Embodied-AI:
Perception, Representation and Action

- **DATA 8010 & ELEC 8111**
- **Instructor: Prof. Yanchao Yang**

Kinematics, Dynamics, and Simulation Environments

The Skeptic's Question

> *"If VLAs and diffusion policies can learn end-to-end, why should I spend time on kinematics and dynamics?"*

Modern vision-language-action models map directly from images and language to robot actions. They appear to bypass classical robotics entirely.

This is a legitimate question. Let's examine what these systems are actually doing—and why understanding the underlying physics still matters.

What Networks Implicitly Learn

When a policy network predicts actions from observations, it must internally represent these physical relationships:

Forward Kinematics

$\theta \rightarrow T(\theta)$

Joint angles determine end-effector pose. The network learns this mapping implicitly.

Jacobian

$J(\theta)$

How joint velocities map to task-space velocities. Varies with configuration.

Dynamics

$\tau = M \cdot a + C + g$

Forces required to produce desired accelerations, including inertia and gravity.

Key insight: End-to-end learning doesn't eliminate physics—it uses data and parameters to approximate relationships that have known analytical forms. Understanding these forms helps you design better systems.

The Debugging Advantage

Without Physical Intuition

Observation: Policy fails in certain arm configurations.

Response: Add more training data. Tune hyperparameters. Try different architectures.

Process: Trial and error until it works (or doesn't).

With Physical Intuition

Observation: Policy fails in certain arm configurations.

Diagnosis: Failures occur near singularities where Jacobian loses rank—small task-space errors require large joint motions.

Solution: Modify action space, add singularity-aware training data, or use nullspace motion.

Knowledge transforms you from a "*hyperparameter tuner*" into an engineer who diagnoses and solves problems systematically.

Design Decisions You Will Face

In every robot learning project, you will encounter these questions. Answering them well requires the concepts we cover today:

Action Space Design

Should the policy output joint positions, joint velocities, joint torques, or Cartesian poses? How does this choice affect learning and sim-to-real transfer?

Observation Space Design

What proprioceptive information should the policy receive? Joint angles only? End-effector pose? Force/torque feedback?

Rotation Representation

Quaternions, Euler angles, rotation matrices, or 6D representation? Each has different continuity properties affecting network training.

Simulation Configuration

What parameters should be randomized for domain randomization? What is the source of the reality gap for your task?

These are not abstract questions—they directly impact whether your system works in the real world.

Today's Roadmap

1. **State and Representation**
   Configuration space, task space, rotation representations for neural networks

2. **Kinematics Core**
   Forward kinematics concept, Jacobian intuition, singularities and their impact

3. **Dynamics and Force Control**
   Dynamics equation intuition, impedance control basics, compliance for contact tasks

4. **Action Space Design**
   Position vs. velocity vs. torque control, Cartesian vs. joint space, design trade-offs

5. **Simulation and Sim-to-Real**
   How simulators work, sources of reality gap, domain randomization principles

State and Representation

How do we mathematically describe a robot's configuration?

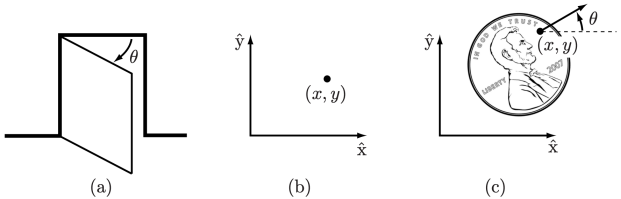
The Two Spaces

Configuration Space (C-space)

Definition: The n-dimensional space of all possible joint configurations.

$$\mathbf{q} = (\theta_1, \theta_2, \ldots, \theta_n)^\top \in \mathbb{R}^n$$

For n-DOF robot: dim(Q) = n. Directly measured by joint encoders.



Task Space (Operational Space)

Definition: The space of end-effector poses (position + orientation).

$$\mathbf{x} = (\mathbf{p}, \mathbf{R}) \in SE(3), \quad \dim = 6$$
$$\mathbf{x} = \begin{pmatrix} \mathbf{p} \\ \boldsymbol{\phi} \end{pmatrix} \in \mathbb{R}^3 \times SO(3)$$

For rigid body in 3D: dim(X) = 6. Where tasks are naturally specified.

The Fundamental Mapping

$$\text{FK} : \mathcal{Q} \to SE(3), \quad T_{ee} = \text{FK}(\mathbf{q})$$
$$T_{0n}(\mathbf{q}) = T_{01}(\theta_1) \cdot T_{12}(\theta_2) \cdots T_{(n-1)n}(\theta_n) \in SE(3)$$

Forward kinematics maps joint angles to end-effector pose. This is the bridge between the two spaces.

Design implication: Policies can operate in either space. Joint-space policies output q directly; task-space policies output x and require inverse kinematics. This choice profoundly affects learning difficulty.

Configuration Space — Formal Definition

Definition (Configuration Space)

The **configuration** of a robot is a complete specification of the position of every point of the robot. The minimum number n of real-valued coordinates needed to represent the configuration is the **degrees of freedom (DOF)**. The n-dimensional space containing all possible configurations is the **configuration space** (C-space).

Joint Types and Topology

$$\mathbf{q} \in \mathcal{Q} = \prod_{i=1}^{n} [\theta_{i,\min}, \theta_{i,\max}] \subset \mathbb{R}^n$$

Revolute: angle wraps (S^1 topology). Prismatic: bounded interval in $\mathbb{R}$.

Joint Limits

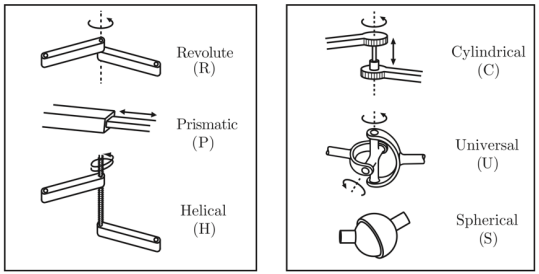
$$\text{DOF} = m(N-1) - \sum_{i=1}^{J} c_i$$

Physical constraints bound each joint variable, where m = 6 (spatial) or 3 (planar), N = number of links including ground, J = number of joints, c_i = constraints imposed by joint i.

Example: n-DOF Robot Arm

$$\mathbf{q} = (\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, \theta_7)^\top \in \mathbb{R}^7$$

7-DOF Franka: $Q \subset \mathbb{R}^7$, each θ_i bounded by mechanical limits.



For learning: Joint encoders directly measure q — this is your most reliable proprioceptive signal. Policy inputs often include (q, $\dot{q}$) as the core state representation.

Task Space — Where Tasks Live

Definition (Task Space)

The **task space** (or operational space) is the space in which the robot's task is naturally expressed. For manipulation, this is typically the space of end-effector poses: position in $\mathbb{R}^3$ and orientation in SO(3).

End-Effector Pose

$$T = \begin{bmatrix} R & \mathbf{p} \\ \mathbf{0}^\top & 1 \end{bmatrix} \in SE(3), \quad \dim = 6$$

Position p ∈ $\mathbb{R}^3$ plus rotation R ∈ SO(3). Total: 6 DOF.

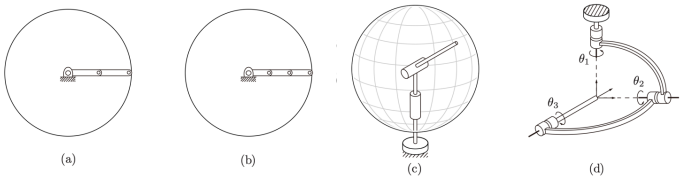
Workspace vs Task Space

Workspace: Set of reachable poses (depends on robot). **Task space:** Where tasks are defined (independent of robot).

The Dimension Gap → Redundancy

$$r = n - m$$

7-DOF arm in 6D task space: 7 - 6 = 1 degree of redundancy. Infinite joint configs can achieve the same EE pose.
(where n = dim(C-space), m = dim(Task-space))



For learning: Task-space actions (Δp, ΔR) are often more intuitive for imitation learning from human demos. But they require solving IK at runtime, which introduces its own challenges (singularities, multiple solutions).

Redundancy and Velocity Kinematics

Definition (Kinematic Redundancy)

A robot is kinematically redundant if dim(C-space) > dim(Task-space). Infinitely many joint configurations can achieve the same end-effector pose.

Velocity Kinematics

Joint velocities map to end-effector velocity via the Jacobian:

$$\dot{\mathbf{x}} = J(\mathbf{q})\,\dot{\mathbf{q}}$$

J(q) is configuration-dependent; its structure determines motion capabilities.

Symbol Definitions

$\dot{\mathbf{q}} \in \mathbb{R}^n$ (joint velocities)

$\dot{\mathbf{x}} \in \mathbb{R}^m$ (end-effector velocity / twist)

$J(\mathbf{q}) \in \mathbb{R}^{m \times n}$ (Jacobian matrix)

Key insight: For 7-DOF arm (n=7) in 6D task space (m=6), J is 6×7. Since n > m, the system is underdetermined: multiple joint velocities produce the same end-effector velocity.

Null Space and Self-Motion

Definition (Null Space)

The set of joint velocities producing zero EE velocity:

$$\mathcal{N}(J) = \{\dot{\mathbf{q}} \in \mathbb{R}^n \mid J(\mathbf{q})\,\dot{\mathbf{q}} = \mathbf{0}\}$$

For redundant robots: dim(N(J)) = n − m > 0

Self-Motion Manifold

All configurations achieving the same pose:

$$\mathcal{M}(T^*) = \{\mathbf{q} \in \mathcal{Q} \mid \mathrm{FK}(\mathbf{q}) = T^*\}$$
$$\dim(\mathcal{M}) = n - m$$

Implications for Learning

Challenge: IK has ∞ solutions; policy must handle ambiguity
Opportunity: Use null space for obstacle avoidance
Design: Policy controls null space, or delegate?

