbf{p}_k, \mathbf{P}_k$).

From DP, we know the Bellman equation:

$$V_k(\mathbf{x}) = \min_{\mathbf{u}} \left[l_k(\mathbf{x}, \mathbf{u}) + V_{k+1}(f_k(\mathbf{x}, \mathbf{u})) \right]$$

We define the **Action-Value Function** (or Q-function) as the term inside the min:

$$Q_k(\mathbf{x}, \mathbf{u}) = l_k(\mathbf{x}, \mathbf{u}) + V_{k+1}(f_k(\mathbf{x}, \mathbf{u}))$$

DDP works by finding a quadratic approximation of Q_k around the nominal state-control pair $(\bar{\mathbf{x}}_k, \bar{\mathbf{u}}_k)$. Let $\delta\mathbf{x} = \mathbf{x} - \bar{\mathbf{x}}_k$ and $\delta\mathbf{u} = \mathbf{u} - \bar{\mathbf{u}}_k$:

$$Q_k(\bar{\mathbf{x}}_k + \delta\mathbf{x}, \bar{\mathbf{u}}_k + \delta\mathbf{u}) \approx Q_k(\bar{\mathbf{x}}_k, \bar{\mathbf{u}}_k) + \begin{bmatrix} \mathbf{g}_x \\ \mathbf{g}_u \end{bmatrix}^\top \begin{bmatrix} \delta\mathbf{x} \\ \delta\mathbf{u} \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \delta\mathbf{x} \\ \delta\mathbf{u} \end{bmatrix}^\top \begin{bmatrix} \mathbf{G}_{xx} & \mathbf{G}_{xu} \\ \mathbf{G}_{ux} & \mathbf{G}_{uu} \end{bmatrix} \begin{bmatrix} \delta\mathbf{x} \\ \delta\mathbf{u} \end{bmatrix}$$

Since, the Hessian is symmetric, we have $\mathbf{G}_{xu} = \mathbf{G}_{ux}^\top$. The new cost-to-go is $V_k(\bar{\mathbf{x}}_k + \delta\mathbf{x}) = \min_{\delta\mathbf{u}} Q_k(\dots)$. Since this is a quadratic function in $\delta\mathbf{u}$, we can find the minimum by setting the gradient to zero:

$$\nabla_{\delta\mathbf{u}} Q_k(\dots) = \mathbf{g}_u + \mathbf{G}_{uu}\delta\mathbf{u} + \mathbf{G}_{ux}\delta\mathbf{x} = 0$$

Solving for $\delta\mathbf{u}$ gives the **optimal control policy** for the deviation:

$$\delta\mathbf{u}^\star = -(\mathbf{G}_{uu})^{-1}\mathbf{g}_u - (\mathbf{G}_{uu})^{-1}\mathbf{G}_{ux}\delta\mathbf{x}$$

This policy has two parts,

$$\mathbf{d}_k = -(\mathbf{G}_{uu})^{-1}\mathbf{g}_u \qquad \text{(Feedforward / open-loop correction)}$$
$$\mathbf{K}_k = -(\mathbf{G}_{uu})^{-1}\mathbf{G}_{ux} \qquad \text{(Feedback gain)}$$

The local policy is $\delta\mathbf{u}^\star = \mathbf{d}_k + \mathbf{K}_k\delta\mathbf{x}$. We plug this policy back into the quadratic expansion for Q_k to get the new quadratic model for V_k, yielding the update rules for $\mathbf{p}_k$ and $\mathbf{P}_k$:

$$\mathbf{p}_k = \mathbf{g}_x + \mathbf{K}_k^\top \mathbf{G}_{uu}\mathbf{d}_k - \mathbf{K}_k^\top \mathbf{g}_u - \mathbf{G}_{xu}\mathbf{d}_k$$
$$\mathbf{P}_k = \mathbf{G}_{xx} + \mathbf{K}_k^\top \mathbf{G}_{uu}\mathbf{K}_k - \mathbf{G}_{xu}\mathbf{K}_k - \mathbf{K}_k^\top \mathbf{G}_{ux}$$

11.3 DDP vs. Iterative LQR (iLQR)

The final piece of the puzzle is: what are the terms $\mathbf{g}$ and $\mathbf{G}$? We find them by applying the chain rule to $Q_k = l_k + V_{k+1}(f_k)$, using our quadratic model for V_{k+1} and a Taylor expansion of f_k.

First-Order Terms (Gradients): Let $\mathbf{A}_k = \nabla_{\mathbf{x}} f_k$ and $\mathbf{B}_k = \nabla_{\mathbf{u}} f_k$. The chain rule gives:

$$\mathbf{g}_x = \nabla_{\mathbf{x}} Q_k = \nabla_{\mathbf{x}} l_k + (\nabla_{\mathbf{x}} f_k)^\top \nabla_{\mathbf{x}} V_{k+1} = \nabla_{\mathbf{x}} l_k + \mathbf{A}_k^\top \mathbf{p}_{k+1}$$
$$\mathbf{g}_u = \nabla_{\mathbf{u}} Q_k = \nabla_{\mathbf{u}} l_k + (\nabla_{\mathbf{u}} f_k)^\top \nabla_{\mathbf{x}} V_{k+1} = \nabla_{\mathbf{u}} l_k + \mathbf{B}_k^\top \mathbf{p}_{k+1}$$

Second-Order Terms (Hessians): This is where DDP and iLQR differ. The full Hessian of Q_k (using the chain rule) is:

$$\mathbf{G}_{xx} = \nabla^2_{\mathbf{xx}} l_k + \mathbf{A}_k^\top \mathbf{P}_{k+1} \mathbf{A}_k + (\nabla_{\mathbf{x}} \mathbf{p}_{k+1})^\top \cdot \nabla^2_{\mathbf{xx}} f_k$$

- **DDP (Differential DP):** The full DDP algorithm includes that last term: $(\dots)\nabla^2 f_k$. This term involves the second-order derivatives of the dynamics ($\nabla^2 f_k$), which is a **third-rank tensor**. This is computationally demanding and more burdensome to implement.
- **iLQR (Iterative LQR):** The iLQR algorithm makes a key simplification: it **ignores all second-order derivatives of the dynamics** f_k. It assumes $\nabla^2 f_k = 0$. This is equivalent to making a *first-order* (linear) approximation of the dynamics, but a *second-order* (quadratic) approximation of the cost.

This simplification yields the following Hessian terms:

$$\mathbf{G}_{xx} = \nabla^2_{\mathbf{xx}} l_k + \mathbf{A}_k^\top \mathbf{P}_{k+1} \mathbf{A}_k$$
$$\mathbf{G}_{uu} = \nabla^2_{\mathbf{uu}} l_k + \mathbf{B}_k^\top \mathbf{P}_{k+1} \mathbf{B}_k$$
$$\mathbf{G}_{ux} = \nabla^2_{\mathbf{ux}} l_k + \mathbf{B}_k^\top \mathbf{P}_{k+1} \mathbf{A}_k$$

This simplification is widely used in practice.

11.4 The Full iLQR Algorithm

Algorithm 8: Iterative LQR (iLQR)

Input: Initial trajectory $\bar{\mathbf{U}} = \{\bar{\mathbf{u}}_1, \dots, \bar{\mathbf{u}}_{N-1}\}$, initial state $\mathbf{x}_1$, dynamics f_k, costs l_k, l_N
Output: Optimal $\mathbf{U}^\star, \mathbf{X}^\star$

Rollout $\bar{\mathbf{X}}$ using $\bar{\mathbf{U}}$ and f_k; Compute $J(\bar{\mathbf{X}}, \bar{\mathbf{U}})$
while *not converged* **do**
  `// 1. Backward Pass (Compute Policy)`
  $\mathbf{p}_N \leftarrow \nabla l_N(\bar{\mathbf{x}}_N)$; $\mathbf{P}_N \leftarrow \nabla^2 l_N(\bar{\mathbf{x}}_N)$; **for** $k = N-1$ **to** 1 **do**
    Compute cost derivatives at $(\bar{\mathbf{x}}_k, \bar{\mathbf{u}}_k)$: $\mathbf{l}_x, \mathbf{l}_u, \mathbf{L}_{xx}, \mathbf{L}_{uu}, \mathbf{L}_{ux}$
    Compute dynamics Jacobians at $(\bar{\mathbf{x}}_k, \bar{\mathbf{u}}_k)$: $\mathbf{A}_k, \mathbf{B}_k$
    Compute Q-function model:
      $\mathbf{g}_x \leftarrow \mathbf{l}_x + \mathbf{A}_k^\top \mathbf{p}_{k+1}$; $\mathbf{g}_u \leftarrow \mathbf{l}_u + \mathbf{B}_k^\top \mathbf{p}_{k+1}$
      $\mathbf{G}_{xx} \leftarrow \mathbf{L}_{xx} + \mathbf{A}_k^\top \mathbf{P}_{k+1} \mathbf{A}_k$
      $\mathbf{G}_{uu} \leftarrow \mathbf{L}_{uu} + \mathbf{B}_k^\top \mathbf{P}_{k+1} \mathbf{B}_k$; $\mathbf{G}_{ux} \leftarrow \mathbf{L}_{ux} + \mathbf{B}_k^\top \mathbf{P}_{k+1} \mathbf{A}_k$
    (Regularize $\mathbf{G}_{uu}$ to be positive-definite)
    Compute policy: $\mathbf{d}_k \leftarrow -(\mathbf{G}_{uu})^{-1} \mathbf{g}_u$; $\mathbf{K}_k \leftarrow -(\mathbf{G}_{uu})^{-1} \mathbf{G}_{ux}$
    Compute new V-model: $\mathbf{p}_k, \mathbf{P}_k$ using update rules from §11.2
  `// 2. Forward Pass (Line Search)`
  $\alpha \leftarrow 1.0$; $\mathbf{x}_1^{\text{new}} \leftarrow \mathbf{x}_1$
  while *true* **do**
    for $k = 1$ **to** $N-1$ **do**
      $\delta \mathbf{x}_k = \mathbf{x}_k^{\text{new}} - \bar{\mathbf{x}}_k$
      $\mathbf{u}_k^{\text{new}} \leftarrow \bar{\mathbf{u}}_k + \alpha \mathbf{d}_k + \mathbf{K}_k \delta \mathbf{x}_k$
      $\mathbf{x}_{k+1}^{\text{new}} \leftarrow f_k(\mathbf{x}_k^{\text{new}}, \mathbf{u}_k^{\text{new}})$
    $J_{\text{new}} \leftarrow J(\mathbf{X}^{\text{new}}, \mathbf{U}^{\text{new}})$
    if $J_{new} < J$ **then**
      break `// Accept step`
    $\alpha \leftarrow 0.5\alpha$; `// Backtrack`
  `// 3. Update Trajectory`
  $J \leftarrow J_{\text{new}}$; $\bar{\mathbf{X}} \leftarrow \mathbf{X}^{\text{new}}$; $\bar{\mathbf{U}} \leftarrow \mathbf{U}^{\text{new}}$

Code 11.1 (Julia Notebook: `acrobot-ilqr.ipynb`).
The provided `acrobot-ilqr.ipynb` notebook implements exactly this iLQR algorithm.

