Foundations of Robotics and Embodied AI (WIP)

Machine Learning, Perception, Planning,
Estimation, and Control

Roman Grigorii

Ph.D., Robotics and Human–Machine Interaction
Senior Software Engineer, Robotics

2026

Preface

Robotics is entering a period of unusually rapid change. For much of its history, capable robotic systems were built by decomposing intelligence into carefully engineered components: perception estimated the state of the world, planner generated trajectories, controllers converted those trajectories into forces and torques, and task logic coordinated the system as a whole. This approach remains fundamental to modern robotics, but it is increasingly being complemented by large learned models capable of connecting perception, language, reasoning, and action within a common architecture.

By mid 2020's, the frontier of the field has moved substantially toward *generalist embodied intelligence*. Vision–language–action models can condition robot behavior directly on images, language instructions, proprioceptive state, and other observations. Transformer-based policies, diffusion and flow-based action generators, and large-scale imitation learning are increasingly being used to produce continuous robot behavior rather than merely semantic predictions.

At the same time, robotics remains fundamentally different from static/digital machine learning. A language model can generate an incorrect token with little consequence. A robot acts through a physical body whose motion is constrained by geometry, dynamics, contact, friction, actuator limits, latency, uncertainty, and safety. A successful embodied system must therefore connect modern learning methods to the mathematical and engineering foundations of robotics.

Several problems remain especially important. Robot data are expensive to collect, and available datasets remain small compared with those used to train language and vision models. Human video, teleoperation, simulation, synthetic data, and autonomous experience are consequently being combined to increase scale and diversity. Transferring knowledge between different robot embodiments remains difficult. Long-horizon reliability, recovery from mistakes, manipulation under uncertainty, contact-rich interaction, and generalization to unfamiliar environments remain substantially harder than performing short demonstrations under controlled conditions.

The emerging picture is therefore not one in which classical robotics has been replaced by machine learning. Instead, modern robotics increasingly combines several closely related areas:

1. **Robot mechanics and kinematics** - modeling configuration, motion, forces, and physical interaction.
2. **Control** - regulating motion, force, stability, and contact behavior.
3. **State estimation and sensor fusion** - inferring pose, velocity, and other hidden states from noisy measurements.
4. **Computer vision and perception** - extracting geometric, semantic, and task-relevant information from sensor data.
5. **Planning and optimization** - generating feasible motions, trajectories, and action sequences under constraints.
6. **Machine learning** - learning perception, representations, policies, dynamics, and decision-making from data.
7. **Systems engineering** - integrating computation, sensors, actuators, communication, and software into reliable real-world robotic platforms.

Modern robotics is increasingly defined by the integration of these areas rather than by any one of them in isolation, and understanding modern embodied AI requires fluency across all of them. I wrote a collection of chapters that I personally use to update myself on concepts quickly, whether for work or for fun, and I hope it can be useful for you as well.

Contents

Chapter 1

Classical Robotics

Classical robotics provides the mathematical and physical foundations for describing, estimating, planning, and controlling robot motion. This chapter develops the core tools used to represent rigid-body geometry, model robot kinematics and dynamics, estimate state from noisy sensors, localize and map the environment, plan feasible motion, and control physical interaction through contact. These methods form the basis of modern robotic systems and provide the structure upon which learning-based approaches, including imitation learning, reinforcement learning, and embodied AI, are built.

Robotics requires precise mathematical language for describing the position and orientation of objects, sensors, links, and coordinate frames in three-dimensional space. A robot manipulator, for example, may contain a base frame, several joint frames, an end-effector frame, one or more camera frames, and coordinate frames attached to objects in the environment. Perception and control algorithms must continually transform geometric quantities between these frames.

The mathematical structure underlying rigid-body motion is built primarily from the rotation group $SO(3)$ and the rigid-motion group $SE(3)$. Rotation matrices, Euler angles, axis–angle representations, and quaternions provide different ways of representing orientation, while homogeneous transformation matrices combine rotation and translation into a single mathematical entity. Lie-group methods, which are en extensions to the above, provide a local representation that is particularly useful for estimation, optimization, control, and simultaneous localization and mapping.

These concepts form the geometric foundation for the kinematics, state estimation, perception, and manipulation methods developed later in this chapter.

1.1 Mathematical Foundations of Robot Motion

1.1.1 Coordinate Frames and Homogeneous Transformations

A coordinate frame consists of an origin together with a set of orthonormal basis vectors. In three dimensions, a frame $\{A\}$ can be written conceptually as

$$\{A\} = (o_A, \hat{x}_A, \hat{y}_A, \hat{z}_A)\,,$$

where o_A is the origin and the three unit vectors define the coordinate axes. A physical point exists independently of the coordinate frame used to describe it. Its numerical coordinates, however, depend on the chosen frame.

Let

$$p_A = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \in \mathbb{R}^3$$

denote the coordinates of a physical point p expressed in frame $\{A\}$. Suppose a second frame $\{B\}$ is rotated and translated relative to $\{A\}$. The coordinates of a point expressed in the two frames are related by

$$p_A = R_{AB} p_B + t_{AB}.$$

Here, $R_{AB} \in \mathbb{R}^{3\times 3}$ describes the orientation of frame B relative to frame A, while $t_{AB} \in \mathbb{R}^3$ gives the position of the origin of frame B, expressed in frame A. This notation is worth interpreting carefully. The transformation T_{AB} maps coordinates expressed in frame B into coordinates expressed in frame A. The rotation and translation can be combined into a single homogeneous transformation matrix,

$$\boxed{T_{AB} = \begin{bmatrix} R_{AB} & t_{AB} \\ 0 & 1 \end{bmatrix}.}$$

A three-dimensional point can then be augmented with an additional homogeneous coordinate,

$$\tilde{p}_B = \begin{bmatrix} p_B \\ 1 \end{bmatrix},$$

so that coordinate transformation becomes a matrix multiplication:

$$\boxed{\tilde{p}_A = T_{AB} \tilde{p}_B.}$$

This representation is extremely useful because rotation and translation can now be composed using ordinary matrix multiplication.

1.1.2 Rigid-Body Transformations and $SE(3)$

A rigid-body transformation changes the position and orientation of an object without changing distances or angles within that object.

A three-dimensional rigid-body pose is represented by

$$T = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix},$$

where $R \in SO(3)$ represents orientation and $t \in \mathbb{R}^3$ represents translation.

The set of all valid three-dimensional rigid transformations forms the *special Euclidean group*

$$\boxed{SE(3) = \left\{ \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \;\middle|\; R \in SO(3),\ t \in \mathbb{R}^3 \right\}.}$$

A rigid-body transformation therefore acts on a point according to

$$p' = Rp + t.$$

The transformation preserves Euclidean distance. If two points p_1 and p_2 are transformed,

$$p_1' = Rp_1 + t, \qquad p_2' = Rp_2 + t,$$

then

$$p_1' - p_2' = R(p_1 - p_2).$$

Because R is orthogonal,

$$\|R(p_1 - p_2)\| = \|p_1 - p_2\|.$$

Hence,

$$\|p_1' - p_2'\| = \|p_1 - p_2\|.$$

This distance-preserving property is what distinguishes rigid motion from more general affine transformations such as scaling or shearing.

A rigid-body pose has six degrees of freedom: three translational and three rotational. Although the matrix representation of T contains sixteen entries, these entries are highly constrained and therefore do not represent sixteen independent parameters.

1.1.3 Rotation Matrices and $SO(3)$

The orientation component of a rigid-body transformation is described by a rotation matrix

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}.$$

The set of valid three-dimensional rotation matrices is the *special orthogonal group*

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

The condition $R^T R = I$ means that the rows and columns of R form orthonormal bases. Consequently, $R^{-1} = R^T$. The determinant condition $\det R = 1$ distinguishes proper rotations from reflections.

A rotation matrix can be interpreted as a change of basis. For example,

$$R_{AB} = \begin{bmatrix} | & | & | \\ [\hat{x}_B]_A & [\hat{y}_B]_A & [\hat{z}_B]_A \\ | & | & | \end{bmatrix}$$

contains, as its columns, the axes of frame B expressed in frame A.

This interpretation makes the transformation

$$p_A = R_{AB} p_B$$

particularly intuitive: the coordinates of a vector expressed in basis B are rewritten using the basis vectors of frame A.

Elementary Rotations

Rotations about the coordinate axes are commonly written as

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix},$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix},$$

and

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

More complicated orientations can be formed by composing these rotations.

Because matrix multiplication is not commutative, $R_1 R_2 \neq R_2 R_1$ in general. The order of rotations is therefore essential.

1.1.4 Euler Angles

A three-dimensional orientation can be parameterized using three successive elementary rotations. Such representations are generally called *Euler-angle representations*.

One common convention in robotics is roll–pitch–yaw, (ϕ, θ, ψ), where ϕ is roll, θ is pitch, and ψ is yaw.

For a commonly used Z-Y-X convention,

$$\boxed{R = R_z(\psi) R_y(\theta) R_x(\phi).}$$

Euler angles are attractive because they provide a minimal and intuitive three-parameter description of orientation.

However, they have important disadvantages. The representation depends on the chosen rotation convention, and certain configurations produce singularities. For example, the roll–pitch–yaw representation becomes singular when the pitch approaches $\theta = \pm\frac{\pi}{2}$.

This phenomenon is commonly associated with *gimbal lock*. The physical orientation remains perfectly valid, but the chosen coordinate parameterization loses one degree of freedom locally.

Euler angles are therefore convenient for visualization and human interpretation but are often avoided internally in estimation, interpolation, and optimization algorithms.

1.1.5 Axis–Angle Representation

Euler's rotation theorem states that any orientation in three dimensions can be represented as a rotation by an angle θ about some unit axis

$$\hat{\omega} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}, \qquad \|\hat{\omega}\| = 1.$$

The orientation can therefore be represented by $(\hat{\omega}, \theta)$.

A compact representation is the rotation vector

$$\phi = \theta \hat{\omega} \in \mathbb{R}^3.$$

The direction of ϕ gives the rotation axis, while $\|\phi\| = \theta$ gives the rotation angle.

Axis–angle representations provide an important connection between ordinary three-dimensional vectors and the nonlinear rotation group $SO(3)$. This relationship becomes particularly useful when introducing the exponential map.

1.1.6 Quaternions

A quaternion provides another representation of three-dimensional orientation. A quaternion can be written as

$$q = \begin{bmatrix} q_w \\ q_x \\ q_y \\ q_z \end{bmatrix} = \begin{bmatrix} q_w \\ q_v \end{bmatrix},$$

where q_w is the scalar component and

$$q_v = \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix}$$

is the vector component.

A quaternion represents a rotation when it has unit norm:

$$\boxed{q_w^2 + q_x^2 + q_y^2 + q_z^2 = 1.}$$

Given an axis–angle representation $(\hat{\omega}, \theta)$, the corresponding unit quaternion is

$$\boxed{q = \begin{bmatrix} \cos(\theta/2) \\ \hat{\omega}\sin(\theta/2) \end{bmatrix}.}$$

The inverse conversion satisfies $\theta = 2\arccos(q_w)$, with the rotation axis determined from the vector component when $\sin(\theta/2) \neq 0$.

The quaternion conjugate is

$$q^* = \begin{bmatrix} q_w \\ -q_v \end{bmatrix}.$$

For a unit quaternion, $q^{-1} = q^*$.

Quaternion Composition

Let

$$q_1 = \begin{bmatrix} w_1 \\ v_1 \end{bmatrix}, \qquad q_2 = \begin{bmatrix} w_2 \\ v_2 \end{bmatrix}.$$

Their quaternion product is

$$q_1 \otimes q_2 = \begin{bmatrix} w_1 w_2 - v_1^T v_2 \\ w_1 v_2 + w_2 v_1 + v_1 \times v_2 \end{bmatrix}.$$

Quaternion multiplication corresponds to rotation composition, and like rotation-matrix multiplication it is generally noncommutative: $q_1 \otimes q_2 \neq q_2 \otimes q_1$.

Unit quaternions are widely used in robotics because they are compact, avoid the singularities of Euler-angle parameterizations, and are convenient for interpolation and numerical integration.

They are not, however, a unique representation. The quaternions q and $-q$ represent exactly the same physical orientation. This is known as the quaternion *double-cover* property.

Quaternion to Rotation Matrix

For $q = [w, x, y, z]^T$, the equivalent rotation matrix is

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

Thus rotation matrices, axis–angle representations, and unit quaternions all describe the same underlying orientation even though their numerical representations differ.

1.1.7 Composition of Rigid-Body Transformations

Coordinate transformations frequently occur in chains.

Suppose T_{AB} maps coordinates from frame B into frame A, and T_{BC} maps coordinates from frame C into frame B.

Then

$$\tilde{p}_A = T_{AB} T_{BC} \tilde{p}_C,$$

so the composite transformation is

$$\boxed{T_{AC} = T_{AB} T_{BC}.}$$

Writing the transformations explicitly,

$$T_{AB} = \begin{bmatrix} R_{AB} & t_{AB} \\ 0 & 1 \end{bmatrix}, \qquad T_{BC} = \begin{bmatrix} R_{BC} & t_{BC} \\ 0 & 1 \end{bmatrix},$$

their product is

$$T_{AC} = \begin{bmatrix} R_{AB} R_{BC} & R_{AB} t_{BC} + t_{AB} \\ 0 & 1 \end{bmatrix}.$$

Therefore, $R_{AC} = R_{AB} R_{BC}$, while

$$t_{AC} = R_{AB} t_{BC} + t_{AB}.$$

The translation t_{BC} must first be rotated into frame A before it can be added to t_{AB}.

This frame-composition rule is fundamental in forward kinematics. For a serial manipulator,

$$T_{0n} = T_{01}T_{12}\cdots T_{(n-1)n}.$$

1.1.8 Inversion of Rigid-Body Transformations

If

$$T_{AB} = \begin{bmatrix} R_{AB} & t_{AB} \\ 0 & 1 \end{bmatrix}$$

maps coordinates from B to A, its inverse maps coordinates from A to B: $T_{BA} = T_{AB}^{-1}$.

Because $R_{AB}^{-1} = R_{AB}^{T}$, the inverse rigid transformation has the particularly simple form

$$\boxed{T_{BA} = T_{AB}^{-1} = \begin{bmatrix} R_{AB}^{T} & -R_{AB}^{T}t_{AB} \\ 0 & 1 \end{bmatrix}.}$$

This result can be understood geometrically. The rotation is undone by applying the transpose, while the translation must first be negated and then expressed in the rotated coordinate frame.

Transformation inversion is used constantly in robotics. For example, if a perception system estimates the pose of a camera relative to the world, T_{WC}, then

$$T_{CW} = T_{WC}^{-1}$$

describes the world relative to the camera.

1.1.9 Lie Groups and Lie Algebras

The sets $SO(3)$ and $SE(3)$ are not ordinary vector spaces. For example, adding two rotation matrices does not generally produce another valid rotation matrix.

Instead, they are examples of *Lie groups*: sets that are simultaneously smooth manifolds and algebraic groups.

This structure is particularly useful because it allows robotic poses to remain on their physically valid nonlinear manifolds while small perturbations are represented using ordinary vectors.

Associated with each Lie group is a *Lie algebra*, which can be interpreted as the tangent space of the group at the identity element.

The Lie algebra associated with $SO(3)$ is denoted $\mathfrak{so}(3)$, and the Lie algebra associated with $SE(3)$ is denoted $\mathfrak{se}(3)$.

These tangent spaces provide local linear representations of rotation and rigid motion.

1.1.10 The Lie Algebra $\mathfrak{so}(3)$

A vector

$$\omega = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \in \mathbb{R}^3$$

can be associated with a skew-symmetric matrix using the *hat operator*:

$$\boxed{\omega^\wedge = [\omega]_\times = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}.}$$

This matrix satisfies $(\omega^\wedge)^T = -\omega^\wedge$.

The set of all such skew-symmetric matrices forms

$$\boxed{\mathfrak{so}(3) = \left\{ \Omega \in \mathbb{R}^{3\times 3} \mid \Omega^T = -\Omega \right\}.}$$

The hat operator is useful because $\omega^\wedge v = \omega \times v$. The inverse operation is commonly called the *vee operator*, with $(\omega^\wedge)^\vee = \omega$.

Thus,

$$\mathbb{R}^3 \longleftrightarrow \mathfrak{so}(3).$$

This allows small rotations and angular velocities to be manipulated using ordinary three-dimensional vectors.

1.1.11 The Lie Algebra $\mathfrak{se}(3)$

A rigid-body motion contains both rotational and translational components.

A six-dimensional vector,

$$\xi = \begin{bmatrix} \rho \\ \phi \end{bmatrix} \in \mathbb{R}^6,$$

contains a translational component $\rho \in \mathbb{R}^3$ and a rotational component $\phi \in \mathbb{R}^3$.

Its matrix representation is

$$\boxed{\xi^\wedge = \begin{bmatrix} \phi^\wedge & \rho \\ 0 & 0 \end{bmatrix} \in \mathfrak{se}(3).}$$

The set of matrices of this form constitutes the Lie algebra $\mathfrak{se}(3)$.

This six-dimensional representation is often called a *twist coordinate*. It provides a local vector description of rigid-body displacement and velocity.

The relationship

$$SE(3) \longleftrightarrow \mathfrak{se}(3) \longleftrightarrow \mathbb{R}^6$$

is fundamental to modern robotics optimization and estimation.

1.1.12 Exponential Map for $SO(3)$

The exponential map converts an element of the Lie algebra into an element of the corresponding Lie group. For rotations,

$$\mathrm{Exp} : \mathfrak{so}(3) \to SO(3).$$

Given a rotation vector $\phi = \theta\hat{\omega}$, the corresponding rotation matrix is

$$R = \exp(\phi^\wedge).$$

Rodrigues' rotation formula provides a closed-form expression:

$$\boxed{R = I + \frac{\sin\theta}{\theta}\phi^\wedge + \frac{1-\cos\theta}{\theta^2}(\phi^\wedge)^2,}$$

where $\theta = \|\phi\|$.

Equivalently, if the unit rotation axis $\hat{\omega}$ is separated from the angle,

$$R = I + \sin\theta\,\hat{\omega}^\wedge + (1-\cos\theta)(\hat{\omega}^\wedge)^2.$$

The exponential map therefore provides a direct relationship between axis–angle vectors and rotation matrices:

$$\boxed{\phi \in \mathbb{R}^3 \quad \xrightarrow{\mathrm{Exp}} \quad R \in SO(3).}$$

For very small rotations, $\theta \approx 0$, the exponential map has the first-order approximation

$$R \approx I + \phi^\wedge.$$

This approximation is important in estimation and optimization because it allows small orientation corrections to be represented linearly.

1.1.13 Logarithmic Map for $SO(3)$

The logarithmic map performs the inverse operation:

$$\mathrm{Log} : SO(3) \to \mathfrak{so}(3).$$

Given a rotation matrix R, the rotation angle satisfies

$$\boxed{\theta = \cos^{-1}\left(\frac{\mathrm{tr}(R) - 1}{2}\right).}$$

Away from singular cases, the corresponding skew-symmetric matrix can be recovered using

$$\phi^{\wedge} = \frac{\theta}{2\sin\theta}(R - R^T).$$

Applying the vee operator gives the rotation vector,

$$\phi = \mathrm{Log}(R)^{\vee}.$$

Thus,

$$R \in SO(3) \quad \overset{\mathrm{Log}}{\longrightarrow} \quad \phi \in \mathbb{R}^3.$$

The logarithmic map is especially useful for defining orientation error.

Given desired and actual orientations R_d and R, a relative rotation can be written as

$$R_{\mathrm{err}} = R_d^T R,$$

and the corresponding local error vector is

$$e_R = \mathrm{Log}(R_{\mathrm{err}})^{\vee}.$$

Unlike direct subtraction of rotation matrices or Euler angles, this error respects the geometry of $SO(3)$.

1.1.14 Exponential and Logarithmic Maps for $SE(3)$

The same idea extends to rigid transformations.

The exponential map

$$\mathrm{Exp} : \mathfrak{se}(3) \to SE(3)$$

maps a six-dimensional local motion vector to a rigid-body transformation:

$$\boxed{T = \operatorname{Exp}(\xi^\wedge).}$$

If

$$\xi = \begin{bmatrix} \rho \\ \phi \end{bmatrix},$$

then

$$\operatorname{Exp}(\xi^\wedge) = \begin{bmatrix} \operatorname{Exp}(\phi^\wedge) & J(\phi)\rho \\ 0 & 1 \end{bmatrix},$$

where $J(\phi)$ is the left Jacobian of $SO(3)$.

For $\theta = \|\phi\|$, the left Jacobian can be written as

$$J(\phi) = I + \frac{1 - \cos\theta}{\theta^2}\phi^\wedge + \frac{\theta - \sin\theta}{\theta^3}(\phi^\wedge)^2.$$

For small ϕ, $J(\phi) \approx I$.

The inverse mapping is

$$\operatorname{Log} : SE(3) \to \mathfrak{se}(3).$$

Thus,

$$\xi = \operatorname{Log}(T)^\vee \in \mathbb{R}^6$$

provides a local six-dimensional representation of a rigid transformation.

This construction is widely useful because optimization algorithms generally operate on vectors in Euclidean space. Instead of directly optimizing the constrained entries of a transformation matrix, an algorithm can optimize a local perturbation $\delta\xi \in \mathbb{R}^6$ and apply it through the exponential map.

For example,

$$T_{\text{new}} = \operatorname{Exp}(\delta\xi^\wedge)T.$$

The resulting T_{new} remains a valid element of $SE(3)$.

1.1.15 Relative Poses and Pose Error

Many robotics problems depend not on absolute poses but on the relative transformation between two poses.

Suppose $T_A, T_B \in SE(3)$. The transformation from A to B can be written as

$$T_{AB} = T_A^{-1} T_B.$$

A local six-dimensional representation of this relative pose is

$$\xi_{AB} = \operatorname{Log}\left(T_A^{-1} T_B\right)^{\vee}.$$

Similarly, if T_d is a desired pose and T is the actual pose, a geometrically meaningful pose error can be formed using

$$\boxed{e = \operatorname{Log}\left(T_d^{-1} T\right)^{\vee}.}$$

The error is represented as a six-dimensional vector rather than by subtracting transformation matrices directly.

This construction appears throughout modern robotics, including pose-graph optimization, visual odometry, SLAM, calibration, trajectory optimization, and nonlinear control.

1.1.16 Summary

Three-dimensional robot geometry is fundamentally the geometry of rigid-body motion.

Rotation matrices belong to $SO(3)$, while rigid transformations belong to $SE(3)$. A pose transformation has the form

$$T = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix},$$

and transforms coordinates according to $p' = Rp + t$.

Multiple coordinate transformations compose through matrix multiplication,

$$T_{AC} = T_{AB} T_{BC},$$

while inversion is given by

$$T^{-1} = \begin{bmatrix} R^T & -R^T t \\ 0 & 1 \end{bmatrix}.$$

Orientations can be represented using rotation matrices, Euler angles, axis–angle coordinates, or quaternions. Each representation has different numerical and conceptual advantages.

The Lie algebras $\mathfrak{so}(3)$ and $\mathfrak{se}(3)$ provide local vector representations of rotation and rigid-body motion. Exponential and logarithmic maps connect these local representations to the nonlinear groups:

$$\mathbb{R}^3 \leftrightarrow \mathfrak{so}(3) \leftrightarrow SO(3),$$

and

$$\mathbb{R}^6 \leftrightarrow \mathfrak{se}(3) \leftrightarrow SE(3).$$

These mathematical tools provide the foundation for the forward kinematics, inverse kinematics, Jacobians, state estimation, visual geometry, SLAM, trajectory optimization, and manipulation methods developed in the remainder of this chapter.

1.2 Robot Kinematics

Robot kinematics describes the geometry of motion without considering the forces or torques that produce that motion. The central variables are the robot's *configuration*, usually described by its joint coordinates, and the position and orientation of points or coordinate frames attached to the robot.

For a manipulator with n joints, the configuration is represented by the generalized-coordinate vector

$$q = \begin{bmatrix} q_1 & q_2 & \cdots & q_n \end{bmatrix}^T.$$

For a revolute joint, q_i is typically an angle. For a prismatic joint, q_i is a linear displacement.

The complete set of allowable configurations forms the robot's *configuration space*, commonly denoted

$$\mathcal{Q}.$$

A robot's kinematic model establishes relationships between configuration space and quantities expressed in physical or task space. The most basic relationship is the forward-kinematics map

$$T_{WE} = f(q),$$

which determines the pose of the end effector E relative to a world or base frame W.

Kinematics is commonly studied at two related levels. *Position kinematics* relates joint configuration to pose, while *differential kinematics* relates joint velocities to Cartesian linear and angular velocities. These relationships provide the foundation for inverse kinematics, motion planning, manipulation, and robot control.

1.2.1 Kinematic Chains

A robot mechanism consists of rigid links connected by joints. Each joint constrains the relative motion between neighboring links.

The most common joint types in manipulation are revolute and prismatic joints.

A revolute joint permits rotation about a fixed axis,

$$q_i = \theta_i,$$

while a prismatic joint permits translation along a fixed axis,

$$q_i = d_i.$$

A sequence of links and joints forms a *kinematic chain*.

In a serial manipulator, each link is connected to the next in a single chain,

$$\text{base} \to \text{joint } 1 \to \text{link } 1 \to \cdots \to \text{joint } n \to \text{end effector}.$$

Examples include conventional six-axis industrial manipulators and many seven-degree-of-freedom collaborative arms.

Tree-structured mechanisms contain branches. Humanoid robots are a common example: the torso connects to several chains corresponding to the arms, legs, and head.

Closed-chain mechanisms contain one or more kinematic loops. Parallel manipulators and mechanisms in which two robot hands rigidly hold the same object are examples.

Serial chains are particularly convenient mathematically because the pose of each successive link can be computed recursively through transformation composition.

Suppose frame $\{i\}$ is attached to link i. Let

$$^{i-1}T_i(q_i)$$

describe the transformation from frame i to frame $i-1$. The pose of the end effector is obtained by multiplying the transformations along the chain:

$$^0T_E(q) = {}^0T_1(q_1)\,{}^1T_2(q_2)\cdots{}^{n-1}T_E(q_n).$$

This product is the basis of forward kinematics.

1.2.2 Forward Kinematics

Forward kinematics determines the pose of a robot link or end effector from the robot configuration.

For an n-joint manipulator,

$$q = [q_1, \ldots, q_n]^T,$$

and the forward-kinematics function is

$$\boxed{T_{WE} = f(q).}$$

The transformation

$$T_{WE} = \begin{bmatrix} R_{WE} & p_{WE} \\ 0 & 1 \end{bmatrix}$$

contains both the end-effector orientation

$$R_{WE} \in SO(3)$$

and its position

$$p_{WE} \in \mathbb{R}^3.$$

For a serial manipulator,

$$T_{WE}(q) = T_{W0}T_{01}(q_1)T_{12}(q_2)\cdots T_{n-1,E}(q_n).$$

Each transformation depends only on the geometry of the corresponding link and the associated joint variable.

Forward kinematics is normally deterministic: once q is known, the pose of every link is determined.

Example: Two-Link Planar Manipulator

Consider two revolute links with lengths l_1 and l_2, and joint angles q_1 and q_2.

The end-effector position is

$$x = l_1 \cos q_1 + l_2 \cos(q_1 + q_2),$$

$$y = l_1 \sin q_1 + l_2 \sin(q_1 + q_2).$$

The end-effector orientation in the plane is

$$\phi = q_1 + q_2.$$

Thus,

$$f(q) = \begin{bmatrix} l_1 \cos q_1 + l_2 \cos(q_1 + q_2) \\ l_1 \sin q_1 + l_2 \sin(q_1 + q_2) \\ q_1 + q_2 \end{bmatrix}.$$

Even for this simple robot, the mapping is nonlinear because the joint angles enter through sine and cosine functions.