Human analogy: Place your hand flat on a table. Your hand (end-effector) is fixed, but your elbow can still move in an arc. That elbow motion is null-space self-motion.

Why Representation Matters

Core Insight

Neural networks are function approximators. The difficulty of learning a function depends critically on its smoothness. A good representation makes the target function smooth; a bad representation introduces discontinuities and ambiguities.

Good Representation

Small changes in state → small changes in representation

Unique mapping: each state has one representation

Bad Representation

Small changes in state → large jumps in representation (discontinuities)

Ambiguous: same state has multiple representations

Practical impact: Zhou et al. (CVPR 2019) showed that rotation representation alone can change pose estimation error by 25%+ with the same network architecture.

The Challenge of Representing Orientation

Position: Simple

$$\mathbf{p} = (x, y, z) \in \mathbb{R}^3$$

Euclidean space. Continuous, unique, no singularities. Networks handle this perfectly.

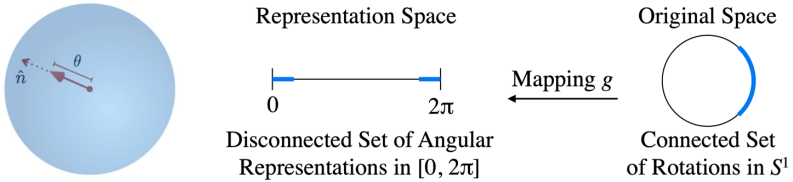
Orientation: Complicated

$$R \in SO(3) = \{R \in \mathbb{R}^{3\times3} \mid R^\top R = I,\ \det(R) = +1\}$$

Non-Euclidean manifold. No global coordinates without singularities or discontinuities.

Topological Fact (Zhou et al., 2019)

Any representation of SO(3) with fewer than 5 dimensions must have discontinuities. There is no way to continuously parameterize all 3D rotations with 3 or 4 numbers.



Implication: We must choose a representation. Each has tradeoffs. The next slides examine: Euler angles, quaternions, rotation matrices, and 6D representations.

Euler Angles

Definition

Three successive rotations about specified axes. Common conventions: ZYX, ZYZ, XYZ (roll-pitch-yaw).

$$R(\alpha,\beta,\gamma) = R_z(\alpha)\,R_y(\beta)\,R_x(\gamma)$$

$$R(\alpha,\beta,\gamma) = \begin{bmatrix} c_\alpha c_\beta & c_\alpha s_\beta s_\gamma - s_\alpha c_\gamma & c_\alpha s_\beta c_\gamma + s_\alpha s_\gamma \\ s_\alpha c_\beta & s_\alpha s_\beta s_\gamma + c_\alpha c_\gamma & s_\alpha s_\beta c_\gamma - c_\alpha s_\gamma \\ -s_\beta & c_\beta s_\gamma & c_\beta c_\gamma \end{bmatrix}$$

$$c_\alpha = \cos\alpha, \quad s_\alpha = \sin\alpha, \quad \text{etc.}$$

Advantages

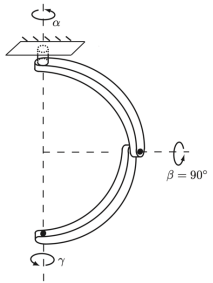
Compact (3 parameters). Intuitive for humans. Common in aerospace, graphics.

$$\text{At } \beta = \tfrac{\pi}{2}: \quad R_z(\alpha)\,R_y\!\left(\tfrac{\pi}{2}\right)R_x(\gamma) = R_z(\alpha+\gamma)\,R_y\!\left(\tfrac{\pi}{2}\right)$$

Problems for Learning

Gimbal lock: At β = ±90°, one DOF is lost. Jacobian becomes singular.

Discontinuities: Angles wrap around (e.g., 359° → 0°). Small rotation causes large numerical jump.



Verdict: Not recommended for neural network inputs/outputs due to discontinuities.

Quaternions

Definition

A unit quaternion is a 4D vector on the unit sphere S^3, encoding axis-angle rotation.

$$\mathbf{q} = \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} \in S^3 \subset \mathbb{R}^4, \quad \|\mathbf{q}\|^2 = w^2 + x^2 + y^2 + z^2 = 1$$

Advantages

Compact (4 parameters). No gimbal lock. Efficient composition and interpolation (SLERP). Standard in game engines, physics simulators.

Problems for Learning

Double cover: q and −q represent the same rotation. Network may oscillate between them.

Unit constraint: ||q|| = 1 is hard to enforce. Normalization has gradient issues at q ≈ 0.

Verdict: Better than Euler angles, but use with care. Consider mapping to consistent hemisphere.

Quaternion to Rotation

$$R(\mathbf{q}) = \begin{bmatrix} 1-2(y^2+z^2) & 2(xy-wz) & 2(xz+wy) \\ 2(xy+wz) & 1-2(x^2+z^2) & 2(yz-wx) \\ 2(xz-wy) & 2(yz+wx) & 1-2(x^2+y^2) \end{bmatrix}$$

Rotation by angle θ about unit axis $\hat{\omega}$ gives: w = cos(θ/2), (x,y,z) = $\hat{\omega}$ sin(θ/2)

$$\mathbf{q} = \begin{pmatrix} \cos(\theta/2) \\ \hat{\boldsymbol{\omega}} \sin(\theta/2) \end{pmatrix} = \begin{pmatrix} \cos(\theta/2) \\ \omega_x \sin(\theta/2) \\ \omega_y \sin(\theta/2) \\ \omega_z \sin(\theta/2) \end{pmatrix}$$

Axis-Angle Representation

Definition

Any rotation can be described as a rotation by angle θ about a unit axis ω̂. The exponential coordinates combine these into a single 3-vector.

$$\boldsymbol{\omega} = \hat{\boldsymbol{\omega}}\theta \in \mathbb{R}^3, \quad \text{where } \hat{\boldsymbol{\omega}} \in S^2,\ \theta \in [0, \pi]$$
$$R = \exp([\boldsymbol{\omega}]_\times) = I + \frac{\sin\theta}{\theta}[\boldsymbol{\omega}]_\times + \frac{1-\cos\theta}{\theta^2}[\boldsymbol{\omega}]_\times^2$$

$$[\boldsymbol{\omega}]_\times = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}$$

Advantages

Compact (3 parameters). Intuitive geometric meaning.

Natural for angular velocities: ω = ω̂θ directly gives rotation rate.

Problems for Learning

Singularity at θ = 0: Axis ω̂ is undefined for identity rotation.

Discontinuity at θ = π: ω̂ and −ω̂ give same rotation when θ = π.

Verdict: Use for angular velocities and Lie algebra computations. Not recommended for network orientation outputs.

Connection to Lie Theory

Axis-angle is the **exponential coordinates** for SO(3). The vector ω lives in the Lie algebra so(3).

exp: so(3) → SO(3) maps axis-angle to rotation matrix

log: SO(3) → so(3) maps rotation matrix to axis-angle

Common Usage

Angular velocity vectors, twist representations, some policy outputs (e.g., delta rotations in some RT-X variants)

Rotation Matrices

Definition

A 3×3 orthogonal matrix with determinant +1. The columns form an orthonormal basis.

$$SO(3) = \left\{ R \in \mathbb{R}^{3\times 3} \;\middle|\; R^\top R = I,\ \det(R) = +1 \right\}$$

Advantages

No singularities or discontinuities. Unique representation (no double cover). Direct geometric meaning: columns are transformed basis vectors.

Problems for Learning

Over-parameterized: 9 numbers for 3 DOF. Six constraints: $R^TR = I$, $\det(R) = 1$.

Constraint enforcement: Network output may not satisfy constraints. Projection via SVD adds complexity.

Orthogonality Constraints

$$R^\top R = I \implies \underbrace{\begin{cases} \|\mathbf{r}_1\|^2 = 1 \\ \|\mathbf{r}_2\|^2 = 1 \\ \|\mathbf{r}_3\|^2 = 1 \end{cases}}_{\text{(unit length)}} \quad \underbrace{\begin{cases} \mathbf{r}_1 \cdot \mathbf{r}_2 = 0 \\ \mathbf{r}_2 \cdot \mathbf{r}_3 = 0 \\ \mathbf{r}_1 \cdot \mathbf{r}_3 = 0 \end{cases}}_{\text{(orthogonality)}}$$

Three unit-length constraints + three orthogonality constraints = 9 − 6 = 3 DOF

Verdict: Clean mathematically, but constraint enforcement is awkward for networks.

The 6D Rotation Representation

Key Paper: Zhou et al., CVPR 2019

"On the Continuity of Rotation Representations in Neural Networks" — Proves that any representation with < 5 dimensions must have discontinuities. Proposes a continuous 6D representation.

Method

Output first two columns of rotation matrix (6 numbers). Recover third via cross product, then orthonormalize. [6D representation and Gram-Schmidt recovery]

$$\mathbf{a}_1, \mathbf{a}_2 \in \mathbb{R}^3 \quad (6 \text{ numbers total})$$

$$\mathbf{b}_1 = \frac{\mathbf{a}_1}{\|\mathbf{a}_1\|}, \mathbf{b}_2 = \frac{\mathbf{a}_2 - (\mathbf{b}_1 \cdot \mathbf{a}_2)\,\mathbf{b}_1}{\|\mathbf{a}_2 - (\mathbf{b}_1 \cdot \mathbf{a}_2)\,\mathbf{b}_1\|}, \mathbf{b}_3 = \mathbf{b}_1 \times \mathbf{b}_2$$

$$R = \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \mathbf{b}_3 \end{bmatrix} \in SO(3)$$

Continuous

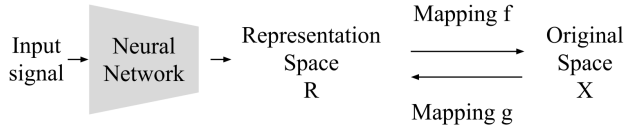
No discontinuities anywhere in SO(3)

Unique

No double-cover ambiguity

Simple

Easy to implement



Recommendation

Use 6D representation by default for neural network orientation inputs/outputs. This is current best practice.