- **System:** It defines the nonlinear `acrobot_dynamics` and its `dynamics_rk4` discretization. The Acrobot is a classic underactuated system (like a pendulum on a pendulum, but only the "elbow" is motorized), which is highly nonlinear.
- **Goal:** The goal is to swing up from the hanging position ($\mathbf{x}_0 = [-\pi/2, 0, 0, 0]$) to the upright, balanced position ($\mathbf{x}_{\text{goal}} = [\pi/2, 0, 0, 0]$).
- **Derivatives:** It uses `ForwardDiff.jacobian` (`dfdx`) and `ForwardDiff.derivative` (`dfdu`) to compute $\mathbf{A}_k$ and $\mathbf{B}_k$ at each point along the trajectory. It never computes second derivatives of the dynamics, confirming it is iLQR, not DDP.
- **Backward Pass:** The loop `for k = (Nt-1):-1:1` computes `gx`, `gu`, `Gxx`, `Guu`, `Gxu` using the iLQR equations. It then computes the feedforward gain `d[k]` and feedback

gain `K[:,:,k]`.

- **Forward Pass:** The line search performs robust backtracking when

$$extisnan(J_{\text{new}}) \text{ or } J_{\text{new}} > J - 10^{-2}\,\alpha\,\Delta J,$$

rolls out a new trajectory using the computed policy $\mathbf{u}_{\text{new}} = \bar{\mathbf{u}} + \alpha\mathbf{d} + \mathbf{K}(\mathbf{x}_{\text{new}} - \bar{\mathbf{x}})$, and accepts the step only if it makes sufficient progress.

- **Result:** The algorithm converges in a number of iterations, finding a dynamic swing-up motion that gets the Acrobot to its goal state. The plots and animation show this complex, nonlinear behavior.

Lecture 12: Free-Time Problems and Direct Collocation

We shift our focus from "shooting-based" methods (like DDP) to "direct" methods. These methods re-frame the entire trajectory optimization problem as a single, large Nonlinear Program (NLP) and solve it with off-the-shelf optimizers.

12.1 Handling Free/Minimum-Time Problems

A common problem in robotics is to find the *fastest* path to a goal, not just *a* path. This is a minimum-time problem.

Definition 12.1 (Minimum-Time Optimal Control). The continuous-time minimum-time problem is often formulated as:

$$\begin{aligned} \min_{\mathbf{x}(t),\mathbf{u}(t),t_f} \quad & J = \int_0^{t_f} 1\, dt = t_f \\ \text{subject to} \quad & \dot{\mathbf{x}}(t) = f(\mathbf{x}(t), \mathbf{u}(t)) \\ & \mathbf{x}(0) = \mathbf{x}_0 \\ & \mathbf{x}(t_f) = \mathbf{x}_{\text{goal}} \\ & \mathbf{u}_{\min} \le \mathbf{u}(t) \le \mathbf{u}_{\max} \end{aligned}$$

When we discretize this, the number of time steps N is no longer fixed. A common trick is to fix the number of knot points N but make the time step h_k between knot points a decision variable.

Let the total time be $T = \sum_{k=1}^{N-1} h_k$. The problem becomes:

$$\begin{aligned} \min_{\mathbf{X},\mathbf{U},\mathbf{h}} \quad & J = \sum_{k=1}^{N-1} h_k \\ \text{subject to} \quad & \mathbf{x}_{k+1} = f_d(\mathbf{x}_k, \mathbf{u}_k, h_k) \quad \text{(Dynamics depend on } h_k\text{)} \\ & \mathbf{x}_1 = \mathbf{x}_0, \quad \mathbf{x}_N = \mathbf{x}_{\text{goal}} \\ & \mathbf{u}_{\min} \le \mathbf{u}_k \le \mathbf{u}_{\max} \\ & h_k \ge 0 \quad \text{(Time must move forward)} \end{aligned}$$

The stage cost $l(\mathbf{x}_k, \mathbf{u}_k)$ must also be scaled by the time step, as it represents a cost *rate*.

$$J = \sum_{k=1}^{N-1} h_k l(\mathbf{x}_k, \mathbf{u}_k) + l_F(\mathbf{x}_N)$$

If we do not scale the cost by h_k, the solver may exploit this by making h_k arbitrarily large or negative to minimize cost, which is physically meaningless.

The inclusion of h_k as a decision variable makes the dynamics constraint $\mathbf{x}_{k+1} = f_d(\mathbf{x}_k, \mathbf{u}_k, h_k)$ (e.g., from an RK4 step) non-convex, even if the original dynamics were linear. This firmly places free-time problems in the realm of nonlinear optimization.

12.2 Direct Trajectory Optimization

The strategy we just described is an example of **Direct Trajectory Optimization**. The basic strategy is to transcribe the continuous-time optimal control problem into a standard, large-scale Nonlinear Program (NLP).

Note (Transcription Strategy).

1. **Discretize:** Choose a number of knot points N.
2. **Define Decision Vector:** Stack all states and controls into one massive decision vector $\mathbf{z}$:

$$\mathbf{z} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{u}_1 \\ \mathbf{x}_2 \\ \mathbf{u}_2 \\ \vdots \\ \mathbf{x}_{N-1} \\ \mathbf{u}_{N-1} \\ \mathbf{x}_N \end{bmatrix} \in \mathbb{R}^{N(n+m)-m}$$

3. **Formulate NLP:** Transcribe the OCP into the standard NLP form:

$$\begin{aligned} \min_{\mathbf{z}} \quad & f(\mathbf{z}) = \sum_{k=1}^{N-1} l(\mathbf{x}_k, \mathbf{u}_k) + l_F(\mathbf{x}_N) \\ \text{s.t.} \quad & \mathbf{c}_{\text{dyn}}(\mathbf{z}) = 0 \quad \text{(Dynamics as equality constraints)} \\ & \mathbf{d}(\mathbf{z}) \leq 0 \quad \text{(State/control path constraints)} \end{aligned}$$

This NLP is large but **sparse**. The dynamics constraint for time k, for example, only involves variables from time k and $k+1$.

We can then use powerful, general-purpose NLP solvers like **IPOPT** (Interior-Point Optimizer), **SNOPT**, or **KNITRO** to find $\mathbf{z}^\star$.

12.2.1 Sequential Quadratic Programming (SQP)

The most common class of algorithms for solving these NLPs is **Sequential Quadratic Programming (SQP)**. SQP is to constrained optimization what Newton's method is to unconstrained optimization.

Intuition. SQP is a generalization of Newton's method that handles inequality constraints.

- **Goal:** Solve the KKT conditions for the NLP.
- **Strategy:** At each iteration k, form a **Quadratic Program (QP)** subproblem by:
  1. Taking a 2nd-order Taylor expansion of the **Lagrangian** $\mathcal{L}(\mathbf{z}, \lambda, \mu)$ around $\mathbf{z}_k$.
  2. Taking a 1st-order (linear) expansion of the constraints $\mathbf{c}(\mathbf{z})$ and $\mathbf{d}(\mathbf{z})$.
  3. If, the inequalities are convex (i.e. conic), we can generalise SQP into **Sequential Convex Programming (SCP)** by keeping the original inequalities instead of linearising them.
- **Subproblem:** Solve this QP (which is convex if $\nabla^2_{\mathbf{zz}}\mathcal{L}$ is positive definite) to find a primal-dual search direction $\Delta\mathbf{z} = (\delta\mathbf{z}, \delta\lambda, \delta\mu)$.
- **Update:** Perform a line search along this direction using a **merit function** (which balances objective decrease vs. constraint violation) to find $\mathbf{z}_{k+1}$.

For trajectory optimization, the KKT system formed by this NLP is massive but has a strong, sparse, block-banded structure. High-performance solvers for trajectory optimization (like iLQR, DDP, and those used in DIRCOL) are all fundamentally about solving this sparse

KKT system efficiently.

12.3 Direct Collocation (DIRCOL)

The shooting methods (such as DDP) and simple direct transcription (using Euler integration) have a limitation: they are only first-order accurate. If we use a simple $\mathbf{x}_{k+1} = \mathbf{x}_k + hf(\mathbf{x}_k, \mathbf{u}_k)$ as the dynamics constraint, we need a very small h (and thus a very large NLP) to obtain an accurate trajectory.

Direct Collocation methods provide a much more accurate transcription.

Definition 12.2 (Direct Collocation). Direct Collocation methods represent the state trajectory $\mathbf{x}(t)$ as a **piecewise polynomial spline**. The dynamics $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$ are not enforced everywhere, but are enforced as equality constraints at a finite number of **collocation points**.

The most common variant is **Hermite-Simpson Collocation**.

- **State Spline:** The state $\mathbf{x}(t)$ over the interval $[t_k, t_{k+1}]$ is represented by a **cubic Hermite polynomial**. This spline is uniquely defined by four values: the state and its derivative at the start and end of the interval, $(\mathbf{x}_k, \dot{\mathbf{x}}_k)$ and $(\mathbf{x}_{k+1}, \dot{\mathbf{x}}_{k+1})$.
- **Control Spline:** The control $\mathbf{u}(t)$ is typically represented by a piecewise-linear spline, i.e., linear interpolation between $\mathbf{u}_k$ and $\mathbf{u}_{k+1}$.

The key approach in DIRCOL is that we make $\mathbf{x}_k$ and $\mathbf{u}_k$ the decision variables, and then *approximate* the derivatives $\dot{\mathbf{x}}_k$ using the dynamics:

$$\dot{\mathbf{x}}_k \approx f(\mathbf{x}_k, \mathbf{u}_k) \quad \text{and} \quad \dot{\mathbf{x}}_{k+1} \approx f(\mathbf{x}_{k+1}, \mathbf{u}_{k+1})$$

With these four values $(\mathbf{x}_k, f(\mathbf{x}_k, \mathbf{u}_k), \mathbf{x}_{k+1}, f(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}))$, the cubic spline is fully defined.

The Collocation Constraint We now enforce the dynamics $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$ at a single **collocation point** in the middle of the interval: $t_{k+1/2} = t_k + h/2$.