For a general three-dimensional manipulator, the same principle applies, but homogeneous transformations are used to compose arbitrary rotations and translations.

1.2.3 Denavit–Hartenberg Parameters

For a serial manipulator, one must specify the rigid transformation between every pair of neighboring joint frames. The Denavit–Hartenberg (DH) convention provides a systematic method for doing this using four parameters per link.

Under the standard DH convention, the transformation from frame $i-1$ to frame i is written as

$$^{i-1}T_i = R_z(\theta_i)T_z(d_i)T_x(a_i)R_x(\alpha_i).$$

The four DH parameters are

$$\theta_i, \qquad d_i, \qquad a_i, \qquad \alpha_i.$$

Their geometric meanings are:

- θ_i: rotation about the previous z-axis,
- d_i: translation along the previous z-axis,
- a_i: translation along the new x-axis, often called the link length,
- α_i: rotation about the new x-axis, often called the link twist.

The resulting homogeneous transformation is

$$^{i-1}T_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

For a revolute joint,

$$\theta_i$$

is the joint variable, while the remaining DH parameters are constant.

For a prismatic joint,

$$d_i$$

is the joint variable.

Once a DH table has been constructed, forward kinematics follows directly from

$$^0T_n = {}^0T_1\,{}^1T_2 \cdots {}^{n-1}T_n.$$

The principal value of DH parameters is not that they change the underlying geometry, but that they provide a standardized procedure for assigning coordinate frames and constructing transformation matrices.

Modern robotics software may instead use direct $SE(3)$ transformations, product-of-exponentials formulations, or URDF-defined joint transforms. Nevertheless, DH notation remains useful for analytical derivation and for understanding classical manipulator kinematics.

1.2.4 Inverse Kinematics

Inverse kinematics solves the reverse problem.

Given a desired end-effector pose

$$T_{WE}^{\star},$$

we seek a robot configuration

$$q^{\star}$$

satisfying

$$\boxed{f(q^{\star}) = T_{WE}^{\star}.}$$

Unlike forward kinematics, inverse kinematics is generally not guaranteed to have exactly one solution.

A target pose may have

- no solution,
- one solution,
- several discrete solutions,
- infinitely many solutions.

A pose has no solution if it lies outside the robot's reachable workspace or violates joint constraints.

Several configurations may produce the same end-effector pose. A planar two-link arm, for example, may often reach the same point using an 'elbow-up" or 'elbow-down" configuration.

For a redundant manipulator, there may be infinitely many configurations satisfying the same Cartesian task.

Analytical Inverse Kinematics

For manipulators with sufficiently simple geometry, inverse kinematics can sometimes be solved analytically.

For the two-link planar arm,

$$x = l_1 \cos q_1 + l_2 \cos(q_1 + q_2),$$

$$y = l_1 \sin q_1 + l_2 \sin(q_1 + q_2).$$

Define

$$r^2 = x^2 + y^2.$$

Using the law of cosines,

$$\cos q_2 = \frac{r^2 - l_1^2 - l_2^2}{2 l_1 l_2}.$$

Thus,

$$q_2 = \pm \cos^{-1}\left(\frac{r^2 - l_1^2 - l_2^2}{2 l_1 l_2}\right).$$

The two signs correspond to different kinematic branches.

Once q_2 is known,

$$q_1 = \operatorname{atan2}(y, x) - \operatorname{atan2}\left(l_2 \sin q_2, l_1 + l_2 \cos q_2\right).$$

Analytical solutions are fast and precise but depend strongly on manipulator geometry and become difficult to derive for general robots.

Numerical Inverse Kinematics

General inverse kinematics is often solved iteratively.

Starting from an initial configuration q_k,

$$q_{k+1} = q_k + \Delta q_k.$$

Let

$$e_k$$

denote the current task-space pose error.

For a sufficiently small change in configuration,

$$e_k \approx J(q_k)\Delta q_k.$$

A common update is

$$\Delta q_k = J^\dagger(q_k) e_k,$$

so that

$$\boxed{q_{k+1} = q_k + J^\dagger(q_k) e_k.}$$

Here $J^\dagger$ is the Moore–Penrose pseudoinverse.

In practice, the pose error should respect the geometry of $SE(3)$. For example, if $T(q_k)$ is the current pose and T_d is the desired pose, one possible local error representation is

$$e_k = \operatorname{Log}\left(T(q_k)^{-1} T_d\right)^\vee.$$

Numerical inverse kinematics is flexible and can incorporate joint limits, collision constraints, preferred postures, and regularization. Its solution, however, depends on initialization and may converge to a local solution or fail near singular configurations.

1.2.5 Differential Kinematics

Forward kinematics describes position and orientation,

$$x = f(q).$$

Differential kinematics describes how small changes in configuration affect the task variables.

Differentiating with respect to time gives

$$\dot{x} = \frac{\partial f}{\partial q} \dot{q}.$$

Define the Jacobian

$$J(q) = \frac{\partial f}{\partial q}.$$

Then

$$\boxed{\dot{x} = J(q)\dot{q}.}$$

For a complete Cartesian pose, velocity is represented by a six-dimensional twist

$$\xi = \begin{bmatrix} v \\ \omega \end{bmatrix},$$

where

$$v \in \mathbb{R}^3$$

is translational velocity and

$$\omega \in \mathbb{R}^3$$

is angular velocity.

The differential kinematic relationship becomes

$$\boxed{\xi = J(q)\dot{q}.}$$

For an n-joint manipulator,

$$J(q) \in \mathbb{R}^{6\times n}.$$

Differentiating once more gives the acceleration relationship

$$\boxed{\dot{\xi} = J(q)\ddot{q} + \dot{J}(q,\dot{q})\dot{q}.}$$

The term

$$J\ddot{q}$$

describes Cartesian acceleration produced directly by joint acceleration, while

$$\dot{J}\dot{q}$$

accounts for the changing geometry of the manipulator as it moves.

1.2.6 The Manipulator Jacobian

The manipulator Jacobian is one of the most important objects in robot kinematics.

Each column of

$$J = \begin{bmatrix} J_1 & J_2 & \cdots & J_n \end{bmatrix}$$

describes how velocity of one joint contributes to the Cartesian velocity of the end effector.

Consider a revolute joint whose axis is z_i, with a point on the joint axis located at p_i. Let the end-effector position be p_E.

Rotation about that axis generates translational velocity

$$v_i = z_i \times (p_E - p_i)$$

and angular velocity

$$\omega_i = z_i.$$

Therefore, the corresponding Jacobian column is

$$\boxed{J_i = \begin{bmatrix} z_i \times (p_E - p_i) \\ z_i \end{bmatrix}.}$$

For a prismatic joint, motion along z_i produces translation but no angular velocity:

$$\boxed{J_i = \begin{bmatrix} z_i \\ 0 \end{bmatrix}.}$$

The complete Jacobian is formed by stacking these columns.

Geometric and Analytical Jacobians

It is useful to distinguish the *geometric Jacobian* from an *analytical Jacobian*.

The geometric Jacobian maps joint velocity to physical linear and angular velocity:

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = J_g(q)\dot{q}.$$

An analytical Jacobian may instead relate joint velocity to the derivative of some chosen pose coordinates, such as Euler angles:

$$\begin{bmatrix} \dot{p} \\ \dot{\phi} \end{bmatrix} = J_a(q)\dot{q}.$$

Because Euler-angle derivatives are not equal to angular velocity in general,

$$\dot{\phi} \neq \omega,$$

the two Jacobians are generally different.

The geometric Jacobian is typically preferred when reasoning directly about rigid-body motion because the angular component is the physical angular velocity rather than the derivative of a particular orientation parameterization.

1.2.7 Inverse Differential Kinematics

Suppose a desired end-effector twist is given by

$$\xi_d.$$

We seek a joint velocity satisfying

$$J\dot{q} = \xi_d.$$

If J is square and nonsingular,

$$\dot{q} = J^{-1}\xi_d.$$

More generally, the Moore–Penrose pseudoinverse is used:

$$\boxed{\dot{q} = J^{\dagger}\xi_d.}$$

When the system is underdetermined or overdetermined, the pseudoinverse provides a least-squares solution.

For a full-row-rank Jacobian with more columns than rows,

$$J^{\dagger} = J^T(JJ^T)^{-1}.$$

For a full-column-rank Jacobian with more rows than columns,

$$J^{\dagger} = (J^TJ)^{-1}J^T.$$

The singular-value decomposition provides the most general numerical construction.

If

$$J = U\Sigma V^T,$$

then

$$J^{\dagger} = V\Sigma^{\dagger}U^T,$$

where nonzero singular values are inverted in $\Sigma^{\dagger}$.

This representation is also particularly useful for understanding singularities.

1.2.8 Jacobian Singularities

A kinematic singularity occurs when the Jacobian loses rank:

$$\boxed{\operatorname{rank}(J) < \min(m, n),}$$

where m is the task dimension and n is the number of joints.

The velocity equation

$$\xi = J\dot{q}$$

shows the geometric consequence directly. The columns of J describe the Cartesian velocity directions that can be produced by the joints. If those columns become linearly dependent, at least one independent task-space direction is lost.

Near a singularity, attempting to solve

$$\dot{q} = J^{\dagger}\xi$$

can require extremely large joint velocities for even moderate Cartesian velocities.

Using the singular-value decomposition,

$$J = U\Sigma V^{T},$$

a singularity occurs when one or more singular values approach zero.

If

$$\sigma_{\min} \to 0,$$

then inversion produces the factor

$$\frac{1}{\sigma_{\min}} \to \infty.$$

This explains why joint velocities become poorly conditioned near singular configurations.

Damped Least Squares

A commonly used regularized inverse is

$$\boxed{J_{\lambda}^{\dagger} = J^{T}\left(JJ^{T} + \lambda^{2}I\right)^{-1}.}$$

The damping parameter

$$\lambda > 0$$

prevents the inverse from becoming arbitrarily large.

Increasing λ improves numerical robustness but sacrifices exact Cartesian tracking.

Damped least squares is therefore useful near singularities or whenever the Jacobian is poorly conditioned.

1.2.9 Manipulability

The Jacobian also describes how joint-space motion maps into task-space motion.

Suppose joint velocities satisfy

$$\|\dot{q}\| \leq 1.$$

Because

$$\xi = J\dot{q},$$

the resulting set of Cartesian velocities forms an ellipsoid known as the *manipulability ellipsoid*.

Its principal directions are given by the left singular vectors of J, and the lengths of the corresponding axes are proportional to the singular values.

Large singular values indicate directions in which Cartesian velocity can be produced easily. Small singular values indicate directions requiring comparatively large joint motion.

A common scalar manipulability measure is

$$\boxed{\mu(q) = \sqrt{\det\left(JJ^T\right).}}$$

As the manipulator approaches a singularity,

$$\mu(q) \to 0.$$

Manipulability is therefore useful both as an analysis tool and as a secondary optimization objective.

1.2.10 Redundancy and Null-Space Motion

A robot is kinematically redundant with respect to a task when it has more controllable joint degrees of freedom than are required by the task.

For example, a seven-joint manipulator controlling a six-dimensional end-effector pose has one degree of kinematic redundancy under ordinary nonsingular conditions.

The velocity equation

$$J\dot{q} = \xi$$

then has infinitely many solutions.

The general pseudoinverse solution can be written as

$$\boxed{\dot{q} = J^{\dagger}\xi + \left(I - J^{\dagger}J\right)z.}$$

The first term

$$J^{\dagger}\xi$$

produces the desired Cartesian motion.

The second term lies in the null space of J. Define

$$N = I - J^{\dagger}J.$$

Then

$$JN = 0,$$

so

$$JNz = 0.$$

Consequently,

$$\dot{q}_{\text{null}} = Nz$$

changes the robot configuration without changing the instantaneous end-effector twist.

This additional freedom can be used to accomplish secondary objectives such as

- avoiding joint limits,
- avoiding self-collision,
- maintaining distance from obstacles,
- maximizing manipulability,
- maintaining a preferred posture.

Suppose a scalar objective

$$h(q)$$

should be increased while preserving the primary task. One possible choice is

$$z = k\nabla_q h(q),$$

giving

$$\dot{q} = J^{\dagger}\xi + k\left(I - J^{\dagger}J\right)\nabla_q h(q).$$

The primary task is preserved instantaneously because the secondary gradient is projected into the Jacobian null space.

1.2.11 Velocity and Force Relationships

The Jacobian has a dual role in robot mechanics.

It maps joint velocities into Cartesian velocities:

$$\boxed{\xi = J\dot{q}.}$$

Its transpose maps Cartesian forces and moments into generalized joint forces.

Let the end effector experience a wrench

$$F = \begin{bmatrix} f \\ \tau_c \end{bmatrix} \in \mathbb{R}^6.$$

Consider an infinitesimal joint displacement

$$\delta q.$$

The corresponding Cartesian displacement is

$$\delta x = J\delta q.$$

The work performed by the Cartesian wrench is

$$\delta W = F^T \delta x.$$

Substituting the differential kinematics gives

$$\delta W = F^T J \delta q.$$

The same work expressed in joint coordinates is

$$\delta W = \tau^T \delta q.$$

Because the two expressions must agree for arbitrary δq,

$$\tau^T = F^T J,$$

and therefore

$$\boxed{\tau = J^T F.}$$

Thus,

$$J: \quad \text{joint velocity} \rightarrow \text{Cartesian velocity},$$

whereas

$$J^T : \quad \text{Cartesian wrench} \rightarrow \text{joint torque.}$$

This duality follows from conservation of instantaneous power:

$$\tau^T \dot{q} = F^T \xi.$$

Using

$$\xi = J\dot{q},$$

we obtain

$$F^T \xi = F^T J\dot{q} = (J^T F)^T \dot{q}.$$

Hence,

$$\tau = J^T F.$$

This relationship appears throughout manipulation, force control, impedance control, contact modeling, and operational-space control.

1.2.12 Workspace and Reachability

The *workspace* of a robot is the set of end-effector poses or positions that can be achieved by some valid configuration.

For a position-only task, the reachable workspace may be defined as

$$\boxed{\mathcal{W}_p = \{p \mid p = p(q),\ q \in \mathcal{Q}\}\,.}$$

For full pose,

$$\mathcal{W} = \{T \in SE(3) \mid T = f(q),\ q \in \mathcal{Q}\}\,.$$

Two related notions are commonly distinguished.

The *reachable workspace* is the set of points that the end effector can reach in at least one orientation.

The *dexterous workspace* is the set of points at which the end effector can achieve a specified set of orientations, or in the strongest definition, all possible orientations.

Consequently,

$$\mathcal{W}_{\text{dexterous}} \subseteq \mathcal{W}_{\text{reachable}}.$$

Workspace depends on

- link lengths,
- joint types,
- joint limits,
- mechanical interference,
- self-collision constraints,
- environmental obstacles,
- orientation requirements.

Example: Two-Link Planar Arm

For a two-link planar arm,

$$x = l_1 \cos q_1 + l_2 \cos(q_1 + q_2),$$

$$y = l_1 \sin q_1 + l_2 \sin(q_1 + q_2).$$

The distance from the base is

$$r = \sqrt{x^2 + y^2}.$$

Assuming unrestricted revolute joints, the endpoint is reachable only if

$$\boxed{|l_1 - l_2| \leq r \leq l_1 + l_2.}$$

The outer boundary

$$r = l_1 + l_2$$

occurs when the links are fully extended.

The inner boundary

$$r = |l_1 - l_2|$$

occurs when the links fold back against each other.

These boundaries are also associated with singular configurations because the two links become collinear.

This illustrates an important relationship between workspace geometry and the Jacobian: singularities frequently occur at workspace boundaries, although singular configurations can also occur inside a robot's workspace.

1.2.13 Reachability and Inverse Kinematics

Reachability determines whether an inverse-kinematics problem has a solution.

Given

$$T_d,$$

inverse kinematics asks for

$$q$$

such that

$$f(q) = T_d.$$

A solution exists only if

$$T_d \in \mathcal{W}.$$

Even if the desired position lies inside the positional workspace, a complete pose may still be unreachable because the required orientation cannot be produced at that position.

Likewise, a mathematically reachable pose may be practically infeasible because the corresponding configuration violates

$$q_{\min} \leq q \leq q_{\max},$$

collision constraints, or other mechanical restrictions.

It is therefore useful to distinguish geometric reachability from practical feasibility.

1.2.14 Kinematic Constraints and Joint Limits

Joint coordinates are normally constrained:

$$q_i^{\min} \leq q_i \leq q_i^{\max}.$$

Velocity limits may also apply:

$$|\dot{q}_i| \leq \dot{q}_i^{\max}.$$

These constraints matter even when a Cartesian target is theoretically reachable.

For redundant manipulators, null-space motion provides one mechanism for remaining away from joint limits while maintaining an end-effector task.

For example, define a cost that penalizes displacement from the center of each joint range:

$$H(q) = \frac{1}{2} \sum_{i=1}^{n} \left(\frac{q_i - q_i^{\text{mid}}}{q_i^{\max} - q_i^{\min}} \right)^2 .$$

A secondary velocity can be selected as

$$z = -k \nabla H(q),$$

and projected into the null space:

$$\dot{q} = J^{\dagger} \xi - k \left(I - J^{\dagger} J \right) \nabla H(q).$$

The end-effector task is maintained while the internal posture is driven toward configurations with greater joint-limit margin.

1.2.15 Summary

Robot kinematics relates internal robot configuration to externally meaningful motion.

Forward kinematics defines the mapping

$$q \longrightarrow T_{WE},$$

typically through a chain of rigid transformations,

$${}^{0}T_{n} = {}^{0}T_{1}\, {}^{1}T_{2} \cdots {}^{n-1}T_{n}.$$

Denavit–Hartenberg parameters provide a classical systematic representation of these link-to-link transformations.

Inverse kinematics solves the reverse problem,

$$T_d \longrightarrow q,$$

and may admit zero, one, several, or infinitely many solutions.

Differential kinematics describes local motion through the manipulator Jacobian:

$$\xi = J(q)\dot{q}.$$

The pseudoinverse provides a general inverse velocity mapping,

$$\dot{q} = J^{\dagger} \xi,$$

while redundant robots admit additional null-space motion,

$$\dot{q} = J^{\dagger} \xi + \left(I - J^{\dagger} J \right) z.$$

When the Jacobian loses rank, the robot enters a kinematic singularity. Near such configurations, some Cartesian directions become difficult or impossible to generate and inverse differential kinematics becomes poorly conditioned.

The same Jacobian also relates Cartesian wrenches and joint torques:

$$\tau = J^T F.$$

Finally, the robot workspace describes which positions and poses can be generated by valid configurations. Reachability, joint limits, singularities, and redundancy therefore connect the global geometry of the robot with its local differential behavior.

Together, these ideas provide the mathematical foundation needed for the dynamics, planning, and control methods that follow.

1.3 Robot Dynamics

Kinematics describes how a robot can move without considering the physical causes of that motion. Dynamics extends this description by relating motion to forces and torques.

For a rigid-link manipulator, the configuration is described by generalized coordinates

$$q = \begin{bmatrix} q_1 & q_2 & \cdots & q_n \end{bmatrix}^T,$$

together with the corresponding joint velocities

$$\dot{q}$$

and accelerations

$$\ddot{q}.$$

The central problem of robot dynamics is to understand the relationship

$$(q, \dot{q}, \ddot{q}) \quad \longleftrightarrow \quad \tau,$$

where

$$\tau = \begin{bmatrix} \tau_1 & \cdots & \tau_n \end{bmatrix}^T$$

contains the generalized forces applied at the joints. For revolute joints, these generalized forces are torques; for prismatic joints, they are linear forces.

For a rigid manipulator, this relationship is commonly written in the form

$$\boxed{M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) + \tau_f = \tau + J^T(q)F_{\text{ext}}.}$$

This equation summarizes several physical effects:

$$\text{inertia} + \text{velocity-dependent forces} + \text{gravity} + \text{friction} = \text{actuator forces} + \text{external forces}.$$

Understanding where these terms come from is the central subject of rigid-body dynamics.

1.3.1 Rigid-Body Dynamics

A rigid body has both translational and rotational motion.

For a body of mass m whose center of mass has position p_C, Newton's second law gives

$$F = m\ddot{p}_C.$$

Rotational motion is governed by the corresponding angular equation. In a body-fixed representation,

$$\tau = I\dot{\omega} + \omega \times I\omega,$$

where

$$I \in \mathbb{R}^{3\times 3}$$

is the rotational inertia tensor and

$$\omega \in \mathbb{R}^3$$

is angular velocity.

The translational and rotational equations together describe the dynamics of one rigid body.

A robot manipulator, however, contains many rigid bodies connected through joints. Motion of one link changes the motion of all downstream links, and the forces generated by one link are transmitted through the kinematic chain.

The dynamics of a manipulator therefore cannot generally be treated as a collection of independent joint equations.

Instead, the acceleration of the robot is coupled through a configuration-dependent inertia matrix.

1.3.2 Newton–Euler Formulation

One classical method for deriving robot dynamics is the Newton–Euler formulation.

The basic idea is to apply Newton's translational equation and Euler's rotational equation to every link in the robot.

For link i,

$$F_i = m_i a_{C_i},$$

where a_{C_i} is the acceleration of the link's center of mass.

The rotational equation is

$$N_i = I_i\dot{\omega}_i + \omega_i \times I_i\omega_i.$$

Here,

$$N_i$$

is the net moment acting on the link.

For a serial manipulator, the Newton–Euler algorithm is naturally organized as two recursive passes.

Forward Recursion

The forward pass propagates motion from the base toward the end effector:

$$\text{base} \to \text{link } 1 \to \text{link } 2 \to \cdots \to \text{link } n.$$

Using the joint positions, velocities, and accelerations, the recursion computes quantities such as

$$\omega_i, \qquad \dot{\omega}_i, \qquad a_i, \qquad a_{C_i}.$$

These determine the inertial force

$$F_i = m_i a_{C_i}$$

and inertial moment

$$N_i = I_i \dot{\omega}_i + \omega_i \times I_i \omega_i.$$

Backward Recursion

The backward pass propagates forces and moments from the end effector back toward the base:

$$\text{end effector} \to \text{link } n \to \cdots \to \text{link } 1.$$

Forces transmitted from downstream links are combined with the inertial and gravitational forces acting on each link.

For a revolute joint, the required joint torque is obtained by projecting the transmitted moment onto the joint axis z_i:

$$\tau_i = z_i^T n_i.$$

For a prismatic joint,

$$\tau_i = z_i^T f_i.$$

The Newton–Euler method therefore computes joint forces from link-level force and moment balances.

Its recursive structure makes it computationally efficient, particularly for inverse dynamics.

1.3.3 Lagrangian Mechanics

A second major approach derives the equations of motion from energy rather than directly summing forces.

Define the Lagrangian

$$\boxed{L(q, \dot{q}) = K(q, \dot{q}) - V(q),}$$

where

$$K$$

is the total kinetic energy and

$$V$$

is the total potential energy.

For each generalized coordinate q_i, the Euler–Lagrange equation is

$$\boxed{\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = \tau_i.}$$

For a robot with n generalized coordinates, this produces n coupled equations.

The kinetic energy of the manipulator can be written in the quadratic form

$$\boxed{K = \frac{1}{2}\dot{q}^T M(q)\dot{q}.}$$

This expression introduces the configuration-dependent inertia matrix $M(q)$.

The gravitational potential energy can be written as

$$V(q),$$

and its gradient produces the generalized gravity forces:

$$\boxed{g(q) = \frac{\partial V(q)}{\partial q}.}$$

Substituting these expressions into the Euler–Lagrange equations produces the familiar manipulator equation

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) = \tau$$

in the ideal rigid, frictionless case.

The Newton–Euler and Lagrangian approaches therefore describe the same physical system from different perspectives. Newton–Euler mechanics emphasizes forces and moments on individual bodies, whereas Lagrangian mechanics emphasizes the kinetic and potential energy of the complete system.

1.3.4 Manipulator Equations of Motion

The dynamics of a rigid-link robot are commonly written as

$$\boxed{M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) + \tau_f = \tau + J^T(q)F_{\text{ext}}.}$$

Each term has a distinct physical meaning.

The term

$$M(q)\ddot{q}$$

represents inertial forces associated with accelerating the links.

The term

$$C(q,\dot{q})\dot{q}$$

contains velocity-dependent Coriolis and centrifugal effects.

The vector

$$g(q)$$

contains generalized forces caused by gravity.

The vector

$$\tau_f$$

represents joint friction or other dissipative effects.

The vector

$$\tau$$

contains the actuator torques applied by the motors.

Finally,

$$J^T(q)F_{\text{ext}}$$

represents generalized joint torques produced by an external Cartesian wrench.

The robot dynamics can therefore be interpreted as a generalized force balance:

$$\text{required generalized forces} = \text{actuator forces} + \text{external loading}.$$

1.3.5 The Inertia Matrix

The matrix

$$M(q) \in \mathbb{R}^{n \times n}$$

is the joint-space inertia matrix, sometimes called the mass matrix.

It describes how difficult it is to accelerate the robot in different joint-space directions.

Because the geometry of the robot changes as the joints move,

$$M = M(q).$$

A configuration with the arm fully extended, for example, may have substantially different rotational inertia about a base joint than a configuration in which the arm is folded close to the body.

The kinetic energy is

$$\boxed{K = \frac{1}{2}\dot{q}^T M(q)\dot{q}.}$$

This relation provides an important physical interpretation of the mass matrix.

For a conventional unconstrained rigid manipulator, $M(q)$ is symmetric:

$$\boxed{M(q) = M^T(q),}$$

and positive definite:

$$\boxed{x^T M(q) x > 0}$$

for every nonzero vector x.

Positive definiteness means that every nonzero joint velocity corresponds to positive kinetic energy.

Coupling Between Joints

The inertia matrix is generally not diagonal.

For a two-joint manipulator,

$$M(q) = \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix}.$$

The off-diagonal terms

$$M_{12}, \qquad M_{21}$$

represent inertial coupling.

Consequently, accelerating one joint may require torque at another joint even if that second joint is not itself accelerating.

This coupling is one reason manipulator dynamics are substantially more complicated than independent scalar motor equations.

1.3.6 Coriolis and Centrifugal Terms

The term

$$C(q, \dot{q})\dot{q}$$

contains generalized forces caused by motion through a configuration-dependent mechanical system.

These effects arise because the inertia matrix itself changes with configuration.

The terminology usually distinguishes two related phenomena.

Centrifugal terms typically contain squared joint velocities such as

$$\dot{q}_i^2,$$

while *Coriolis terms* involve products of different joint velocities,

$$\dot{q}_i \dot{q}_j.$$

Together they are represented compactly as

$$C(q, \dot{q})\dot{q}.$$

One construction uses the Christoffel symbols

$$c_{ijk} = \frac{1}{2}\left(\frac{\partial M_{ij}}{\partial q_k} + \frac{\partial M_{ik}}{\partial q_j} - \frac{\partial M_{jk}}{\partial q_i}\right),$$

from which

$$C_{ij} = \sum_k c_{ijk}\dot{q}_k.$$

Then

$$C(q, \dot{q})\dot{q}$$

collects the velocity-dependent generalized forces.

An important subtlety is that the matrix $C(q, \dot{q})$ itself is not uniquely defined. Different matrices C can produce the same vector

$$C(q, \dot{q})\dot{q}.$$

The physically meaningful quantity is therefore usually the complete velocity-dependent force vector rather than an individual matrix entry.

1.3.7 Gravity

Gravity acts on the mass of every robot link.