Implementing 6D Rotation

Converting between 6D representation and rotation matrices is straightforward. Here's the PyTorch implementation.

6D → Rotation Matrix (Gram-Schmidt)

```python
def rotation_6d_to_matrix(r6d):
    """Convert 6D representation to 3x3 rotation matrix."""
    a1 = r6d[..., :3]
    a2 = r6d[..., 3:6]

    # Gram-Schmidt orthonormalization
    b1 = F.normalize(a1, dim=-1)
    dot = (b1 * a2).sum(-1, keepdim=True)
    b2 = F.normalize(a2 - dot * b1, dim=-1)
    b3 = torch.cross(b1, b2, dim=-1)

    return torch.stack([b1, b2, b3], dim=-2)

def matrix_to_rotation_6d(R):
    """Convert 3x3 rotation matrix to 6D."""
    return R[..., :2, :].flatten(start_dim=-2)
```

Rotation Matrix → 6D (for labels)

easy

Loss Functions

L2 on 6D: Simple, works well in practice

Frobenius on R: $||R_{pred} - R_{target}||_F$

Geodesic: $\arccos((\mathrm{tr}(R^T\hat{R}) - 1) / 2)$

Key Properties

Differentiable: Gram-Schmidt has well-defined gradients

Batch-friendly: Works with any batch dimensions

No constraints: Any 6 numbers → valid rotation

Rotation Representation Comparison

Representation	Params	Singularities	Continuous	Unique	Constraints	Verdict
Euler angles	3	✗ Yes	✗ No	✓ Yes	None	**Avoid**
Axis-angle	3	✗ Yes	✗ No	~ Near π	$\|\omega\|=1$	**Velocities**
Quaternion	4	✓ No	✗ No	✗ No	$\|q\|=1$	**Caution**
Rotation matrix	9	✓ No	✓ Yes	✓ Yes	R^TR=I	**Constrained**
6D (Zhou)	**6**	✓ No	✓ Yes	✓ Yes	**None!**	**★ Best**

For Neural Networks, We Need:

✓ No singularities (stable gradients) · ✓ Continuous (smooth optimization landscape) · ✓ Unique (unambiguous learning target) · ✓ No constraints (unconstrained output layer)

Only 6D satisfies all criteria. Zhou et al. (2019) proved that representations below 5D must have discontinuities; 6D is the practical minimum that also provides uniqueness and requires no constraints

Practical Guide: When to Use What

Use 6D Representation

Network I/O: Policy inputs/outputs, learned embeddings

Training data: Orientation labels for imitation learning

Loss computation: When comparing orientations during training

Default choice for anything touching neural networks

Use Quaternions

Interpolation: SLERP for smooth trajectories

Library APIs: MuJoCo, PyBullet, Isaac expect quaternions

Composition: Efficient rotation chaining (Hamilton product)

Convert at boundaries: 6D ↔ R ↔ quaternion

Use Axis-Angle

Angular velocity: ω represents rotation rate directly

Delta rotations: Small incremental rotations

Lie algebra: Optimization on SO(3), twist representations

Natural for velocities, not for absolute orientations

Typical Conversion Flow in a Policy Pipeline

6D (network output) → Rotation Matrix R → Quaternion (for simulator)

Key Principle: Use 6D at the learning boundary (network inputs/outputs), convert to whatever format downstream components need. The rotation matrix R serves as the universal intermediate—all representations convert to/from it easily.

SE(3) — The Full Pose

Now that we understand rotation representations, we combine rotation with translation to describe the **full pose** of a rigid body.

Special Euclidean Group SE(3)

SE(3) is the group of all rigid-body transformations in 3D: rotations (SO(3)) combined with translations ($\mathbb{R}^3$). It has 6 DOF: 3 rotation, 3 translation.

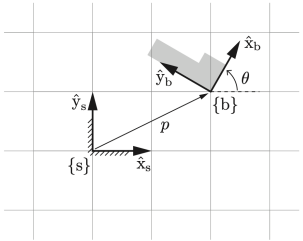
Homogeneous Transformation Matrix

$$T = \begin{bmatrix} R & \mathbf{p} \\ \mathbf{0}^\top & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \in SE(3) \qquad T^{-1} = \begin{bmatrix} R^\top & -R^\top \mathbf{p} \\ \mathbf{0}^\top & 1 \end{bmatrix}$$

Packages R ∈ SO(3) and p ∈ $\mathbb{R}^3$ into a single matrix. Composition is matrix multiplication.

Key Operations

$$T_{AC} = T_{AB} \cdot T_{BC}$$



Why SE(3) Matters for Embodied AI

SE(3) is the universal language for poses across simulators (MuJoCo, Isaac), libraries (Drake, Pinocchio), and research literature.

Coordinate Frames

Every pose is expressed relative to some reference frame. Keeping track of frames is essential—frame mismatches are among the most common bugs in robotics code.

Common Frames in Manipulation

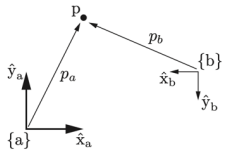
{W} World frame — Origin at a fixed point in the environment (e.g., corner of table or room)
{B} Base frame — At the robot's mounting point, with z-axis pointing up
{E} End-effector frame — At the gripper/tool center point (TCP), oriented along the approach direction
{C} Camera frame — Mounted on the wrist or externally, with z-axis along the optical axis
{O} Object frame — Attached to the object being manipulated

Notation Convention

$$T_{AB} : \text{pose of frame } \{B\} \text{ expressed in frame } \{A\}$$

$$T_{AC} = T_{AB} \cdot T_{BC}$$

Subscript order matters: T_AB transforms points from {B} to {A}.



$$\mathbf{p}^A = T_{AB}\,\mathbf{p}^B = R_{AB}\,\mathbf{p}^B + \mathbf{t}_{AB}$$

⚠ Common Bug: Frame Mismatch

Using camera-frame coordinates when base-frame is expected, or vice versa. Always ask: *"What frame is this pose expressed in?"* When debugging, print frame names alongside poses.

Summary — Representation for Learning

Representation determines learning difficulty. Neural networks approximate functions; the smoother and more unique the mapping, the easier it is to learn.

Rotation is fundamentally hard. SO(3) is a non-Euclidean manifold. Any representation with < 5 dimensions has discontinuities or non-uniqueness. This is topology, not implementation.

Use 6D for neural network I/O. It is continuous, unique, and unconstrained—the only representation satisfying all requirements for learning. Convert to quaternions or rotation matrices at system boundaries.

SE(3) is the universal pose language. All simulators, libraries, and papers use homogeneous transformation matrices. Know the notation: T_AB means pose of frame {B} expressed in frame {A}.

Frame awareness prevents bugs. Always track which frame poses are expressed in. Frame mismatches are a leading cause of robotics bugs that can be subtle and dangerous.

Next: We now have the mathematical language. Next, we study *kinematics*—how joint configurations map to end-effector poses (forward kinematics) and how to invert this mapping (inverse kinematics).

Kinematics

Bridging Joint Space and Task Space

The Central Question

Should your policy operate in joint space or task space?

Joint Space (Configuration Space)

Policy outputs: joint positions, velocities, or torques

Dimension: n (number of joints)

Direct motor commands — no IK needed

Task Space (Operational Space)

Policy outputs: end-effector pose or velocity

Dimension: 6 (position + orientation) or less

Requires IK to convert to joint commands

Kinematics: The Bridge Between Spaces

Forward Kinematics (FK): joint configuration → end-effector pose (always well-defined)

Inverse Kinematics (IK): end-effector pose → joint configuration (may have 0, 1, or many solutions)

This choice determines your policy's output dimension, whether you need an IK solver, and how the policy generalizes across robot morphologies.

Forward Kinematics: Concept and Structure

Definition: Forward Kinematics (FK)

Given the joint configuration $\mathbf{q} = (q_1, q_2, ..., q_n)^T \in \mathbb{R}^n$, compute the end-effector pose $\mathbf{T} \in SE(3)$.

The Chain of Transformations

A serial manipulator is a chain of rigid bodies connected by joints. Each joint i contributes a transformation $T_i(q_i)$ that depends on its joint variable. FK computes the composition of all these transformations from base to end-effector.

Conceptual Form

$$T(\mathbf{q}) = T_1(q_1) \cdot T_2(q_2) \cdots T_n(q_n)$$

Each $T_i(q_i) \in SE(3)$ is a 4×4 homogeneous transformation matrix

Key Properties

Differentiable: Smooth function of q everywhere | **Non-linear:** Products of rotations | **Well-defined:** Every q → exactly one T

Variable Definitions:

Let n denote the number of joints in the robot. For the conceptual form of forward kinematics:

- $\mathbf{q_i \in \mathbb{R}}$: Joint variable for joint i (angle in radians for revolute joints, linear displacement in meters for prismatic joints)
- $\mathbf{q = (q_1, q_2, ..., q_n)^T \in \mathbb{R}^n}$: The joint configuration vector containing all joint variables
- $\mathbf{T_i(q_i) \in SE(3)}$: The 4×4 homogeneous transformation matrix contributed by joint i, which depends on the joint variable q_i
- $\mathbf{T(q) \in SE(3)}$: The end-effector pose (4×4 homogeneous transformation matrix) as a function of the full joint configuration q

Next: To write each $T_i(q_i)$ precisely, we need a way to describe how each joint moves a rigid body — this leads to *screw axes* and the *product of exponentials* formula.

Screw Motion: Why and What

Why Screw Motion? — Chasles' Theorem (1830)

Every rigid-body displacement can be expressed as a **screw motion**: rotation about an axis combined with translation along that same axis. This means we can describe any joint's motion — and thus any robot's FK — using screw motions.

What is Screw Motion?

A screw motion has three elements: (1) a **screw axis** — a line in 3D space, (2) a **rotation angle** θ about that axis, and (3) a **translation distance** d along that axis.