1. **Spline value at midpoint:** The cubic Hermite spline equations give the state *value* at the midpoint as:
$$\mathbf{x}_{k+1/2} = \frac{1}{2}(\mathbf{x}_k + \mathbf{x}_{k+1}) + \frac{h}{8}(\dot{\mathbf{x}}_k - \dot{\mathbf{x}}_{k+1})$$
2. **Spline derivative at midpoint:** The derivative of the spline at the midpoint is:
$$\dot{\mathbf{x}}_{k+1/2} = -\frac{3}{2h}(\mathbf{x}_k - \mathbf{x}_{k+1}) - \frac{1}{4}(\dot{\mathbf{x}}_k + \dot{\mathbf{x}}_{k+1})$$
3. **Control at midpoint:** From linear interpolation:
$$\mathbf{u}_{k+1/2} = \frac{1}{2}(\mathbf{u}_k + \mathbf{u}_{k+1})$$

Now, we substitute $\dot{\mathbf{x}}_k \approx f_k = f(\mathbf{x}_k, \mathbf{u}_k)$ and $\dot{\mathbf{x}}_{k+1} \approx f_{k+1} = f(\mathbf{x}_{k+1}, \mathbf{u}_{k+1})$ into these equations.

The **collocation constraint** (or defect) requires that the spline's derivative $\dot{\mathbf{x}}_{k+1/2}$ must equal the dynamics evaluated at the spline's midpoint value $f(\mathbf{x}_{k+1/2}, \mathbf{u}_{k+1/2})$.

$$\mathbf{c}_k(\mathbf{x}_k, \mathbf{u}_k, \mathbf{x}_{k+1}, \mathbf{u}_{k+1}) = \dot{\mathbf{x}}_{k+1/2} - f(\mathbf{x}_{k+1/2}, \mathbf{u}_{k+1/2}) = 0$$

Writing this out in full, the dynamics constraint for the NLP is:

$$\mathbf{c}_k = \left(-\frac{3}{2h}(\mathbf{x}_k - \mathbf{x}_{k+1}) - \frac{1}{4}(f_k + f_{k+1}) \right) - f\left(\frac{1}{2}(\mathbf{x}_k + \mathbf{x}_{k+1}) + \frac{h}{8}(f_k - f_{k+1}),\ \ \frac{1}{2}(\mathbf{u}_k + \mathbf{u}_{k+1}) \right) = 0$$

This single, complex equation provides a 3rd-order accurate integration scheme!

> **Note** (Efficiency). This approach appears computationally expensive, but it is actually very efficient. To evaluate the constraint and its Jacobian, we need f_k, f_{k+1}, and $f_{k+1/2}$. However, the f_k and f_{k+1} terms are shared with the adjacent constraints (c_{k-1} and c_{k+1}). The net cost is approximately 2 dynamics evaluations per time step, which is *less* than a standard RK4 (which requires 4).

Code 12.1 (Julia Notebook: `dircol.ipynb` (Acrobot Swing-up)).
The provided Julia notebook, `dircol.ipynb`, demonstrates the implementation of the Hermite-Simpson Direct Collocation method.

Objective The notebook solves the Acrobot swing-up problem. The goal is to find a trajectory from the hanging-down state ($\mathbf{x}_0 = [-\pi/2, 0, 0, 0]$) to the upright, balanced state ($\mathbf{x}_{\text{goal}} = [\pi/2, 0, 0, 0]$). This is a classic, highly nonlinear and underactuated problem.

Transcription The notebook transcribes this problem into an NLP to be solved by **Ipopt**.

- **Decision Vector:** It defines `n_nlp = (Nx+Nu)*Nt` as the total number of variables in the flattened decision vector $\mathbf{z}$, which contains all $\mathbf{x}_k$ and $\mathbf{u}_k$.
- **NLP Functions:** It provides two key functions to Ipopt:
  1. `cost(ztraj)`: This is the NLP objective $f(\mathbf{z})$. It sums a simple quadratic stage cost $\frac{1}{2}(\mathbf{x}_k - \mathbf{x}_{\text{goal}})^\top \mathbf{Q}(\mathbf{x}_k - \mathbf{x}_{\text{goal}}) + \frac{1}{2}\mathbf{u}_k^\top \mathbf{R}\mathbf{u}_k$ over all time steps.
  2. `con!(c, ztraj)`: This is the NLP constraint function $\mathbf{g}(\mathbf{z})$. It fills the constraint vector `c` with:
     - The initial condition: $\mathbf{c}[1 : Nx] = \mathbf{x}_1 - \mathbf{x}_0$
     - The terminal condition: $\mathbf{c}[end - Nx + 1 : end] = \mathbf{x}_N - \mathbf{x}_{\text{goal}}$
     - The dynamics: The middle of the vector is filled by the `dircol_dynamics` function for each time step.
- **Collocation Constraint:** The function `dircol_dynamics(x1,u1,x2,u2)` implements the constraint we derived. It computes `xm`, `um`, `xdotm` (from the spline) and `fm` (from the dynamics) and returns the defect `fm - xdotm`.

Solving and Results The notebook uses `ForwardDiff.jacobian!` to compute the sparse Jacobians of the cost and constraint functions automatically. It passes all this information to the `Ipopt.Optimizer`.

Lecture 13: Algorithm Recap and Attitude Kinematics

We have covered a wide range of algorithms, each suited to different problem types. The following provides a decision-making framework for choosing an algorithm:

- **Problem: Stabilization (Time-Invariant)**
  - If the system is linear and has no constraints: Use LQR (Linear Quadratic Regulator). This involves solving the Discrete Algebraic Riccati Equation (DARE) once to get a single, stabilizing gain $\mathbf{K}$.
  - If the system is linear and has constraints (e.g., actuator limits): Use Model Predictive Control (MPC). The subproblem to solve at each time step is a Quadratic Program (QP) or a Second-Order Cone Program (SOCP) for conic constraints. The LQR solution is typically used as the terminal cost.
- **Problem: Reference Tracking (Time-Varying)**
  - If the system is linear (or linearized about the reference trajectory) and has no constraints: Use TVLQR (Time-Varying LQR). This involves solving the Riccati recursion backward in time to get a time-varying gain sequence $\mathbf{K}_k$.
- **Problem: Nonlinear Trajectory Optimization (far from equilibrium)**
  - This is the most general problem, and the choice is between the two main families of algorithms we've studied: DDP and DIRCOL.

The following table provides a direct comparison of the trade-offs between Direct Collocation (as a sparse NLP) and DDP/iLQR.

Table 1: Comparison of Nonlinear Trajectory Optimization Methods

Property	**Direct Collocation (DIRCOL)**	**DDP / iLQR**
Dynamics	Enforced as constraints $\mathbf{c}(\mathbf{x}_k, \mathbf{u}_k, \mathbf{x}_{k+1}) = 0$.	Always dynamically feasible (rollouts use exact dynamics).
Initial Guess	Can use any infeasible guess for states and controls.	Can only guess the control sequence $\mathbf{U}$.
Constraints	Handles arbitrary state/control constraints easily (as NLP constraints).	State constraints are difficult to handle; requires methods like AL or penalties.
Tracking Controller	Must be designed separately (e.g., by running TVLQR on the result).	Produces a TVLQR tracking controller ($\mathbf{K}_k$) for free.
Convergence	Numerically robust; relies on mature large-scale SQP solvers.	Very fast (quadratic) local convergence, but can be ill-conditioned.
Implementation	Complex; relies on an external sparse NLP solver (e.g., IPOPT).	Relatively simple to implement; self-contained.

13.1 The Challenge of Attitude

Many robotic systems, such as quadrotors, spacecraft, and legged robots, undergo large-angle 3D rotations.

- **The Problem:** Naive parameterizations like Euler angles (ϕ, θ, ψ) have **singularities** (e.g., gimbal lock at $\theta = \pm 90^\circ$). An optimizer or controller will fail at these points.
- **The Need:** We require a singularity-free representation to optimize and control motion through all possible attitudes.

This leaves two primary choices: **Rotation Matrices** and **Unit Quaternions**.

13.2 Representation 1: Rotation Matrices (SO(3))

Attitude describes the orientation of a body-fixed coordinate frame $\{\mathcal{B}\}$ relative to a fixed world or navigation frame $\{\mathcal{N}\}$. We can represent this with a 3×3 **rotation matrix** (or Direction Cosine Matrix, DCM), $\mathbf{R}$, that rotates vectors from the body frame to the world frame:

$$\mathbf{v}_{\mathcal{N}} = \mathbf{R}\mathbf{v}_{\mathcal{B}}$$

Definition 13.1 (Special Orthogonal Group, SO(3)). A valid rotation matrix $\mathbf{R}$ must belong to the **Special Orthogonal Group**, $SO(3)$. This imposes two constraints:

1. **Orthogonal:** The matrix's columns are orthonormal, $\mathbf{R}^\top\mathbf{R} = \mathbf{I}$. This ensures it preserves lengths and angles.
2. **Special:** The determinant is positive one, $\det(\mathbf{R}) = 1$. This ensures it's a right-handed system (i.e., not a reflection).

Kinematics. To use $\mathbf{R}$ in a dynamic model, we must know how it evolves in time given a body-frame angular velocity $\omega \in \mathbb{R}^3$ (which is what a gyro measures).

The kinematic relationship is:

$$\dot{\mathbf{R}} = \mathbf{R}\hat{\omega}$$

where $\hat{\omega}$ is the **hat map**, which converts the vector ω into its skew-symmetric matrix form:

$$\omega = \begin{bmatrix}\omega_x \\ \omega_y \\ \omega_z\end{bmatrix} \quad \Rightarrow \quad \hat{\omega} = \begin{bmatrix}0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0\end{bmatrix}$$

such that $\hat{\omega}\mathbf{p} = \omega \times \mathbf{p}$ for any vector $\mathbf{p}$.

Note (Derivation of Rotation Kinematics). Consider a point $\mathbf{p}$ that is fixed in the body frame ($\dot{\mathbf{p}}_{\mathcal{B}} = 0$). Its representation in the world frame is $\mathbf{p}_{\mathcal{N}}(t) = \mathbf{R}(t)\mathbf{p}_{\mathcal{B}}$.

Differentiating this using the product rule gives:

$$\dot{\mathbf{p}}_{\mathcal{N}} = \dot{\mathbf{R}}\mathbf{p}_{\mathcal{B}} + \mathbf{R}\dot{\mathbf{p}}_{\mathcal{B}} = \dot{\mathbf{R}}\mathbf{p}_{\mathcal{B}}$$

From physics, we also know that the linear velocity of the point is $\dot{\mathbf{p}}_{\mathcal{N}} = \omega_{\mathcal{N}} \times \mathbf{p}_{\mathcal{N}}$.