For link i with center-of-mass position

$$p_{C_i}(q),$$

the gravitational potential energy can be written as

$$V_i(q) = -m_i g^T p_{C_i}(q),$$

depending on the sign convention used for the gravity vector.

The total potential energy is

$$V(q) = \sum_i V_i(q).$$

The generalized gravity torque is

$$\boxed{g(q) = \frac{\partial V(q)}{\partial q}.}$$

The required gravitational torque therefore changes as the manipulator moves.

A horizontal arm held far from the base may require significant torque, while the same arm hanging vertically may require much less torque about some joints.

1.3.8 Gravity Compensation

A robot holding a stationary configuration satisfies approximately

$$\dot{q} = 0, \qquad \ddot{q} = 0.$$

Ignoring friction and external forces, the manipulator dynamics reduce to

$$g(q) = \tau.$$

Thus the torque required simply to hold the robot against gravity is

$$\boxed{\tau_{\text{grav}} = g(q).}$$

A controller may therefore explicitly add gravity compensation:

$$\tau = \tau_{\text{feedback}} + g(q).$$

For example, a PD joint controller becomes

$$\boxed{\tau = K_P(q_d - q) + K_D(\dot{q}_d - \dot{q}) + g(q).}$$

The feedback controller then primarily corrects deviations from the desired trajectory rather than continuously supplying the static torque needed to support the robot.

If the gravity model is accurate, this can significantly improve tracking and reduce the feedback gains required for a given performance level.

1.3.9 External Forces and Generalized Torques

Suppose the end effector experiences an external wrench

$$F_{\text{ext}} = \begin{bmatrix} f_{\text{ext}} \\ \tau_{\text{ext}}^{C} \end{bmatrix}.$$

The corresponding generalized joint torque is

$$\boxed{\tau_{\text{ext}} = J^T(q) F_{\text{ext}}.}$$

This relationship follows from virtual work.

For an infinitesimal joint displacement,

$$\delta x = J \delta q.$$

The Cartesian work is

$$\delta W = F^T \delta x = F^T J \delta q.$$

The same work written in joint coordinates is

$$\delta W = \tau^T \delta q.$$

Because the expressions must agree for arbitrary δq,

$$\boxed{\tau = J^T F.}$$

Thus the manipulator Jacobian plays two complementary roles:

$$\dot{x} = J\dot{q}$$

maps joint velocities into Cartesian velocities, while

$$\tau = J^T F$$

maps Cartesian forces into generalized joint forces.

1.3.10 Forward Dynamics

The *forward dynamics* problem asks:

Given the robot state and applied forces, what acceleration will result?

Starting from

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) + \tau_f = \tau + J^T F_{\text{ext}},$$

we solve for acceleration:

$$\boxed{\ddot{q} = M^{-1}(q)\left[\tau + J^T F_{\text{ext}} - C(q,\dot{q})\dot{q} - g(q) - \tau_f\right].}$$

Forward dynamics is fundamental to simulation.

Given

$$q_t, \qquad \dot{q}_t, \qquad \tau_t,$$

the acceleration

$$\ddot{q}_t$$

is computed from the dynamics.

The state can then be integrated forward in time:

$$\dot{q}_{t+\Delta t} \approx \dot{q}_t + \ddot{q}_t \Delta t,$$

$$q_{t+\Delta t} \approx q_t + \dot{q}_t \Delta t + \frac{1}{2}\ddot{q}_t \Delta t^2.$$

More accurate numerical integration methods are generally preferred in simulation, but the principle is the same:

$$\boxed{\text{forces} \rightarrow \text{accelerations} \rightarrow \text{motion.}}$$

1.3.11 Inverse Dynamics

The *inverse dynamics* problem reverses this relationship.

Instead of asking what acceleration results from a torque, inverse dynamics asks:

What actuator torques are required to produce a desired acceleration?

Ignoring external forces and friction,

$$\boxed{\tau = M(q)\ddot{q}_d + C(q,\dot{q})\dot{q} + g(q).}$$

More generally,

$$\tau = M(q)\ddot{q}_d + C(q,\dot{q})\dot{q} + g(q) + \tau_f - J^T F_{\text{ext}}.$$

Inverse dynamics is particularly useful in model-based control because the desired trajectory supplies

$$q_d, \qquad \dot{q}_d, \qquad \ddot{q}_d,$$

and the model predicts the torque required to realize that motion.

The distinction between the two problems is therefore

$$\boxed{\text{Forward dynamics:} \quad (q,\dot{q},\tau) \to \ddot{q}}$$

and

$$\boxed{\text{Inverse dynamics:} \quad (q,\dot{q},\ddot{q}_d) \to \tau.}$$

Forward dynamics predicts what the robot will do. Inverse dynamics computes what must be applied to make the robot do something.

1.3.12 Computed-Torque Interpretation

The inverse-dynamics equation can also be used inside a feedback controller.

Define a desired acceleration

$$\ddot{q}^\star = \ddot{q}_d + K_D(\dot{q}_d - \dot{q}) + K_P(q_d - q).$$

The commanded torque is

$$\boxed{\tau = M(q)\ddot{q}^\star + C(q,\dot{q})\dot{q} + g(q).}$$

Substituting this into the ideal robot dynamics gives approximately

$$\ddot{e} + K_D\dot{e} + K_P e = 0,$$

where

$$e = q_d - q.$$

Thus, an accurate dynamics model can be used to cancel much of the robot's nonlinear mechanical behavior, leaving an approximately linear tracking-error system.

This provides one important connection between robot dynamics and model-based control.

1.3.13 Contact Dynamics

The free-motion manipulator equation assumes that the robot moves without additional kinematic constraints imposed by the environment.

When a robot makes contact, the environment can exert forces on the robot.

If a contact has Jacobian

$$J_c(q)$$

and contact wrench

$$F_c,$$

then its generalized contribution is

$$\boxed{\tau_c = J_c^T(q)F_c.}$$

The dynamics therefore become

$$\boxed{M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = \tau + J_c^T F_c}$$

for the idealized frictionless-joint case.

For several simultaneous contacts,

$$J_c = \begin{bmatrix} J_{c_1} \\ J_{c_2} \\ \vdots \end{bmatrix}, \qquad F_c = \begin{bmatrix} F_{c_1} \\ F_{c_2} \\ \vdots \end{bmatrix},$$

and the same expression remains valid:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = \tau + J_c^T F_c.$$

Contact therefore couples the motion of the robot with forces supplied by the environment.

1.3.14 Normal and Tangential Contact Forces

A contact force can be decomposed into normal and tangential components:

$$f = f_n n + f_t.$$

The normal component

$$f_n$$

acts perpendicular to the contact surface, while

$$f_t$$

lies in the tangent plane.

Under the Coulomb friction model, the tangential force is constrained by

$$\boxed{\|f_t\| \leq \mu f_n,}$$

where

$$\mu$$

is the coefficient of friction.

The admissible tangential forces form a *friction cone*.

If the required tangential force lies outside this cone, the contact cannot remain static and sliding occurs.

For computational optimization, the friction cone is frequently approximated by a polyhedral set of linear inequalities.

These constraints are important in grasping, pushing, walking, door opening, and bimanual manipulation.

1.3.15 Contact Constraints

When a point on the robot is constrained to remain stationary relative to the environment, its Cartesian velocity must satisfy

$$J_c(q)\dot{q} = 0.$$

Differentiating gives the corresponding acceleration constraint

$$J_c(q)\ddot{q} + \dot{J}_c(q, \dot{q})\dot{q} = 0.$$

The robot dynamics and the contact constraint must then be satisfied simultaneously:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = \tau + J_c^T F_c,$$

$$J_c\ddot{q} + \dot{J}_c\dot{q} = 0.$$

These equations illustrate an important difference between free motion and constrained motion.

During free motion, actuator torques determine acceleration directly through the robot dynamics.

During contact, the unknown contact forces

$$F_c$$

must also be determined so that the robot motion satisfies the environmental constraints.

This is the basis of constrained rigid-body dynamics used in locomotion and manipulation.

1.3.16 Contact Forces and Sensing

A six-axis force/torque sensor measures a wrench of the form

$$w = \begin{bmatrix} F \\ \tau \end{bmatrix} \in \mathbb{R}^6.$$

These measurements allow the robot to detect contact, regulate interaction forces, and estimate external disturbances.

Joint torque measurements can also be used to infer external loading.

Given a sufficiently accurate internal dynamics model,

$$\boxed{\tau_{\text{ext}} \approx \tau_{\text{measured}} - \tau_{\text{model}}.}$$

If the external load is applied through a known contact Jacobian,

$$\tau_{\text{ext}} = J^T F_{\text{ext}}.$$

Thus, force sensing and dynamics modeling provide complementary ways of estimating physical interaction with the environment.

1.3.17 Dynamics in Simulation and Control

Robot dynamics play different roles depending on the application.

In simulation, forward dynamics predicts how actuator commands change the robot state:

$$\tau \to \ddot{q} \to \dot{q} \to q.$$

In model-based control, inverse dynamics predicts which actuator commands are needed to realize a desired motion:

$$q_d, \dot{q}_d, \ddot{q}_d \to \tau.$$

During physical interaction, the dynamics must additionally account for contact forces:

$$\tau + J_c^T F_c \to \ddot{q}.$$

These three views correspond to prediction, control, and physical interaction.

1.3.18 Summary

Robot dynamics describes the relationship between forces, torques, and motion.

The Newton–Euler formulation derives dynamics by applying force and moment balance recursively to each rigid link, while the Lagrangian formulation derives the same motion equations from kinetic and potential energy.

Both lead to the standard manipulator equation

$$\boxed{M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) + \tau_f = \tau + J^T F_{\text{ext}}.}$$

The inertia matrix

$$M(q)$$

captures configuration-dependent inertial coupling between the joints.

The term

$$C(q,\dot{q})\dot{q}$$

captures Coriolis and centrifugal effects, while

$$g(q)$$

contains generalized gravitational forces.

Gravity compensation applies approximately

$$\tau_{\text{grav}} = g(q)$$

to support the robot's own weight.

Forward dynamics computes acceleration from applied torques,

$$\boxed{\ddot{q} = M^{-1}\left[\tau + J^T F_{\text{ext}} - C\dot{q} - g - \tau_f\right],}$$

whereas inverse dynamics computes the required torque from a desired acceleration,

$$\boxed{\tau = M\ddot{q}_d + C\dot{q} + g.}$$

When the robot contacts its environment, the dynamics include generalized contact forces

$$J_c^T F_c,$$

and the contact motion may simultaneously be constrained by

$$J_c\ddot{q} + \dot{J}_c\dot{q} = 0.$$

Friction further restricts feasible contact forces through conditions such as

$$\|f_t\| \leq \mu f_n.$$

These equations form the mechanical foundation for joint-space control, operational-space control, impedance control, whole-body control, trajectory optimization, and physically grounded robot learning.

1.4 Robot Control

Robot control concerns the design of actuator commands that cause a physical robot to follow desired motions, regulate interaction forces, reject disturbances, and remain stable despite modeling error and uncertainty. Kinematics determines how joint motion maps into Cartesian motion, while dynamics determines how applied forces and torques produce acceleration. A controller closes the loop between the desired behavior and the measured behavior of the physical system.

For a rigid-link manipulator, the dynamics developed in the previous section can be written as

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) + \tau_f = \tau + J^T(q)F_{\text{ext}},$$

where q, $\dot{q}$, and $\ddot{q}$ are joint position, velocity, and acceleration; $M(q)$ is the joint-space inertia matrix; $C(q,\dot{q})\dot{q}$ represents Coriolis and centrifugal effects; $g(q)$ represents gravity; τ_f represents friction; τ is the commanded actuator torque; and F_{ext} is an external Cartesian wrench.

A controller chooses τ, or some lower-level command that ultimately produces τ, so that the robot behaves according to a desired objective. Depending on the application, the controlled quantity may be a joint position q_d, a Cartesian pose x_d, a desired contact force F_d, or a combination of motion and force objectives.

1.4.1 Feedback Control

The central idea of feedback control is to measure the actual behavior of the system and compare it with the desired behavior. If $x_d(t)$ is a desired quantity and $x(t)$ is the measured quantity, the tracking error is $e(t) = x_d(t) - x(t)$. The controller uses this error to modify the actuator command.

Conceptually, a feedback loop has the form

$$\text{reference} \rightarrow \text{controller} \rightarrow \text{robot} \rightarrow \text{measurement} \rightarrow \text{error}.$$

This differs from *open-loop control*, in which commands are applied without using measurements to correct the resulting motion. Open-loop control can work when the model and environment are known accurately, but physical robots experience disturbances, friction, calibration error, payload variation, actuator uncertainty, and contact forces. Feedback allows the controller to respond to these effects.

A simple proportional controller has the form $u = K_P e$. If the error increases, the corrective action increases proportionally. For mechanical systems this often resembles a virtual spring: a larger displacement from the desired configuration produces a larger restoring effort.

Feedback introduces an important tradeoff. Larger gains generally increase responsiveness and reduce tracking error, but excessively aggressive gains can amplify measurement noise, excite unmodeled dynamics, saturate actuators, or destabilize the closed-loop system. Robot control therefore involves not merely reducing error, but shaping the complete dynamic response of the robot.

1.4.2 PID Control

The proportional–integral–derivative controller is one of the most widely used feedback-control structures. For a scalar tracking error $e(t) = x_d(t) - x(t)$, the controller is

$$\boxed{u(t) = K_P e(t) + K_I \int_0^t e(\tau)\, d\tau + K_D \dot{e}(t).}$$

The proportional term $K_P e$ responds directly to the current error. Increasing K_P generally makes the system respond more strongly to displacement from the target. In a mechanical interpretation, proportional feedback behaves similarly to stiffness.

The derivative term $K_D \dot{e}$ responds to how rapidly the error is changing. In mechanical systems it behaves similarly to damping. If the robot is moving rapidly toward the desired state, derivative feedback reduces the command before the system overshoots substantially. Derivative action therefore tends to suppress oscillation and improve transient behavior.

The integral term accumulates error over time. If a disturbance produces a persistent error, the integral contribution continues to grow until sufficient control effort is generated to remove that bias. Integral action is therefore useful for eliminating steady-state error caused by effects such as constant loads, imperfect gravity models, or actuator bias.

For many robot joints, proportional–derivative control is used more frequently than full PID because the mechanical system already contains integrative dynamics and because integral action can create undesirable windup during saturation. A joint controller may therefore use

$$\tau = K_P(q_d - q) + K_D(\dot{q}_d - \dot{q}),$$

where K_P and K_D are commonly diagonal positive-definite gain matrices.

If integral action is required, the controller can be extended to

$$\tau = K_P e_q + K_D \dot{e}_q + K_I \int e_q\, dt,$$

with $e_q = q_d - q$.

In real implementations, integral action is often combined with *anti-windup*. If an actuator reaches its torque limit, the integral state should not continue accumulating without bound, because doing so can produce a large transient once the actuator leaves saturation.

Derivative feedback also requires care because numerical differentiation amplifies high-frequency sensor noise. Joint velocity from encoders, filtered velocity estimates, or derivative filtering is therefore generally preferable to differentiating raw position measurements directly.

1.4.3 Joint-Space Control

Joint-space control specifies desired behavior directly in generalized coordinates. A desired trajectory consists of $q_d(t)$, and often desired velocity $\dot{q}_d(t)$ and acceleration $\ddot{q}_d(t)$.

A basic PD torque controller is

$$\boxed{\tau = K_P(q_d - q) + K_D(\dot{q}_d - \dot{q}).}$$

This controller does not explicitly compensate for the nonlinear robot dynamics. The feedback terms must therefore both correct tracking error and overcome gravity, inertial coupling, friction, and disturbances.

For small motions or robots with strong internal servo loops, this approach can be sufficient. However, the same numerical gains can produce different Cartesian behavior in different robot configurations because the effective inertia seen by each joint changes with q.

Gravity compensation provides an important improvement. If the model predicts the generalized gravitational torque $g(q)$, the controller can apply

$$\boxed{\tau = K_P(q_d - q) + K_D(\dot{q}_d - \dot{q}) + g(q).}$$

The feedback terms then primarily regulate deviations from the desired configuration instead of continuously supplying the torque required to support the robot's own weight.

This distinction is particularly noticeable on manipulators. Without gravity compensation, a low-gain arm may sag substantially under its own weight. With reasonably accurate compensation, the same low feedback gains can maintain configuration while preserving compliance.

Joint-space controllers are attractive because joint position and velocity are directly measured by encoders, joint limits are naturally represented, and the resulting controller is comparatively simple. The main limitation is that many robot tasks are naturally defined in Cartesian space. A command such as "move the end effector 5 cm forward" must first be converted into an appropriate joint trajectory before a purely joint-space controller can execute it.

1.4.4 Feedforward and Computed-Torque Control

Feedback corrects errors after they appear. Feedforward control instead uses a model of the robot to predict the effort that should be required to produce the desired motion.

Suppose a desired trajectory provides q_d, $\dot{q}_d$, and $\ddot{q}_d$. Ignoring friction and external forces, inverse dynamics predicts the nominal torque

$$\tau_{\text{ff}} = M(q)\ddot{q}_d + C(q, \dot{q})\dot{q} + g(q).$$

A practical controller can combine this feedforward torque with feedback corrections. One common formulation defines the commanded acceleration

$$\ddot{q}^\star = \ddot{q}_d + K_D(\dot{q}_d - \dot{q}) + K_P(q_d - q),$$

and then applies

$$\boxed{\tau = M(q)\ddot{q}^\star + C(q, \dot{q})\dot{q} + g(q).}$$

This is commonly called *computed-torque control* or inverse-dynamics control.

To understand the idea, substitute the command into the ideal manipulator dynamics $M\ddot{q} + C\dot{q} + g = \tau$. The nonlinear terms cancel, leaving approximately

$$\ddot{e} + K_D\dot{e} + K_P e = 0,$$

where $e = q_d - q$.

The controller has therefore transformed the nonlinear manipulator into an approximately linear second-order tracking system.

The advantage is that the feedback gains no longer need to compensate for the complete robot dynamics. The limitation is that cancellation is only as accurate as the model. Errors in link masses, payloads, friction, motor dynamics, elasticity, or external contact forces create residual dynamics that the feedback controller must still reject.

This illustrates an important principle used throughout robotics: model-based feedforward handles predictable physics, while feedback handles uncertainty.

1.4.5 Cartesian and Operational-Space Control

Joint-space control regulates individual joint coordinates, but manipulation objectives are often more naturally stated in Cartesian coordinates. Examples include moving the hand to a desired pose, maintaining a tool orientation, following a surface, or applying a specified force normal to an object.

Let x denote a Cartesian task variable. Differential kinematics give $\dot{x} = J(q)\dot{q}$, and differentiating once more gives

$$\boxed{\ddot{x} = J(q)\ddot{q} + \dot{J}(q,\dot{q})\dot{q}.}$$

A simple Cartesian feedback law can define a desired task-space acceleration as

$$\ddot{x}^\star = \ddot{x}_d + K_D(\dot{x}_d - \dot{x}) + K_P(x_d - x).$$

One possibility is to convert this acceleration into joint acceleration through inverse differential kinematics. Operational-space control goes further by accounting explicitly for the robot's dynamics as seen from the task coordinates.

Starting from the joint-space inertia $M(q)$, the effective task-space inertia is

$$\boxed{\Lambda(q) = \left(JM^{-1}J^T\right)^{-1}.}$$

The matrix $\Lambda(q)$ describes the apparent inertia of the robot at the controlled Cartesian frame. The robot may be easy to accelerate in one Cartesian direction and difficult to accelerate in another; Λ captures this configuration-dependent behavior.

A simplified operational-space wrench command is $F^\star = \Lambda(q)\ddot{x}^\star$, with the corresponding joint torque $\tau = J^T F^\star$. A more complete controller compensates for velocity-dependent and gravitational effects,

$$\boxed{F^\star = \Lambda \ddot{x}^\star + \mu(q, \dot{q}) + p(q),}$$

followed by

$$\tau = J^T F^\star.$$

Here μ contains task-space Coriolis and centrifugal effects, while p contains the task-space gravity contribution.

Operational-space control is especially useful because the controlled behavior can be specified directly in physically meaningful coordinates. Rather than commanding individual joints, the controller can regulate quantities such as end-effector translation, orientation, or Cartesian force.

For a redundant robot, additional joint-space objectives can be introduced without disturbing the primary Cartesian task. A general form is $\tau = J^T F^\star + N^T \tau_0$, where N projects the secondary command into a dynamically appropriate null space. Secondary objectives may include posture regulation, joint-limit avoidance, collision avoidance, or maintaining favorable manipulability.

1.4.6 Impedance Control

Pure position control attempts to eliminate position error regardless of the external forces required to do so. This behavior can be undesirable during physical interaction. If a rigidly position-controlled robot encounters an object that prevents it from reaching the commanded position, the position error can create a very large contact force.

Impedance control instead specifies a relationship between motion and force. The robot is commanded to behave as though its end effector were connected to the desired trajectory by a virtual mechanical system.

A simple Cartesian spring–damper relation is

$$\boxed{F^\star = K(x_d - x) + D(\dot{x}_d - \dot{x}),}$$

where K is the desired stiffness and D is the desired damping. The corresponding actuator torque may be produced through $\tau = J^T F^\star$, together with gravity or dynamics compensation.

If the end effector is displaced from the desired pose, the robot does not necessarily attempt to eliminate the displacement with arbitrarily large effort. Instead, displacement produces a force determined by the chosen virtual stiffness.

A more complete desired impedance is

$$\boxed{M_d(\ddot{x} - \ddot{x}_d) + D_d(\dot{x} - \dot{x}_d) + K_d(x - x_d) = F_{\text{ext}}.}$$

The matrices M_d, D_d, and K_d specify the apparent inertia, damping, and stiffness that the robot presents to its environment.

A high stiffness causes the end effector to behave almost like a rigid position-controlled device. A lower stiffness permits greater displacement under external force and therefore produces more compliant interaction.

This is useful in tasks such as assembly, grasping, human–robot interaction, surface contact, and manipulation in uncertain environments. If an object's position differs slightly from the estimate, compliance allows the robot to adapt physically rather than fighting the geometric error with large contact forces.

Conceptually, impedance control implements the mapping

$$\text{motion deviation} \longrightarrow \text{interaction force}.$$

1.4.7 Admittance Control

Admittance control reverses the input-output relationship used in impedance control. Instead of using motion error to generate force, admittance control measures force and generates desired motion.

A virtual mechanical system may be defined as

$$\boxed{M_d\ddot{x}_d + D_d\dot{x}_d + K_d(x_d - x_0) = F_{\text{ext}},}$$

where x_0 is a nominal equilibrium pose. The measured external force therefore moves the desired trajectory according to the chosen virtual mass, damping, and stiffness.

Conceptually,

$$\text{measured force} \longrightarrow \text{desired motion}.$$

This distinction is strongly related to the underlying robot hardware. A torque-controlled robot can often implement impedance directly by commanding actuator torques. A robot with a strong internal position servo may instead be easier to control through admittance: a force sensor measures interaction force, the admittance model converts that force into a modified position command, and the existing position controller follows the result.

Consider a robot handle intended to be moved by a person. If the person pushes with force F_{ext}, an admittance controller can move the commanded position in the direction of the push. Increasing the virtual mass makes the system respond more slowly, increasing damping reduces oscillation, and increasing stiffness causes the robot to return more strongly toward its nominal pose.

Impedance and admittance therefore aim to create similar physical interaction behavior but differ in which quantity is treated as the controller input and which is produced as the output.

1.4.8 Force Control

Some manipulation tasks require direct regulation of contact force rather than position. Examples include maintaining a specified normal force while polishing a surface, applying a controlled insertion force during assembly, or maintaining a desired grasp force.

Suppose a force/torque sensor measures the external wrench F_{ext}, and the desired wrench is F_d. The force error is $e_F = F_d - F_{\text{ext}}$.

A simple proportional force controller may generate a correction proportional to this error. For a scalar contact direction, one might write $u_F = K_F e_F$. Integral action can be added to eliminate steady-state force error, giving $u_F = K_F e_F + K_I \int e_F dt$.

The precise interpretation of u_F depends on the underlying control architecture. On a torque-controlled manipulator, a desired Cartesian wrench can be converted into joint torque through

$$\boxed{\tau_F = J^T F^\star.}$$

On a position-controlled robot, the force controller may instead modify the desired position, effectively producing an admittance-like outer loop.

Direct force regulation generally requires contact with the environment. Before contact, the environment cannot produce the commanded reaction force. This makes transitions between free-space motion and contact important. A controller that blindly increases commanded force while no object is present can cause the robot to accelerate into the environment once contact occurs.

Force-control performance also depends on environment stiffness. If the environment is extremely stiff, a very small position change may produce a large force change. High force-feedback gains can therefore destabilize the interaction, especially in the presence of sensor delay, actuator dynamics, structural compliance, or communication latency.

1.4.9 Hybrid Position–Force Control

Many manipulation tasks require position regulation in some directions and force regulation in others.

Consider wiping a table. Along the surface, the robot should follow a desired trajectory in x and y. In the direction normal to the surface, however, exact position is less important than maintaining an appropriate contact force.

Trying to control both exact normal position and exact normal force simultaneously is generally contradictory when the environment constrains the contact geometry. Instead, the task space can be divided into position-controlled and force-controlled subspaces.

Let S_p denote a selection matrix for position-controlled directions, and let S_f select force-controlled directions. For complementary task directions,

$$\boxed{S_p + S_f = I, \qquad S_p S_f = 0.}$$

A Cartesian motion controller can produce a wrench F_p associated with position tracking, while a force controller produces F_f. The combined command can then be written conceptually as

$$\boxed{F^\star = S_p F_p + S_f F_f.}$$

Joint torque is obtained through

$$\tau = J^T F^\star.$$

For the table-wiping example, one might select tangential x- and y-motion for position control while selecting the surface-normal z-direction for force control.

The same principle applies to constrained manipulation. When turning a crank, opening a door, sliding an object along a wall, or inserting a peg, the environment constrains some motion directions while leaving others relatively free.

Hybrid position–force control explicitly represents this geometric structure rather than asking a single position controller to resolve the interaction implicitly.

1.4.10 Model Predictive Control

Model Predictive Control (MPC) chooses control actions by repeatedly predicting the future evolution of the robot and optimizing behavior over a finite horizon.

Suppose the robot dynamics are represented in discrete time as $x_{k+1} = f(x_k, u_k)$, where x_k is the state and u_k is the control input. MPC solves for a sequence of future inputs $u_0, \ldots, u_{H-1}$.

A typical objective is

$$\boxed{\min_{u_{0:H-1}} \sum_{k=0}^{H-1} \Big[(x_k - x_k^{\text{ref}})^T Q (x_k - x_k^{\text{ref}}) + u_k^T R u_k \Big] + (x_H - x_H^{\text{ref}})^T Q_f (x_H - x_H^{\text{ref}})}$$

subject to the system dynamics and constraints such as $x_{\min} \le x_k \le x_{\max}$ and $u_{\min} \le u_k \le u_{\max}$.

The matrix Q penalizes tracking error, R penalizes excessive control effort, and Q_f penalizes terminal-state error.

Although MPC computes an entire sequence

$$u_0^\star, u_1^\star, \ldots, u_{H-1}^\star,$$

only the first command $u_0^\star$ is normally executed. The system is measured again, the horizon moves forward, and the optimization is solved again. This process is known as *receding-horizon control*.

The important distinction from a conventional feedback controller is that MPC reasons explicitly about future consequences. A simple PD controller determines the current command from current tracking error. MPC asks how the current command will influence states several steps into the future.

Its major advantage is the natural treatment of constraints. Joint limits, velocity limits, torque limits, collision constraints, friction constraints, and contact conditions can potentially be included directly in the optimization problem.