The Screw Axis is a Line in Space

The axis does *not* necessarily pass through the origin. It is a line defined by its direction and location. We encode both in a 6D vector.

Pitch: h = d / θ

The **pitch** describes how much translation per radian of rotation.

h = 0: Pure rotation (revolute joint)
0 < h < ∞: General screw (rotation + translation)
h = ∞: Pure translation (prismatic joint)

Robot joints are typically h=0 (revolute) or h=∞ (prismatic).

Can you imagine?

Next: How do we represent a screw axis mathematically? We encode both axis direction and location in a 6D vector.

Screw Axis Representation

Key Idea: Encode the Velocity Field

A screw axis $S = (\omega, v) \in \mathbb{R}^6$ encodes the velocity of every point in space when the rigid body moves along the screw. For any point at position r, its velocity is: **$v(r) = \omega \times r + v$**

The Components of S = (ω, v)

$\omega \in \mathbb{R}^3$: Angular velocity vector. Direction = axis direction. For revolute joints, $\|\omega\| = 1$.

$v \in \mathbb{R}^3$: Linear velocity of the point at the origin. This encodes where the axis is located in space.

Prismatic Joint (h = ∞)

Pure translation along direction $\hat{v}$. No rotation, so $\omega = 0$:

$$\mathcal{S} = (\mathbf{0},\ \hat{\mathbf{v}})$$

All points move with the same velocity $\hat{v}$ (uniform translation).

Revolute Joint (h = 0)

Axis passes through point p with direction $\hat{\omega}$. The velocity of any point r due to rotation is $v(r) = \hat{\omega} \times (r - p)$. Setting $r = 0$:

$$\mathbf{v} = \hat{\boldsymbol{\omega}} \times (\mathbf{0} - \mathbf{p}) = -\hat{\boldsymbol{\omega}} \times \mathbf{p}$$

Any point p on the axis gives the same v (since $\hat{\omega} \times \hat{\omega} = 0$).

Why v Encodes Axis Location

If axis passes through origin: $p = 0 \Longrightarrow v = 0$.
If axis is offset from origin: $v \neq 0$, and $\|v\|$ equals the perpendicular distance from origin to axis.
The 6D vector S encodes both direction and location.

From Screw Axis to Transformation

Given screw axis S and motion amount q (radians or meters), the **matrix exponential** $e^{[S]q} \in SE(3)$ computes the resulting 4×4 transformation. The bracket [S] converts S to a 4×4 matrix in se(3), and the exponential "integrates" the infinitesimal motion into a finite transformation.

The Product of Exponentials Formula

Given a screw axis $S = (\omega, v) \in \mathbb{R}^6$ where $\omega = (\omega_1, \omega_2, \omega_3)^T \in \mathbb{R}^3$ is the angular velocity component and $v \in \mathbb{R}^3$ is the linear velocity component, the 4×4 matrix representation $[S] \in se(3)$ is:

$$[\mathcal{S}] = \begin{bmatrix} [\omega] & v \\ 0 & 0 \end{bmatrix} \in \mathbb{R}^{4\times4} \qquad [\omega] = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix}$$

Given the matrix form $[S] \in se(3)$ and joint variable $q \in \mathbb{R}$, the matrix exponential $e^{[S]q} \in SE(3)$ is a 4×4 homogeneous transformation matrix. For a revolute joint with $\|\omega\| = 1$:

$$e^{[\mathcal{S}]q} = \begin{bmatrix} e^{[\omega]q} & (Iq + (1-\cos q)[\omega] + (q - \sin q)[\omega]^2)v \\ 0 & 1 \end{bmatrix}$$

$$e^{[\omega]q} = I + \sin q\,[\omega] + (1-\cos q)\,[\omega]^2$$

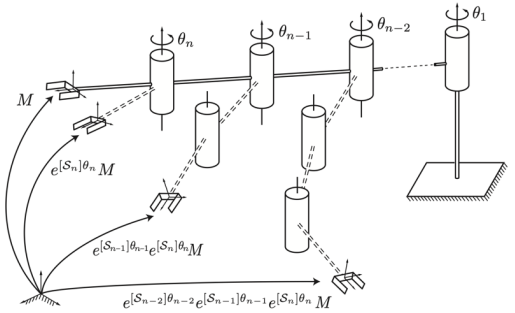
Product of Exponentials (Space Frame)

$$T(\mathbf{q}) = e^{[\mathcal{S}_1]q_1}\, e^{[\mathcal{S}_2]q_2} \cdots e^{[\mathcal{S}_n]q_n}\, M$$

Apply joint transformations in sequence from base to tip, then apply home configuration M.

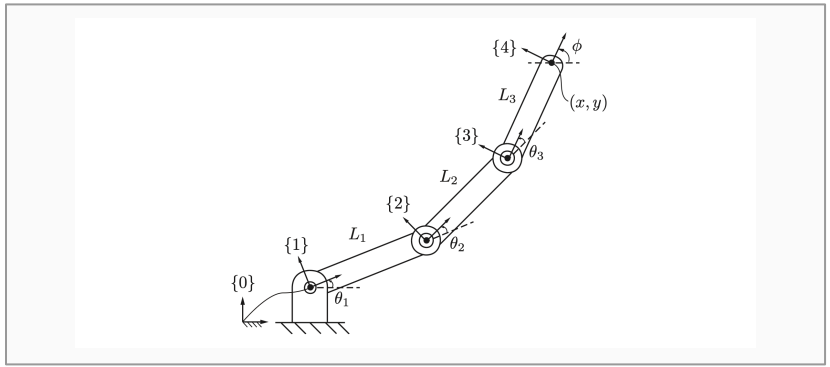
Why "Space Frame"?

All screw axes S_i are expressed in the fixed space frame {s} at home configuration. An alternative "body frame" formulation exists but is less common.



In Practice: Libraries (Pinocchio, Drake, MuJoCo) read robot descriptions (URDF/MJCF), extract screw axes automatically, and compute FK. Understanding the structure helps with debugging and designing observations.

Forward Kinematics: Example



Setup

For a 3R planar arm with link lengths $L_1, L_2, L_3 \in \mathbb{R}^+$ (meters) and joint angles $\theta_1, \theta_2, \theta_3 \in \mathbb{R}$ (radians), where θ_1 is measured from the positive x-axis and θ_2, θ_3 are measured relative to the previous link, the x-coordinate of the end-effector is?

Forward Kinematics Equations

$$x = L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2) + L_3 \cos(\theta_1 + \theta_2 + \theta_3)$$
$$y = L_1 \sin\theta_1 + L_2 \sin(\theta_1 + \theta_2) + L_3 \sin(\theta_1 + \theta_2 + \theta_3)$$

End-Effector Orientation

$$\phi = \theta_1 + \theta_2 + \theta_3$$

Observation: FK is non-linear (sines/cosines). Small changes in θ can produce very different Δx, Δy depending on configuration.

Key Insight: Even this simple 2-link arm has non-linear FK. For a 7-DOF arm with 3D rotations, the mapping is far more complex—but the principle is the same: FK is always well-defined, differentiable, and computable.

Forward Kinematics in Practice

In practice, you call library functions to compute FK. Here's how FK appears in common frameworks.

MuJoCo

mujoco.mj_forward(model, data)
data.xpos[body_id] → position
data.xquat[body_id] → quaternion

PyBullet

p.getLinkState(robot, link_id)
Returns (pos, orn, ...) for specified link

Pinocchio

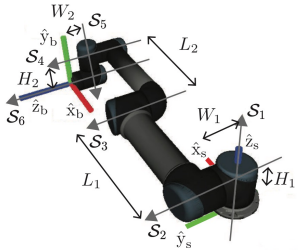
pin.forwardKinematics(model, data, q)
data.oMi[frame_id] → SE(3) pose

FK in Observation Construction

A common pattern: the policy receives end-effector pose as part of its observation, even though the robot is controlled in joint space. This requires calling FK every timestep to convert the raw joint state to task-space coordinates.

Why Include EE Pose in Observations?

Tasks are often defined in task space (reach position X, grasp object at Y)
Relieves network from learning FK implicitly
Makes policy more interpretable and debuggable



The Jacobian: Motivation

FK tells us where the end-effector is. But how do we relate **velocities**? If joints move at certain rates, how fast does the end-effector move?

The Velocity Relationship

Given joint velocities $\dot{\mathbf{q}}$, what is the resulting end-effector velocity $\dot{\mathbf{x}}$?

Why Velocities Matter

Control: Task-space controllers command end-effector velocities

Safety: End-effector speed limits for human interaction

IK: Numerical IK algorithms use velocity relationships

Differential Kinematics

Since x = FK(q), we can differentiate with respect to time:

$$\dot{\mathbf{x}} = \frac{\partial \mathrm{FK}}{\partial \mathbf{q}} \dot{\mathbf{q}} = J(\mathbf{q})\, \dot{\mathbf{q}}$$

J(q) is the **Jacobian matrix** — the local linear approximation to FK

Key Insight

The Jacobian linearizes the non-linear FK mapping at the current configuration. It tells us how infinitesimal joint motions translate to infinitesimal end-effector motions *at this specific pose*. Move to a different configuration, and J changes.

The Jacobian: Definition

Definition: Manipulator Jacobian

$$\dot{\mathbf{x}} = J(\mathbf{q})\,\dot{\mathbf{q}}, \quad \text{where } J(\mathbf{q}) \in \mathbb{R}^{m \times n}$$

J maps joint velocities (n-dimensional) to task-space velocities (m-dimensional, typically m = 6)

Geometric Jacobian Structure

$$J = \begin{bmatrix} J_v \\ J_\omega \end{bmatrix} \in \mathbb{R}^{6 \times n}$$

- $J_v \in \mathbb{R}^{3xn}$ maps joint velocities to end-effector linear velocity ($\dot{p} = J_v \dot{q}$)
- $J\omega \in \mathbb{R}^{3xn}$ maps joint velocities to end-effector angular velocity ($\omega = J\omega \dot{q}$)

Critical Property: Configuration-Dependent

J is a function of q. The same joint velocity produces *different* end-effector velocities at different configurations.