We can express the angular velocity and position in the body frame: $\omega_{\mathcal{N}} = \mathbf{R}\omega$ and $\mathbf{p}_{\mathcal{N}} = \mathbf{R}\mathbf{p}_{\mathcal{B}}$.

$$\dot{\mathbf{p}}_{\mathcal{N}} = (\mathbf{R}\omega) \times (\mathbf{R}\mathbf{p}_{\mathcal{B}}) = \mathbf{R}(\omega \times \mathbf{p}_{\mathcal{B}}) = \mathbf{R}\hat{\omega}\mathbf{p}_{\mathcal{B}}$$

Equating the two expressions for $\dot{\mathbf{p}}_{\mathcal{N}}$, we get $\dot{\mathbf{R}}\mathbf{p}_{\mathcal{B}} = \mathbf{R}\hat{\omega}\mathbf{p}_{\mathcal{B}}$. Since this must hold for any $\mathbf{p}_{\mathcal{B}}$, we find $\dot{\mathbf{R}} = \mathbf{R}\hat{\omega}$.

Problem with SO(3): We can use $\mathbf{R}$ as part of our state, but:

1. **Redundancy:** We use 9 numbers (and 6 constraints) to represent 3 DOFs.
2. **Numerical Drift:** When we numerically integrate $\dot{\mathbf{R}} = \mathbf{R}\hat{\omega}$, small integration errors will accumulate. After many steps, the resulting $\mathbf{R}$ will no longer be perfectly orthogonal (i.e., $\mathbf{R}^\top\mathbf{R} \neq \mathbf{I}$). The matrix "drifts" off the $SO(3)$ manifold, leading to simulation errors.

13.3 Representation 2: Unit Quaternions (S^3)

Quaternions are a 4-dimensional number system that is extremely efficient and stable for representing rotations.

Definition 13.2 (Unit Quaternion). Based on Euler's Rotation Theorem, any rotation can be defined by an angle θ about a unit axis $\mathbf{a}$. A unit quaternion $\mathbf{q} \in \mathbb{R}^4$ is defined as:

$$\mathbf{q} = \begin{bmatrix} s \\ \mathbf{v} \end{bmatrix} = \begin{bmatrix} \cos(\theta/2) \\ \mathbf{a}\sin(\theta/2) \end{bmatrix}$$

where s is the "scalar part" and $\mathbf{v}$ is the "vector part".

All valid rotations correspond to **unit quaternions**, which live on the 4D unit sphere (called S^3), satisfying the constraint:

$$\mathbf{q}^\top \mathbf{q} = s^2 + v_x^2 + v_y^2 + v_z^2 = \cos^2(\theta/2) + \sin^2(\theta/2) = 1$$

Note. Quaternions are a **double cover** of $SO(3)$. $\mathbf{q}$ and $-\mathbf{q}$ represent the exact same 3D rotation (corresponding to a rotation by θ and $\theta + 2\pi$). This is a topological ambiguity but not a singularity.

Key Operations.

- **Conjugate:** $\mathbf{q}^* = \begin{bmatrix} s \\ -\mathbf{v} \end{bmatrix}$.
- **Multiplication:** Quaternion multiplication $\mathbf{q}_a \otimes \mathbf{q}_b$ (which corresponds to composing rotations) is defined as:

  $$\mathbf{q}_a \otimes \mathbf{q}_b = \begin{bmatrix} s_a s_b - \mathbf{v}_a^\top \mathbf{v}_b \\ s_a \mathbf{v}_b + s_b \mathbf{v}_a + \mathbf{v}_a \times \mathbf{v}_b \end{bmatrix}$$

  This can be written in matrix form as $\mathbf{L}(\mathbf{q}_a)\mathbf{q}_b$ or $\mathbf{R}(\mathbf{q}_b)\mathbf{q}_a$, where:

  $$\mathbf{L}(\mathbf{q}) = \begin{bmatrix} s & -\mathbf{v}^\top \\ \mathbf{v} & s\mathbf{I} + \hat{\mathbf{v}} \end{bmatrix}, \quad \mathbf{R}(\mathbf{q}) = \begin{bmatrix} s & -\mathbf{v}^\top \\ \mathbf{v} & s\mathbf{I} - \hat{\mathbf{v}} \end{bmatrix}$$

- **Rotate a Vector:** To rotate $\mathbf{x} \in \mathbb{R}^3$, we "embed" it as a pure quaternion $\mathbf{x}_p = \begin{bmatrix} 0 \\ \mathbf{x} \end{bmatrix}$ and compute the "sandwich product":

  $$\begin{bmatrix} 0 \\ \mathbf{y} \end{bmatrix} = \mathbf{q} \otimes \mathbf{x}_p \otimes \mathbf{q}^*$$

  The vector part $\mathbf{y}$ is the rotated vector.

Quaternion Kinematics. This is the central equation for dynamics. The derivative of the quaternion $\mathbf{q}$ is related to the body-frame angular velocity ω by:

$$\dot{\mathbf{q}} = \frac{1}{2}\mathbf{q} \otimes \begin{bmatrix} 0 \\ \omega \end{bmatrix}$$

This is most useful when written in matrix form. Let $\mathbf{H} = \begin{bmatrix} 0^\top \\ \mathbf{I}_{3\times 3} \end{bmatrix}$ be the matrix that embeds ω into a pure quaternion. The kinematics are:

$$\dot{\mathbf{q}} = \frac{1}{2}\mathbf{L}(\mathbf{q}) \begin{bmatrix} 0 \\ \omega \end{bmatrix} = \frac{1}{2}\mathbf{L}(\mathbf{q})\mathbf{H}\omega$$

This equation is linear in ω, which is extremely convenient for control design.

Code 13.1 (Julia Notebook: `rbsim.ipynb`).
Goal: torque-free rigid-body simulation; compare DCM vs. quaternion integration.

Part 1 — DCM

- State: $\mathbf{x} \in \mathbb{R}^{12} = [\text{vec}(\mathbf{R}); \omega]$.
- Dynamics: $\dot{\mathbf{R}} = \mathbf{R}\hat{\omega}$, $\dot{\omega} = -\mathbf{J}^{-1}(\hat{\omega}\,\mathbf{J}\,\omega)$.
- Integrator: RK4.
- Result: after 10^4 steps, $\mathbf{R}_k^\top \mathbf{R}_k \neq \mathbf{I}$ (drift off $SO(3)$).

Part 2 — Quaternion

- State: $\mathbf{x} \in \mathbb{R}^7 = [\mathbf{q}; \omega]$.
- Dynamics: $\dot{\mathbf{q}} = \frac{1}{2}\mathbf{L}(\mathbf{q})\mathbf{H}\omega$, $\dot{\omega}$ as above.
- Integrator: RK4 + re-normalize $\mathbf{q} \leftarrow \mathbf{q}/\|\mathbf{q}\|$ each step.

Conclusion Projection enforces $\mathbf{q}^\top\mathbf{q} = 1$, so $\|\mathbf{q}_k\| = 1$ and $\mathbf{R}(\mathbf{q}_k)^\top\mathbf{R}(\mathbf{q}_k) = \mathbf{I}$. Quaternions with re-normalization are the robust default for attitude simulation.

Note. Note that optimisation w.r.t. to qwuaterinions and rotation matrixes are often non-convex, as the space of valid soluitions is a shell of unit sphere, hence the the set ofg solutions is non-convex in lature. But often these roatations optimisation problems admit strong convex relaxations. In particular you can show that when dealing with linear constraints, tthe solution will be pushed to the bundary of the set, allowing for converting the unit norm consstraint to a unit ball one. Wahba's problem is one such example.

Lecture 14: Optimization with Quaternions

We will develop the tools necessary to perform optimization (e.g., Gauss-Newton, DDP, SQP) directly on functions involving quaternions.

14.1 The Geometry of Quaternion Optimization

The central challenge in optimizing with quaternions is that they live on a 3D manifold (the sphere S^3) embedded in a 4D space.

- We cannot simply add an unconstrained update vector $\delta\mathbf{q} \in \mathbb{R}^4$ to $\mathbf{q}_k$, as $\mathbf{q}_k + \delta\mathbf{q}$ will not be a unit quaternion.
- Any valid update $\dot{\mathbf{q}}$ must lie in the 3D *tangent space* at $\mathbf{q}$.

Our optimization algorithms (like Newton or Gauss-Newton) are designed to solve for unconstrained correction vectors $\phi \in \mathbb{R}^3$. We need a way to map these 3D corrections to the 4D tangent space.

> **Intuition** (Analogy to 2D Rotations). This is analogous to 2D rotations on the unit circle S^1. A 2D rotation can be represented by a vector $\mathbf{v} = [\cos\theta, \sin\theta]^\top$. A 1D update (an angular velocity $\dot{\theta} \in \mathbb{R}^1$) is mapped to the 2D tangent space at $\mathbf{v}$ by the kinematics:
>
> $$\dot{\mathbf{v}} = \frac{\partial \mathbf{v}}{\partial \theta}\dot{\theta} = \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix} \dot{\theta}$$
>
> We must do the same for quaternions: map a 3D update $\phi \in \mathbb{R}^3$ to a 4D derivative $\delta\mathbf{q} \in \mathbb{R}^4$.

14.2 Differentiating Functions of Quaternions

> **Note.** This currently does not make sense to me. Come back and revisit this section.

Our optimization update $\phi \in \mathbb{R}^3$ is an unconstrained 3D vector. We must define how this 3D vector perturbs our 4D quaternion $\mathbf{q}$. A natural choice is to use a **multiplicative update**:

$$\mathbf{q}_{k+1} = \mathbf{q}_k \otimes \delta\mathbf{q}(\phi)$$

where $\delta\mathbf{q}(\phi)$ is a small rotation quaternion that is parameterized by ϕ.

The "Attitude Jacobian" $\mathbf{G}(\mathbf{q})$ We need to apply the chain rule. For any function $f(\mathbf{q})$, we want to find its derivative with respect to the 3D perturbation ϕ:

$$\nabla_\phi f = \frac{\partial f}{\partial \mathbf{q}} \frac{\partial \mathbf{q}}{\partial \phi}$$

Let's find the term $\frac{\partial \mathbf{q}}{\partial \phi}$. This is the "Attitude Jacobian" $\mathbf{G}(\mathbf{q})$.

Let $\mathbf{q}'(\phi) = \mathbf{q} \otimes \delta\mathbf{q}(\phi)$. We need to choose a parameterization for $\delta\mathbf{q}(\phi)$.