The cost is computation. Linear MPC with quadratic costs can often be solved quickly, while nonlinear MPC for a high-degree-of-freedom robot with contact dynamics may require substantial computation at every control step.

For robotics, MPC is therefore particularly attractive when predicting ahead and respecting constraints are important enough to justify the additional computational cost.

1.4.11 Whole-Body Control

Whole-body control extends task-space control to robots with many degrees of freedom and multiple simultaneous objectives. Humanoids, quadrupeds, and mobile manipulators may need to control end-effector motion, base motion, balance, posture, contact forces, and joint limits at the same time.

For task i, differential kinematics give

$$\ddot{x}_i = J_i\ddot{q} + \dot{J}_i\dot{q}.$$

If the desired task acceleration is $\ddot{x}_i^\star$, a whole-body controller may solve an optimization such as

$$\boxed{\min_{\ddot{q},\tau,F_c} \sum_i \left\| J_i\ddot{q} + \dot{J}_i\dot{q} - \ddot{x}_i^\star \right\|_{W_i}^2}$$

subject to the rigid-body dynamics

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = \tau + J_c^T F_c,$$

along with actuator, joint, and contact constraints.

For a legged robot, the contact force F_c may be constrained to remain inside a friction cone so that the feet do not slip. Actuator torques may satisfy $\tau_{\min} \le \tau \le \tau_{\max}$, and joint accelerations may also be bounded.

Different tasks can be combined using weights W_i. For example, maintaining balance may receive a larger weight than tracking a preferred arm posture.

An alternative is strict task prioritization. A high-priority task is solved first, and lower-priority commands are allowed only in directions that do not interfere with that task. This extends the null-space idea from inverse kinematics into constrained dynamic control.

Whole-body control can therefore be viewed as a bridge between classical feedback control and online constrained optimization. Rather than designing a separate independent controller for every degree of freedom, the robot solves for a physically consistent action satisfying many objectives simultaneously.

1.4.12 Learned Policies and Low-Level Control

Modern robot-learning systems often separate learned decision-making from high-frequency physical control.

A learned policy maps observations into actions,

$$a_t = \pi(o_t),$$

but the action need not be a motor torque. It may instead be a desired joint configuration q_d, a desired end-effector pose x_d, a relative displacement Δx, a desired velocity, or a short trajectory.

A common architecture is therefore

$$\text{observations} \rightarrow \text{learned policy} \rightarrow \text{desired robot motion} \rightarrow \text{classical controller} \rightarrow \tau.$$

For example, a vision-language-action model may process images and language at 10–30 Hz and produce Cartesian end-effector targets, while an impedance or torque controller runs at several hundred hertz or more.

This separation is useful because high-level learning and low-level control solve different problems. The learned policy can focus on perception, semantic understanding, task sequencing, and generalization across environments. The lower-level controller handles fast feedback, actuator dynamics, model compensation, contact disturbances, and stability.

The action representation determines where responsibility lies between these layers. A high-level pose action such as $a_t = x_d$ gives substantial authority to the classical controller. The learned model does not need to learn the fast mechanical response of the robot explicitly.

At the opposite extreme, a torque-level policy directly outputs $a_t = \tau_t$. Such a policy has greater control authority and can exploit nonlinear dynamics that may be difficult to model, but it must generally operate at higher frequency and learn behaviors that would otherwise be handled by the controller.

This produces a useful spectrum:

$$\text{pose targets} \rightarrow \text{velocity targets} \rightarrow \text{joint targets} \rightarrow \text{torques},$$

with progressively more responsibility transferred from the classical controller to the learned policy.

Intermediate approaches are also possible. A learned policy may predict nominal actions while a conventional controller provides stabilizing feedback, or learning may supply residual corrections to a model-based controller. The key architectural question is therefore not simply whether a robot uses learning or classical control, but which parts of the control hierarchy are learned and which remain explicitly model based.

1.4.13 Relationship Between the Control Methods

The control methods described above are not mutually exclusive. They address different levels of the robot-control problem.

PID and joint-space control provide direct feedback around joint variables. Feedforward and computed-torque methods use a dynamics model to anticipate the torque required for a desired motion. Operational-space control expresses dynamic objectives directly in Cartesian coordinates. Impedance and admittance control shape the physical relationship between motion and interaction force. Force and hybrid position–force control explicitly regulate contact behavior. MPC reasons over a future time horizon while accounting for constraints. Whole-body control coordinates several simultaneous tasks across many robot degrees of freedom. Learned policies can operate above, alongside, or in place of portions of this classical hierarchy.

A modern manipulation system might therefore use several of these ideas simultaneously. A learned policy could select an end-effector target, an MPC layer could generate a feasible short-

horizon trajectory, an operational-space impedance controller could track that trajectory, and a high-rate torque loop could execute the resulting commands.

The overall control stack can therefore be viewed as a hierarchy,

$$\text{task objective} \rightarrow \text{motion generation} \rightarrow \text{feedback control} \rightarrow \text{actuator command} \rightarrow \text{physical robot}.$$

1.4.14 Summary

Robot control closes the loop between desired and measured behavior. Feedback control responds to tracking error, while feedforward control uses a model to anticipate the physical effort required to execute a desired trajectory.

PID and joint-space controllers regulate joint-level errors directly. Gravity compensation and computed-torque control incorporate increasingly detailed models of the robot dynamics. Cartesian and operational-space control move the control objective into task coordinates, where manipulation objectives can be specified more naturally.

During contact, position regulation alone is often insufficient. Impedance control specifies how motion deviations create interaction forces, whereas admittance control maps measured forces into desired motion. Direct force control regulates interaction wrench, and hybrid position–force control separates task directions into those that should follow motion commands and those that should regulate force.

MPC adds explicit prediction and constraints, while whole-body control coordinates several simultaneous objectives under the complete robot dynamics. Learned policies introduce another layer of abstraction: they can generate high-level targets for conventional controllers or directly produce lower-level actions.

The central principle across these approaches is that successful robot control requires matching the control representation to the physical task. Free-space motion, high-speed trajectory tracking, compliant manipulation, constrained contact, locomotion, and learned behavior place different requirements on the control system, and modern robots frequently combine several of these methods within a single hierarchy.

1.5 State Estimation and Sensor Fusion

A robot rarely has direct access to every physical quantity required for navigation and control. Joint encoders measure internal configuration, an IMU measures angular velocity and specific force, cameras observe image intensities, LiDAR measures ranges, and GPS may provide global position. Quantities such as global pose, velocity, sensor bias, object motion, or contact state must therefore be inferred from incomplete and noisy measurements.

State estimation combines a model of how the system evolves with measurements from one or more sensors to estimate these hidden quantities. The underlying problem is inherently probabilistic. Even if the robot has an accurate motion model, disturbances and unmodeled dynamics create uncertainty during prediction. Likewise, sensors provide imperfect observations corrupted by noise, limited resolution, calibration error, occlusion, and environmental effects.

A useful abstract model is to let x_t denote the hidden state, u_t the control or motion input, and z_t the sensor measurement. The system evolves according to a transition model $p(x_t \mid x_{t-1}, u_{t-1})$, while measurements are described by a likelihood $p(z_t \mid x_t)$. State estimation then seeks the posterior distribution $p(x_t \mid z_{1:t}, u_{1:t-1})$.

1.5.1 Probabilistic State Estimation

A deterministic estimator produces a single state estimate $\hat{x}_t$. A probabilistic estimator additionally represents uncertainty about that estimate. This distinction is important because two estimates with the same numerical value may have very different reliability.

For example, a mobile robot may estimate its position as $\hat{x} = 10$ m. If the associated standard deviation is 0.01 m, the estimate is highly certain; if the standard deviation is 5 m, the same mean conveys much less information.

In a Gaussian representation, uncertainty is described by a mean $\hat{x}$ and covariance P, written $x \sim \mathcal{N}(\hat{x}, P)$. For an estimation error $e = x - \hat{x}$, the covariance is $P = \mathbb{E}[ee^T]$. Diagonal terms of P describe uncertainty in individual state variables, while off-diagonal terms describe correlations between their errors.

These correlations matter physically. If a mobile robot has uncertainty in heading, then after moving forward its uncertainty in Cartesian position may increase. Similarly, camera observations may couple uncertainty in depth, position, and orientation.

Probabilistic estimation therefore attempts to track not only the most likely state but also how uncertainty evolves as the robot moves and receives measurements.

1.5.2 Bayes Filtering

The general recursive solution to the state-estimation problem is the Bayes filter. At each timestep the estimator first predicts the state using the motion model and then corrects that prediction using the new measurement.

Suppose the previous posterior is $p(x_{t-1} \mid z_{1:t-1}, u_{1:t-2})$. The prediction step propagates this distribution through the dynamics:

$$\boxed{p(x_t \mid z_{1:t-1}, u_{1:t-1}) = \int p(x_t \mid x_{t-1}, u_{t-1}) p(x_{t-1} \mid z_{1:t-1}, u_{1:t-2})\, dx_{t-1}.}$$

The result is the *prior* distribution before observing z_t. Once a measurement arrives, Bayes' rule gives

$$\boxed{p(x_t \mid z_{1:t}, u_{1:t-1}) \propto p(z_t \mid x_t) p(x_t \mid z_{1:t-1}, u_{1:t-1}).}$$

The measurement likelihood $p(z_t \mid x_t)$ assigns larger probability to states that better explain the observation. The measurement therefore reweights the predicted state distribution.

Conceptually, Bayesian filtering follows the repeated cycle

$$\text{prior state} \rightarrow \text{motion prediction} \rightarrow \text{measurement comparison} \rightarrow \text{posterior state}.$$

Different state-estimation algorithms differ primarily in how they represent the probability distribution and how accurately they perform these two operations. The Kalman filter represents the distribution as a Gaussian and assumes linear models. The EKF retains the Gaussian representation but linearizes nonlinear models. The UKF propagates a set of deterministically chosen sigma points. Particle filters approximate the entire posterior using weighted random samples.

1.5.3 Kalman Filter

The Kalman filter is a recursive estimator for linear dynamical systems with Gaussian uncertainty. Its importance comes not merely from the equations themselves, but from the way those equations combine two uncertain sources of information: a prediction from the system model and a measurement from a sensor.

Consider the linear stochastic system

$$x_t = A_t x_{t-1} + B_t u_t + w_t, \qquad w_t \sim \mathcal{N}(0, Q_t),$$

with measurement model

$$z_t = H_t x_t + v_t, \qquad v_t \sim \mathcal{N}(0, R_t).$$

Here x_t is the hidden system state, u_t is a known control input, and z_t is the measurement. The matrices Q_t and R_t describe uncertainty in the motion model and sensor respectively.

The filter maintains an estimated state $\hat{x}_t$ together with its covariance P_t. The estimation error is $e_t = x_t - \hat{x}_t$, so the covariance is

$$P_t = \mathbb{E}\left[e_t e_t^T\right].$$

The Kalman-filter recursion consists of two stages: prediction and correction.

Prediction of the State

Suppose the posterior estimate at time $t-1$ is $\hat{x}_{t-1}$. Taking the expectation of the state-transition equation and using $\mathbb{E}[w_t] = 0$ gives

$$\mathbb{E}[x_t] = A_t \mathbb{E}[x_{t-1}] + B_t u_t.$$

Replacing the unknown expectation with the current estimate gives the predicted state

$$\boxed{\hat{x}_t^- = A_t \hat{x}_{t-1} + B_t u_t.}$$

The superscript $(-)$ indicates that this is the *prior* estimate, before incorporating measurement z_t.

Prediction of the Covariance

The predicted state alone is not sufficient; the estimator must also determine how uncertain that prediction is.

The true state evolves according to

$$x_t = A_t x_{t-1} + B_t u_t + w_t,$$

while the filter predicts

$$\hat{x}_t^- = A_t \hat{x}_{t-1} + B_t u_t.$$

Subtracting gives the prior estimation error

$$e_t^- = x_t - \hat{x}_t^- = A_t(x_{t-1} - \hat{x}_{t-1}) + w_t = A_t e_{t-1} + w_t.$$

The prior covariance is therefore

$$P_t^- = \mathbb{E}\left[e_t^- (e_t^-)^T\right].$$

Substituting $e_t^- = A_t e_{t-1} + w_t$ gives

$$P_t^- = \mathbb{E}\left[(A_t e_{t-1} + w_t)(A_t e_{t-1} + w_t)^T\right].$$

Assuming the process noise is independent of the previous estimation error, the cross terms vanish. Therefore,

$$\boxed{P_t^- = A_t P_{t-1} A_t^T + Q_t.}$$

The interpretation is important. The term $A_t P_{t-1} A_t^T$ propagates existing uncertainty through the system dynamics, while Q_t adds uncertainty introduced by disturbances and model imperfections.

Prediction therefore usually increases uncertainty.

The Measurement Innovation

When measurement z_t arrives, the predicted state implies a predicted measurement

$$\hat{z}_t = H_t \hat{x}_t^- .$$

The discrepancy

$$\boxed{y_t = z_t - H_t \hat{x}_t^-}$$

is called the *innovation* or measurement residual.

The innovation measures how much new information the sensor provides relative to the current prediction.

Using $z_t = H_t x_t + v_t$,

$$y_t = H_t(x_t - \hat{x}_t^-) + v_t = H_t e_t^- + v_t.$$

Its covariance is therefore

$$S_t = \mathbb{E}[y_t y_t^T].$$

Assuming measurement noise is independent of the predicted state error gives

$$\boxed{S_t = H_t P_t^- H_t^T + R_t.}$$

Thus, uncertainty in the measurement residual comes from two sources: uncertainty in the predicted state and uncertainty in the sensor itself.

Choosing the State Correction

The corrected estimate is assumed to have the form

$$\boxed{\hat{x}_t = \hat{x}_t^- + K_t y_t,}$$

where K_t is a matrix that determines how strongly the innovation changes the state estimate.

Substituting the innovation gives

$$\hat{x}_t = \hat{x}_t^- + K_t \left(z_t - H_t \hat{x}_t^- \right) .$$

The problem is therefore to determine the optimal K_t.

The posterior estimation error is

$$e_t = x_t - \hat{x}_t.$$

Using the update equation,

$$e_t = x_t - \hat{x}_t^- - K_t \left(z_t - H_t \hat{x}_t^- \right).$$

Since $e_t^- = x_t - \hat{x}_t^-$ and $z_t = H_t x_t + v_t$,

$$e_t = (I - K_t H_t) e_t^- - K_t v_t.$$

The resulting posterior covariance is

$$P_t = (I - K_t H_t) P_t^- (I - K_t H_t)^T + K_t R_t K_t^T.$$

This expression describes how the choice of K_t determines the uncertainty remaining after the measurement update.

The Kalman gain is chosen to minimize the posterior uncertainty. Minimizing the trace of P_t with respect to K_t gives

$$K_t \left(H_t P_t^- H_t^T + R_t \right) = P_t^- H_t^T.$$

Therefore,

$$\boxed{K_t = P_t^- H_t^T \left(H_t P_t^- H_t^T + R_t \right)^{-1}.}$$

Since the innovation covariance is $S_t = H_t P_t^- H_t^T + R_t$, this can also be written compactly as

$$K_t = P_t^- H_t^T S_t^{-1}.$$

The Kalman gain is therefore not an arbitrary weighting factor. It follows from choosing the linear correction that minimizes the uncertainty of the resulting state estimate.

Interpretation of the Kalman Gain

The scalar case makes the result intuitive. Suppose the state is observed directly, so $H = 1$. Then

$$\boxed{K = \frac{P^-}{P^- + R}.}$$

If the prediction is highly uncertain relative to the sensor, $P^- \gg R$, then $K \approx 1$. The update becomes approximately $\hat{x} \approx z$, meaning the estimator trusts the measurement.

If the sensor is much noisier than the prediction, $R \gg P^-$, then $K \approx 0$, and the estimator changes the predicted state only slightly.

Thus, the Kalman gain performs an uncertainty-weighted compromise between model prediction and sensor observation.

Posterior Covariance

After applying the optimal gain, the covariance may be updated using

$$P_t = (I - K_t H_t) P_t^-.$$

A numerically safer form is the Joseph covariance update,

$$\boxed{P_t = (I - K_t H_t) P_t^- (I - K_t H_t)^T + K_t R_t K_t^T.}$$

The Joseph form follows directly from the posterior error equation and better preserves covariance symmetry and positive semidefiniteness under finite-precision computation.

The complete Kalman-filter recursion can therefore be summarized as

$$\boxed{\begin{aligned} \hat{x}_t^- &= A_t \hat{x}_{t-1} + B_t u_t, \\ P_t^- &= A_t P_{t-1} A_t^T + Q_t, \\ y_t &= z_t - H_t \hat{x}_t^-, \\ S_t &= H_t P_t^- H_t^T + R_t, \\ K_t &= P_t^- H_t^T S_t^{-1}, \\ \hat{x}_t &= \hat{x}_t^- + K_t y_t, \\ P_t &= (I - K_t H_t) P_t^-. \end{aligned}}$$

The important conceptual sequence is not merely the equations themselves:

predict state

$\downarrow$

propagate uncertainty

$\downarrow$

compare prediction with measurement

$\downarrow$

quantify residual uncertainty

$\downarrow$

compute optimal weighting

$\downarrow$

correct state and uncertainty

The Kalman filter is therefore best understood as an optimal uncertainty-weighted fusion of a dynamical prediction and a noisy observation under the assumptions of linear dynamics and Gaussian noise.

The Kalman filter gives the exact recursive posterior mean and covariance for a linear dynamical system with Gaussian process and measurement noise. The system model is $x_t = A_t x_{t-1} + B_t u_t + w_t$, and the measurement model is $z_t = H_t x_t + v_t$, where $w_t \sim \mathcal{N}(0, Q_t)$ and $v_t \sim \mathcal{N}(0, R_t)$.

The filter maintains a state estimate $\hat{x}_t$ and covariance P_t. Each update consists of prediction followed by measurement correction.

Prediction

The predicted state is

$$\boxed{\hat{x}_t^- = A_t \hat{x}_{t-1} + B_t u_t, \qquad P_t^- = A_t P_{t-1} A_t^T + Q_t.}$$

The superscript $(-)$ denotes the prior estimate before incorporating the current measurement.

The covariance equation has a direct interpretation. The term $A_t P_{t-1} A_t^T$ propagates existing uncertainty through the system dynamics, while Q_t accounts for new uncertainty caused by imperfect modeling and disturbances.

Thus, even if no measurement is received, uncertainty normally grows as the system evolves.

Measurement Innovation

Given the predicted state, the expected sensor measurement is $\hat{z}_t = H_t \hat{x}_t^-$. The difference between the actual and predicted measurement,

$$y_t = z_t - H_t \hat{x}_t^-,$$

is called the *innovation* or measurement residual.

The innovation covariance is

$$\boxed{S_t = H_t P_t^- H_t^T + R_t.}$$

The first term represents uncertainty in the predicted state expressed in measurement coordinates, while R_t represents measurement noise.

A large innovation does not automatically mean that the state estimate is wrong. The importance of that innovation depends on how uncertain the predicted state and measurement are.

Kalman Gain and Correction

The Kalman gain determines how strongly the state should move toward the new measurement:

$$\boxed{K_t = P_t^- H_t^T \left(H_t P_t^- H_t^T + R_t \right)^{-1}.}$$

The corrected estimate is then

$$\boxed{\hat{x}_t = \hat{x}_t^- + K_t\left(z_t - H_t\hat{x}_t^-\right).}$$

For a scalar system with $H = 1$, the gain reduces to $K = P^-/(P^- + R)$. If $P^- \gg R$, the prediction is relatively uncertain and $K \approx 1$, so the filter trusts the measurement. If $R \gg P^-$, the measurement is relatively noisy and $K \approx 0$, so the estimator trusts the model.

After the state update, the covariance can be written in the simplified form $P_t = (I - K_tH_t)P_t^-$. In numerical implementations, the Joseph form

$$P_t = (I - K_tH_t)P_t^-(I - K_tH_t)^T + K_tR_tK_t^T$$

is often preferred because it better preserves symmetry and positive semidefiniteness under finite-precision arithmetic.

The essential Kalman-filter cycle is therefore prediction of both state and uncertainty, comparison with a new measurement, uncertainty-weighted correction, and reduction of posterior uncertainty.

1.5.4 Extended Kalman Filter

Most robotic systems are nonlinear. A mobile robot's Cartesian motion depends on $\sin\theta$ and $\cos\theta$, camera projection contains division by depth, and rigid-body orientation evolves on a nonlinear manifold. The system is therefore more naturally written as $x_t = f(x_{t-1}, u_t) + w_t$ with measurement $z_t = h(x_t) + v_t$.

The Extended Kalman Filter preserves the Gaussian structure of the Kalman filter but handles nonlinear dynamics by repeatedly constructing a local linear approximation.

The mean is propagated through the true nonlinear model,

$$\boxed{\hat{x}_t^- = f(\hat{x}_{t-1}, u_t),}$$

while uncertainty is propagated using the Jacobian

$$F_t = \left.\frac{\partial f(x, u_t)}{\partial x}\right|_{x=\hat{x}_{t-1}}.$$

The covariance prediction becomes

$$\boxed{P_t^- = F_tP_{t-1}F_t^T + Q_t.}$$

The measurement model is treated similarly. The predicted measurement is $\hat{z}_t = h(\hat{x}_t^-)$, and the measurement Jacobian is

$$H_t = \left.\frac{\partial h(x)}{\partial x}\right|_{x=\hat{x}_t^-}.$$

The innovation, innovation covariance, gain, and state correction are then $y_t = z_t - h(\hat{x}_t^-)$, $S_t = H_t P_t^- H_t^T + R_t$, $K_t = P_t^- H_t^T S_t^{-1}$, and $\hat{x}_t = \hat{x}_t^- + K_t y_t$.

The central idea is therefore

nonlinear model for the state, local linear model for the uncertainty.

The EKF does not replace the nonlinear system with a globally linear one. It repeatedly linearizes around the current estimate.

Why Linearization Can Fail

The EKF uses first-order approximations such as $f(x) \approx f(\hat{x}) + F(x - \hat{x})$. This approximation is accurate only when the uncertainty is sufficiently local and the nonlinear function is reasonably well approximated by a tangent plane over that region.

If the state uncertainty becomes large, the predicted Gaussian may differ substantially from the true transformed distribution. Incorrect Jacobians, poor initialization, underestimated noise, and highly nonlinear measurements can therefore cause inconsistency or divergence.

The EKF nevertheless remains one of the most important estimation methods in robotics because it is computationally efficient and works well when the estimate stays near the true state.

1.5.5 Unscented Kalman Filter

The Unscented Kalman Filter addresses nonlinear estimation without explicitly computing first-order Jacobians. Like the EKF, it represents the state distribution using a mean and covariance and assumes an approximately Gaussian posterior. The difference lies in how a nonlinear transformation of that Gaussian is approximated.

Suppose $x \sim \mathcal{N}(\hat{x}, P)$. Instead of linearizing a nonlinear function $f(x)$, the UKF chooses a deterministic collection of *sigma points* distributed around $\hat{x}$. For an n-dimensional state, a common construction uses $2n + 1$ points,

$$\chi_0 = \hat{x}, \qquad \chi_i = \hat{x} + \left[\sqrt{(n+\lambda)P}\right]_i, \qquad \chi_{i+n} = \hat{x} - \left[\sqrt{(n+\lambda)P}\right]_i.$$

Each point is propagated through the full nonlinear model, $\chi_i' = f(\chi_i, u)$, and the transformed mean and covariance are reconstructed from weighted combinations of the propagated sigma points.

The motivation is that approximating the probability distribution and then passing representative points through the exact nonlinear function can be more accurate than approximating the nonlinear function itself with a first-order Taylor expansion.

The same idea is applied to measurements. Predicted sigma points are passed through $h(\cdot)$, producing a predicted measurement distribution and a state–measurement cross covariance. A Kalman-like gain is then formed from these quantities.

The UKF is therefore useful when nonlinearities are strong enough that first-order EKF linearization becomes inaccurate but the posterior remains approximately unimodal and Gaussian. Its disadvantages are greater computational cost than a basic EKF and the need to carefully handle states such as rotations that do not naturally live in Euclidean vector spaces.

1.5.6 Particle Filters

Kalman-style filters compress uncertainty into a single Gaussian. This representation cannot naturally describe several widely separated hypotheses. A particle filter instead represents the posterior with weighted samples $\{x_t^{(i)}, w_t^{(i)}\}_{i=1}^N$, approximating

$$\boxed{p(x_t \mid z_{1:t}) \approx \sum_{i=1}^{N} w_t^{(i)} \delta(x_t - x_t^{(i)}).}$$

During prediction, each particle is propagated through the motion model. For stochastic dynamics this can be written $x_t^{(i)} \sim p(x_t \mid x_{t-1}^{(i)}, u_{t-1})$.

When a measurement arrives, particles that better explain the observation receive larger weights. In a bootstrap particle filter the unnormalized update is

$$\tilde{w}_t^{(i)} = w_{t-1}^{(i)} p(z_t \mid x_t^{(i)}),$$

followed by normalization $w_t^{(i)} = \tilde{w}_t^{(i)} / \sum_j \tilde{w}_t^{(j)}$.

Over time, most particles may acquire nearly zero weight while only a few dominate. This phenomenon is called *particle degeneracy*. A useful diagnostic is the effective sample size $N_{\text{eff}} \approx 1/\sum_i (w_t^{(i)})^2$. When N_{eff} becomes too small, the filter resamples particles in proportion to their weights.

Resampling removes low-probability hypotheses and duplicates high-probability hypotheses. After resampling, the particles are commonly assigned equal weights.

The major advantage of particle filters is their ability to represent multimodal posteriors. If a robot could plausibly occupy either of two distant corridors, the particle distribution can contain one cluster in each corridor. A single Gaussian estimate would instead tend to summarize these possibilities using one mean and covariance, possibly placing the mean in a location where the robot is unlikely to be.

The principal disadvantage is scaling. A large number of particles may be required to represent uncertainty accurately in a high-dimensional state space, making particle filters especially useful for lower-dimensional localization problems but less attractive for very large inertial-navigation states.

1.5.7 Sensor Fusion

No single sensor provides perfect information about a robot's state. Sensor fusion combines measurements with complementary strengths so that one sensor compensates for limitations in another.

An IMU provides high-rate angular velocity and specific-force measurements but accumulates drift when integrated. Cameras provide rich geometric constraints but depend on scene texture, lighting, and successful feature or image matching. LiDAR provides accurate geometric range information but may be lower-rate and computationally expensive. GPS provides global position outdoors but may be unavailable or degraded near buildings and indoors. Wheel encoders provide precise local motion information but accumulate error through slip.

A fused estimator combines these sources through a common latent state. A practical navigation state may contain

$$x = \begin{bmatrix} p & v & q & b_a & b_g \end{bmatrix}^T,$$

where p is position, v velocity, q orientation, and b_a and b_g are accelerometer and gyroscope biases.

High-rate IMU measurements can propagate the state, while lower-rate measurements from cameras, GPS, wheel odometry, or LiDAR correct accumulated drift. The estimator therefore separates sensors according to their information content rather than merely averaging their outputs.

A measurement should influence the state according to both its uncertainty and the correlation structure of the state covariance. This is the deeper meaning of probabilistic sensor fusion: information is combined in proportion to what each sensor can actually constrain.

1.5.8 Observability

Before designing an estimator, it is important to ask whether the desired state can be inferred from the available measurements at all. This property is called *observability*.

For the linear system $x_{t+1} = Ax_t$ with measurements $y_t = Cx_t$, the observability matrix is

$$\boxed{\mathcal{O} = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix}.}$$

The complete state is observable if $\operatorname{rank}(\mathcal{O}) = n$.