This is because FK is non-linear — its derivative (the Jacobian) varies across the configuration space.

Dimensions

n = number of joints (e.g., 7 for Franka) **m** = task-space dimension (typically 6) **J ∈ ℝ^{6×7}** for a 7-DOF arm

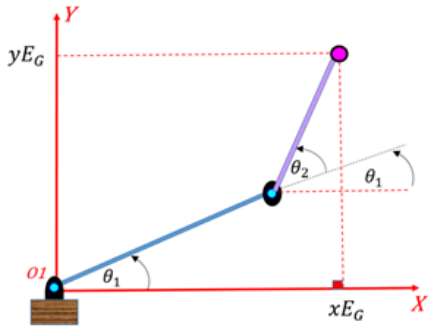
For Learning: When policies output task-space velocities, the Jacobian (or its pseudoinverse) converts these to joint velocities. Libraries compute J automatically.

The Jacobian: Example

2R Planar Arm Jacobian

For a 2R planar arm with FK equations x = $L_1\cos(\theta_1) + L_2\cos(\theta_1+\theta_2)$ and y = $L_1\sin(\theta_1) + L_2\sin(\theta_1+\theta_2)$, the Jacobian is the matrix of partial derivatives

$$J(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial x}{\partial \theta_1} & \frac{\partial x}{\partial \theta_2} \\ \frac{\partial y}{\partial \theta_1} & \frac{\partial y}{\partial \theta_2} \end{bmatrix}$$



Result

$$J = \begin{bmatrix} -L_1 s_1 - L_2 s_{12} & -L_2 s_{12} \\ L_1 c_1 + L_2 c_{12} & L_2 c_{12} \end{bmatrix}$$

J is 2×2: maps 2 joint velocities to 2D end-effector velocity

Numerical Example

At $\theta_1 = 45°$, $\theta_2 = 30°$, $L_1 = L_2 = 1$:
J ≈ [−1.67, −0.97; 0.97, 0.26]

Key Observation: J Changes with Configuration

At $\theta_1 = 0°$, $\theta_2 = 0°$ (arm straight):
J = [0, 0; 2, 1] — can only move in y-direction!

At $\theta_1 = 0°$, $\theta_2 = 90°$ (elbow bent):
J = [−1, −1; 1, 0] — can move in both x and y

Implication: The same joint velocity command produces completely different end-effector motions depending on configuration. This is why task-space control requires recomputing J at every timestep.

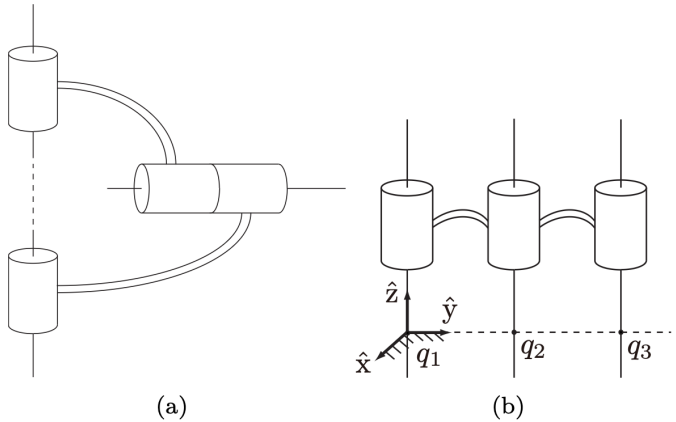
Jacobian Singularities

Definition: Kinematic Singularity

A configuration q where J(q) loses rank. The robot loses ability to move in certain task-space directions.

Common Singularity Types

Boundary: Arm fully extended — workspace boundary
Internal: Joint axes align — e.g., elbow straight
Wrist: Wrist axes align (6-DOF arms)



Mathematical Condition

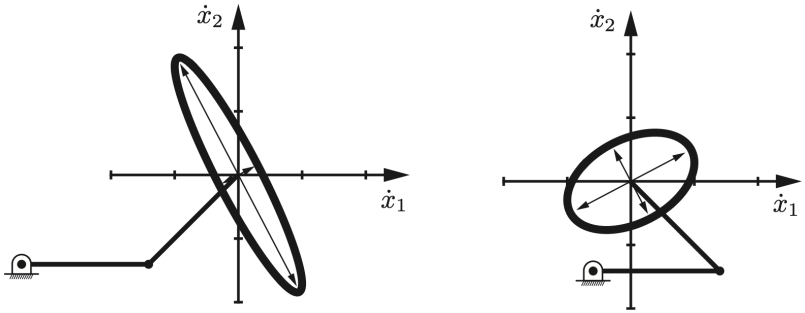
$$\text{rank}(J(\mathbf{q})) < \min(m, n) \implies \text{singularity}$$

For 6-DOF task with 7-DOF arm: singularity when rank(J) < 6.

Why Singularities Matter for Learning

Near singularities, small task-space motions require very large joint velocities (J^+ blows up). Policies may produce unrealizable commands. Learned policies must avoid singularities or handle them gracefully.

Singularity Example: 2R Arm at Full Extension



Setup: $\theta_2 = 0$ (arm straight)

At full extension, both links point in the same direction. The elbow joint rotation axis is parallel to the shoulder rotation axis effect.

Jacobian at $\theta_2 = 0$

For a 2R planar arm at full extension where $\theta_2 = 0$, the Jacobian simplifies using $s_{12} = \sin(\theta_1 + 0) = s_1$ and $c_{12} = \cos(\theta_1 + 0) = c_1$:

$$J|_{\theta_2=0} = \begin{bmatrix} -(L_1 + L_2)s_1 & -L_2 s_1 \\ (L_1 + L_2)c_1 & L_2 c_1 \end{bmatrix}$$

Key observation: columns are proportional → det(J) = 0 → rank deficient

Physical Consequence

Both joints produce velocity in the **same direction** (perpendicular to the arm). Motion along the arm direction is impossible — no joint velocity combination achieves it.

Velocity Ellipse Collapses

Away from singularity, unit joint velocities trace an ellipse in task space. At singularity, this ellipse collapses to a line — the robot can only move in one direction.

Practical Implications

If a task-space policy requests motion toward the arm (radially inward), the IK solver has no valid solution. Reward shaping or workspace constraints should discourage learned policies from entering singular regions.

Manipulability

Manipulability: Quantifying Motion Capability

Not all configurations are equally good. Manipulability measures how "well" the robot can move — far from singularities is better.

Manipulability Measure

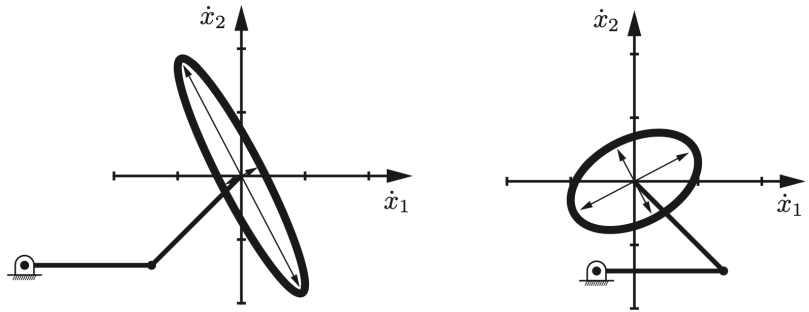
$$w(\mathbf{q}) = \sqrt{\det(J(\mathbf{q})\, J(\mathbf{q})^\top)}$$

w = 0 at singularity (det = 0)

w large when robot can move easily in all directions

Manipulability Ellipsoid

The set of end-effector velocities achievable with unit joint velocity forms an ellipsoid. Principal axes = singular vectors of J. Axis lengths = singular values of J.



Applications in Learning

Observation: Include w(q) to help policy understand motion capability

Reward: Bonus for high manipulability keeps robot from singularities

Null-space: Optimize manipulability as secondary objective

Inverse Kinematics: Problem Statement

Definition: Inverse Kinematics (IK)

Given a desired end-effector pose **T_target** ∈ SE(3), find joint configuration(s) **q** such that FK(q) = T_target.

The Challenge: IK is Hard

Unlike FK (which is always well-defined), IK may have **zero**, **one**, **multiple**, or **infinitely many** solutions. This makes it fundamentally more difficult than FK.

No Solution

Target is outside the workspace, or orientation is unreachable at that position.

Multiple Solutions

Same pose achievable with different configurations (elbow-up vs elbow-down).

Infinite Solutions

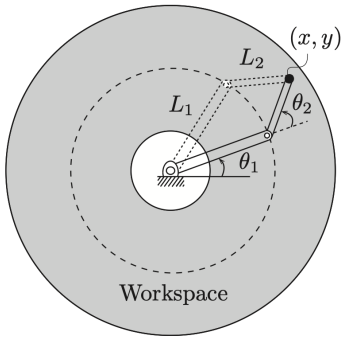
Redundant robots (n > 6) have a continuous manifold of solutions — the null space.

Mathematical Formulation

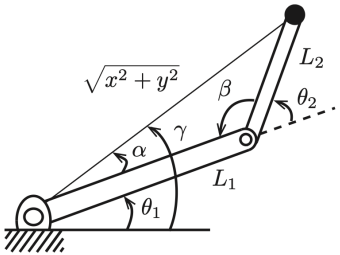
$$\text{Find } \mathbf{q} \text{ such that } \mathrm{FK}(\mathbf{q}) = T_{\text{target}} \qquad \text{or} \qquad \mathbf{q}^* = \arg\min_{\mathbf{q}} \|\mathrm{FK}(\mathbf{q}) - T_{\text{target}}\|$$

IK can be formulated as root-finding (find exact solution) or optimization (minimize error). Numerical methods typically use the optimization view.