- **Axis-Angle:** One choice is to let ϕ be an axis-angle vector. For small ϕ, $\delta\mathbf{q}(\phi) \approx [1, \frac{1}{2}\phi]^\top$.
- **Vector Part (Rodrigues):** A simpler choice (used in the `wahba.ipynb` notebook) is to let ϕ be the **vector part** of the update quaternion:

$$\delta\mathbf{q}(\phi) = \begin{bmatrix} \sqrt{1 - ||\phi||^2} \\ \phi \end{bmatrix}$$

Let's use the second definition, as it matches the code. We need the derivative *at the identity* ($\phi = 0$):

$$\left.\frac{\partial \delta\mathbf{q}}{\partial \phi}\right|_{\phi=0} = \frac{\partial}{\partial \phi}\begin{bmatrix}\sqrt{1-\phi^\top\phi}\\ \phi\end{bmatrix}\Bigg|_{\phi=0} = \begin{bmatrix}\frac{-\phi^\top}{\sqrt{1-\phi^\top\phi}}\\ \mathbf{I}\end{bmatrix}\Bigg|_{\phi=0} = \begin{bmatrix}0^\top\\ \mathbf{I}\end{bmatrix} = \mathbf{H}$$

Now we can find $\mathbf{G}(\mathbf{q})$ using the chain rule for $\mathbf{q}' = \mathbf{L}(\mathbf{q})\delta\mathbf{q}(\phi)$:

$$\mathbf{G}(\mathbf{q}) = \left.\frac{\partial \mathbf{q}'}{\partial \phi}\right|_{\phi=0} = \mathbf{L}(\mathbf{q})\left.\frac{\partial \delta\mathbf{q}}{\partial \phi}\right|_{\phi=0} = \mathbf{L}(\mathbf{q})\mathbf{H}$$

This 4×3 matrix $\mathbf{G}(\mathbf{q})$ is the crucial "Attitude Jacobian". It maps 3D unconstrained corrections ϕ from our solver into the 4D tangent space at our current attitude $\mathbf{q}$.

Chain Rules for Optimization Now we can differentiate any function with respect to our 3D parameterization ϕ.

- **Scalar-Valued Function ($f : \mathbb{R}^4 \to \mathbb{R}$):** The 3D tangent-space gradient $\nabla_\phi f \in \mathbb{R}^{1\times3}$ is:

$$\nabla_\phi f(\mathbf{q}) = \frac{\partial f}{\partial \mathbf{q}}\mathbf{G}(\mathbf{q})$$

- **Vector-Valued Function ($\mathbf{r} : \mathbb{R}^4 \to \mathbb{R}^m$):** The 3D tangent-space Jacobian $\mathbf{J}_\phi \in \mathbb{R}^{m\times3}$ is:

$$\mathbf{J}_\phi(\mathbf{q}) = \frac{\partial \mathbf{r}}{\partial \mathbf{q}}\mathbf{G}(\mathbf{q})$$

14.3 Example: Wahba's Problem & Gauss-Newton

A classic robotics problem is "pose estimation" or "attitude determination".

Definition 14.1 (Wahba's Problem). Given a set of m unit vectors known in a world frame, $\{\mathbf{v}_1^N, \dots, \mathbf{v}_m^N\}$, and a corresponding set of measurements of those same vectors in the robot's body frame, $\{\mathbf{v}_1^B, \dots, \mathbf{v}_m^B\}$, find the attitude $\mathbf{q}$ that best aligns them.

This is a nonlinear least-squares problem. We seek to minimize the sum of squared errors:

$$J(\mathbf{q}) = \sum_{i=1}^{m} ||\mathbf{v}_i^N - \mathbf{Q}(\mathbf{q})\mathbf{v}_i^B||_2^2$$

where $\mathbf{Q}(\mathbf{q})$ is the rotation matrix for quaternion $\mathbf{q}$.

To solve this with the Gauss-Newton method, we first stack all errors into a single residual vector $\mathbf{r}(\mathbf{q}) \in \mathbb{R}^{3m}$:

$$\mathbf{r}(\mathbf{q}) = \begin{bmatrix}\mathbf{v}_1^N - \mathbf{Q}(\mathbf{q})\mathbf{v}_1^B\\ \vdots\\ \mathbf{v}_m^N - \mathbf{Q}(\mathbf{q})\mathbf{v}_m^B\end{bmatrix}$$

The cost is $J(\mathbf{q}) = \frac{1}{2}||\mathbf{r}(\mathbf{q})||_2^2$.

Gauss-Newton Method Recall that the Gauss-Newton step $\Delta\mathbf{x}$ for a cost $\frac{1}{2}||\mathbf{r}(\mathbf{x})||^2$ is:

$$\Delta\mathbf{x} = -\left(\mathbf{J}(\mathbf{x})^\top\mathbf{J}(\mathbf{x})\right)^{-1}\mathbf{J}(\mathbf{x})^\top\mathbf{r}(\mathbf{x})$$

where $\mathbf{J}(\mathbf{x}) = \frac{\partial \mathbf{r}}{\partial \mathbf{x}}$.

In our case, the unconstrained variable is the 3D correction ϕ, not the 4D state $\mathbf{q}$. The Jacobian we need is the $3m \times 3$ attitude Jacobian, $\mathbf{J}_\phi$:

$$\mathbf{J}_\phi(\mathbf{q}) = \frac{\partial \mathbf{r}}{\partial \mathbf{q}} \mathbf{G}(\mathbf{q})$$

The Gauss-Newton step $\phi \in \mathbb{R}^3$ is the solution to the 3×3 linear system:

$$\left(\mathbf{J}_\phi^\top \mathbf{J}_\phi\right) \phi = -\mathbf{J}_\phi^\top \mathbf{r}(\mathbf{q})$$

The Full Algorithm (Gauss-Newton for Wahba's Problem): 1. Start with an initial guess $\mathbf{q}_k$ (e.g., $\mathbf{q}_0 = [1, 0, 0, 0]^\top$). 2. **while** $||\mathbf{J}_\phi^\top \mathbf{r}|| > \epsilon$: 3. Compute residual: $\mathbf{r} = \mathbf{r}(\mathbf{q}_k)$. 4. Compute 4D Jacobian: $\frac{\partial \mathbf{r}}{\partial \mathbf{q}}$ (e.g., using auto-differentiation). 5. Compute Attitude Jacobian: $\mathbf{G}(\mathbf{q}_k) = \mathbf{L}(\mathbf{q}_k)\mathbf{H}$. 6. Compute 3D Jacobian: $\mathbf{J}_\phi = \frac{\partial \mathbf{r}}{\partial \mathbf{q}} \mathbf{G}(\mathbf{q}_k)$. 7. Solve for 3D update: $\phi = -\left(\mathbf{J}_\phi^\top \mathbf{J}_\phi\right)^{-1} \mathbf{J}_\phi^\top \mathbf{r}$. 8. Form 4D update quaternion: $\delta\mathbf{q} = \begin{bmatrix} \sqrt{1 - ||\phi||^2} \\ \phi \end{bmatrix}$. 9. Apply multiplicative update: $\mathbf{q}_{k+1} = \mathbf{L}(\mathbf{q}_k)\delta\mathbf{q}$. 10. **end while**

Code 14.1 (Julia Notebook: `wahba.ipynb`).
The `wahba.ipynb` notebook implements this exact algorithm.

- **Setup:** It defines the helper functions `Q(q)` (quaternion to DCM), `L(q)`, `H`, and `G(q) = L(q)*H`.
- **Data:** It generates a random `qtrue`, then creates $m = 10$ pairs of corresponding world vectors `vN` and body vectors `vB`.
- **Residual:** The `residual(q)` function computes $\mathbf{r}(\mathbf{q}) = \text{vec}(\texttt{vN} - \mathbf{Q}(\mathbf{q})\texttt{vB})$.
- **Gauss-Newton Loop:** The `while` loop implements the algorithm described above:
  - `r = residual(q)`: Computes the 30×1 residual vector.
  - `dr = ForwardDiff.jacobian(residual, q)`: Auto-differentiates to get the 30×4 Jacobian $\frac{\partial \mathbf{r}}{\partial \mathbf{q}}$.
- **Result:** The algorithm converges in a few iterations. The final cell shows that `q` is either very close to `qtrue` or `-qtrue`, correctly identifying the true attitude (up to the double-cover ambiguity).

Lecture 15: LQR with Quaternions and Quadrotor Control

> **As previously seen.** In previous lectures, we explored trajectory optimization methods like DDP and Direct Collocation. We also introduced unit quaternions as a singularity-free representation for attitude. However, we noted that using quaternions introduces a constraint $\|\mathbf{q}\| = 1$, which complicates standard control techniques like LQR.

Today, we bridge the gap between quaternion kinematics and optimal control. We will derive how to apply LQR to systems with quaternion states by linearizing in the tangent space (error state). Then, we will apply this to a full 3D quadrotor model.

15.1 LQR with Quaternions

A naive approach to linearizing a system with quaternion states leads to issues. Consider a state vector $\mathbf{x} \in \mathbb{R}^{13}$ containing a position $\mathbf{r} \in \mathbb{R}^3$, a quaternion $\mathbf{q} \in \mathbb{R}^4$, a linear velocity $\mathbf{v} \in \mathbb{R}^3$, and an angular velocity $\omega \in \mathbb{R}^3$.

$$\mathbf{x} = \begin{bmatrix} \mathbf{r} \\ \mathbf{q} \\ \mathbf{v} \\ \omega \end{bmatrix}$$

If we linearize the discrete-time dynamics $\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k)$ directly to obtain $\mathbf{A} \in \mathbb{R}^{13\times 13}$ and $\mathbf{B} \in \mathbb{R}^{13\times m}$, the resulting linear system will be **uncontrollable**.

> **Intuition.** The quaternion constraint $\mathbf{q}^\top \mathbf{q} = 1$ confines the state to a manifold. There is no control authority that can move the state off this manifold (i.e., change the norm of the quaternion). Consequently, the controllability matrix will be rank-deficient (rank < 13), and the standard Riccati recursion will fail (specifically, $\mathbf{P}$ matrices may become singular or ill-conditioned).

To solve this, we must linearize the dynamics in the **error state** space, often called the "tangent space." While the global orientation is represented by a quaternion (4 numbers), the *error* in orientation is 3-dimensional (like a rotation vector).

15.1.1 Error State Dynamics

Let the nominal state be $\bar{\mathbf{x}}$ and the true state be $\mathbf{x}$. We define an error state $\delta\tilde{\mathbf{x}} \in \mathbb{R}^{12}$ (note the dimension reduction from 13 to 12):

$$\delta\tilde{\mathbf{x}} = \begin{bmatrix} \delta\mathbf{r} \\ \delta\phi \\ \delta\mathbf{v} \\ \delta\omega \end{bmatrix}.$$

The position, velocity, and angular velocity errors are standard arithmetic differences (e.g., $\delta\mathbf{r} = \mathbf{r} - \bar{\mathbf{r}}$). The orientation error $\delta\phi \in \mathbb{R}^3$ is defined such that it relates the nominal quaternion $\bar{\mathbf{q}}$ to the true quaternion $\mathbf{q}$.