The key point is that observability is a property of the system and measurements, not of the estimator algorithm. If some component of the state never influences the sensor measurements, no Kalman filter, neural network, or optimizer can recover it uniquely from those measurements alone.

Robotics frequently contains partially observable directions. Monocular visual odometry, for example, cannot recover absolute metric scale without additional information. Pure inertial integration cannot determine global position over long periods because small bias errors integrate into drift. Visual–inertial systems can constrain metric scale through accelerometer measurements, but global position and absolute yaw may remain unobservable without external references such as GPS, landmarks, or magnetometers.

Observability can also depend on motion. A sensor configuration may theoretically provide information about a parameter, yet particular trajectories may fail to excite that parameter sufficiently. Calibration and state-estimation systems therefore benefit from trajectories that create informative motion.

1.5.9 IMU Integration

An inertial measurement unit typically provides gyroscope measurements ω_m and accelerometer measurements a_m. These measurements are commonly modeled as $\omega_m = \omega + b_g + n_g$ and $a_m = R^T(a - g) + b_a + n_a$, where b_g and b_a are slowly varying biases and n_g and n_a are measurement noise.

After bias correction, angular velocity is approximately $\omega = \omega_m - b_g$. Over a short timestep Δt, the incremental rotation vector is $\Delta\phi \approx \omega\Delta t$.

Using quaternions, the corresponding incremental rotation is

$$\Delta q = \begin{bmatrix} \cos(\|\omega\|\Delta t/2) \\ \dfrac{\omega}{\|\omega\|}\sin(\|\omega\|\Delta t/2) \end{bmatrix},$$

and the orientation can be propagated using $q_{k+1} = q_k \otimes \Delta q$. For sufficiently small Δt, $\Delta q \approx [1, \frac{1}{2}\omega\Delta t]^T$.

Accelerometer measurements require more care because an accelerometer measures *specific force*, not simply world-frame linear acceleration. After removing bias and rotating the measurement into the world frame, the acceleration estimate is

$$\boxed{a_W = R_{WB}(a_m - b_a) + g_W.}$$

Velocity and position can then be propagated approximately as $v_{k+1} = v_k + a_W\Delta t$ and $p_{k+1} = p_k + v_k\Delta t + \frac{1}{2}a_W\Delta t^2$.

These equations appear simple, but repeated integration makes inertial navigation extremely sensitive to error. A small constant gyroscope bias creates an orientation error that grows with time. That orientation error causes gravity to be projected incorrectly into the horizontal acceleration estimate. The resulting acceleration error is then integrated once into velocity error and again into position error.

Similarly, a constant accelerometer bias b_a produces velocity error that grows approximately linearly with time and position error that grows approximately quadratically.

For this reason, inertial measurements are excellent for high-rate short-term propagation but require external measurements to prevent long-term drift.

1.5.10 Bias Estimation

IMU biases are usually included directly in the estimator state because treating them as known constants leads to rapid drift. A common bias model is a random walk such as $b_{g,k+1} = b_{g,k} + w_{bg}$ and $b_{a,k+1} = b_{a,k} + w_{ba}$.

The estimator then learns the biases indirectly from disagreement between inertial predictions and external measurements. For example, if visual measurements repeatedly indicate that the inertial prediction rotates slightly too fast, the estimator can attribute part of the discrepancy to gyroscope bias.

Bias estimation illustrates an important principle in sensor fusion: states do not need to be measured directly to be estimated. They need only influence measurements in a sufficiently observable way.

1.5.11 Visual-Inertial Estimation

Visual–inertial estimation combines the complementary characteristics of cameras and IMUs. The IMU provides high-frequency motion information and remains useful during rapid motion, while visual observations provide geometric constraints that correct inertial drift.

A typical visual–inertial state contains position p, velocity v, orientation R or quaternion q, accelerometer bias b_a, and gyroscope bias b_g. Depending on the system, it may also contain camera–IMU calibration parameters, landmarks, past camera poses, or timing offsets.

Between camera frames, IMU data propagate the state forward. When a new image arrives, visual measurements compare predicted scene geometry with observed image features or image intensities. The resulting residual corrects the inertial state.

For a landmark P_W observed by a camera, the predicted camera-frame point is $P_C = R_{CW}(P_W - p_W)$, followed by perspective projection into the image. A visual residual can then be formed from the difference between the observed and predicted pixel location.

Thus the estimator combines an inertial motion model with a geometric camera measurement model.

Why the Sensors Complement Each Other

A camera can infer relative motion from changes in image geometry, but monocular vision alone has difficulty determining metric scale. An accelerometer measures motion in physical units and therefore introduces metric information.

The IMU, however, drifts rapidly when integrated because of bias and noise. Camera observations constrain that drift by repeatedly tying the trajectory back to scene geometry.

The relationship can therefore be summarized conceptually as

$$\text{IMU} \rightarrow \text{high-rate propagation}, \qquad \text{vision} \rightarrow \text{drift correction}.$$

Visual–inertial odometry estimates local motion over time, while visual–inertial SLAM additionally maintains persistent map information and may incorporate loop closures that correct accumulated long-term drift.

1.5.12 Filtering and Optimization-Based Estimation

Visual–inertial estimation can be implemented with recursive filters such as an error-state EKF or with nonlinear optimization over a sliding time window.

A filter maintains a current state and covariance and incorporates measurements sequentially. This provides bounded computational cost and is attractive for real-time embedded systems.

A sliding-window optimizer instead retains a recent set of poses, velocities, biases, and visual landmarks and minimizes the joint measurement error over that window. Schematically, the estimator solves

$$\min_X \sum_k \|r_k(X)\|^2_{\Sigma_k^{-1}},$$

where the residuals may include visual reprojection errors, IMU integration errors, priors, and other sensor measurements.

Optimization-based estimators can relinearize previous measurements as the state estimate changes, which is often advantageous for strongly nonlinear problems. Their main cost is additional computation and memory.

Both approaches use the same underlying physical information; they differ primarily in how measurements are retained and how the estimation problem is numerically solved.

1.5.13 State Estimation on $SO(3)$ and $SE(3)$

Robot pose does not naturally live in a Euclidean vector space. Orientation belongs to $SO(3)$, while a rigid-body pose belongs to $SE(3)$. Directly adding Gaussian perturbations to the entries of a rotation matrix can violate the constraints $R^T R = I$ and $\det R = 1$.

Modern estimators therefore commonly distinguish between the *nominal state*, which lives on the manifold, and a small *error state*, which lives in a local Euclidean tangent space.

Suppose the nominal orientation is $\hat{R} \in SO(3)$. A nearby orientation can be represented as

$$R = \hat{R}\,\mathrm{Exp}(\delta\phi^\wedge),$$

where $\delta\phi \in \mathbb{R}^3$ is a small rotational error.

For small error, $\mathrm{Exp}(\delta\phi^\wedge) \approx I + \delta\phi^\wedge$. The orientation therefore remains a valid rotation matrix while the uncertainty can be represented using the ordinary three-dimensional vector $\delta\phi$.

The same construction extends to pose. If $\hat{T} \in SE(3)$ is the nominal transformation and $\delta\xi \in \mathbb{R}^6$ is a small pose perturbation, one may write

$$\boxed{T = \hat{T}\,\mathrm{Exp}(\delta\xi^\wedge).}$$

The covariance is then maintained over $\delta\xi$, not directly over the constrained entries of T.

When an update produces a correction $\delta\xi$, the state is retracted back onto the manifold using the exponential map. The error state is then reset around the updated nominal pose.

This is the central idea behind error-state filters and manifold-based estimators.

1.5.14 Pose Residuals on $SE(3)$

Pose differences should likewise respect the group structure. If T_d is a measured or desired pose and T is the current estimate, ordinary matrix subtraction $T_d - T$ does not produce a meaningful

six-dimensional pose error.

Instead, a relative transformation can be formed as $T_d^{-1}T$, and the logarithmic map converts it into a local vector:

$$\boxed{e = \operatorname{Log}\left(T_d^{-1}T\right)^{\vee} \in \mathbb{R}^6.}$$

This type of residual appears in pose-graph SLAM, calibration, nonlinear state estimation, and trajectory optimization.

For example, if two poses T_i and T_j are connected by a measured relative pose $\hat{T}_{ij}$, a pose-graph residual can be written as

$$e_{ij} = \operatorname{Log}\left(\hat{T}_{ij}^{-1}T_i^{-1}T_j\right)^{\vee}.$$

The logarithmic map transforms the nonlinear group discrepancy into a local Euclidean error that can be used in least-squares optimization.

1.5.15 Error-State Filtering

The error-state formulation is particularly common in inertial navigation. The nominal state may contain p, v, R, b_a, and b_g, while the small error state contains

$$\delta x = \begin{bmatrix}\delta p & \delta v & \delta\phi & \delta b_a & \delta b_g\end{bmatrix}^T.$$

The nominal state is propagated using the nonlinear IMU equations. Meanwhile the covariance of δx is propagated using a linearized error-dynamics model.

When another sensor produces a correction, the filter estimates δx. Translational, velocity, and bias errors are added normally, while the rotation correction is applied through $R \leftarrow R\operatorname{Exp}(\delta\phi^{\wedge})$. The error state is then reset to zero and the covariance is adjusted accordingly.

This architecture avoids treating orientation as an unconstrained Euclidean quantity while retaining the computational structure of a Kalman filter.

1.5.16 Choosing an Estimation Method

The different estimators described above solve the same Bayesian inference problem using different approximations.

The linear Kalman filter is appropriate when the system and measurement models are linear and the uncertainty is well described by a Gaussian distribution. The EKF extends the same representation to nonlinear systems through local linearization. The UKF avoids explicit Jacobians and instead propagates representative sigma points through the nonlinear functions. Particle filters can represent strongly non-Gaussian or multimodal posteriors but scale poorly with state dimension.

For robot navigation, the EKF and error-state EKF remain common when computational efficiency and high update rate are important. Optimization-based estimators and factor graphs are frequently used when many nonlinear geometric measurements must be jointly reconciled. Particle

filters remain particularly valuable when the central challenge is ambiguity between several distinct pose hypotheses.

The choice is therefore not determined simply by which method is most mathematically sophisticated. It depends on the state dimension, degree of nonlinearity, expected posterior shape, sensor rates, computational resources, and whether the application requires maintaining multiple hypotheses.

1.5.17 Summary

State estimation reconstructs hidden robot state from imperfect models and noisy measurements. The general Bayesian filtering framework alternates between a prediction step,

$$p(x_t \mid z_{1:t-1}) = \int p(x_t \mid x_{t-1}) p(x_{t-1} \mid z_{1:t-1}) dx_{t-1},$$

and a measurement correction,

$$p(x_t \mid z_{1:t}) \propto p(z_t \mid x_t) p(x_t \mid z_{1:t-1}).$$

The Kalman filter implements this recursion exactly for linear Gaussian systems. The Extended Kalman Filter applies the same structure to nonlinear systems by propagating the mean through the true nonlinear model and propagating uncertainty through local Jacobians. The Unscented Kalman Filter instead propagates carefully selected sigma points through the nonlinear model, while particle filters represent the posterior using weighted samples and can maintain multiple distinct hypotheses.

Sensor fusion combines complementary measurements through a shared latent state. IMUs provide high-frequency motion propagation but drift under integration; cameras, LiDAR, GPS, and other sensors provide observations that constrain this drift. Visual–inertial estimation is a particularly important example in which inertial propagation and visual geometric correction work together.

Observability determines which components of the state can actually be inferred from the available motion and measurements. No estimator can reliably recover a fundamentally unobservable quantity.

Finally, robot pose estimation requires respecting the geometry of $SO(3)$ and $SE(3)$. Rather than treating rotations and poses as ordinary Euclidean vectors, modern estimators commonly maintain nominal states on the manifold and represent small errors in the corresponding tangent space. Exponential and logarithmic maps then connect local uncertainty with globally valid rigid-body transformations.

Together, probabilistic estimation, sensor fusion, observability analysis, inertial propagation, and manifold-aware pose representation provide the state information required by perception, mapping, planning, and feedback control.

1.6 Localization and Mapping

A mobile robot must continually answer two closely related questions: *where am I?* and *what does the environment around me look like?* Localization estimates the robot's pose relative to some reference frame, while mapping constructs a representation of the surrounding environment. In many practical systems these problems are coupled, because localization requires knowledge of the environment while accurate mapping requires knowledge of the robot trajectory.

Odometry provides short-term relative motion estimates. Mapping integrates observations into a persistent representation. Simultaneous Localization and Mapping, or SLAM, estimates both the trajectory and the map while accounting for uncertainty and revising previous estimates when new information becomes available.

The geometric state of the robot is commonly represented by a rigid transformation $T_{WB} \in SE(3)$, describing the pose of the robot body frame B relative to a world frame W. A localization system therefore estimates a sequence $T_0, T_1, \ldots, T_N$, while a mapping system additionally estimates environmental variables such as landmarks, surfaces, occupancy, or dense geometry.

1.6.1 Odometry

Odometry estimates incremental motion from measurements generated as the robot moves. Rather than determining an absolute global pose directly, it typically estimates the relative transformation between consecutive states.

Let $\hat{T}_{k-1,k}$ denote the estimated motion from timestep $k-1$ to k. The global pose can be propagated recursively as

$$\boxed{T_{W,k} = T_{W,k-1}\hat{T}_{k-1,k}.}$$

The relative transform may be estimated from wheel encoders, inertial sensors, cameras, LiDAR, joint kinematics, or combinations of these sensors.

Wheel odometry, for example, estimates motion from wheel rotations. For a differential-drive robot with left and right wheel travel Δs_L and Δs_R, the approximate forward displacement is $\Delta s = (\Delta s_R + \Delta s_L)/2$, while the heading change for wheel separation b is $\Delta\theta = (\Delta s_R - \Delta s_L)/b$. These increments can then be integrated into the robot pose.

Odometry is fundamentally incremental, so errors accumulate. If each estimated transform contains a small error,

$$\hat{T}_{k-1,k} = T_{k-1,k}\Delta T_k,$$

then repeatedly composing these estimates also composes their errors. The resulting trajectory gradually drifts away from the true trajectory.

This distinction is important:

$$\text{odometry} \quad \text{provides locally consistent relative motion,}$$

while

global localization requires external or repeated environmental constraints.

A robot may therefore have excellent short-term odometry while accumulating substantial long-term position or heading error.

The nature of the drift depends on the sensor. Wheel odometry is affected by slip and imperfect wheel calibration. IMU odometry is affected strongly by bias and integration drift. Visual odometry can fail under poor texture, lighting changes, blur, or insufficient parallax. LiDAR odometry can become weak in geometrically repetitive environments such as long corridors.

1.6.2 Visual Odometry

Visual odometry estimates camera motion from a sequence of images. Given images I_{k-1} and I_k, the objective is to estimate the relative camera transformation $T_{k-1,k}$.

Visual odometry can broadly be divided into *feature-based* and *direct* approaches.

Feature-based methods first identify repeatable image features, match them between frames, and infer camera motion from the resulting geometric correspondences. A typical pipeline is

detect image features
$\downarrow$
match or track features
$\downarrow$
reject outliers
$\downarrow$
estimate relative pose
$\downarrow$
refine motion estimate.

For two calibrated monocular views, corresponding normalized image coordinates x_1 and x_2 satisfy the epipolar constraint $x_2^T E x_1 = 0$, where the essential matrix is $E = [t]_\times R$. Estimating E therefore constrains the relative rotation R and translation direction t.

A fundamental limitation is that monocular two-view geometry does not determine the magnitude of translation. If t satisfies the epipolar geometry, then so does αt for positive scale α. Monocular visual odometry therefore lacks absolute metric scale unless additional information is introduced.

Stereo cameras, RGB-D cameras, known object geometry, wheel odometry, or inertial sensing can provide metric scale.

Once three-dimensional landmarks have been reconstructed, subsequent camera poses can often be estimated through Perspective-n-Point. Given world points P_i and image observations p_i, PnP estimates R, t satisfying $\lambda_i \tilde{p}_i = K(RP_i + t)$.

Direct visual odometry avoids explicitly extracting sparse features and instead minimizes photometric inconsistency between images. If pixel u in one frame corresponds to a projected location

u' in another, a direct residual may be written as $r(u) = I_k(u') - I_{k-1}(u)$. Motion is then estimated by minimizing the photometric residual over many pixels.

Feature methods tend to be robust to moderate illumination changes and can operate on sparse measurements. Direct methods use more image information but require a more accurate photometric model and a sufficiently good initialization.

Regardless of the method, visual odometry produces primarily local motion estimates. Without global constraints, the resulting trajectory still accumulates drift.

1.6.3 Mapping

Mapping constructs a persistent representation of the environment from robot observations. The appropriate map representation depends strongly on the task.

A landmark map stores sparse geometric features such as points $P_j \in \mathbb{R}^3$. This representation is common in visual SLAM because image observations can be related directly to landmark projections.

A point-cloud map stores large collections of three-dimensional measurements,

$$\mathcal{P} = \{p_1, p_2, \ldots, p_N\}, \qquad p_i \in \mathbb{R}^3.$$

Such maps are natural for LiDAR and depth-camera systems. Each new scan can be transformed into the global frame and accumulated into the existing map.

An occupancy map represents whether regions of space are free or occupied. In a two-dimensional occupancy grid, the environment is divided into cells, each storing a probability such as $p(m_i = \text{occupied})$. Sensor measurements update these probabilities as new evidence is observed.

For numerical convenience, occupancy-grid mapping often uses log odds. If p_i is the occupancy probability of cell i, define $l_i = \log[p_i/(1 - p_i)]$. Independent measurement updates then become additive rather than requiring repeated multiplication of probabilities.

Dense maps may instead represent surfaces using meshes, signed-distance functions, truncated signed-distance fields, surfels, or learned implicit representations.

The central difficulty in mapping is that every measurement must be placed in a common coordinate system. If a point P_C is measured in camera coordinates and the camera pose is T_{WC}, then its world position is determined by $P_W = T_{WC}P_C$ in homogeneous notation.

Consequently, pose error directly creates map error. If localization drifts, repeated observations of the same physical wall may appear as several misaligned walls in the map. This coupling motivates SLAM.

1.6.4 Simultaneous Localization and Mapping

Simultaneous Localization and Mapping estimates the robot trajectory and the environment jointly from noisy measurements.

Given observations $z_{1:T}$ and controls or motion inputs $u_{1:T}$, SLAM can be written probabilistically as

$$\boxed{p(x_{1:T}, m \mid z_{1:T}, u_{1:T}),}$$

where $x_{1:T}$ is the trajectory and m represents map variables.

The coupling arises because observations of the environment improve the trajectory estimate, while the trajectory estimate determines where those observations should be placed in the map.

A useful conceptual decomposition separates a SLAM system into a *front end* and a *back end*. The front end converts raw sensor data into geometric constraints. It may perform feature detection, feature matching, scan registration, place recognition, landmark association, or visual tracking. The back end optimizes the robot trajectory and map so that these constraints are mutually consistent.

Thus,

raw sensor measurements
↓
data association and relative constraints
↓
graph or optimization problem
↓
globally consistent trajectory and map.

This separation is not absolute, but it is useful conceptually. Front-end failures often arise from incorrect correspondence or place recognition, while back-end errors arise from optimization, modeling, or inconsistent uncertainty assumptions.

1.6.5 Local Registration

Many SLAM systems obtain relative-pose constraints by aligning successive sensor observations.

For point clouds, Iterative Closest Point estimates the rigid transform aligning a source cloud p_i to a target cloud q_i. Point-to-point ICP minimizes

$$\min_{R,t} \sum_i \|Rp_i + t - q_i\|^2.$$

A commonly used point-to-plane form instead minimizes

$$\min_{R,t} \sum_i \left[n_i^T (Rp_i + t - q_i)\right]^2,$$

where n_i is the target surface normal.

ICP alternates between estimating correspondences and updating the rigid transformation. Because the optimization is local, it generally requires a sufficiently accurate initial alignment.

When registration is performed between consecutive scans, the result provides LiDAR odometry. When a current scan is aligned against a persistent local or global map, the same geometric principle can provide localization relative to that map.

1.6.6 Pose Graphs

As a robot travels, relative-pose measurements can be organized as a graph. Each node represents a robot pose $T_i \in SE(3)$, while an edge between nodes i and j represents a measured relative transformation $\hat{T}_{ij}$.

If the current pose estimates were perfectly consistent with the measurement, they would satisfy $T_i^{-1} T_j = \hat{T}_{ij}$.

Measurement noise and accumulated drift prevent exact consistency, so an error is defined on $SE(3)$:

$$\boxed{e_{ij} = \operatorname{Log}\left(\hat{T}_{ij}^{-1} T_i^{-1} T_j\right)^{\vee}.}$$

The transformation inside the logarithm compares the relative pose predicted by the current trajectory with the measured relative pose. The logarithmic map converts this discrepancy into a local vector in $\mathbb{R}^6$.

If each measurement has information matrix Ω_{ij}, pose-graph optimization solves

$$\boxed{\min_{\{T_i\}} \sum_{(i,j)} e_{ij}^T \Omega_{ij} e_{ij}.}$$

The information matrix is approximately the inverse of the measurement covariance, so reliable constraints receive greater weight than uncertain constraints.

Most odometry edges connect nearby states,

$$T_1 \leftrightarrow T_2 \leftrightarrow T_3 \leftrightarrow \cdots,$$

but SLAM becomes especially powerful when additional edges connect states observed far apart in time.

Because relative measurements do not by themselves determine an absolute world coordinate system, a pose graph has a gauge freedom. One pose, typically T_0, is therefore fixed or given a strong prior to define the reference frame.

1.6.7 Loop Closure

A loop closure occurs when the robot recognizes a location that it has previously visited. Suppose the robot returns near pose T_i after traveling for a long time and is currently at T_j. Local odometry may have accumulated substantial drift, so the two pose estimates may disagree even though the physical locations are nearly the same.

Recognizing the place allows the SLAM system to add a constraint $\hat{T}_{ij}$ connecting the distant portions of the trajectory.

Before loop closure, the graph may resemble a long chain of local odometry constraints. Adding a loop-closure edge creates a cycle:

$$T_i \longleftrightarrow T_j.$$

Graph optimization can then distribute the accumulated inconsistency across many previous poses rather than applying one abrupt correction at the current timestep.

Loop closure therefore transforms SLAM from simple incremental odometry into a globally corrective estimator.

The process usually contains two distinct stages. *Place recognition* determines whether the current observation might correspond to a previously visited location. *Geometric verification* then checks whether the proposed match is physically consistent and estimates the relative transformation.

This distinction is critical because a false loop closure can be much more damaging than a missed loop closure. An incorrect constraint may force unrelated areas of the map together and corrupt the entire optimized trajectory.

For this reason, practical systems commonly require strong geometric verification, inlier testing, consistency checks, and sometimes robust loss functions before accepting a loop closure.

1.6.8 Robust Estimation

Data association is imperfect. Feature matches, scan correspondences, and loop closures can contain outliers, while ordinary least squares heavily penalizes large residuals.

Instead of minimizing only $e^T \Omega e$, a SLAM system may minimize a robustified objective

$$\sum_k \rho\left(e_k^T \Omega_k e_k\right),$$

where $\rho(\cdot)$ grows more slowly than the quadratic loss for large residuals.

The purpose is not to ignore all large errors. A large residual may represent a legitimate constraint that exposes accumulated drift. Rather, robust losses reduce the influence of measurements whose residuals are far outside what the assumed Gaussian noise model predicts.

Robust estimation is especially important for loop closure because perceptual aliasing can produce confident but incorrect place matches.

1.6.9 Bundle Adjustment

Pose-graph optimization estimates poses from relative-pose constraints. Visual SLAM often contains a richer geometric problem in which both camera poses and three-dimensional landmarks are estimated directly from image measurements.

This joint optimization is called *bundle adjustment*.

Let T_i denote camera pose i, P_j landmark j, and z_{ij} the observed pixel location of landmark j in camera i. The predicted image location is obtained by transforming the landmark into the camera frame and projecting it using the camera model.

If $\pi(\cdot)$ denotes perspective projection, the reprojection residual is

$$r_{ij} = z_{ij} - \pi\left(T_i^{-1} P_j\right).$$

Bundle adjustment solves

$$\boxed{\min_{\{T_i\},\{P_j\}} \sum_{(i,j)\in\mathcal{O}} \rho\left(r_{ij}^T \Sigma_{ij}^{-1} r_{ij}\right),}$$

where $\mathcal{O}$ is the set of observed pose–landmark pairs and ρ may be a robust loss.

The optimization adjusts both camera poses and landmark positions so that the reconstructed scene explains the measured image locations as consistently as possible.

Bundle adjustment is powerful because every landmark seen from several viewpoints couples those viewpoints together. Moving one camera pose changes the reprojection residuals of every landmark observed in that frame, while moving one landmark changes residuals across every camera that sees it.

The resulting optimization can be very large, but its Jacobian is sparse because each image observation depends only on one camera pose and one landmark. Modern solvers exploit this sparse structure, often eliminating landmark variables using the Schur complement before solving for camera-pose updates.

Bundle adjustment is commonly used for local refinement in visual odometry and for global refinement in visual SLAM and structure-from-motion systems.

1.6.10 Pose Graph Optimization and Bundle Adjustment

Pose graphs and bundle adjustment are related but optimize different variables.

A pose graph typically contains only robot poses as variables, with relative-pose measurements between them. It is compact and therefore well suited for large-scale trajectory correction.

Bundle adjustment retains the original geometric landmark observations and optimizes both camera poses and landmarks. It preserves more information but generally creates a larger optimization problem.

A visual SLAM system may therefore use local bundle adjustment for accurate geometric refinement while maintaining a higher-level pose graph for large-scale loop closure and global consistency.

1.6.11 Factor Graphs

A factor graph provides a general probabilistic representation for state-estimation problems. Variable nodes represent unknown quantities such as poses, velocities, IMU biases, or landmarks. Factor nodes represent measurements or probabilistic relationships between subsets of these variables.

Suppose the unknown variables are collectively denoted X and the measurements are Z. Under conditional independence assumptions, the posterior can be factored as

$$\boxed{p(X \mid Z) \propto \prod_k \phi_k(X_k),}$$

where factor ϕ_k depends only on a small subset X_k of the complete state.

If each measurement has approximately Gaussian error, a factor can be expressed through a residual $r_k(X_k)$ and covariance Σ_k. Maximum a posteriori estimation then becomes a nonlinear least-squares problem,

$$\boxed{X^\star = \arg\min_X \sum_k r_k(X_k)^T \Sigma_k^{-1} r_k(X_k).}$$

A pose-graph edge is therefore one example of a factor. An odometry factor relates two neighboring poses. A landmark observation factor relates a pose and a landmark. An IMU factor may relate pose, velocity, and bias states at consecutive keyframes. GPS may produce a unary position factor, while a prior factor anchors the coordinate frame.

The power of the factor-graph representation comes from sparsity. A particular measurement usually depends on only a small subset of the complete trajectory. The corresponding Jacobian therefore contains large regions of zeros, allowing sparse numerical methods to solve estimation problems involving many thousands of variables.

Factor graphs also provide a unified language for combining heterogeneous sensors. Rather than designing a separate estimation procedure for every sensor combination, each sensor contributes factors encoding its measurement model and uncertainty.

1.6.12 Linearization in Graph-Based Estimation

SLAM measurement models are nonlinear, so graph optimization proceeds iteratively.

Let the current state estimate be X, and let δX be a local perturbation. A residual can be approximated as

$$r(X \oplus \delta X) \approx r(X) + J\delta X,$$

where J is the residual Jacobian and $\oplus$ denotes an appropriate state update, including manifold-aware updates for poses.