IK: Multiple Solutions



(a) A workspace, and lefty and righty configurations.



(b) Geometric solution.

The 2R Arm Example

For a 2R planar arm reaching a target (x, y), there are typically **two solutions**: one with elbow bent "up" and one with elbow bent "down."

Analytical IK for 2R Arm

$$\theta_2 = \pm \arccos\left(\frac{x^2+y^2-L_1^2-L_2^2}{2L_1L_2}\right)$$

The ± gives two solutions. Once θ_2 is chosen, θ_1 is determined.

For 6-DOF Arms: Up to 8 Solutions

A standard 6-DOF arm with spherical wrist can have up to **8** distinct IK solutions for a single end-effector pose: shoulder left/right × elbow up/down × wrist flip/no-flip = $2^3 = 8$ configurations. Non-standard geometries may have additional solutions.

How to Choose Among Solutions?

Additional criteria are needed: closest to current configuration (for smooth motion), highest manipulability (avoid singularities), avoids obstacles (collision-free), satisfies joint limits, or minimizes energy. IK solvers typically return the solution closest to a provided initial guess.

Numerical IK: Jacobian Methods

Core Idea: Iterative Linearization

Iteratively update q by computing the task-space error, then using the Jacobian to find the joint-space correction: $\Delta q = J^{+} \Delta x$. Repeat until error is small.

- Given task-space error $\Delta x \in \mathbb{R}^{m}$ and Jacobian $J(q) \in \mathbb{R}^{m \times n}$, the minimum-norm joint update is computed using the Moore-Penrose pseudoinverse $J^{+} \in \mathbb{R}^{n \times m}$.

Jacobian Pseudoinverse

$$\Delta \mathbf{q} = J^{+} \Delta \mathbf{x} = J^{\top} (J J^{\top})^{-1} \Delta \mathbf{x}$$

Pros: Minimum-norm solution; exact when invertible

Cons: Blows up near singularities (JJ^{T} nearly singular)

Damped Least Squares (DLS)

$$\Delta \mathbf{q} = J^{\top} (J J^{\top} + \lambda^{2} I)^{-1} \Delta \mathbf{x}$$

Pros: Numerically stable; graceful degradation at singularities

Cons: Introduces tracking error; λ requires tuning

Iteration Loop

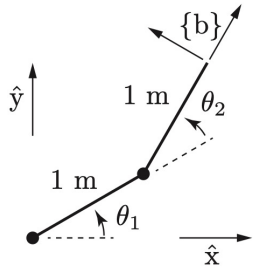
1. Compute error: $\Delta x = x_{target} - FK(q)$

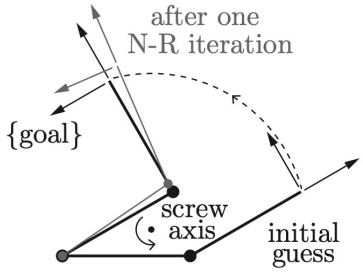
2. Compute update: $\Delta q = J^{+}(q)\, \Delta x$ (or DLS variant)

3. Update configuration: $q \leftarrow q + \alpha \cdot \Delta q$ (α = step size)

In Practice: Use library implementations (Pinocchio, Drake, MuJoCo IK). They handle numerical details, joint limits, and singularity avoidance.

Numerical IK: Example





Iteration Trace

Setup: $L_1 = L_2 = 1$m, Target: (1.2, 0.8)

Initial guess: θ = (0°, 30°), gives FK = (1.87, 0.5)

Iter 1: error = (−0.67, 0.30), ||e|| = 0.73
Iter 2: error = (−0.12, 0.08), ||e|| = 0.14
Iter 3: error = (−0.01, 0.01), ||e|| = 0.01
Converged: θ = (33.7°, 58.2°) ✓

What Can Go Wrong

Near singularity: Convergence slows dramatically; Δq becomes very large; may oscillate or diverge.

Unreachable target: Algorithm never converges; settles at closest reachable pose with residual error.

Local minima: For complex robots, may converge to wrong solution branch depending on initial guess.

Practical Advice for Learning Pipelines

Always use the current robot configuration as the initial guess — this ensures smooth trajectories and avoids jumping between IK solution branches. Set reasonable iteration limits and handle IK failure gracefully (e.g., hold previous command or trigger safety stop).

IK for Redundant Robots

Recall from Part 2: Redundancy and Null Space

When n > m (more joints than task DOF), infinitely many IK solutions exist. The extra freedom forms the **null space** — joint motions that don't affect the end-effector.
For a redundant robot with n joints and m task-space dimensions (n > m), the joint velocity that achieves desired end-effector velocity $\dot{x} \in \mathbb{R}^m$ while also pursuing a secondary objective $\dot{q}_0 \in \mathbb{R}^n$.

Null-Space Projection

$$\dot{\mathbf{q}} = J^{+}\dot{\mathbf{x}} + (I - J^{+}J)\dot{\mathbf{q}}_0$$

First term achieves the task. Second term $(I - J^{+}J)$ projects $\dot{q}_0$ onto the null space — achieving **secondary objectives** without affecting the end-effector.

- $J^{+} \in \mathbb{R}^{nxm}$ is the pseudoinverse of the Jacobian $J(q) \in \mathbb{R}^{mxn}$
- $I \in \mathbb{R}^{nxn}$ is the identity matrix
- $(I - J^{+}J) \in \mathbb{R}^{nxn}$ is the null-space projection matrix
- $\dot{q}_0 \in \mathbb{R}^{n}$ is an arbitrary joint velocity representing the secondary objective

Common Secondary Objectives ($\dot{q}_0$)

Maximize manipulability: $\dot{q}_0 = \nabla w(q)$ **Joint centering:** $\dot{q}_0 = k(q_{mid} - q)$ **Obstacle avoidance:** move elbow away

Redundancy Resolution in Learning

Option 1: Policy outputs task-space only; IK handles null-space via hand-designed objectives (e.g., maximize manipulability).

Option 2: Policy outputs joint-space, implicitly learning how to use redundancy for the task.

Option 3: Policy outputs task-space Δx *plus* a null-space gradient $\dot{q}_0$ (e.g., scalar weights on predefined objectives), learning both explicitly.

Kinematics in Policy Architectures

Where FK and IK appear in your system depends on your policy architecture. Here are three common patterns:

Pattern 1: Joint-Space

Output: joint positions/velocities
FK: observation (EE pose)
IK: **not needed**
Simple, direct

Pattern 2: Task-Space + IK

Output: EE pose/velocity
FK: observation
IK: **converts output → joints**
Task-aligned, transfers well

Pattern 3: Learned IK

Output: task-space target
Second network learns IK
Or: differentiable IK layer
Flexible, robot-specific

Comparison

Aspect	Joint-Space	Task-Space+IK	Learned IK
Action dimension	n joints	6 (SE(3))	6 (SE(3))
Sample efficiency	Lower (learns FK)	Higher (task-aligned)	Medium
Morphology transfer	Hard	Easier	Medium
Singularity handling	Implicit	IK solver	Learned

Key Decision: If tasks are defined in task space (reach X, maintain orientation Y), Pattern 2 works well. If you need fine-grained joint control or want to avoid IK complexity, Pattern 1 is simpler.

Summary — Kinematics Decisions

Forward Kinematics is always available. Given joint configuration q, FK computes end-effector pose T. It's differentiable, non-linear, and computed automatically by simulators and libraries. Use FK to include task-space information in observations.

The Jacobian connects velocity spaces. J(q) is configuration-dependent. It enables task-space control and numerical IK. Singularities (rank-deficient J) limit motion capability and cause control/IK difficulties.

Inverse Kinematics is the hard direction. IK may have zero, one, multiple, or infinite solutions. Numerical methods (pseudoinverse, DLS) iterate to find solutions. Always seed with current configuration for smooth motion.

Choose your policy space deliberately. Joint-space policies are simpler (no IK needed) but must learn the FK relationship implicitly. Task-space policies align with task definitions but require IK in the loop. Neither is universally better — choose based on your task and transfer requirements.

Quick Decision Guide

Use joint-space actions if: Task involves fine joint control, avoid IK complexity, or high-frequency policy.

Use task-space actions if: Task defined in Cartesian space, need cross-morphology transfer, or demo data is task-space.

Next: We turn to *dynamics* — how forces and torques produce motion, and how this shapes action space design.

Dynamics

How Forces Produce Motion

The Dynamics Equation

- $\theta \in \mathbb{R}^n$ — joint positions (angles)
- $\dot{\theta} \in \mathbb{R}^n$ — joint velocities
- $\ddot{\theta} \in \mathbb{R}^n$ — joint accelerations
- $\tau \in \mathbb{R}^n$ — joint torques (commanded)

- $M(\theta) \in \mathbb{R}^{n \times n}$ — mass/inertia matrix (symmetric positive definite)
- $c(\theta, \dot{\theta}) \in \mathbb{R}^n$ — Coriolis and centrifugal forces
- $g(\theta) \in \mathbb{R}^n$ — gravity torques

Equations of Motion (for an n-joint robot)

$$\tau = M(\theta)\ddot{\theta} + c(\theta, \dot{\theta}) + g(\theta)$$

All vectors are n×1, M(θ) is n×n symmetric positive definite (always invertible)

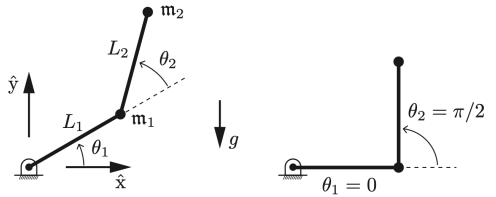
$M(\theta)\ddot{\theta}$ — Inertial forces: mass matrix maps joint accelerations to required torques

$c(\theta, \dot{\theta}) = C(\theta, \dot{\theta})\dot{\theta}$ — Velocity-dependent forces arising from moving reference frames

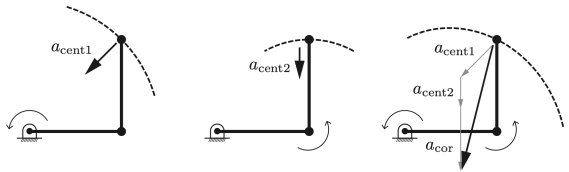
$g(\theta) = \frac{\partial P(\theta)}{\partial \theta}$ — Gradient of gravitational potential energy

Example: 7-DOF Panda Arm

θ ∈ $\mathbb{R}^7$ (7 joint angles), τ ∈ $\mathbb{R}^7$ (7 torques), M(θ) is 7×7. At full extension, shoulder may need 50+ Nm just to hold against gravity.