Commonly, the error quaternion $\delta\mathbf{q}$ is defined as the rotation required to go from the nominal frame to the true frame:

$$\delta\mathbf{q} = \bar{\mathbf{q}}^{-1} \otimes \mathbf{q} = \bar{\mathbf{q}}^{*} \otimes \mathbf{q} \tag{15.1}$$

For small errors, $\delta\mathbf{q} \approx \begin{bmatrix} 1 \\ \frac{1}{2}\delta\phi \end{bmatrix}$. We define a mapping matrix $\mathbf{E}(\mathbf{q})$ that projects variations from the full state space to the error state space. For the quaternion part, we use the matrix

$\mathbf{G}(\mathbf{q})$ discussed in previous lectures (where $\dot{\mathbf{q}} = \frac{1}{2}\mathbf{G}(\mathbf{q})\omega$). Typically, $\mathbf{G}(\mathbf{q}) = \mathbf{L}(\mathbf{q})\mathbf{H}$, where $\mathbf{H} = [0_{1\times 3}; \mathbf{I}_3]$. The full projection matrix is chosen block-diagonal:

$$\mathbf{E}(\mathbf{x}) = \text{blockdiag}(\mathbf{I}_3, \mathbf{G}(\mathbf{q}), \mathbf{I}_6) \in \mathbb{R}^{13\times 12}.$$

Near the nominal trajectory we can write, to first order,

$$\delta\mathbf{x}_k \approx \mathbf{E}(\bar{\mathbf{x}}_k)\, \delta\tilde{\mathbf{x}}_k, \tag{15.2}$$

meaning that $\mathbf{E}(\bar{\mathbf{x}}_k)$ "lifts" the error state into a perturbation in the original coordinates. Conversely, the transpose $\mathbf{E}(\bar{\mathbf{x}}_k)^\top$ acts as a local projection from the ambient 13D perturbation back onto the 12D tangent space.

Chain-rule view. The nonlinear dynamics are

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k). \tag{15.3}$$

Linearizing around the nominal trajectory $(\bar{\mathbf{x}}_k, \bar{\mathbf{u}}_k)$ gives the usual 13-state Jacobians

$$\delta\mathbf{x}_{k+1} \approx \mathbf{A}_k\, \delta\mathbf{x}_k + \mathbf{B}_k\, \delta\mathbf{u}_k, \tag{15.4}$$

where

$$\mathbf{A}_k = \left.\frac{\partial f}{\partial \mathbf{x}}\right|_{\bar{\mathbf{x}}_k, \bar{\mathbf{u}}_k}, \qquad \mathbf{B}_k = \left.\frac{\partial f}{\partial \mathbf{u}}\right|_{\bar{\mathbf{x}}_k, \bar{\mathbf{u}}_k}.$$

We want dynamics directly in the error variables $\delta\tilde{\mathbf{x}}$. Using the projection at the next time step,

$$\delta\tilde{\mathbf{x}}_{k+1} \approx \mathbf{E}(\bar{\mathbf{x}}_{k+1})^\top\, \delta\mathbf{x}_{k+1}, \tag{15.5}$$

and substituting the linearized dynamics, we obtain

$$\begin{aligned}
\delta\tilde{\mathbf{x}}_{k+1} &\approx \mathbf{E}(\bar{\mathbf{x}}_{k+1})^\top \big(\mathbf{A}_k\, \delta\mathbf{x}_k + \mathbf{B}_k\, \delta\mathbf{u}_k\big). && (15.6)\\
&\approx \mathbf{E}(\bar{\mathbf{x}}_{k+1})^\top \big(\mathbf{A}_k\, \mathbf{E}(\bar{\mathbf{x}}_k)\, \delta\tilde{\mathbf{x}}_k + \mathbf{B}_k\, \delta\mathbf{u}_k\big), && (15.7)
\end{aligned}$$

where in the last step we used $\delta\mathbf{x}_k \approx \mathbf{E}(\bar{\mathbf{x}}_k)\, \delta\tilde{\mathbf{x}}_k$. Collecting terms, the error dynamics take the standard linear form

$$\delta\tilde{\mathbf{x}}_{k+1} = \tilde{\mathbf{A}}_k\, \delta\tilde{\mathbf{x}}_k + \tilde{\mathbf{B}}_k\, \delta\mathbf{u}_k, \tag{15.8}$$

with

$$\tilde{\mathbf{A}}_k = \mathbf{E}(\bar{\mathbf{x}}_{k+1})^\top \left(\left.\frac{\partial f}{\partial \mathbf{x}}\right|_{\bar{\mathbf{x}}_k, \bar{\mathbf{u}}_k}\right) \mathbf{E}(\bar{\mathbf{x}}_k) \tag{15.9}$$

$$\tilde{\mathbf{B}}_k = \mathbf{E}(\bar{\mathbf{x}}_{k+1})^\top \left(\left.\frac{\partial f}{\partial \mathbf{u}}\right|_{\bar{\mathbf{x}}_k, \bar{\mathbf{u}}_k}\right) \tag{15.10}$$

Geometrically, we first lift a small tangent-space perturbation at time k into the ambient 13D space, evolve it forward through the linearized dynamics, and finally project the result back onto the tangent space at time $k+1$. The resulting system $\delta\tilde{\mathbf{x}}_{k+1} = \tilde{\mathbf{A}}_k\delta\tilde{\mathbf{x}}_k + \tilde{\mathbf{B}}_k\delta\mathbf{u}_k$ is a standard linear system of dimension 12. It is controllable (assuming the underlying physical system is), and we can apply the standard LQR algorithm to find the gain matrix $\mathbf{K} \in \mathbb{R}^{m\times 12}$.

15.1.2 Attitude Control Loop

The resulting control law is:

$$\mathbf{u}_k = \bar{\mathbf{u}}_k - \mathbf{K}_k \delta\tilde{\mathbf{x}}_k$$

To implement this, we must compute $\delta\tilde{\mathbf{x}}_k$ given the current state $\mathbf{x}$ and reference $\bar{\mathbf{x}}$:

1. Compute standard errors: $\delta\mathbf{r} = \mathbf{r} - \bar{\mathbf{r}}$, $\delta\mathbf{v} = \mathbf{v} - \bar{\mathbf{v}}$, etc.
2. Compute orientation error $\delta\phi$. First, compute the error quaternion: $\delta\mathbf{q} = \bar{\mathbf{q}}^* \otimes \mathbf{q}$. Then, convert the vector part of $\delta\mathbf{q}$ to the error vector $\delta\phi$. A simple approximation often used is $\delta\phi = 2 \cdot \text{vector}(\delta\mathbf{q}) \cdot \text{sgn}(\text{scalar}(\delta\mathbf{q}))$ (to ensure we take the shortest path).
3. Stack these to form $\delta\tilde{\mathbf{x}}$ and multiply by $\mathbf{K}$.

15.2 3D Quadrotor Dynamics

We apply this theory to a 3D quadrotor.

15.2.1 State and Inputs

The state vector is $\mathbf{x} = [\mathbf{r}^\top, \mathbf{q}^\top, \mathbf{v}^\top, \omega^\top]^\top \in \mathbb{R}^{13}$.

- $\mathbf{r} \in \mathbb{R}^3$: Position in World frame $\mathcal{N}$.
- $\mathbf{q} \in \mathbb{R}^4$: Attitude quaternion (Rotation from $\mathcal{B}$ to $\mathcal{N}$).
- $\mathbf{v} \in \mathbb{R}^3$: Linear velocity in **Body** frame $\mathcal{B}$ (Note: some formulations use World frame; we follow the provided notes/notebook which use Body frame velocity).
- $\omega \in \mathbb{R}^3$: Angular velocity in Body frame $\mathcal{B}$.

The inputs are the motor forces (or speeds squared) $\mathbf{u} \in \mathbb{R}^4$. Typically, $F_i = k_f u_i$ and $\tau_i = k_\tau u_i$.

15.2.2 Equations of Motion

Kinematics.

$$\dot{\mathbf{r}} = \mathbf{R}(\mathbf{q})\mathbf{v} \tag{15.11}$$

$$\dot{\mathbf{q}} = \frac{1}{2}\mathbf{L}(\mathbf{q})\mathbf{H}\omega = \frac{1}{2}\mathbf{q} \otimes \begin{bmatrix} 0 \\ \omega \end{bmatrix} \tag{15.12}$$

where $\mathbf{R}(\mathbf{q})$ is the rotation matrix corresponding to $\mathbf{q}$.

Dynamics. The translational dynamics are derived from Newton's Second Law. Since $\mathbf{v}$ is in the body frame, we must account for the rotating frame (Coriolis effect). Force balance in world frame: $m\mathbf{a}_\mathcal{N} = \mathbf{F}_{\text{gravity}} + \mathbf{R}\mathbf{F}_{\text{thrust}}$. Rotating to body frame:

$$m(\dot{\mathbf{v}} + \omega \times \mathbf{v}) = \mathbf{R}^\top \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \sum F_i \end{bmatrix} \tag{15.13}$$

$$\dot{\mathbf{v}} = \mathbf{R}(\mathbf{q})^\top \mathbf{g} + \frac{1}{m}\mathbf{F}_{\text{thrust}} - \omega \times \mathbf{v} \tag{15.14}$$

The rotational dynamics follow Euler's equations:

$$\mathbf{J}\dot{\omega} + \omega \times (\mathbf{J}\omega) = \tau_{\text{total}} \tag{15.15}$$

where τ_{total} comes from the differential thrust of the motors.

Input Mapping. The mapping from motor inputs $\mathbf{u}$ to body wrench (Force F_z and Torques τ) is constant:

$$\begin{bmatrix} F_z \\ \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} = \begin{bmatrix} k_f & k_f & k_f & k_f \\ 0 & lk_f & 0 & -lk_f \\ -lk_f & 0 & lk_f & 0 \\ k_\tau & -k_\tau & k_\tau & -k_\tau \end{bmatrix} \mathbf{u}$$

(Note: The signs depend on the specific motor numbering and spin directions).

15.3 Code Analysis: `quadrotor.ipynb`

The notebook provides a concrete implementation of these concepts.

Code 15.1 (Julia Notebook: Quadrotor LQR).
The notebook demonstrates the failure of naive linearization and the success of the quaternion-error approach.

- **Naive Linearization:** Computing the Jacobian $\mathbf{A}$ of the 13-state system results in a matrix that is rank-deficient. The controllability matrix $\mathcal{C}$ is also rank-deficient. This confirms that we cannot control the system "off the manifold".
- **Reduced Linearization:** The code constructs the projection matrix $\mathbf{E}(\mathbf{q})$ (denoted as `E` in the code) using the quaternion kinematics map $\mathbf{G}(\mathbf{q})$.

  $$\tilde{\mathbf{A}} = \mathbf{E}^\top \mathbf{A} \mathbf{E}, \quad \tilde{\mathbf{B}} = \mathbf{E}^\top \mathbf{B}$$

  The resulting system matrices $\tilde{\mathbf{A}}$ (12x12) and $\tilde{\mathbf{B}}$ (12x4) form a controllable pair.
- **LQR Gain:** The function `dlqr` is called on the reduced system to compute $\mathbf{K}$.
- **Controller Implementation:** Inside the simulation loop, the controller computes the orientation error using the quaternion product $\bar{\mathbf{q}}^* \otimes \mathbf{q}$ (implemented as `L(q0)'*q`). It extracts the vector part to form the error state $\delta\tilde{\mathbf{x}}$ and applies $\mathbf{u} = \mathbf{u}_{\text{hover}} - \mathbf{K}\delta\tilde{\mathbf{x}}$.
- **Simulation:** The dynamics are integrated using RK4. Crucially, the quaternion is re-normalized ($\mathbf{q} \leftarrow \mathbf{q}/\|\mathbf{q}\|$) after every integration step to prevent numerical drift from violating the unit norm constraint.