The nonlinear least-squares problem then produces linear normal equations of the form

$$\boxed{H\delta X = -b,}$$

where H is an approximate Hessian assembled from measurement Jacobians and information matrices.

After solving for δX, the states are updated and the system is linearized again. Gauss–Newton and Levenberg–Marquardt are common algorithms for this iterative process.

For pose variables $T \in SE(3)$, the update should preserve the rigid-body structure. One possible update is $T \leftarrow T\,\mathrm{Exp}(\delta\xi^\wedge)$, with $\delta\xi \in \mathbb{R}^6$.

This connects graph-based localization directly to the $SE(3)$ machinery introduced earlier in the chapter.

1.6.13 Visual–Inertial SLAM

Visual–inertial SLAM combines camera observations with inertial measurements while maintaining both a trajectory and a persistent representation of the environment.

A typical state at keyframe k contains pose T_k, velocity v_k, gyroscope bias $b_{g,k}$, and accelerometer bias $b_{a,k}$. Landmark positions may also be included explicitly.

The IMU provides high-rate motion measurements between camera frames. Images provide geometric constraints that prevent the inertial trajectory from drifting without bound.

At a high level,

IMU measurements
↓
high-rate motion propagation
↓
camera keyframe
↓
visual geometric constraints
↓
joint trajectory and map optimization.

1.6.14 IMU Constraints Between Keyframes

A camera may operate at tens of frames per second while the IMU operates at hundreds or thousands of hertz. Optimizing one state for every IMU measurement would create an unnecessarily large problem.

Visual–inertial systems therefore commonly summarize the many IMU measurements between two keyframes into a compact relative-motion constraint. This idea is known as *IMU preintegration*.

Between keyframes i and j, the gyroscope and accelerometer measurements are integrated to construct approximate relative rotation, velocity, and position increments. Conceptually these quantities represent

$$\Delta R_{ij}, \qquad \Delta v_{ij}, \qquad \Delta p_{ij}.$$

Rather than repeatedly reintegrating all raw IMU samples whenever the global pose estimate changes, the preintegrated quantities provide a reusable constraint between the two keyframe states.

An IMU factor therefore connects variables such as T_i, v_i, b_i and T_j, v_j, b_j.

The camera contributes visual factors, while the IMU contributes inertial factors. Joint optimization finds a trajectory that explains both.

1.6.15 Visual Factors

Suppose landmark P_j is observed in camera keyframe i at pixel z_{ij}. The visual residual is a reprojection error,

$$r_{ij}^{\text{vis}} = z_{ij} - \pi\left(T_{CW,i} P_j\right).$$

These visual constraints provide strong information about relative geometry.

However, monocular vision alone has an unknown metric scale. The same image geometry can be produced by simultaneously scaling the scene and camera translations.

The accelerometer measures motion in metric units and couples the visual reconstruction to gravity and physical acceleration. As a result, visual–inertial estimation can recover metric scale once sufficient informative motion has occurred.

1.6.16 Bias Estimation in Visual–Inertial SLAM

Gyroscope and accelerometer biases are typically included in the optimization state. Even small biases create substantial drift when inertial measurements are repeatedly integrated.

A visual–inertial system may model the biases as slowly varying random walks. Visual observations then indirectly constrain them.

For example, if integrated gyroscope measurements consistently predict more rotation than the images support, the optimizer can adjust the gyroscope bias rather than incorrectly changing the scene geometry.

The estimator therefore solves not only for where the robot moved, but also for systematic errors in the sensor model.

1.6.17 Observability in Visual–Inertial SLAM

Combining vision and inertial sensing resolves several ambiguities but not every possible degree of freedom.

Monocular visual SLAM alone has an arbitrary global scale. Visual–inertial estimation can make this scale observable through acceleration measurements.

Gravity also provides information about roll and pitch. Once the direction of gravity is estimated, orientation relative to the vertical direction is constrained.

However, without an external global reference, absolute position remains arbitrary because translating the complete trajectory and map together does not change the internal measurements. Absolute yaw about the gravity direction may likewise remain unobservable in many visual–inertial configurations.

These freedoms are not estimation failures. They are properties of the information available to the system.

The estimator must therefore distinguish between

quantities that are uncertain

and

quantities that are fundamentally unobservable.

Adding GPS, magnetometer information, known landmarks, or another global reference can constrain some of these remaining degrees of freedom.

1.6.18 Keyframes and Sliding Windows

Keeping every image and every past state in an online optimization problem would eventually make computation unbounded. Practical visual SLAM systems therefore select a subset of images as *keyframes*.

Keyframes are retained because they provide useful geometric information or sufficient viewpoint change relative to existing frames.

A sliding-window visual–inertial estimator may optimize only the most recent N keyframes together with their velocities, biases, and observed landmarks. Older states are removed from the active problem while their information is summarized into a prior through marginalization.

This allows computational cost to remain approximately bounded while retaining information from past observations.

A complete SLAM system may additionally maintain a longer-term global pose graph containing many historical keyframes. Local estimation then provides accurate short-term motion, while the global graph handles long-range loop closures.

1.6.19 Local and Global Consistency

An important distinction in mapping systems is the difference between *local accuracy* and *global consistency*.

Visual or LiDAR odometry may produce highly accurate relative motion over short periods while still drifting globally. Local bundle adjustment can further improve the relative geometry within a small region.

Loop closure introduces long-range constraints, and global pose-graph or bundle-adjustment optimization redistributes accumulated drift across the trajectory.

A modern SLAM architecture can therefore be understood as several coupled estimation scales:

frame-to-frame tracking

↓

local odometry

↓

local mapping / bundle adjustment

↓

loop closure

↓

global optimization.

Each stage solves a different part of the same geometric estimation problem.

1.6.20 Localization Against an Existing Map

SLAM is required when the map is unknown. If a sufficiently accurate map already exists, the problem can instead be treated as localization.

A LiDAR-equipped robot may align each current scan against a previously constructed point-cloud map. A camera may localize using known visual landmarks through PnP. A mobile robot may compare observations against an occupancy map.

The unknown variables can then be limited primarily to the robot pose rather than the pose and map simultaneously.

This is usually a simpler estimation problem, although challenges such as perceptual aliasing, environmental change, and poor initialization remain.

A practical robotic system may alternate between these modes: it may localize against a known map during ordinary operation but update or rebuild portions of the map when the environment changes.

1.6.21 Relationship Between Odometry, Mapping, and SLAM

Odometry, mapping, localization, and SLAM are closely related but should not be treated as interchangeable terms.

Odometry estimates incremental motion,

$$T_{k-1,k}.$$

Localization estimates the robot's pose relative to an external map or reference frame,

$$T_{W,k}.$$

Mapping estimates an environmental representation m, assuming sufficiently accurate robot poses are available.

SLAM estimates both,

$$(T_{1:T}, m),$$

using repeated environmental observations to correct trajectory drift.

Loop closure supplies long-range geometric constraints, pose graphs provide a compact representation of those constraints, bundle adjustment jointly refines visual poses and landmarks, and factor graphs provide a general probabilistic framework in which all of these measurement types can be combined.

Visual–inertial SLAM extends this framework by coupling image geometry with high-rate inertial motion constraints.

1.6.22 Summary

Localization and mapping connect short-term motion estimation with persistent geometric understanding of the environment.

Odometry estimates relative motion and provides locally smooth trajectories, but incremental error causes long-term drift. Visual odometry derives these relative motions from image geometry using feature correspondences, PnP, epipolar geometry, or direct photometric alignment.

Mapping transforms sensor measurements into a common reference frame to construct representations such as landmark maps, occupancy grids, point clouds, meshes, or dense surface models. Because mapping depends on accurate poses and localization often depends on an accurate map, the two problems naturally become coupled.

SLAM estimates the trajectory and map simultaneously. Modern SLAM systems commonly separate front-end measurement processing and data association from back-end nonlinear optimization.

Pose graphs represent robot poses as nodes and relative measurements as edges. Their residuals can be defined directly on $SE(3)$,

$$e_{ij} = \operatorname{Log}\left(\hat{T}_{ij}^{-1} T_i^{-1} T_j\right)^{\vee},$$

and global optimization adjusts the poses to make the graph mutually consistent.

Loop closure adds long-range constraints when the robot revisits previously observed locations, allowing accumulated odometric drift to be corrected.

Bundle adjustment jointly optimizes camera poses and three-dimensional landmarks using reprojection error, while factor graphs generalize the estimation problem to arbitrary collections of sparse probabilistic constraints.

Visual–inertial SLAM combines visual geometry with high-rate IMU measurements. The IMU provides short-term motion propagation and metric information, while vision constrains inertial drift. Bias estimation, IMU preintegration, keyframe selection, local optimization, loop closure, and global graph optimization together allow the robot to maintain an accurate trajectory and map over extended operation.

The overall estimation hierarchy can therefore be summarized as

sensor measurements

↓

relative motion and geometric constraints

↓

local odometry

↓

mapping

↓

loop closure and global optimization

↓

consistent trajectory and environment model.

1.7 Robot Manipulation

Robot manipulation concerns the deliberate physical interaction between a robot and objects in its environment. Unlike free-space motion, manipulation requires the robot to reason not only about where its end effector should move, but also about contact geometry, friction, force transmission, object motion, compliance, uncertainty, and task constraints.

A manipulation system therefore combines many of the ideas developed earlier in this chapter. Kinematics determines whether a grasp or contact configuration is reachable. Dynamics describes how forces applied through the robot affect the object. State estimation determines the pose and motion of the object. Motion planning generates collision-free approach trajectories. Force and impedance control regulate interaction after contact.

Manipulation is therefore not a single algorithm. It is a coupled problem involving geometry, mechanics, sensing, planning, and control.

A simple manipulation pipeline can be summarized as

perceive object and environment
↓
select grasp or contact
↓
plan feasible motion
↓
establish contact
↓
regulate motion and force
↓
verify task progress.

The difficulty increases substantially once contact occurs because the robot and environment become mechanically coupled.

1.7.1 Manipulation Kinematics

Manipulation kinematics extends ordinary robot kinematics to include objects and contacts.

Suppose the world frame is W, the end-effector frame is E, and an object frame is O. Their poses are related through rigid transformations such as T_{WE}, T_{WO}, and T_{EO}.

If the robot must place its end effector at a particular pose relative to the object, then the desired world-frame end-effector pose is

$$\boxed{T_{WE}^{d} = T_{WO} T_{OE}^{d}.}$$

Thus, an object-relative manipulation command can be transformed into an ordinary end-effector inverse-kinematics problem.

This representation is useful because many manipulation tasks are naturally object centered. A grasp might require the gripper to approach the object from a particular direction, a screwdriver must align with a screw axis, and a hand opening a door must maintain a relationship with the handle rather than with a fixed world coordinate.

A grasp configuration can therefore be described by the relative transform T_{EO} together with the robot configuration q.

Once contact becomes rigid, the object and end effector may be kinematically constrained. If the grasp transform remains constant,

$$T_{EO} = \text{constant},$$

then the object pose satisfies

$$T_{WO} = T_{WE} T_{EO}.$$

Motion of the end effector therefore directly determines object motion.

For velocities, if the grasp is rigid and there is no slip, the object and end-effector twists are related through the appropriate adjoint transformation. Conceptually, the rigid grasp imposes

$$\text{end-effector motion} \longleftrightarrow \text{object motion}.$$

This becomes especially important in dual-arm manipulation, where two end effectors simultaneously constrain the same object.

1.7.2 Pre-Grasp and Grasp Configurations

A manipulation trajectory often distinguishes between a *pre-grasp pose* and the final grasp pose.

The pre-grasp pose lies near the object but avoids contact. From there, the robot executes a constrained approach motion into the grasp.

This separation is useful because the motion-planning problem changes near contact. Large free-space motions primarily require collision avoidance, while final approach may require precise alignment with the object geometry.

A typical sequence is

collision-free free-space approach
$\downarrow$
pre-grasp pose
$\downarrow$
constrained approach
$\downarrow$
contact
$\downarrow$
gripper closure or force regulation.

A candidate grasp is feasible only if the corresponding end-effector pose lies inside the robot workspace and admits a valid inverse-kinematics solution that respects joint limits and collision constraints.

Thus, a geometrically good grasp on the object may still be unusable if the robot cannot reach it.

1.7.3 Grasping

A grasp is a set of contacts between the robot and an object that allows the robot to constrain, hold, or manipulate that object.

A grasp configuration may be represented as $G = (T_{EO}, q_{\text{gripper}})$, where T_{EO} specifies the gripper pose relative to the object and q_{gripper} describes the internal configuration of the fingers or gripper.

The quality of a grasp depends on several factors: contact locations, contact normals, friction, object geometry, object mass, actuator limits, collision clearance, expected disturbances, and the manipulation task.

A parallel-jaw gripper, for example, often seeks two opposing surface contacts. A multifingered hand may use several contacts distributed across the object. A suction gripper relies on a different contact model and requires a sufficiently suitable surface.

A grasp should not be evaluated solely by whether the fingers geometrically touch the object. The relevant question is whether the available contacts can generate the forces and moments necessary to resist disturbances and accomplish the desired task.

1.7.4 Contact Models

The mechanical behavior of a grasp depends on the assumed contact model.

A *frictionless point contact* can generate force only along the surface normal. If n_i is the normal at contact i, the contact force has the form $f_i = \lambda_i n_i$, with $\lambda_i \geq 0$.

A *point contact with friction* can additionally generate tangential forces subject to a friction constraint.

A *soft-finger contact* can transmit normal and tangential forces together with a limited moment about the contact normal, approximating the behavior of a deformable fingertip with finite contact area.

The selected contact model directly changes which object wrenches can be produced by the grasp.

1.7.5 Object Wrenches and the Grasp Matrix

A contact force contributes both a force and a moment to the object.

Suppose contact i applies force f_i at object-frame position r_i. The wrench about the object origin is

$$w_i = \begin{bmatrix} f_i \\ r_i \times f_i \end{bmatrix}.$$

For several contacts, stack the contact forces into a vector f. Their combined wrench on the object can then be written

$$\boxed{w = Gf,}$$

where G is the *grasp matrix*.

The grasp matrix plays a role analogous to the manipulator Jacobian transpose. It describes how individual contact forces combine to create a net force and moment on the object.

The manipulator Jacobian maps contact wrench to robot joint torque through $\tau = J^T F$, while the grasp matrix maps contact forces to object wrench through $w = Gf$.

Together, these relationships connect actuator forces to contact forces and contact forces to object motion.

1.7.6 Grasp Quality and Force Closure

A central question in grasp analysis is whether the contacts can resist external disturbances.

A grasp has *force closure* if the allowable contact forces can generate wrenches that oppose an arbitrary external wrench on the object, subject to the assumed friction and force constraints.

In simplified terms,

force closure $\quad \Rightarrow \quad$ arbitrary small object wrench can be resisted.

Force closure depends on both geometry and friction. Two frictionless point contacts, for example, cannot generally resist arbitrary three-dimensional object wrenches. Adding friction expands the set of achievable forces and can transform an otherwise unstable grasp into a force-closure grasp.

The complete set of wrenches that can be generated by the contacts is called the *grasp wrench space*.

If the origin lies strictly inside the set of achievable wrenches, then the grasp can generate restoring wrench in any direction around equilibrium, which is one way of expressing force closure.

1.7.7 Grasp Quality Metrics

Force closure is largely a binary property: a grasp either satisfies the condition or does not. In practice, several force-closure grasps may exist, so a continuous grasp-quality metric is useful.

One common idea is to ask how large a disturbance wrench the grasp can resist in its weakest direction. Geometrically, this corresponds to measuring the radius of the largest wrench-space ball centered at the origin that fits inside the achievable grasp-wrench set.

A larger radius indicates greater robustness to disturbances.

Other grasp metrics may penalize large required finger forces, poor contact geometry, proximity to friction limits, difficult robot postures, or sensitivity to object-pose uncertainty.

Thus, the "best" grasp is task dependent. A grasp suitable for simply lifting an object may not be suitable for applying a large torque, inserting the object into a fixture, or handing it to another person.

Manipulation planning therefore often evaluates a grasp using both *object-level stability* and *robot-level feasibility*.

1.7.8 Contact and Friction

Contact introduces forces that constrain relative motion between the robot and object.

A contact force can be decomposed into normal and tangential components as $f = f_n n + f_t$, where n is the contact normal.

For unilateral rigid contact, the normal force must satisfy

$$f_n \geq 0,$$

because ordinary contact can push two bodies apart but cannot pull them together.

Tangential force is limited by friction. Under the Coulomb friction model,

$$\boxed{\|f_t\| \leq \mu f_n,}$$

where μ is the coefficient of friction.

When the inequality is strict, the contact can remain static. When the tangential force reaches the boundary, slipping becomes possible.

The friction coefficient therefore determines how much tangential force can be transmitted for a given normal force.

This explains why increasing grip force may prevent an object from sliding. Larger normal force f_n increases the maximum admissible tangential force μf_n.

However, excessive gripping force may deform or damage an object, waste actuator capacity, or reduce manipulation dexterity. Force regulation therefore often seeks the minimum normal force that preserves sufficient friction margin.

1.7.9 Friction Cones

In three dimensions, the Coulomb condition $\|f_t\| \leq \mu f_n$ defines a cone of admissible contact-force directions known as the *friction cone*.

The cone is centered on the inward surface normal. Its angular width increases with the friction coefficient.

High friction produces a wide cone and permits a large range of tangential forces. Low friction produces a narrow cone, making slip more likely.

A contact force that lies inside the cone can be supported without sliding. A force outside the cone violates the static-friction model.

Manipulation planning therefore frequently imposes

$$f_i \in \mathcal{C}_i,$$

where $\mathcal{C}_i$ denotes the friction cone at contact i.

Exact circular cones are nonlinear constraints. Optimization-based controllers often approximate them using a polyhedral cone constructed from several linear facets. This converts the friction constraint into a set of linear inequalities and makes it easier to use inside quadratic programs or trajectory optimizers.

The accuracy of the approximation improves as more facets are added, at the cost of a larger optimization problem.

1.7.10 Slip and Contact Modes

Contact behavior is not limited to 'contact" or 'no contact." Several distinct modes can occur.

A contact may stick, slide, separate, or roll. Each mode imposes different velocity and force constraints.

For sticking contact, the relative tangential velocity is approximately zero and the force remains inside the friction cone.

For sliding contact, tangential velocity is nonzero and friction acts near the cone boundary opposite the direction of slip.

For separation, the normal contact force becomes zero and the bodies move apart.

Manipulation tasks such as pushing and in-hand manipulation often intentionally transition between these modes. Modeling those transitions makes contact-rich manipulation significantly more difficult than free-space motion.

1.7.11 Force and Torque Sensing

Physical interaction can be measured directly using force and torque sensors.

A six-axis force/torque sensor measures the wrench

$$w = \begin{bmatrix} F \\ \tau \end{bmatrix} \in \mathbb{R}^6.$$

Such sensors are frequently mounted at the robot wrist between the arm and end effector. They can measure contact forces generated during insertion, surface following, assembly, or human interaction.

Joint torque sensors provide another source of interaction information. If a sufficiently accurate robot dynamics model predicts the torque required for free motion, then external torque can be estimated as

$$\boxed{\tau_{\text{ext}} \approx \tau_{\text{measured}} - \tau_{\text{model}}.}$$

The corresponding Cartesian wrench is related by $\tau_{\text{ext}} = J^T F_{\text{ext}}$.

If J is appropriately conditioned, the external wrench can be estimated from the measured joint torque using a pseudoinverse relationship.

These approaches allow contact detection even without a dedicated wrist force sensor, although performance depends strongly on model accuracy, friction compensation, and actuator sensing quality.

1.7.12 Tactile Sensing

Force/torque sensing measures the resultant wrench transmitted through the robot structure, but it generally does not reveal the detailed pressure distribution across individual contacts.

Tactile sensors provide more local information, such as contact location, pressure distribution, normal force, shear force, or incipient slip.

For dexterous manipulation this information can be critical. Two grasps may produce the same net wrist force while having very different fingertip contact distributions.

Tactile feedback can therefore support tasks such as detecting slip, adjusting finger force, localizing contact, estimating object pose within the hand, and controlling rolling or sliding contact.

1.7.13 Compliant Manipulation

Manipulation often occurs under geometric uncertainty. A hole may be slightly displaced from its estimated position, a table surface may not be perfectly modeled, or an object may move slightly during grasping.

A perfectly rigid position controller reacts to this mismatch by generating potentially large contact forces.

Compliant manipulation instead allows some deviation in motion in response to environmental forces.

Impedance control provides one common framework. A desired Cartesian behavior may be written

$$M_d(\ddot{x} - \ddot{x}_d) + D_d(\dot{x} - \dot{x}_d) + K_d(x - x_d) = F_{\text{ext}}.$$

The robot then behaves approximately like a virtual mechanical system with inertia M_d, damping D_d, and stiffness K_d.

For many manipulation tasks, the important quantity is not whether the robot follows the nominal position exactly but whether the resulting contact behavior remains stable and useful.

Consider inserting a peg into a hole. A small lateral pose error can cause the peg to strike the rim. A rigid controller may increase the force while continuing to push in the incorrect direction. A compliant controller allows the peg to shift laterally under the resulting contact force and potentially self-align with the hole.

Compliance can therefore transform geometric uncertainty into controlled physical motion.

1.7.14 Passive and Active Compliance

Compliance can arise mechanically or through feedback control.

Passive compliance is created by physical elasticity in springs, flexures, soft fingertips, series elastic actuators, or deformable structures. Its behavior exists even if the controller fails.

Active compliance is generated by the controller, for example through impedance or admittance control.

Passive compliance provides naturally fast physical response but is limited by hardware design. Active compliance is programmable and can vary across tasks but depends on sensor quality, controller bandwidth, and stability.

Many robotic systems combine both approaches.

1.7.15 In-Hand Manipulation

A robot does not always need to release an object in order to change its pose. *In-hand manipulation* changes the object's configuration relative to the hand while maintaining one or more contacts.

Examples include rotating a tool within the fingers, repositioning a grasped object, rolling an object across fingertips, or shifting contact locations before performing another task.

In-hand manipulation is more complex than rigid grasping because the relative transform T_{EO} is no longer constant.

Instead, the system must reason about how contact forces and motions change the object pose inside the hand.

Several basic mechanisms are possible:

- rolling contact, in which the surfaces move without tangential slip,
- sliding contact, in which the object intentionally moves across a fingertip,
- finger gaiting, in which some fingers release and establish new contacts while others maintain the grasp,
- pivoting, in which the object rotates around one or more contacts.

Each mechanism creates different kinematic and friction constraints.

For example, controlled sliding requires the applied contact force to approach the friction limit in the intended direction while maintaining sufficient normal force to prevent loss of contact.

In-hand manipulation therefore requires coordinated control of both object motion and internal contact forces.

1.7.16 Internal Forces

When several contacts act on the same object, some combinations of contact forces may cancel at the object level.

If the contact-force vector is f and the grasp matrix is G, the object wrench is $w = Gf$.

Any contact-force vector f_{int} satisfying

$$\boxed{Gf_{\text{int}} = 0}$$

produces no net object wrench.

Such forces are called *internal forces*.

Internal forces do not accelerate the object but can change how strongly it is squeezed or how close individual contacts are to slipping.

This distinction is fundamental in multifinger and bimanual manipulation. A robot may independently control the object wrench and, within the remaining force freedom, regulate internal force to maintain grasp stability.

1.7.17 Bimanual Manipulation

Bimanual manipulation coordinates two manipulators acting on the same object or task.

If the left and right end effectors have Jacobians J_L and J_R, their velocities can be represented jointly as

$$\begin{bmatrix} v_L \\ v_R \end{bmatrix} = \begin{bmatrix} J_L & 0 \\ 0 & J_R \end{bmatrix} \begin{bmatrix} \dot{q}_L \\ \dot{q}_R \end{bmatrix}.$$

When both hands rigidly grasp the same object, their motions are no longer independent. Both must remain consistent with a single object motion.

If the grasp transforms T_{E_LO} and T_{E_RO} remain fixed, motion of either hand constrains the object pose and therefore constrains the other hand.

The controller must therefore distinguish two types of behavior: *object motion*, which changes the net wrench and motion of the object, and *internal motion or force*, which changes how the two arms load the object without necessarily changing its global pose.

This distinction is important when carrying a heavy box. The two hands should coordinate to generate the required net upward wrench while also maintaining sufficient inward force to prevent the object from slipping.

Too little internal force may lose the grasp; too much may damage the object or unnecessarily load the actuators.

1.7.18 Coordinated Object-Level Control

A useful approach is to formulate the task around the object rather than the individual robot hands.

Suppose the desired object pose is T_{WO}^{d}. The required left- and right-hand poses can be obtained from the fixed grasp transforms,

$$T_{WE_L}^{d} = T_{WO}^{d} T_{OE_L}, \qquad T_{WE_R}^{d} = T_{WO}^{d} T_{OE_R}.$$

This guarantees that the two desired hand motions remain geometrically consistent with the same rigid object.

At the force level, individual contact wrenches can be chosen so that their combined effect produces the desired object wrench while an additional internal-force component maintains grasp stability.

Bimanual manipulation is therefore naturally expressed in object coordinates rather than as two unrelated arm-control problems.

1.7.19 Mobile Manipulation

A mobile manipulator combines an arm with a movable base. Examples include an arm mounted on a wheeled platform, a humanoid robot, or a legged robot performing manipulation.

The configuration may be written as

$$q = \begin{bmatrix} q_b \\ q_a \end{bmatrix},$$

where q_b describes the mobile base and q_a describes the manipulator.

The end-effector pose depends on both,

$$T_{WE} = f(q_b, q_a).$$

This creates additional redundancy. A target that is difficult to reach with the arm alone may become easy after moving the base.

The key challenge is deciding how motion should be divided between base and arm.

For example, when reaching toward a distant shelf, the robot may move the base forward while keeping the arm near a high-manipulability posture. During precise contact, the base may remain mostly stationary while the arm performs the final motion.

A unified Jacobian can relate both base and arm velocities to end-effector velocity,

$$\xi_E = J_b \dot{q}_b + J_a \dot{q}_a.$$

This allows the complete system to be treated as one redundant kinematic chain.

1.7.20 Base Placement and Reachability

Mobile manipulation introduces a planning problem that fixed-base manipulation does not have: where should the base be placed before reaching?

A candidate object may be reachable from many base configurations, but these configurations can differ substantially in manipulability, collision clearance, visibility, and stability.

Good base placement may maximize metrics such as expected reachability or manipulability while maintaining sufficient clearance from obstacles.

This is particularly important in household manipulation. A robot approaching a cabinet must place its base close enough to reach the handle but far enough away to allow the door to open.

Thus, navigation and manipulation planning become coupled.

1.7.21 Whole-Body Mobile Manipulation

For highly redundant systems, the base and arm can move simultaneously.

Rather than executing

$$\text{navigate} \rightarrow \text{stop} \rightarrow \text{manipulate},$$

the robot may perform coordinated whole-body motion.

For example, a robot carrying a large object may continuously adjust both base and arms to avoid obstacles while maintaining the object's orientation.

This requires planning and control over the combined configuration space, often subject to constraints on base motion, arm motion, contact, balance, and collision.

Mobile manipulation therefore connects the manipulation problem directly to whole-body control and trajectory optimization.

1.7.22 Manipulation Under Uncertainty

Manipulation is particularly sensitive to perception error because small geometric errors can become large contact forces.

Suppose the estimated object pose is $\hat{T}_{WO}$, while the actual pose differs by a small transformation ΔT. A grasp or insertion trajectory planned from the estimate may therefore encounter the object earlier or at a different orientation than expected.