2R open chain under gravity



Centripetal and Coriolis accelerations

Key Insight: Dynamics is nonlinear and configuration-dependent — the same torque produces different accelerations depending on robot pose and velocity.

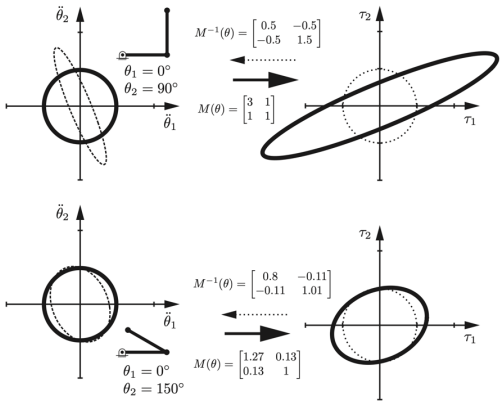
Configuration-Dependent Dynamics

Mass Matrix M(θ) Changes with Configuration

The same torque produces different accelerations depending on robot pose. When the arm is extended, it has more rotational inertia and is harder to accelerate.

Concrete Example

A 2-link arm extended horizontally might need **50 Nm** at the shoulder for 1 rad/s² acceleration. The same arm folded back might need only **15 Nm** for the same acceleration — a 3× difference!



Reading the Ellipsoid

Ellipsoid principal axes show directions of easy/hard acceleration. Larger extent = more torque required.

FORWARD DYNAMICS

$$\ddot{\theta} = M^{-1}(\theta)\left[\tau - c(\theta, \dot{\theta}) - g(\theta)\right]$$

Torques in → accelerations out. **MuJoCo/PyBullet do this** every timestep to simulate physics.

INVERSE DYNAMICS

$$\tau = M(\theta)\ddot{\theta} + c(\theta, \dot{\theta}) + g(\theta)$$

Desired motion → required torques. **Robot controllers use this** for trajectory tracking.

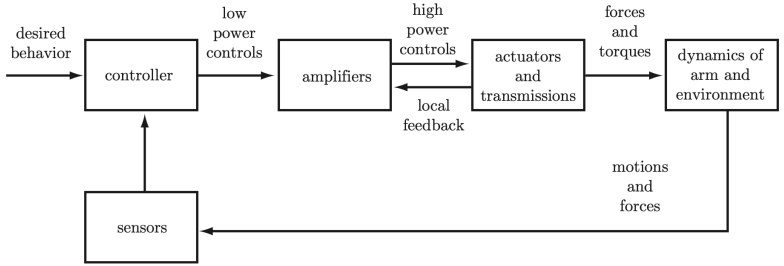
Computational Considerations

Both forward and inverse dynamics can be computed efficiently using recursive Newton-Euler algorithms. Complexity is O(n) in the number of joints.

Why This Matters for Learning

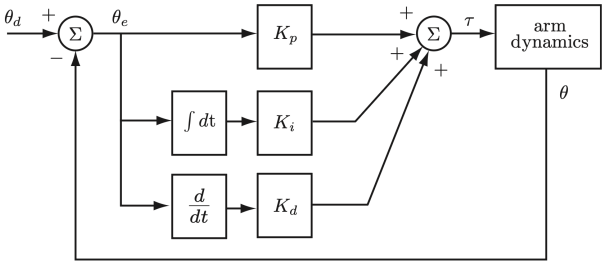
If your policy outputs torques, it must implicitly learn these configuration-dependent dynamics. If your policy outputs positions, the low-level controller handles this for you.

How Classical Control Uses Dynamics



Control system block diagram

Sensors → Controller → Actuators → Dynamics

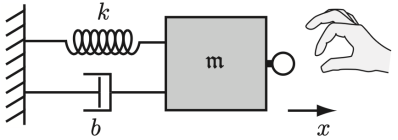


PID controller block diagram

Computed Torque Control (Track a Trajectory)

$$\tau = M(\theta)\left(\ddot{\theta}_d + K_p(\theta_d - \theta) + K_d(\dot{\theta}_d - \dot{\theta})\right) + c(\theta,\dot{\theta}) + g(\theta), \quad K_p, K_d \in \mathbb{R}^{n\times n}$$

Uses inverse dynamics to cancel nonlinearities. The feedforward term (M, c, g) compensates for known dynamics; the feedback term (K_p, K_d) corrects for errors. *Requires accurate dynamics model.* θd is the desired joint position trajectory



Virtual spring-damper at end-effector

Impedance Control (Behave Like a Spring-Damper)

$$F = K(x_d - x) + B(\dot{x}_d - \dot{x})$$
$$x_d, \dot{x}_d \in \mathbb{R}^m \text{ (desired position, velocity)}$$
$$K, B \in \mathbb{R}^{m\times m} \text{ (stiffness, damping)}$$

Makes robot behave like a mass-spring-damper system. Useful for contact tasks where position tracking alone is insufficient.

Classical Control vs. Learning

Classical methods assume **known dynamics** — the model is hand-derived or identified. Learning methods can work with **unknown dynamics** — the policy implicitly learns to compensate through experience.

What's Inside the Robot

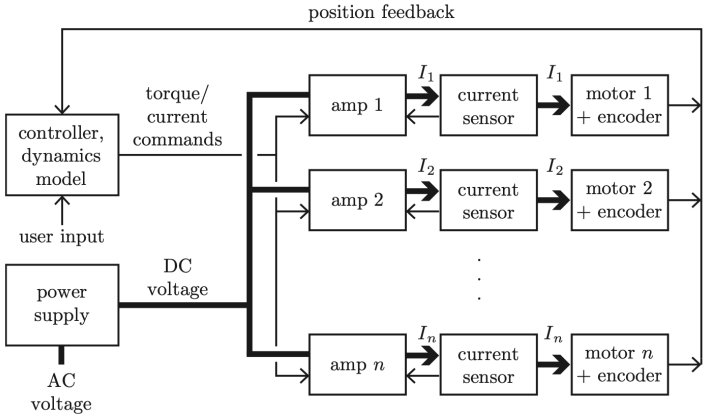


Figure 8.7: A block diagram of a typical n-joint robot. The bold lines correspond to high-power signals while the thin lines correspond to communication signals.

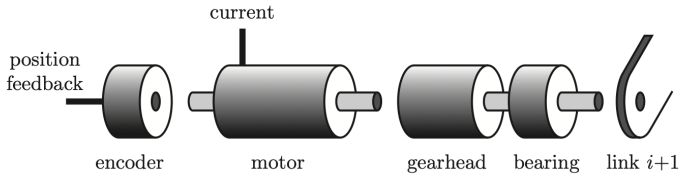


Figure 8.8: The outer cases of the encoder, motor, gearhead, and bearing are fixed in link i, while the gearhead output shaft supported by the bearing is fixed in link $i+1$.

Most Robots Don't Expose Direct Torque Control

Commercial robots typically provide position or velocity command interfaces. Your commands pass through multiple layers:

Command → Controller → Amplifier → Motor → Gearhead → Link

What the Low-Level Controller Handles

The robot's internal controller runs at 1+ kHz and handles: gravity compensation, friction compensation, trajectory interpolation, and safety limits. When you send a position command, all of this happens automatically.

Implication for Learning

Your "action" is processed through this entire pipeline before affecting the robot. The action space you choose (position, velocity, or torque) determines how much complexity your policy must handle vs. how much is handled for you.

Action Space Choices

Position Control

$$a = \theta_d \in \mathbb{R}^n, \quad \text{(target joint angles in radians, 10-100 Hz)}$$

Policy outputs target joint angles θ_d. Robot's controller handles dynamics. *action = [θ_1, θ_2, ...] in radians*

↓

Velocity Control

$$a = \dot{\theta}_d \in \mathbb{R}^n, \quad \text{(target joint velocities in rad/s, 100-500 Hz)}$$

Policy outputs target velocities θ̇_d. More responsive, still abstracted. *action = [$\dot{\theta}_1$, $\dot{\theta}_2$, ...] in rad/s*

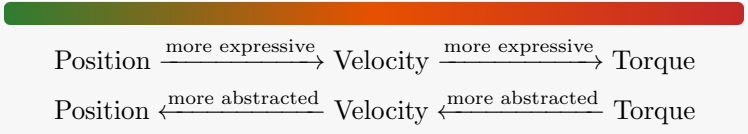
↓

Torque Control

$$a = \tau \in \mathbb{R}^n, \quad \text{(joint torques in Nm, 500-1000+ Hz)}$$

Policy outputs joint torques τ directly. Full control but must handle dynamics. *action = [τ_1, τ_2, ...] in Nm*

Abstraction vs. Expressiveness



The Fundamental Trade-off

Higher abstraction (position control) makes learning easier and safer, but limits what behaviors can be expressed. Lower abstraction (torque control) enables dynamic motions but requires handling complex dynamics.

What Most Systems Use

The majority of successful robot learning deployments use position or velocity control. Torque control is primarily used in simulation-heavy research or on specialized hardware.

Design Choice: Your action space is not given — it's a decision that significantly affects learning difficulty, safety, and achievable behaviors.

Position/Velocity Control for Learning

✓ Stable and Safe

The robot's built-in controller ensures smooth motion and enforces safety limits. Even if the policy outputs poor commands, the low-level controller prevents dangerous accelerations.