Lecture 16: Contact Dynamics and Hybrid Systems

We address systems that interact with the world through contact. This introduces discontinuities in the dynamics (impacts), rendering standard smooth ODE solvers and optimization techniques insufficient. We will explore two primary modeling approaches: **Hybrid Systems** (Event-based) and **Time-Stepping** (Contact-Implicit), and then apply the hybrid approach to trajectory optimization for a legged robot.

16.1 Contact Dynamics

In the air, the dynamics are described by a smooth ODE (e.g., $\dot{v} = -g$). However, when the ball hits the ground at time t_I, the velocity changes instantaneously from v^- (downward) to v^+ (upward).

$$v^+ = -ev^-$$

where $e \in [0, 1]$ is the coefficient of restitution. Because of these discontinuities, we cannot write down a single smooth ODE $\dot{\mathbf{x}} = f(\mathbf{x})$ that is valid everywhere.

16.2 Simulation Approaches

There are two main paradigms for simulating such systems.

16.2.1 Event-Based / Hybrid Simulation

In this approach, we integrate the smooth dynamics until a specific event (contact) occurs. We define a **guard function** $\phi(\mathbf{x})$ (e.g. in the case of the bouncing ball it will be the height of the ball from ground, or some sort of signed distance function.) such that $\phi(\mathbf{x}) \geq 0$ implies no contact (feasible), and $\phi(\mathbf{x}) = 0$ is the contact surface.

Algorithm:

1. Integrate $\dot{\mathbf{x}} = f(\mathbf{x})$ while $\phi(\mathbf{x}) > 0$.
2. Detect when $\phi(\mathbf{x})$ crosses zero (event detection).
3. Use a root-finding algorithm to locate the precise impact time t_I.
4. Apply a **Jump Map** (or Reset Map) to update the state discontinuously:

$$\mathbf{x}^+ = \Delta(\mathbf{x}^-)$$

5. Resume integration with $\mathbf{x}^+$ as the new initial condition.

- **Pros:** Can use high-order, variable-step integrators (e.g., RK45) for high accuracy between impacts.
- **Cons:** Requires explicit enumeration of modes and mode transitions. Scaling to many contacts is difficult (combinatorial explosion of mode sequences).

16.2.2 Time-Stepping / Contact-Implicit

This method solves a constrained optimization problem at each time step to resolve forces and satisfy non-penetration constraints. Consider a brick sliding on a surface. The dynamics are:

$$m\dot{v} = -mg + \mathbf{J}(\mathbf{q})^\top \lambda$$

where λ is the contact force and $\mathbf{J}$ is the contact Jacobian. Discretizing with Backward Euler:

$$m\frac{v_{n+1}-v_n}{h} = -mg + \mathbf{J}^\top \lambda_{n+1}$$

We must satisfy the Karush-Kuhn-Tucker (KKT) conditions for non-penetration, known as **Linear Complementarity Problems (LCP)** in this context:

$$\begin{aligned} \phi(\mathbf{q}_{n+1}) &\geq 0 \quad \text{(Non-penetration)} \\ \lambda_{n+1} &\geq 0 \quad \text{(Pushing force only)} \\ \phi(\mathbf{q}_{n+1}) \cdot \lambda_{n+1} &= 0 \quad \text{(Complementarity: Force is zero if not touching)} \end{aligned}$$

This can be formulated as an optimization problem (QP):

$$\min_{v_{n+1}} \frac{1}{2} v_{n+1}^\top \mathbf{M} v_{n+1} + \dots \quad \text{s.t.} \quad \phi(\mathbf{q}_n + h v_{n+1}) \geq 0$$

- **Pros:** Robust; handles simultaneous contacts and changing contact modes naturally without explicit logic. Widely used in simulators like PyBullet, Gazebo.
- **Cons:** Usually limited to first-order integrators (Euler), so requires small time steps for accuracy. Impact times are not resolved precisely.

16.3 Code Analysis: `hybrid-ball.ipynb`

The notebook provides a simple implementation of the Event-Based method.

Code 16.1 (Julia Notebook: Hybrid Bouncing Ball).
The code simulates a ball falling under gravity with a ground at $y = 0$.

- **Dynamics:** $\dot{\mathbf{x}} = [v, -g]^\top$. Implemented as `dynamics_rk4`.
- **Guard Function:** $\phi(\mathbf{x}) = y$ (vertical position).
- **Jump Map:** Implemented in `jump(x)`. It reflects the vertical velocity:

  $$v_y^+ = -\gamma v_y^-$$

  where $\gamma = 0.9$ is the coefficient of restitution.
- **Simulation Loop:** It iterates through time steps. After each step, it checks 'if guard(x) <= 0'. If true, it applies the jump map immediately. *Note: The provided implementation is a simplified hybrid method; it applies the jump at the next discrete step rather than finding the precise root time, which is acceptable for small time steps.*

16.4 Hybrid Trajectory Optimization for Legged Robots

For control, the Hybrid formulation is often easier to implement within standard trajectory optimization frameworks (like DIRCOL) if we **pre-specify the mode sequence**.

Consider a one-legged hopper with state $\mathbf{x} = [r_b, r_f, v_b, v_f]^\top$ (body and foot positions/velocities) and input $\mathbf{u} = [F, \tau]^\top$.

Modes:

1. **Flight:** Foot is in the air. Dynamics are ballistic.
2. **Stance:** Foot is on the ground. Dynamics involve ground reaction forces.

Jump Map (Touchdown): When transitioning from Flight to Stance, an inelastic collision occurs. The foot velocity is instantaneously zeroed out:

$$\mathbf{x}^+ = \Delta(\mathbf{x}^-) = \begin{bmatrix} r_b \\ r_f \\ v_b \\ 0 \end{bmatrix}$$

Trajectory Optimization Formulation: We assume a fixed sequence of modes (e.g., Flight $\rightarrow$ Stance $\rightarrow$ Flight). We partition the knot points $1 \dots N$ into phases.

- **Phase 1 (Flight):** For $k = 1 \dots N_1 - 1$, enforce flight dynamics $\mathbf{x}_{k+1} = f_{\text{flight}}(\mathbf{x}_k, \mathbf{u}_k)$.
- **Transition:** At $k = N_1$, enforce the jump map $\mathbf{x}_{N_1+1} = \Delta(f_{\text{flight}}(\mathbf{x}_{N_1}, \mathbf{u}_{N_1}))$.
- **Phase 2 (Stance):** For $k = N_1+1 \dots N_2$, enforce stance dynamics $\mathbf{x}_{k+1} = f_{\text{stance}}(\mathbf{x}_k, \mathbf{u}_k)$.

We also add phase-specific constraints:

- During Flight: $\phi(\mathbf{x}_k) > 0$ (Foot above ground).
- During Stance: $\phi(\mathbf{x}_k) = 0$ (Foot on ground).

Code 16.2 (Julia Notebook: `hopper.ipynb`).
The notebook implements Multi-Mode Trajectory Optimization for a 2D hopper.

- **Setup:** Uses `Ipopt` to solve the NLP. The decision variables `z` include states and controls for all time steps.
- **Dynamics Functions:** Separate functions `flight_dynamics` and `stance_dynamics` are defined.
- **Constraints:**
  - `dynamics_constraint!`: This function explicitly loops through the pre-defined modes. If the current segment is Flight, it applies flight dynamics. If Stance, it applies stance dynamics. At the interface indices (multiples of `Nm`), it applies the `jump_map`.
  - `stance_constraint!`: Enforces $y_{\text{foot}} = 0$ during stance phases.
  - `length_constraint!`: Enforces kinematic limits on the leg length.
- **Result:** The optimizer finds a trajectory where the robot jumps, makes contact (handling the impact discontinuity via the jump map constraint), and jumps again, tracking the reference while respecting physics.

Intuition. By fixing the mode sequence, we convert a hard "decision" problem (when to make contact?) into a standard smooth NLP (where to place the foot given we *must* make contact at step K). This is why Hybrid Traj Opt is powerful for gaits like walking or running where the sequence is known (Left-Right-Left...), but fails for complex manipulation where the contact sequence is the solution itself.

Lecture 17: Iterative Learning Control (ILC)

We address a fundamental reality in robotics: **models are never perfect**. Even with the best physics engines and parameter identification, there will always be a "sim-to-real" gap due to unmodeled friction, aerodynamic effects, flexibility, or sensor noise. We will explore strategies to handle these errors, culminating in **Iterative Learning Control (ILC)**, a powerful technique for improving performance over repeated executions of a task.

17.1 Strategies for Model Mismatch

When our nominal model $f_{\text{nom}}(\mathbf{x}, \mathbf{u})$ differs from the true system dynamics $f_{\text{true}}(\mathbf{x}, \mathbf{u})$, the optimal policy computed for the model will be suboptimal or even unstable on the real system. Feedback control (like LQR or MPC) provides some robustness, but it reacts to errors rather than anticipating them. If we need high performance (e.g., tracking a trajectory very aggressively), feedback gains alone may not be sufficient.

There are several standard approaches to close this gap:

1. **Parameter Estimation (System ID):** Assumes a "gray-box" model structure (e.g., a rigid body chain) and fits physical parameters (masses, lengths, friction coefficients) from data.
   - **Pros:** Interpretable, generalizes well to new tasks.
   - **Cons:** Limited by the assumed model structure. If the physics model doesn't include a specific phenomenon (e.g., stiction), fitting parameters won't fix it.
2. **Learn a Model (Black-box / Residual):** Fits a generic function approximator (like a Neural Network or Gaussian Process) to the dynamics $\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k)$ or the error residual $\mathbf{e}_{k+1} = \mathbf{x}_{k+1}^{\text{true}} - f_{\text{nom}}(\mathbf{x}_k, \mathbf{u}_k)$.
   - **Pros:** Flexible, can capture complex nonlinear effects.
   - **Cons:** Data-hungry, often fails to generalize outside the training distribution, "black-box" nature makes stability analysis hard.
3. **Learn a Policy (Model-Free RL):** Optimizes the policy $\pi(\mathbf{x})$ directly based on rewards, ignoring the dynamics model entirely.
   - **Pros:** Requires very few assumptions.
   - **Cons:** Extremely sample inefficient (requires millions of trials), typically does not generalize to new tasks.
4. **Transfer / Iterative Learning Control (ILC):** Assumes we have a decent nominal model and a specific reference trajectory we want to track. We use data from the real system to iteratively refine the **feedforward control** for that specific task.
   - **Pros:** Very sample efficient (often converges in $<$10 iterations), high precision.
   - **Cons:** Task-specific (doesn't generalize to new trajectories).