Successful manipulation systems compensate for such uncertainty through several mechanisms: repeated visual estimation, tactile sensing, compliant control, online replanning, force-based contact detection, and active exploratory motion.

This produces an important distinction between purely geometric planning and closed-loop manipulation.

A purely geometric system may execute

$$\text{estimate} \rightarrow \text{plan} \rightarrow \text{execute},$$

whereas a robust manipulation system repeatedly cycles through

estimate

↓

act

↓

sense consequences

↓

update estimate and plan.

Manipulation is therefore inherently feedback driven.

1.7.23 Manipulation as Integrated Perception, Planning, and Control

Complex manipulation requires perception, planning, and control to operate together.

Consider opening a door. The robot must estimate the door plane, handle position, handle orientation, and possibly the hinge axis. It must select a reachable grasp and generate a collision-free approach trajectory.

Once contact is established, the task changes. The gripper must maintain the grasp while the hand follows the constrained motion imposed by the door hinge. Contact force must remain sufficient to maintain the handle grasp but low enough to avoid excessive loading.

The complete process can be represented as

camera / depth / tactile measurements

↓

object and contact-state estimation

↓

grasp or interaction selection

↓

motion planning

↓

contact establishment

↓

force-aware feedback control

↓

updated perception and task state.

The same structure appears in insertion, grasping, tool use, pushing, folding, and bimanual tasks.

The relative importance of each component changes with the problem. Free-space pick-and-place may depend strongly on visual pose accuracy and grasp planning. Peg insertion depends heavily on contact sensing and compliance. In-hand manipulation requires detailed contact-state

reasoning. Mobile manipulation additionally requires coordination between navigation and arm reachability.

1.7.24 Manipulation of Articulated Objects

Many objects are not free rigid bodies but contain mechanical constraints. Doors rotate about hinges, drawers translate along rails, knobs rotate about fixed axes, and tools may be constrained by contact with a workpiece.

The robot should therefore avoid treating every manipulation problem as unconstrained Cartesian motion.

If an object's allowable configuration is parameterized by a variable s, its pose may be written $T_O(s)$. The end effector then needs to follow a corresponding constrained manifold rather than an arbitrary path through $SE(3)$.

This information can be known from a model or inferred through interaction.

For example, while opening an unknown door, the robot can observe that contact forces rise when commanded motion deviates from the hinge-compatible trajectory. Force and motion measurements can therefore help identify the object's kinematic constraint online.

1.7.25 Manipulation of Deformable Objects

Rigid-body manipulation assumes the object pose can be described by a transformation in $SE(3)$. This assumption fails for ropes, cloth, cables, food, soft packaging, and other deformable objects.

A deformable object's state may require a large set of points, mesh vertices, keypoints, or a lower-dimensional latent representation.

Manipulation then requires estimating how contact changes this distributed state.

A rope, for example, may contain the same endpoints at two moments while having very different intermediate configurations. A single rigid transform cannot represent this distinction.

Deformable manipulation therefore places greater emphasis on perception, learned dynamics, feedback, and replanning because accurate analytical models can be difficult to obtain.

1.7.26 Relationship Between Grasping and Manipulation

Grasping and manipulation should not be treated as identical problems.

Grasping establishes a useful set of contacts. Manipulation uses those contacts to change the state of the object or environment.

A grasp that is maximally stable may not be ideal for the intended manipulation task. A very rigid power grasp may prevent the object from rotating within the hand, while a less restrictive grasp may permit useful dexterity.

Task-aware manipulation therefore asks not merely

Can the robot hold the object?

but

Can the chosen contacts generate the motions and forces required by the task?

This distinction motivates task-oriented grasp planning, in which grasp quality is evaluated relative to anticipated object wrenches and future motion.

1.7.27 Summary

Robot manipulation combines kinematics, contact mechanics, sensing, planning, and feedback control to enable purposeful physical interaction with the environment.

Manipulation kinematics relates robot, end-effector, and object frames, allowing object-relative tasks to be converted into robot motion objectives. A rigid grasp constrains relative motion between the hand and object, while more dexterous manipulation may deliberately permit rolling or sliding contacts.

Grasp mechanics can be expressed through the grasp matrix,

$$w = Gf,$$

which maps individual contact forces into a net object wrench. A force-closure grasp can resist arbitrary disturbance wrenches within the limits of its contact and friction model.

Coulomb friction constrains tangential contact force according to

$$\|f_t\| \leq \mu f_n,$$

and the corresponding set of admissible forces forms a friction cone. These constraints determine whether contacts stick, slip, or lose contact.

Force/torque and tactile sensors provide information about physical interaction that cannot be obtained reliably from vision alone. Combined with impedance, admittance, and force control, these measurements enable compliant manipulation in the presence of geometric uncertainty.

In-hand manipulation changes the object pose relative to the hand through rolling, sliding, pivoting, or finger gaiting. Multifinger and bimanual systems additionally control internal forces satisfying $Gf_{\text{int}} = 0$, allowing grasp stability to be changed without altering the net object wrench.

Bimanual manipulation coordinates multiple end effectors through a shared object model, while mobile manipulation extends the configuration space to include both base and arm motion.

Ultimately, successful manipulation is a closed-loop process. Perception estimates the object and environment, planning selects feasible contacts and motions, control regulates motion and force, and new sensor measurements continuously update the robot's understanding of the interaction.

The complete manipulation loop can therefore be summarized as

perception

↓

grasp and contact reasoning

↓

motion planning

↓

physical interaction

↓

force / tactile / visual feedback

↓

state update and replanning.

Manipulation is therefore best understood not as motion toward an object, but as the controlled transfer of motion and force through contact.

Chapter 2

Intelligence in Robotics

This chapter develops the machine-learning and decision-making foundations used in modern intelligent robotic systems. It begins with core machine-learning concepts and Transformer architectures, then extends these ideas to multimodal and Vision–Language–Action models for embodied control. It covers generative action models based on diffusion and flow matching, imitation-learning methods such as behavior cloning and DAgger, and reinforcement-learning methods for sequential decision-making. The chapter also examines how robot-learning datasets are collected and constructed, including teleoperation, human video, simulation, replay buffers, and dataset balancing, before concluding with methods for evaluating learned policies under distribution shift, perturbations, and long-horizon deployment.

Together, these topics describe how modern robots learn representations, model possible behaviors, select actions, learn from demonstrations and interaction, and generalize beyond the conditions represented in their training data.

2.1 Core Machine Learning Concepts

Most modern machine-learning systems can be understood as parameterized functions whose parameters are adjusted so that the model performs well on observed data. Let

$$\hat{y} = f_\theta(x)$$

denote a model with parameters θ, input x, and prediction $\hat{y}$. Training consists of defining an objective that measures the quality of these predictions, computing how that objective changes with respect to the model parameters, and iteratively updating the parameters to reduce the objective.

At a high level, the training loop is

input data
↓
forward pass
↓
loss
↓
backpropagation
↓
parameter gradients
↓
optimizer update.

The concepts in this section describe the major mathematical and computational components of this process.

2.1.1 Loss and Objective Functions

A *loss function* measures how well a model prediction agrees with a target. For a single training example (x_i, y_i), the loss is written

$$\ell(f_\theta(x_i), y_i).$$

Training usually minimizes the average loss over a dataset of N examples,

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(f_\theta(x_i), y_i).$$

This quantity is often called the *empirical risk*. It approximates the expected performance of the model using the finite dataset available for training. Different tasks require different loss functions. For continuous regression, a common choice is mean-squared error,

$$\ell_{\text{MSE}} = \|\hat{y} - y\|_2^2.$$

For classification, cross-entropy is commonly used. If $p_\theta(y = k \mid x)$ is the predicted probability of the correct class k,

$$\ell_{\text{CE}} = -\log p_\theta(y = k \mid x).$$

Minimizing cross-entropy therefore encourages the model to assign high probability to the correct output. Other objectives encode different forms of supervision. Contrastive losses encourage related examples to occupy nearby regions of a representation space while separating unrelated examples. Ranking and margin-based losses, such as hinge loss, encourage correct outputs to score higher than incorrect alternatives.

The training objective may also contain additional terms,

$$J(\theta) = L_{\text{data}}(\theta) + \lambda R(\theta),$$

where $R(\theta)$ introduces a preference for particular classes of solutions and λ controls its strength.

In practice, objectives may contain several components,

$$J = \lambda_1 L_1 + \lambda_2 L_2 + \cdots,$$

which is common in robotics. A learned policy might simultaneously optimize action prediction, representation learning, auxiliary state prediction, and regularization terms.

2.1.2 Backpropagation and Automatic Differentiation

Once an objective has been defined, training requires its gradient with respect to the model parameters,

$$\nabla_\theta J.$$

Backpropagation computes these gradients efficiently by recursively applying the chain rule through the operations that produced the loss.

Consider

$$z = Wx, \qquad y = \sigma(z), \qquad L = \ell(y, t).$$

The dependence of the loss on the weight matrix W forms the computational chain

$$W \to z \to y \to L.$$

The gradient is propagated backward through this chain:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial W}.$$

More generally, if $y = f(x)$, then an upstream gradient $\partial L/\partial y$ is propagated through f according to

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}.$$

Deep neural networks are simply much larger compositions of such operations. Backpropagation avoids independently differentiating the complete network with respect to every parameter. Intermediate derivatives are reused while the graph is traversed backward, making gradient computation efficient even for models containing billions of parameters.

Automatic differentiation, commonly called *autograd*, is the software mechanism used by frameworks such as PyTorch to perform this calculation automatically. During the forward pass, the framework records the operations connecting tensors. The backward pass then applies derivative rules associated with those operations.

Neural networks normally use *reverse-mode* automatic differentiation. This is especially efficient when a function has many inputs but only one or a small number of outputs. Neural-network training has exactly this structure: millions or billions of parameters ultimately contribute to a scalar loss.

2.1.3 Optimization

Once the gradient has been computed, an optimizer determines how the parameters should change. The simplest method is gradient descent,

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t),$$

where η is the learning rate. The negative gradient points in the direction of steepest local decrease in the objective. The learning rate determines how far the optimizer moves in that direction. In effect, we nudge the parameters on each node of the network in the direction that would make the network get closer to predicting the output. However, we don't nudge too far and recompute the direction the weights need to be shifted on ever backpropagation pass.

For large datasets, computing the exact gradient over every training example before each update is expensive. Neural networks therefore normally use *mini-batch stochastic gradient descent*. If $\mathcal{B}$ is a batch containing B examples, the gradient is estimated using

$$\nabla_\theta J_{\mathcal{B}} = \frac{1}{B}\sum_{i \in \mathcal{B}} \nabla_\theta \ell_i.$$

The resulting mini-batch gradient is only an estimate of the gradient over the full dataset, but it is substantially cheaper to compute. Because different mini-batches provide different noisy estimates of the same underlying objective, repeated updates gradually move the parameters toward regions of lower loss.

A complete pass through the training dataset is called an *epoch*. If a dataset contains N examples and the batch size is B, then one epoch contains approximately N/B optimizer steps.

How these gradients are translated into parameter updates is itself a central optimization problem. If the learning rate is too large, the optimizer may overshoot low-loss regions or oscillate instead of converging. If it is too small, training may progress extremely slowly. The difficulty becomes even greater when the loss surface is poorly conditioned: the objective may be steep in some parameter directions and nearly flat in others, causing ordinary gradient descent to zig-zag through narrow valleys rather than move efficiently toward a minimum.

Much of modern optimization can therefore be understood as an attempt to improve how gradient information is converted into parameter motion. Methods such as momentum accumulate consistent gradient directions over time, while adaptive optimizers such as Adam adjust the effective step size separately for different parameters.

$$v_t = \beta v_{t-1} + \nabla_\theta J_t,$$

followed by

$$\theta_{t+1} = \theta_t - \eta v_t.$$

Momentum accumulates gradients that consistently point in the same direction while reducing the effect of rapidly oscillating components. Adam extends this idea by maintaining moving estimates of both the first and second moments of the gradient. Conceptually,

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t,$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2,$$

where $g_t = \nabla_\theta J_t$. After bias correction, the update scales each parameter approximately according to

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}.$$

Parameters whose gradients consistently have large magnitude therefore receive smaller normalized updates than parameters whose gradients remain small. Adam and AdamW are very common in Transformer training, while SGD with momentum remains important in many vision and classical deep-learning settings.

The learning rate itself is often changed during training. Common schedules include warmup, step decay, cosine decay, and exponential decay. The learning rate is one of the most important optimization hyperparameters: a value that is too large can make optimization unstable, while one that is too small can make training extremely slow or cause optimization to stall.

2.1.4 Activation Functions

A neural network would remain a linear function if every layer consisted only of matrix multiplication and addition. For example, two linear layers,

$$h = W_1 x + b_1, \qquad y = W_2 h + b_2,$$

can be combined into a single linear transformation,

$$y = W_2 W_1 x + W_2 b_1 + b_2.$$

Adding more linear layers would therefore not increase the class of functions the network can represent. *Activation functions* introduce nonlinearity between layers, allowing neural networks to approximate complex relationships.

A typical layer has the form

$$z = Wx + b, \qquad h = \phi(z),$$

where z is the *pre-activation*, h is the resulting activation, and ϕ is a nonlinear activation function.

Activation functions are also important during backpropagation because their derivatives determine how gradients propagate through the network. If an upstream gradient $\partial L/\partial h$ arrives at the activation, then

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial h} \odot \phi'(z),$$

where $\odot$ denotes element-wise multiplication. The derivative $\phi'(z)$ therefore directly scales the gradient passed to earlier layers.

For a sequence of many layers, these derivatives are repeatedly multiplied together. Activation functions whose derivatives are consistently very small can therefore contribute to *vanishing gradients*, while poorly controlled activations can contribute to unstable signal propagation.

Linear Activation

The simplest activation is the identity function,

$$\phi(x) = x, \qquad \phi'(x) = 1.$$

A linear activation does not introduce nonlinearity and therefore provides no additional expressive power when placed between linear layers. It is nevertheless useful at the output of regression models when the prediction should be allowed to take arbitrary real values.

Sigmoid

The sigmoid function maps a real number into the interval $(0, 1)$:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Its derivative is

$$\sigma'(x) = \sigma(x)\big(1 - \sigma(x)\big).$$

Sigmoid is useful when an output should represent a probability, particularly for binary classification. However, when $|x|$ becomes large, the sigmoid saturates near 0 or 1, causing

$$\sigma'(x) \approx 0.$$

Gradients passing through many saturated sigmoid units can therefore become extremely small. For this reason, sigmoid is used less frequently as the hidden-layer activation in modern deep networks.

Hyperbolic Tangent

The hyperbolic tangent maps inputs into $(-1, 1)$:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}.$$

Its derivative is

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x).$$

Unlike sigmoid, $\tanh(x)$ is centered around zero, which can improve optimization in some settings. It still saturates for large positive or negative inputs, however, and its derivative approaches zero in those regions.

Both sigmoid and tanh therefore illustrate an important principle:

$$\text{saturated activation} \quad \Rightarrow \quad \text{small derivative} \quad \Rightarrow \quad \text{weak gradient propagation.}$$

ReLU

The *Rectified Linear Unit* (ReLU) is defined as

$$\mathrm{ReLU}(x) = \max(0, x).$$

Its derivative is

$$\mathrm{ReLU}'(x) = \begin{cases} 1, & x > 0, \\ 0, & x < 0. \end{cases}$$

The function is not differentiable exactly at $x = 0$, but implementations simply choose a suitable subgradient there, typically 0.

ReLU became widely used because its derivative is 1 throughout the positive region. Gradients can therefore propagate without continuously shrinking as they do through saturated sigmoid or tanh units.

The negative region creates a different problem. If a unit consistently receives negative pre-activations, its output is always zero and its gradient is also zero. Such a unit may stop learning, a phenomenon sometimes called a *dying ReLU*.

Leaky ReLU

Leaky ReLU modifies the negative side of ReLU by retaining a small nonzero slope:

$$\mathrm{LeakyReLU}(x) = \begin{cases} x, & x \geq 0, \\ \alpha x, & x < 0, \end{cases}$$

where α is a small positive constant.

Its derivative is

$$\mathrm{LeakyReLU}'(x) = \begin{cases} 1, & x > 0, \\ \alpha, & x < 0. \end{cases}$$

Because the derivative remains nonzero for negative inputs, parameters associated with negative activations can continue to receive gradient updates.

GELU

Many Transformer architectures use the *Gaussian Error Linear Unit* (GELU). Conceptually,

$$\mathrm{GELU}(x) = x\,\Phi(x),$$

where $\Phi(x)$ is the cumulative distribution function of a standard Gaussian.

A commonly used approximation is

$$\mathrm{GELU}(x) \approx \frac{x}{2}\left[1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right].$$

Unlike ReLU, which sharply sets all negative inputs to zero, GELU smoothly suppresses negative values while preserving large positive values. This smoothness makes GELU attractive in deep Transformer-style networks.

SiLU and Swish

Another common smooth activation is the *Sigmoid Linear Unit* (SiLU), also known as Swish:

$$\mathrm{SiLU}(x) = x\sigma(x).$$

Its derivative is

$$\mathrm{SiLU}'(x) = \sigma(x) + x\sigma(x)\big(1 - \sigma(x)\big).$$

Like GELU, SiLU provides a smooth transition between suppressed negative inputs and approximately linear behavior for positive inputs. It is widely used in modern vision models, language models, and other deep architectures.

Softmax

Softmax differs from the preceding activations because it operates on an entire vector rather than independently on each element. Given logits $z \in \mathbb{R}^K$,

$$p_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}.$$

The resulting values satisfy

$$0 < p_i < 1, \qquad \sum_i p_i = 1,$$

so they can be interpreted as a categorical probability distribution.

Unlike element-wise activations, changing one logit affects every softmax output. Its derivatives therefore form a Jacobian,

$$\frac{\partial p_i}{\partial z_j} = p_i(\delta_{ij} - p_j),$$

where δ_{ij} is the Kronecker delta.

Softmax is commonly used at classification outputs and inside attention mechanisms, rather than as the main hidden-layer activation.

Why Activation Choice Affects Training

During backpropagation, gradients repeatedly pass through expressions of the form

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial h} \odot \phi'(z).$$

Across many layers, the gradient contains products of weight matrices and activation derivatives. Schematically,

$$\frac{\partial L}{\partial h_k} \sim \frac{\partial L}{\partial h_L} \prod_{j=k+1}^{L} W_j^T \phi'(z_j).$$

If the magnitudes of these terms are repeatedly much smaller than one, the gradient can vanish. If they are repeatedly too large, the gradient can explode.

Activation functions, parameter initialization, normalization, and residual connections are therefore closely related. Each influences how signals and gradients propagate through deep networks. The important point is not that one activation is universally best. Activation functions determine both the nonlinear behavior the network can represent and the derivatives through which learning signals propagate. Their choice is therefore part of both model design and optimization.

2.1.5 Initialization and Signal Propagation

Optimization begins from an initial parameter configuration. Initialization matters because poorly scaled parameters can cause activations or gradients to grow or shrink rapidly as signals propagate through a deep network. Consider a linear layer

$$y = Wx.$$

If the components of x and the entries of W are approximately independent and zero mean, then the variance of the output depends on both the input variance and the number of terms being summed. Ideally, the network should maintain approximately comparable signal magnitude across layers,

$$\mathrm{Var}(y) \approx \mathrm{Var}(x).$$

If the variance grows repeatedly through the network, activations and gradients may explode. If it repeatedly shrinks, they may vanish. Initialization schemes therefore choose the scale of W based on the number of incoming or outgoing connections.

For a layer with n_{in} inputs and n_{out} outputs, Xavier or Glorot initialization uses a variance approximately proportional to

$$\mathrm{Var}(W) \sim \frac{2}{n_{\text{in}} + n_{\text{out}}}.$$

This is well suited to activations that are approximately symmetric around zero. For ReLU-like networks, He or Kaiming initialization instead uses

$$\mathrm{Var}(W) \sim \frac{2}{n_{\text{in}}},$$

accounting for the fact that ReLU suppresses approximately half of its inputs.

Initialization does not need to place the model near the final solution. Its primary purpose is to begin optimization in a regime where information and gradients can propagate effectively.

2.1.6 Normalization

Normalization modifies the scale and distribution of intermediate representations to improve optimization and numerical stability.

A generic normalized quantity takes the form

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}},$$

where μ and σ^2 are a mean and variance calculated over a specified collection of dimensions.

The distinction between normalization methods lies largely in *which elements are used to compute these statistics.*

For Layer Normalization, let

$$x = [x_1, \ldots, x_d] \in \mathbb{R}^d$$

be the representation of one token or feature vector. The mean and variance are computed across its feature dimensions,

$$\mu = \frac{1}{d}\sum_{j=1}^{d} x_j, \qquad \sigma^2 = \frac{1}{d}\sum_{j=1}^{d}(x_j - \mu)^2.$$

The normalized feature is

$$\hat{x}_j = \frac{x_j - \mu}{\sqrt{\sigma^2 + \epsilon}},$$

after which learned scale and shift parameters are applied,

$$y_j = \gamma_j \hat{x}_j + \beta_j.$$

The learned parameters allow the network to recover or modify the normalized scale when useful.

Batch Normalization instead computes statistics across examples in a minibatch, typically separately for each feature or channel. Its behavior therefore depends on batch statistics during training and on running statistics during inference.

LayerNorm does not depend on the other examples in the minibatch, making it particularly suitable for sequence models and Transformers, where sequence lengths and batching patterns may vary.

Normalization should not be interpreted simply as keeping every activation close to zero. Its broader role is to improve the conditioning of optimization and stabilize the scale of representations as they pass through many layers.

2.1.7 Regularization and Generalization

Minimizing training loss is not the ultimate objective of machine learning. The goal is to perform well on new data that were not used to fit the model.

A sufficiently expressive network can sometimes achieve extremely small training error while learning patterns that do not generalize. This behavior is known as *overfitting*.

Regularization describes methods that bias training toward solutions expected to generalize better.

One approach is to modify the objective directly. For L_2 regularization,

$$J(\theta) = L_{\text{data}}(\theta) + \lambda \|\theta\|_2^2.$$

The gradient of the regularization term discourages parameters from becoming unnecessarily large.

Weight decay produces a related effect by directly shrinking parameters during optimization. For ordinary gradient descent, weight decay and L_2 regularization can be equivalent under appropriate definitions, but this equivalence does not generally hold for adaptive optimizers such as Adam. AdamW therefore implements weight decay separately from the gradient-based Adam update.

Other regularization mechanisms operate differently.

Dropout randomly suppresses activations during training, discouraging the network from relying too strongly on particular pathways.

Data augmentation modifies training examples while preserving their semantic labels, exposing the network to a wider range of inputs.

Early stopping terminates training when validation performance stops improving, preventing continued optimization of the training set from degrading generalization.

Regularization therefore includes any mechanism that changes the optimization process or effective training distribution in a way intended to improve performance outside the training examples.

This motivates the usual division of available data into training, validation, and test sets. The training set determines the parameters. The validation set is used to select hyperparameters and training decisions. The test set is reserved for the final estimate of generalization performance.

2.1.8 Embeddings and Learned Representations

Modern neural networks rarely operate directly on symbolic objects such as words, object identities, task labels, or discrete actions. Instead, these quantities are mapped into continuous vector spaces.

An *embedding* is a mapping

$$e : \mathcal{X} \to \mathbb{R}^d.$$

For example,

$$\text{“cat”} \to e_{\text{cat}} \in \mathbb{R}^d.$$

The individual coordinates of an embedding usually do not have a simple human interpretation. What matters is the geometry of the representation space.

Objects that are useful in similar contexts may develop nearby or otherwise systematically related representations. One common measure of similarity is cosine similarity,

$$\text{sim}(a, b) = \frac{a^T b}{\|a\|_2 \|b\|_2}.$$

Cosine similarity measures the angle between two vectors while ignoring their overall magnitude.

For a finite vocabulary containing V discrete elements, embeddings are commonly stored in a learned matrix

$$E \in \mathbb{R}^{V \times d}.$$

If an input has discrete identifier i, its representation is simply the corresponding row,

$$e_i = E_i.$$

The embedding matrix is trained jointly with the rest of the network through backpropagation.

Embeddings are not limited to language. In robotics, continuous representations may encode image patches, objects, robot states, actions, task identities, temporal positions, or entire observations. Multimodal learning attempts to construct representation spaces in which information from several modalities can interact.

For example,

$$\begin{matrix}\text{language} \\ \text{images} \\ \text{proprioception} \\ \text{actions}\end{matrix} \quad \longrightarrow \quad \text{continuous representations} \quad \longrightarrow \quad \text{shared neural computation.}$$

This idea becomes central in Transformer and Vision–Language–Action architectures, where heterogeneous information must ultimately be represented as vectors that can be processed by a common model.

2.1.9 Putting the Training Process Together

The concepts introduced above form a single computational loop. Given a minibatch of training examples, the model first performs a forward pass to produce predictions. The loss function compares those predictions with the training targets. Backpropagation and automatic differentiation compute the gradients of the objective with respect to every trainable parameter. The optimizer then uses those gradients to update the parameters.

Initialization determines the regime in which this process begins. Normalization helps keep intermediate representations well conditioned as they move through deep networks. Regularization and data design influence whether the solution generalizes beyond the examples used for optimization. Embeddings provide the continuous representations on which the neural network ultimately operates.

The complete training process can be summarized as a sequence of six steps:

$$\boxed{(x, y) \sim \mathcal{D}}$$

Sample an input x and target y from the training dataset

$$\Downarrow$$

$$\boxed{\hat{y} = f_\theta(x)}$$

Forward pass: use the current model parameters θ to produce a prediction

$$\Downarrow$$

$$\boxed{J(\hat{y}, y)}$$

Evaluate how well the prediction $\hat{y}$ agrees with the desired target y

$$\Downarrow$$

$$\boxed{\nabla_\theta J}$$

Backpropagate through the model to determine how each parameter influences the loss

$$\Downarrow$$

$$\boxed{\theta \leftarrow \text{Optimizer}\,(\theta, \nabla_\theta J)}$$

Adjust the parameters in a direction expected to reduce the objective

$$\Downarrow$$

$$\boxed{\text{repeat}}$$

Draw another minibatch and repeat the process over many optimization steps

The first step, $(x, y) \sim \mathcal{D}$, represents sampling training data. Here x is the input presented to the model and y is the corresponding target. In practice, training usually operates on a minibatch of many such examples rather than on a single pair. The forward pass,

$$\hat{y} = f_\theta(x),$$

evaluates the model using its current parameters θ. The result $\hat{y}$ is the model's prediction. At this stage no parameters are changed; the network is simply evaluated from input to output. The objective,

$$J(\hat{y}, y),$$

compares the prediction with the desired target. For regression this might be a squared error, while classification might use cross-entropy. The objective converts the quality of the model's

prediction into a scalar quantity that can be optimized. Backpropagation then computes

$$\nabla_\theta J,$$

the gradient of the objective with respect to the model parameters. Each component of this gradient answers a local question:

$$\frac{\partial J}{\partial \theta_i} = \text{how sensitive the loss is to parameter } \theta_i.$$

A positive derivative indicates that increasing the parameter locally increases the objective, while a negative derivative indicates that increasing it locally decreases the objective. The gradient therefore describes the local slope of the objective throughout parameter space.

The optimizer uses this information to update the parameters. For ordinary gradient descent,

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t),$$

where η controls the size of the update. The parameters are moved opposite the gradient because the gradient points toward the steepest local increase in the objective. The updated parameters are then used for another forward pass on another minibatch, until sufficient performance of the model is achieved.

2.2 Transformer Fundamentals

Transformers are neural-network architectures for processing sequences of representations. Their defining mechanism is *attention*, which allows each element of a sequence to directly retrieve information from other elements rather than relying on information to propagate sequentially through a recurrent hidden state.