✓ Better Sim-to-Real Transfer

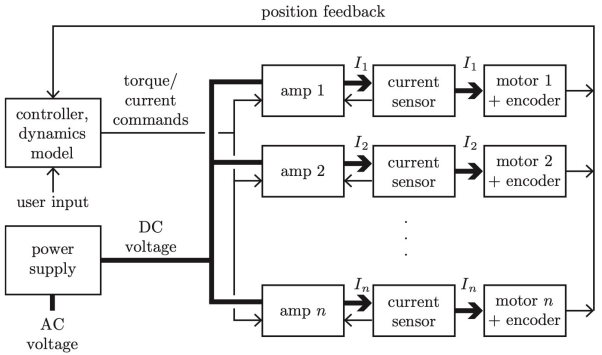
Position/velocity commands are less sensitive to dynamics mismatch. The low-level controller compensates for differences between simulated and real dynamics.

✓ Easier to Learn

The policy doesn't need to learn dynamics — it only needs to learn the mapping from observations to desired positions/velocities. This significantly reduces learning complexity.

△ Limited Bandwidth

Position commands are interpolated by the low-level controller, which smooths rapid changes. This limits responsiveness to fast-changing situations.



△ Can't Exploit Dynamics

Dynamic motions like throwing, jumping, or whip-like movements require precise torque timing. Position control cannot express these behaviors — the controller "fights" against fast accelerations.

When to Use Position/Velocity Control

Use this as your **default choice**. Appropriate for the vast majority of manipulation and locomotion tasks where safety and sim-to-real transfer matter more than dynamic expressiveness.

Pick and place • Assembly • Walking • Tool use • Insertion

Torque Control for Learning

Full Expressiveness

Torque control enables dynamic motions that exploit the robot's inertia and momentum. Behaviors like throwing, catching, jumping, and running require precise control over forces and accelerations.

Direct Force Control

For contact-rich tasks, torque control allows direct specification of interaction forces. Essential for polishing, wiping, or compliant assembly where force regulation matters.

⚠ Must Handle Dynamics

The policy must learn the full nonlinear dynamics: gravity compensation, Coriolis forces, and inertia. Small errors in torque commands produce accelerations that compound over time.

⚠ Safety Concerns

Without the buffer of a position controller, bad torque commands can cause dangerous motions immediately. Requires careful safety monitoring and torque limits.

⚠ Rarely Available on Commercial Robots

Most industrial and collaborative robots only expose position/velocity interfaces. True torque control requires specialized hardware with torque sensors or current-controlled motors.

Requirements for Successful Torque Control

"Torque control policies require simulation-based training because learning the full dynamics demands millions of samples and involves unsafe exploration that cannot be performed on real hardware."

- High-quality simulation with accurate dynamics
- Extensive domain randomization
- Hardware with torque sensing or estimation
- Careful safety monitoring systems
- Often: massive parallel simulation (GPU-based)

When to Use Torque Control

Use when the task fundamentally requires dynamic motions or precise force control, and you have the simulation infrastructure and hardware to support it.

Throwing • Jumping • Running • Batting • Force-sensitive assembly

The Sim-to-Real Gap: Dynamics Mismatch

Mass and Inertia Errors (typically 5-20%)

$$M_{\text{real}}(\theta) \neq M_{\text{sim}}(\theta)$$

CAD models often have incorrect masses. Cables, sensors, and end-effectors add unmodeled mass. Inertia tensors are difficult to measure accurately.

Friction (can vary 2-5× from simulation)

$$f_{\text{real}}(\dot{\theta}) \neq f_{\text{sim}}(\dot{\theta})$$

Real friction is complex: static, viscous, Coulomb, and Stribeck effects. It varies with temperature, wear, and lubrication. Simulators typically use simplified models.

Actuator Dynamics (10-50ms unmodeled delays)

$$\tau_{\text{actual}}(t) = G(s) \cdot \tau_{\text{commanded}}(t) \quad \text{where } G(s) \neq 1$$

Real motors have delays, bandwidth limits, and nonlinear torque curves. Gearbox backlash and compliance are hard to model. Communication delays add latency.

Contact Parameters (highly variable)

$$F_{\text{contact}} = k_{\text{real}}\delta + b_{\text{real}}\dot{\delta}, \quad k_{\text{real}} \neq k_{\text{sim}}, \quad b_{\text{real}} \neq b_{\text{sim}}$$

Contact stiffness, damping, and friction coefficients vary significantly. Surface properties change with wear. Soft contacts are especially challenging.

Kinematics
Transfers well

Dynamics
Transfers poorly

The Core Challenge

Dynamics mismatch is the primary source of sim-to-real failure. A policy trained with perfect torque tracking in simulation will fail when real motors can't deliver commanded torques exactly. This is why action space choice matters — higher abstraction levels absorb more mismatch.

Domain Randomization

The Strategy (Tobin et al. 2017, OpenAI 2018)

Instead of trying to perfectly match simulation to reality, randomize dynamics parameters during training. The policy learns to be robust across a range of dynamics, so it can handle the unknown real dynamics as "just another sample."

Parameters to Randomize

Link Masses ±10–30% of nominal	**Friction Coefficients** 0.5× to 2× nominal
Actuator Delays 0–50ms added latency	**Contact Stiffness** 0.5× to 2× nominal

Key Insight

The policy learns **invariant features** — aspects of the task that hold regardless of exact dynamics. It cannot rely on exploiting specific simulation quirks because those quirks change every episode.

Training with Domain Randomization

Heavy (m+20%) | Light (m-15%) | High friction | Low friction | Delayed (+30ms)

↓

Robust Policy

Each episode samples different dynamics parameters

What Domain Randomization Achieves

The real robot's dynamics fall within (or near) the training distribution. The policy has already learned to handle this variation, so transfer succeeds without explicit system identification.

Trade-off: Robustness vs. Optimality

A robust policy that works across many dynamics is typically more conservative than one optimized for exact dynamics. This is acceptable for most tasks — reliability matters more than peak performance.

Learned vs. Embedded Dynamics

$$\hat{f}_\phi(s_t, a_t) \approx s_{t+1}, \quad \text{learn dynamics model from data}$$

Model-Free RL

Learn everything end-to-end

PPO, SAC, TD3

Policy learns directly from experience. Implicitly compensates for dynamics through trial and error.

Strengths

No modeling assumptions; handles complex dynamics

Weaknesses

Requires 10^6-10^9 samples; poor efficiency

Best For
Cheap sim, complex contact, deformables

$$\pi^*(a|s) = \arg\max_\pi \mathbb{E}\left[\sum_t \gamma^t r(s_t, a_t)\right]$$

Model-Based RL

Learn dynamics model, then plan

MBPO, Dreamer, PETS

Learn f(s,a)→s' from data, then use MPC or planning. Model can transfer across tasks.

Strengths

10-100× more sample efficient; model reusable

Weaknesses

Model errors compound; planning expensive

Best For
Limited real data; long-horizon tasks

Hybrid / Residual

Embed structure, learn residuals

Neural ODE, Residual Physics

Start with known physics (rigid-body dynamics), learn corrections. Physics-informed neural networks.

Strengths

Best of both; generalizes to new scenarios

Weaknesses

Requires correct structural assumptions

Best For
Partially known physics; sim-to-real

$$s_{t+1} = \underbrace{f_{\text{physics}}(s_t, a_t)}_{\text{known rigid-body dynamics}} + \underbrace{f_{\text{residual}}(s_t, a_t; \phi)}_{\text{learned correction}}$$

Physics Simulators

MJ

MuJoCo

Fast, accurate rigid-body dynamics with smooth contact handling. Widely used in RL research. Now free and open-source (DeepMind). **~10M steps/sec (CPU).**

Fast CPU • Smooth contacts • Tendons • XML format

PB

PyBullet

Open-source physics engine with Python bindings. Good URDF support, easy to get started. Popular for manipulation research.

Open source • URDF support • Easy setup • Good docs

IG

Isaac Gym / Isaac Sim

NVIDIA's GPU-accelerated simulator. Enables massive parallel simulation (thousands of environments). State-of-the-art for large-scale RL. **~1B+ steps/sec (GPU, 4096 envs).**

GPU parallelism • Massive throughput • Photorealistic rendering

What They All Do

All physics simulators solve forward dynamics: given current state and applied forces, compute the next state. They differ in how they handle contact, their speed, and their API design.

Contact Handling Matters

Contact is where simulators differ most. MuJoCo uses soft contacts with implicit integration. Bullet uses impulse-based hard contacts. Different choices lead to different sim-to-real gaps.

Simulator Choice Affects Sim-to-Real

Your choice of simulator determines what kind of dynamics mismatch you'll face. Policies trained in different simulators may transfer differently to the same real robot. Consider this when designing your pipeline.

Summary

THE DYNAMICS EQUATION

$$\tau = M(\theta)\ddot{\theta} + c(\theta, \dot{\theta}) + g(\theta)$$

Key Concepts Covered

Mass Matrix M(θ) • Coriolis Forces • Gravity Compensation
Forward Dynamics • Inverse Dynamics • Action Spaces
Sim-to-Real Gap • Domain Randomization

MAIN TAKEAWAY

Dynamics understanding is essential for robot learning, not because you'll derive equations by hand, but because it informs critical design decisions: action space choice, simulation setup, and sim-to-real transfer strategy.

1 Start with Position/Velocity Control

Unless you specifically need torque-level expressiveness, use position or velocity control. It's safer, easier to learn, and transfers better.

2 Dynamics Mismatch is the Main Challenge

Kinematics transfers well; dynamics does not. Focus your sim-to-real efforts on handling mass, friction, and actuator uncertainties.

3 Domain Randomization is Standard

Randomize dynamics parameters during training to learn robust policies. This is the default approach for sim-to-real transfer.

4 Action Space is a Design Choice

Your action space is not given by the problem — it's a decision that affects learning difficulty, safety, and achievable behaviors.

Q & A