17.2 Iterative Learning Control (ILC)

ILC is based on the idea that if we perform the same task repeatedly, the errors we see are likely consistent (systematic). Instead of just reacting to them with feedback, we should "learn" from the previous iteration's error to adjust our plan for the next iteration.

We can view ILC as a specialized form of policy optimization. Consider a tracking controller of the form:

$$\mathbf{u}_k(\mathbf{x}) = \bar{\mathbf{u}}_k - \mathbf{K}_k(\mathbf{x} - \bar{\mathbf{x}}_k)$$

Standard LQR/MPC keeps the feedforward term $\bar{\mathbf{u}}_k$ constant (from the nominal plan). ILC updates $\bar{\mathbf{u}}_k$ based on the error observed in the previous rollout.

17.2.1 Derivation via Sequential Quadratic Programming (SQP)

We can rigorously derive the ILC update by viewing the trajectory optimization problem as a constrained nonlinear program (NLP):

$$\begin{aligned} \min_{\mathbf{z}} \quad & J(\mathbf{z}) = \sum_{k=1}^{N-1} \frac{1}{2}\|\mathbf{x}_k - \bar{\mathbf{x}}_k\|^2_{\mathbf{Q}} + \frac{1}{2}\|\mathbf{u}_k - \bar{\mathbf{u}}_k\|^2_{\mathbf{R}} + \frac{1}{2}\|\mathbf{x}_N - \bar{\mathbf{x}}_N\|^2_{\mathbf{Q}_N} \\ \text{subject to} \quad & \mathbf{c}(\mathbf{z}) = 0 \quad (\text{Dynamics: } \mathbf{x}_{k+1} - f(\mathbf{x}_k, \mathbf{u}_k) = 0) \end{aligned}$$

where $\mathbf{z} = [\mathbf{x}_1, \mathbf{u}_1, \ldots, \mathbf{x}_N]^\top$ is the full decision vector.

To solve this using SQP (which is equivalent to Newton's method on the KKT conditions), we look for a step $\delta\mathbf{z}$ by solving the following linear system:

$$\begin{bmatrix} \mathbf{H} & \mathbf{C}^\top \\ \mathbf{C} & 0 \end{bmatrix} \begin{bmatrix} \delta\mathbf{z} \\ \delta\lambda \end{bmatrix} = \begin{bmatrix} -\nabla J(\mathbf{z}) \\ -\mathbf{c}(\mathbf{z}) \end{bmatrix} \tag{17.1}$$

where:

- $\mathbf{H} \approx \nabla^2_{zz}\mathcal{L}$ is the Hessian of the Lagrangian (often approximated by the Gauss-Newton Hessian of the cost J).
- $\mathbf{C} = \frac{\partial \mathbf{c}}{\partial \mathbf{z}}$ is the Jacobian of the constraints (dynamics).
- $\nabla J(\mathbf{z})$ is the gradient of the cost.
- $\mathbf{c}(\mathbf{z})$ is the constraint violation residual.

The ILC Strategy: In ILC, we exploit the difference between the **nominal model** and the **real system** to populate the terms in Equation 17.1:

1. **RHS (Gradient/Residual):** We perform a rollout on the *real* system.
   - Since the rollout happens on the physical system, it is dynamically feasible with respect to the true dynamics, so $\mathbf{c}_{\text{true}}(\mathbf{z}) = 0$.
   - We compute the gradient $\nabla J(\mathbf{z})$ using the trajectory data $(\mathbf{X}, \mathbf{U})$ collected from the real system. This captures the true tracking error.
2. **LHS (Curvature/Jacobian):** We cannot compute the true system Hessians $\mathbf{H}_{\text{true}}$ or Jacobians $\mathbf{C}_{\text{true}}$.
   - Instead, we approximate them using the *nominal model*:

     $$\mathbf{H} \approx \mathbf{H}_{\text{nom}}, \quad \mathbf{C} \approx \mathbf{C}_{\text{nom}}$$

   - Since our nominal model is a reasonable approximation, and assuming the current trajectory is close to the reference, these matrices (which can be computed offline) are sufficient to determine a good search direction.

Solving this KKT system for $\delta\mathbf{z}$ (specifically the control part $\delta\mathbf{u}$) yields the feedforward update $\mathbf{u}_{\text{new}} = \mathbf{u}_{\text{old}} + \delta\mathbf{u}$. This explains why ILC works: it uses the **true gradient** to drive the optimization, conditioned by the **model's curvature**. In the context of robotics with model mismatch:

- **The Gradient ∇J:** Depends on the *true* system behavior. If we run a rollout on the real robot, we can compute the cost and its sensitivity to the controls around the actual trajectory. This gives us the "true" gradient direction (or a very good approximation of it).
- **The Hessian $\nabla^2 J$:** Computing the true Hessian on a physical system is impossible. However, we have a *nominal model*. We can approximate the curvature of the problem using our model: $\mathbf{H}_{\text{nom}} \approx \nabla^2 J_{\text{true}}$.

Convergence theory tells us that as long as the nominal model is "close enough" (specifically, if $\|\mathbf{I} - \mathbf{H}_{\text{nom}}^{-1}\mathbf{H}_{\text{true}}\| < 1$), this iteration will converge to the local optimum of the *true* system.

17.3 ILC as a Quadratic Program (QP)

We can implement this update step by solving a Linear-Quadratic problem (similar to the subproblem in SQP or iLQR), but mixing real data with model derivatives.

Algorithm 9: Model-Based ILC Algorithm

Input: Nominal model f_{nom}, Initial guess $\mathbf{U}^{(0)}$, Reference $\mathbf{X}^{\text{ref}}$

for *iteration* $i = 0, 1, \ldots$ **do**

`// 1. Rollout on Real System`

Run $\mathbf{U}^{(i)}$ on the robot (possibly with feedback stabilization).

Record actual trajectory $\mathbf{X}^{(i)}$.

Compute tracking error $\mathbf{e}_k = \mathbf{x}_k^{(i)} - \mathbf{x}_k^{\text{ref}}$.

`// 2. Linearize Nominal Model`

Compute $\mathbf{A}_k, \mathbf{B}_k$ from f_{nom} around $\mathbf{x}_k^{(i)}, \mathbf{u}_k^{(i)}$.

`// 3. Formulate Optimization Step`

We want to find corrections $\delta\mathbf{x}, \delta\mathbf{u}$ that reduce the error. We minimize the approximate cost of the *next* iteration:

$$\min_{\delta\mathbf{X}, \delta\mathbf{U}} \sum \frac{1}{2}\|(\mathbf{x}_k^{(i)} + \delta\mathbf{x}_k) - \mathbf{x}_k^{\text{ref}}\|_{\mathbf{Q}}^2 + \frac{1}{2}\|\delta\mathbf{u}_k\|_{\mathbf{R}}^2$$

Expanding the quadratic term $\|\mathbf{x} + \delta\mathbf{x} - \mathbf{x}^{\text{ref}}\|_{\mathbf{Q}}^2$ gives a linear term $\delta\mathbf{x}^\top\mathbf{Q}(\mathbf{x}^{(i)} - \mathbf{x}^{\text{ref}})$. The QP becomes:

$$\begin{aligned} \min_{\delta\mathbf{X}, \delta\mathbf{U}} \quad & \sum_{k=1}^{N-1} \left(\frac{1}{2}\delta\mathbf{x}_k^\top\mathbf{Q}\delta\mathbf{x}_k + \mathbf{q}_k^\top\delta\mathbf{x}_k + \frac{1}{2}\delta\mathbf{u}_k^\top\mathbf{R}\delta\mathbf{u}_k \right) \\ \text{subject to} \quad & \delta\mathbf{x}_{k+1} = \mathbf{A}_k\delta\mathbf{x}_k + \mathbf{B}_k\delta\mathbf{u}_k \\ & \mathbf{u}_{\min} \le \mathbf{u}_k^{(i)} + \delta\mathbf{u}_k \le \mathbf{u}_{\max} \end{aligned}$$

where $\mathbf{q}_k = \mathbf{Q}(\mathbf{x}_k^{(i)} - \mathbf{x}_k^{\text{ref}})$ is the gradient driving the update based on real errors.

`// 4. Update`

Solve QP for $\delta\mathbf{U}$.

$\mathbf{U}^{(i+1)} \leftarrow \mathbf{U}^{(i)} + \delta\mathbf{U}$.

Intuition. The QP asks: "According to my *nominal* model $(\mathbf{A}, \mathbf{B})$, how should I change my inputs $\delta\mathbf{u}$ to eliminate the tracking error $\mathbf{x}^{(i)} - \mathbf{x}^{\text{ref}}$ that I observed on the real system?"

17.4 Code Analysis: `cartpole-ilc.ipynb`

The notebook simulates this process on a Cartpole system with parameter mismatch.

Code 17.1 (Julia Notebook: ILC for Cartpole Swing-up).
Problem Setup:

- **Nominal Model:** Standard Cartpole parameters.
- **True Model:** Perturbed masses ($m_c + 0.02$, $m_p - 0.01$), length ($l + 0.005$), and crucially, **nonlinear friction** (tanh damping) which is completely absent in the nominal model.

Step 1: Nominal Plan (ALTRO)

- Solves the swing-up problem using `Altro.jl` on the nominal model.
- Produces a reference trajectory $\mathbf{X}^{\text{ref}}, \mathbf{U}^{\text{nom}}$.

Step 2: Baseline Performance

- Runs the nominal feedforward $\mathbf{U}^{\text{nom}}$ (plus LQR feedback) on the **True Model**.
- Result: The cartpole fails to reach the top or drift significantly due to the unmodeled friction and mass errors.

Step 3: ILC Update

- **Linearization:** Computes $\mathbf{A}_k, \mathbf{B}_k$ along the *nominal* trajectory using the *nominal* model.
- **QP Formulation:** Uses `OSQP` to solve for $\delta\mathbf{u}$.
- **Cost Vector q:** The linear cost term in the QP is set to `Qilc * (xtraj - Xopt)`. This vector encodes the error between the rollout on the **true** physics (`xtraj`) and the **desired** plan (`Xopt`).
- **Constraints:** Enforces nominal linearized dynamics and torque limits.
- **Result:** The computed $\delta\mathbf{u}$ effectively learns an inverse dynamics term to cancel out the unmodeled friction and gravity errors. After the update, the trajectory tracking on the true system improves dramatically.