At a high level, a Transformer-based sequence model follows the pipeline

raw input
↓
tokens
↓
continuous embeddings
↓
positional information
↓
Transformer blocks
↓
contextual representations
↓
output prediction.

The central computation inside each Transformer block is scaled dot-product attention,

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V,$$

where Q, K, and V are the query, key, and value matrices. Much of Transformer design can be understood in terms of how these representations are constructed, how attention is constrained, and how the resulting information is processed efficiently.

2.2.1 Tokenization and Input Representations

A Transformer does not operate directly on raw symbolic inputs such as text. The input must first be converted into a sequence of vector representations.

For a language model, raw text is divided into discrete tokens. A sentence such as

Transformers are powerful.

might be decomposed conceptually into subword units such as

$$[\text{“Transform”}, \text{“ers”}, \text{“ are”}, \text{“ powerful”}, \text{“.”}].$$

Each token is assigned an integer identifier,

$$[t_1, t_2, \ldots, t_n] \to [i_1, i_2, \ldots, i_n].$$

The model contains a learned embedding matrix

$$E \in \mathbb{R}^{|\mathcal{V}| \times d_{\text{model}}},$$

where $|\mathcal{V}|$ is the vocabulary size and d_{model} is the hidden representation dimension. If token i has identifier i, its embedding is obtained from the corresponding row of E,

$$e_i = E_i.$$

The sequence of token embeddings can be stacked into

$$X \in \mathbb{R}^{n \times d_{\text{model}}},$$

where n is the sequence length.

The same idea extends beyond language. Image patches, robot states, actions, or other modalities can also be projected into vectors of dimension d_{model} and processed by the same sequence architecture.

2.2.2 Positional Information

Attention by itself does not encode sequence order. Without an additional positional representation, sequences containing the same elements in different orders would be difficult to distinguish.

For example,

dog bites man

and

man bites dog

contain the same tokens but have very different meanings.

Transformers therefore incorporate positional information into their representations.

The original Transformer used deterministic sinusoidal positional encodings. For sequence position pos and feature index i,

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right),$$

$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right).$$

If e_i is the token embedding and p_i its positional representation, the input to the Transformer becomes

$$x_i = e_i + p_i.$$

Many modern Transformers instead use *Rotary Positional Embeddings* (RoPE). Rather than adding position directly to the token embedding, RoPE applies position-dependent rotations to the query and key vectors:

$$q_i' = R(i)q_i, \qquad k_j' = R(j)k_j.$$

The resulting attention score depends on

$$(q_i')^T k_j',$$

whose structure naturally incorporates relative positional relationships such as $i - j$.

2.2.3 The Transformer Block

A complete Transformer block contains more than attention. A common modern design combines normalization, attention, residual connections, and a position-wise feed-forward network.

In a pre-normalization architecture, the attention portion can be written approximately as

$$X' = X + \text{Attention}\left(\text{Norm}(X)\right),$$

followed by

$$X'' = X' + \text{MLP}\left(\text{Norm}(X')\right).$$

The residual connections allow information to pass directly through the network while also providing short paths for gradient propagation.

The two major computations inside the block play different roles:

$$\text{attention} \quad \rightarrow \quad \text{information exchange between tokens,}$$

while

$$\text{MLP} \quad \rightarrow \quad \text{nonlinear processing within each token.}$$

A basic feed-forward network has the form

$$\text{MLP}(x) = W_2\phi(W_1x + b_1) + b_2,$$

where ϕ is a nonlinear activation such as GELU. Many modern architectures use gated feed-forward networks such as SwiGLU instead.

The attention and feed-forward operations are repeated across many Transformer blocks, allowing progressively richer contextual representations to emerge.

2.2.4 Queries, Keys, and Values

Attention begins by projecting the input representation into three learned spaces. Given

$$X \in \mathbb{R}^{n\times d_{\text{model}}},$$

the model learns projection matrices

$$W_Q \in \mathbb{R}^{d_{\text{model}}\times d_k}, \qquad W_K \in \mathbb{R}^{d_{\text{model}}\times d_k}, \qquad W_V \in \mathbb{R}^{d_{\text{model}}\times d_v},$$

and constructs

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V.$$

Thus,

$$Q, K \in \mathbb{R}^{n\times d_k}, \qquad V \in \mathbb{R}^{n\times d_v}.$$

For token i, the query vector q_i describes the information that the token attempts to retrieve. For token j, the key k_j describes how that token can be matched against a query, while the value v_j contains the information that can be contributed if the match is strong.

A useful interpretation is therefore

$$\begin{aligned} q_i &: \text{What information am I looking for?} \\ k_j &: \text{How relevant am I to that request?} \\ v_j &: \text{What information should I contribute?} \end{aligned}$$

The compatibility between query i and key j is measured by their dot product,

$$q_i^T k_j.$$

After scaling,

$$s_{ij} = \frac{q_i^T k_j}{\sqrt{d_k}},$$

and the scores for query i are normalized with softmax,

$$a_{ij} = \frac{\exp(s_{ij})}{\sum_{m=1}^{n} \exp(s_{im})}.$$

The output corresponding to query i is then

$$y_i = \sum_{j=1}^{n} a_{ij} v_j.$$

Attention can therefore be interpreted as a content-dependent weighted combination of value vectors.

The complete computation can be summarized as

$QK^T \rightarrow$ compatibility scores $\rightarrow$ softmax $\rightarrow$ attention weights $\rightarrow AV \rightarrow$ aggregated information.

2.2.5 Scaled Dot-Product Attention

The scaling factor $1/\sqrt{d_k}$ is important for numerical stability.

The unscaled dot product is

$$q^T k = \sum_{r=1}^{d_k} q_r k_r.$$

If the components of q and k are approximately independent, zero-mean random variables with comparable variance, then

$$\mathrm{Var}(q^T k) \propto d_k.$$

As the dimension d_k grows, the magnitude of the logits can therefore increase. Large logits cause softmax to become highly concentrated, producing probabilities close to zero or one and reducing useful gradient magnitude.

Dividing by $\sqrt{d_k}$,

$$s_{ij} = \frac{q_i^T k_j}{\sqrt{d_k}},$$

approximately compensates for this growth and keeps the attention logits in a more numerically useful range.

The dimensional structure is also important. For one attention head,

$$Q : (n \times d_k), \qquad K^T : (d_k \times n),$$

so

$$QK^T \in \mathbb{R}^{n \times n}.$$

The resulting matrix contains one pairwise compatibility score for every query–key pair. After softmax,

$$A \in \mathbb{R}^{n \times n},$$

and multiplying by

$$V \in \mathbb{R}^{n \times d_v}$$

produces

$$AV \in \mathbb{R}^{n \times d_v}.$$

Thus QK^T determines *where information should be retrieved from*, while AV performs the corresponding information aggregation.

2.2.6 Self-Attention

In *self-attention*, queries, keys, and values are all derived from the same sequence:

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V.$$

The attention matrix is

$$A = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right),$$

and the resulting contextualized representation is

$$Y = AV.$$

Each element A_{ij} specifies how strongly token i, acting as a query, attends to token j, acting as a key.

For example, consider

> The robot grabbed the cup because it was nearby.

When constructing the representation associated with 'it," an attention head may assign substantial weight to 'cup." The value representation associated with 'cup" can then influence the representation of 'it."

Self-attention therefore allows every output token to contain information aggregated from other locations in the sequence.

2.2.7 Multi-Head Attention

A Transformer generally performs several attention operations in parallel rather than relying on a single query–key–value projection.

For attention head h,

$$Q_h = XW_Q^{(h)}, \qquad K_h = XW_K^{(h)}, \qquad V_h = XW_V^{(h)},$$

and

$$Z_h = \text{Attention}(Q_h, K_h, V_h).$$

The outputs are concatenated,

$$Z = \text{Concat}(Z_1, \ldots, Z_H),$$

and passed through an output projection,

$$Y = ZW_O.$$

Thus,

$$\text{MultiHead}(X) = \text{Concat}(Z_1, \ldots, Z_H)W_O.$$

If the model dimension is divided evenly across H heads,

$$d_h = \frac{d_{\text{model}}}{H},$$

so that

$$Hd_h = d_{\text{model}}.$$

The multiple heads therefore do not usually multiply the overall hidden dimension by H. Instead, the representation is partitioned across several learned attention spaces and then recombined.

Different heads may learn different interaction patterns, although individual heads should not necessarily be assumed to have simple human-interpretable roles.

2.2.8 Causal Attention

Self-attention by itself allows every token to attend to every other token. Autoregressive sequence generation requires an additional constraint: when predicting the next token, the model must not access future tokens.

An autoregressive model factorizes a sequence probability as

$$P(t_1, \ldots, t_n) = \prod_{i=1}^{n} P(t_i \mid t_1, \ldots, t_{i-1}).$$

This constraint is implemented using a causal mask,

$$M_{ij} = \begin{cases} 0, & j \le i, \\ -\infty, & j > i. \end{cases}$$

The attention matrix becomes

$$A = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right).$$

For four tokens, the allowed attention structure is

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}.$$

The first token can attend only to itself, while later tokens can attend to all preceding positions.

Causal attention should therefore be understood as

$$\boxed{\text{causal self-attention} = \text{self-attention} + \text{future-token masking.}}$$

Self-attention itself does not inherently require causality.

2.2.9 Cross-Attention

Cross-attention allows one representation to retrieve information from another.

Let X_q denote the query-side representation and X_c a context representation. Queries are constructed from the first source,

$$Q = X_q W_Q,$$

while keys and values come from the context,

$$K = X_c W_K, \qquad V = X_c W_V.$$

The attention calculation remains

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V.$$

The difference from self-attention is therefore entirely in the source of the representations:

$$\text{self-attention:} \qquad Q, K, V \leftarrow X,$$

whereas

$$\text{cross-attention:} \qquad Q \leftarrow X_q, \qquad K, V \leftarrow X_c.$$

Cross-attention is especially useful in multimodal systems. Suppose a robot receives the language instruction

> Pick up the red cup.

Language features may produce queries while image features provide keys and values. A language representation associated with "red cup" can then assign large attention weights to visual regions whose representations match the instruction.

Cross-attention therefore provides a general mechanism for information exchange between modalities.

2.2.10 Encoder and Decoder Patterns

Transformer architectures differ in how attention is constrained and how information flows through the model.

An *encoder* typically uses bidirectional self-attention, allowing every token to attend to every other token in the input sequence. This is useful when the goal is to construct a contextual representation of an entire observed input.

A *decoder* used for autoregressive generation applies causal self-attention so that each position can access only information available earlier in the sequence.

An encoder–decoder architecture combines the two ideas: an encoder first constructs representations of an input sequence, and a decoder generates an output while using cross-attention to retrieve information from the encoder.

Conceptually,

$$\begin{aligned} \text{encoder} &: \text{bidirectional self-attention}, \\ \text{decoder} &: \text{causal self-attention}, \\ \text{encoder–decoder} &: \text{self-attention + cross-attention}. \end{aligned}$$

Modern multimodal architectures may combine these patterns in many different ways, but the underlying attention mechanisms remain the same.

2.2.11 From Input Tokens to Next-Token Prediction

A decoder-only Transformer performs autoregressive next-token prediction.

Consider a sequence

$$t_1, t_2, \ldots, t_n.$$

Tokenization converts these symbols into discrete identifiers, which are mapped to embeddings,

$$t_i \to e_i \in \mathbb{R}^{d_{\text{model}}}.$$

Positional information is incorporated to form the initial sequence representation X. Each Transformer block repeatedly contextualizes this representation through attention and nonlinear feed-forward processing.

At the final layer, the hidden representation associated with the prediction position is projected into vocabulary logits,

$$z = W_{\text{vocab}} h + b,$$

where

$$z \in \mathbb{R}^{|\mathcal{V}|}.$$

Softmax converts the logits into a probability distribution,

$$P(t_{n+1} = j) = \frac{e^{z_j}}{\sum_{k=1}^{|\mathcal{V}|} e^{z_k}}.$$

A decoding rule then chooses or samples the next token.

Once t_{n+1} has been generated, it becomes part of the context,

$$[t_1, \ldots, t_n] \to [t_1, \ldots, t_n, t_{n+1}],$$

and the process repeats.

The full inference loop is therefore

$$\boxed{\text{context} \to \text{Transformer} \to \text{logits} \to \text{softmax} \to \text{next token} \to \text{updated context.}}$$

2.2.12 Computational Complexity of Attention

Full self-attention compares every query with every key. Since

$$QK^T \in \mathbb{R}^{n \times n},$$

the attention matrix contains n^2 pairwise scores.

Its storage therefore scales as

$$O(n^2),$$

while the principal attention matrix multiplications scale approximately as

$$O(n^2 d),$$

where d denotes the relevant representation dimension.

If sequence length doubles from n to $2n$, the number of pairwise attention scores grows from

$$n^2$$

to

$$4n^2.$$

This quadratic dependence is one of the major computational challenges associated with long-context Transformers.

2.2.13 Autoregressive Inference: Prefill, Decode, and KV Caching

Autoregressive inference is commonly divided into two phases: *prefill* and *decode*.

During prefill, the complete initial prompt

$$[t_1, t_2, \ldots, t_T]$$

is processed together. The Transformer computes the internal representations of all prompt tokens and constructs the key and value tensors needed at each attention layer.

Because many prompt tokens can be processed simultaneously, prefill makes effective use of large parallel matrix operations.

After prefill, the model enters the decode phase. New tokens are generated sequentially,

$$t_{T+1}, \quad t_{T+2}, \quad t_{T+3}, \quad \dots$$

Causal masking ensures that the representation of an earlier token does not depend on tokens generated later. Consequently, the key and value representations previously computed for those earlier tokens do not change and can be reused.

This motivates *KV caching*. Suppose the model has already processed t tokens. At a particular layer, their stored keys and values are

$$K_{\text{cache}} = [K_1, K_2, \dots, K_t],$$

$$V_{\text{cache}} = [V_1, V_2, \dots, V_t].$$

For the new token $t + 1$, only

$$q_{t+1}, \qquad k_{t+1}, \qquad v_{t+1}$$

must be computed. The new key and value are appended to the cache,

$$K_{\text{cache}} \leftarrow [K_{\text{cache}}, k_{t+1}],$$

$$V_{\text{cache}} \leftarrow [V_{\text{cache}}, v_{t+1}],$$

and the new query attends to the accumulated keys and values.

KV caching therefore exchanges additional memory usage for substantially less repeated computation.

A rough scaling law for the number of cached scalar values is

$$O(2BLTH_{kv}d_h),$$

where B is batch size, L is the number of Transformer layers, T is cached sequence length, H_{kv} is the number of key/value heads, and d_h is head dimension. The factor of 2 accounts for storing both keys and values.

The actual memory requirement also depends on numerical precision.

2.2.14 Multi-Head, Multi-Query, and Grouped-Query Attention

Conventional multi-head attention generally assigns separate query, key, and value projections to every attention head. If there are H query heads, there are therefore typically H key and value heads as well.

During autoregressive inference, only keys and values must be retained in the KV cache. Cache memory can therefore be reduced by sharing key/value representations across query heads.

In *Multi-Query Attention* (MQA), all query heads share one set of keys and values,

$$H_{kv} = 1.$$

This can reduce KV-cache memory substantially.

In *Grouped-Query Attention* (GQA), groups of query heads share key/value heads,

$$1 < H_{kv} < H.$$

GQA therefore provides a compromise between ordinary multi-head attention and MQA. Since KV-cache memory is approximately proportional to H_{kv}, reducing the number of key/value heads can substantially reduce inference-memory requirements.

2.2.15 Summary

A Transformer converts discrete or continuous inputs into vector representations, incorporates positional information, and repeatedly applies attention and nonlinear feed-forward transformations.

The central attention operation is

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V.$$

The projections

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V$$

determine what information is requested, how candidate information is matched to that request, and what information is ultimately aggregated.

Self-attention exchanges information within a sequence. Multi-head attention performs this operation in several learned representation spaces. Causal masking restricts information flow for autoregressive prediction, while cross-attention allows information to move between different sequences or modalities.

A Transformer block combines this attention operation with normalization, residual connections, and token-wise nonlinear processing. Repeated blocks produce increasingly contextual representations that can ultimately be mapped to predictions.

During autoregressive inference, the initial context is processed during prefill, after which tokens are generated sequentially during decode. KV caching avoids recomputing key and value representations for previous tokens, while MQA and GQA reduce the memory required to store them.

The complete flow can therefore be summarized as

input

↓

tokenization / embedding

↓

positional representation

↓

attention + MLP blocks

↓

contextual representation

↓

output distribution

↓

prediction.

These mechanisms form the foundation of modern language models and also extend naturally to multimodal systems in which text, images, robot state, and actions are represented within a common Transformer-based architecture.

2.3 Vision-Language-Action Models and Embodied AI

A **Vision-Language-Action (VLA)** model extends a vision-language model from perception and reasoning into control:

VLM: perception + language

VLA: perception + language + action

Instead of only describing the scene, a VLA predicts robot actions:

$$(o_t, \ell) \to \pi_\theta \to a_t,$$

where

- o_t is the current observation,
- ℓ is the language instruction,
- a_t is the robot action,
- π_θ is the learned policy.

A robot observation may contain multiple modalities:

$$o_t = (I_t, D_t, q_t, \dot{q}_t, f_t),$$

where I_t is RGB imagery, D_t is depth, q_t contains joint positions, $\dot{q}_t$ contains joint velocities, and f_t contains force/torque information.

2.3.1 VLA Architecture

A typical VLA architecture separately encodes different modalities:

$$I_t \rightarrow \text{Vision Encoder} \rightarrow z_V,$$

$$\ell \rightarrow \text{Language Encoder} \rightarrow z_L,$$

$$s_t \rightarrow \text{State Encoder} \rightarrow z_S.$$

These representations are fused by a multimodal model:

$$[z_V, z_L, z_S] \rightarrow \text{Multimodal Transformer} \rightarrow \text{Action Head}.$$

Thus,

$$a_t = f_\theta(z_V, z_L, z_S).$$

A useful mental model is

$$\boxed{\text{Observation} \rightarrow \text{Tokens} \rightarrow \text{Multimodal Transformer} \rightarrow \text{Action}}$$

2.3.2 Action Representations

There is no universal robot action representation. The action depends on the control interface.

Joint Position

$$a_t = [q_1^*, q_2^*, \ldots, q_n^*].$$

Joint Velocity

$$a_t = [\dot{q}_1, \ldots, \dot{q}_n].$$

Joint Torque

$$a_t = [\tau_1, \ldots, \tau_n].$$

End-Effector Delta

$$a_t = [\Delta x, \Delta y, \Delta z, \Delta r_x, \Delta r_y, \Delta r_z, g],$$

where g represents a gripper command.

This can be written compactly as

$$a_t = [\Delta p_t, \Delta R_t, g_t].$$

Absolute Cartesian Pose

$$a_t = [x, y, z, q_w, q_x, q_y, q_z, g].$$

End-effector deltas are often useful because they provide a relatively robot-independent action representation.

2.3.3 Observation Tokenization

Transformers operate on sequences of vectors, so robot observations must first be converted into tokens.

For vision,

$$I_t \to \text{image patches} \to x_1, \ldots, x_N.$$

Robot state can be projected using an MLP:

$$s_t = [q_t, \dot{q}_t, g_t],$$

$$x_s = W_s s_t + b_s.$$

Language is tokenized in the usual way.

The resulting multimodal sequence might be

$$X = [x_1^{\text{text}}, \ldots, x_M^{\text{text}}, x_1^{\text{vision}}, \ldots, x_N^{\text{vision}}, x_s].$$

A Transformer then models interactions across modalities:

$$H = \text{Transformer}(X).$$

2.3.4 Temporal Context

A single observation is often insufficient for robotics.

For example, one image may not reveal whether an object is moving left or right.

Therefore, policies may use observation histories:

$$o_{t-k:t} = [o_{t-k}, \ldots, o_t].$$

Each observation can be encoded:

$$x_t = f(o_t),$$

and the temporal sequence becomes

$$[x_{t-k}, \ldots, x_t].$$

This gives the model information about motion and temporal evolution.

2.3.5 Action Tokenization

One approach is to represent continuous actions as discrete tokens.

Suppose

$$\Delta x \in [-0.05, 0.05] \text{ m}.$$

The interval can be discretized into B bins:

$$[-0.05, 0.05] \to \{0, 1, \ldots, B-1\}.$$

Action prediction can then be formulated similarly to next-token prediction:

$$P(a_t \mid o_{\leq t}, \ell).$$

This allows standard autoregressive Transformer machinery and cross-entropy training:

$$\mathcal{L} = -\log P(a_t^{\text{true}}).$$

The conceptual sequence becomes

$$\boxed{\text{Text Tokens} + \text{Vision Tokens} + \text{Action Tokens}}$$

processed by a common sequence model.

2.3.6 Discretization Error

Action tokenization introduces finite resolution.

For

$$a \in [a_{\min}, a_{\max}]$$

with B bins, the approximate action resolution is

$$\Delta a \approx \frac{a_{\max} - a_{\min}}{B}.$$

Therefore, discrete action prediction trades continuous precision for compatibility with token-based sequence modeling.

2.3.7 Continuous Action Prediction

Instead of discretizing actions, the network can directly predict real-valued controls:

$$a_t \in \mathbb{R}^{d_a}.$$

A simple action head is

$$\hat{a}_t = W h_t + b.$$

Training can use regression:

$$\mathcal{L} = \|a_t - \hat{a}_t\|_2^2.$$

Huber loss or other regression losses may also be used.

The major advantage is that there is no quantization error.

2.3.8 Multimodal Action Distributions

Simple deterministic regression can fail when several actions are valid.

Suppose two possible actions are

$$a_1 = [-1, 0], \qquad a_2 = [1, 0].$$

Minimizing mean-squared error may produce their average:

$$\hat{a} = [0, 0],$$

even though this may be a poor action.

The problem is that

$$p(a \mid o)$$

can be multimodal.

Instead of predicting only

$$\hat{a} = f(o),$$

we can model a distribution:

$$p(a \mid o),$$

and sample

$$a \sim p(a \mid o).$$

For example,

$$p(a \mid o) = 0.5p_{\text{left}} + 0.5p_{\text{right}}.$$

This motivates distributional action models such as mixture models, diffusion policies, flow matching, and autoregressive action prediction.

2.3.9 Action Chunking

A single-step policy predicts

$$a_t,$$

then later

$$a_{t+1},$$

and so forth.

Action chunking instead predicts multiple future actions simultaneously:

$$\boxed{A_t = [a_t, a_{t+1}, \ldots, a_{t+H-1}]}$$

where H is the chunk horizon.

If the action dimension is d_a,

$$A_t \in \mathbb{R}^{H \times d_a}.$$

For example,

$$H = 16, \qquad d_a = 7$$

requires predicting

$$16 \times 7 = 112$$

continuous values.

Action chunking provides three important benefits:

- improved temporal coherence,
- reduced model inference frequency,
- representation of short-horizon trajectory intent.

2.3.10 Receding-Horizon Execution

Predicting an entire chunk does not require executing the whole chunk open-loop.

A useful strategy is:

1. predict H future actions,
2. execute only the first few,
3. observe the environment again,
4. predict a new chunk.

Thus,

Predict Long	Execute Short

This combines trajectory coherence with closed-loop feedback.

2.3.11 Temporal Ensembling

When overlapping action chunks are predicted, several predictions may exist for the same future timestep.

For example,

$$A_t = [a_t, a_{t+1}, a_{t+2}, \ldots]$$

and

$$A_{t+1} = [a'_{t+1}, a'_{t+2}, \ldots].$$

Multiple predictions can be combined:

$$\hat{a}_{t+1} = \sum_i w_i a^{(i)}_{t+1}.$$

This can smooth the executed control trajectory.

2.3.12 Hierarchical Policies

Complex tasks naturally operate at multiple levels of abstraction.

For an instruction such as

> Make me coffee.

a high-level policy may produce subgoals such as

1. find mug,
2. grasp mug,
3. move mug to machine,
4. operate machine,
5. retrieve mug.

The high-level policy can be represented as

$$g_t = \pi_H(o_t, \ell),$$

where g_t is a goal or subtask.

A low-level policy then produces motor actions:

$$a_t = \pi_L(o_t, g_t).$$

Therefore,

$$\boxed{\pi_H : \text{observation} \to \text{subgoal}}$$

and

$$\boxed{\pi_L : \text{observation} + \text{subgoal} \to \text{motor action}}$$

The high-level output might be a language instruction, skill ID, waypoint, target pose, or object-centric goal.

2.3.13 Why Hierarchy Matters

Robotic reasoning and control operate at very different timescales.

Semantic planning may operate near

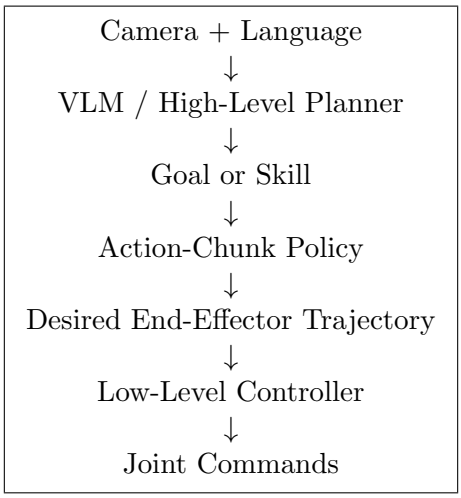
1 Hz,

while learned manipulation policies may operate at roughly

$$10\text{–}50 \text{ Hz},$$

and low-level motor controllers may run at hundreds or thousands of Hertz.

A conceptual robot stack is therefore



This avoids running expensive semantic reasoning at servo-control frequencies.

2.3.14 Action Chunking vs. Hierarchy

These concepts solve different problems.

$$\boxed{\text{Action chunking} = \text{temporal grouping of low-level actions}}$$

while

$$\boxed{\text{Hierarchy} = \text{abstraction across levels of control}}$$

They can be combined:

$$\text{High-Level Policy} \rightarrow g_t \rightarrow \text{Low-Level Policy} \rightarrow [a_t, \ldots, a_{t+H-1}].$$

2.3.15 Behavior Cloning

Robot demonstrations can be represented as trajectories:

$$\tau = (o_1, a_1, o_2, a_2, \ldots, o_T, a_T).$$

With language instruction ℓ, a dataset is

$$\mathcal{D} = \{(\ell_i, \tau_i)\}_{i=1}^{N}.$$

For discrete actions, behavior cloning can minimize

$$\mathcal{L} = -\sum_t \log \pi_\theta(a_t \mid o_{\le t}, \ell).$$

For continuous regression,

$$\mathcal{L} = \sum_t \|a_t - \hat{a}_t\|_2^2.$$

2.3.16 Covariate Shift

A major problem with behavior cloning is that training observations come from the expert distribution:

$$s \sim p_{\text{expert}}(s).$$

During deployment, however, a small policy error can move the robot into a new state s' that was not represented in the demonstrations.

The policy is then operating off-distribution, which can cause another error, followed by another.

This produces compounding error:

$$\boxed{\text{Small Error} \rightarrow \text{New State} \rightarrow \text{Distribution Shift} \rightarrow \text{Larger Error}}$$

This phenomenon is commonly called **covariate shift** in imitation learning.

Strategies mentioned for addressing it include closed-loop replanning, DAgger, reinforcement-learning fine-tuning, and data augmentation.

2.3.17 Discrete vs. Continuous Actions

Discrete / Tokenized Actions Advantages:

- naturally integrate with autoregressive Transformers,
- use cross-entropy objectives,
- can represent multimodality through token probabilities.

Disadvantages:

- introduce quantization error,
- may require large action vocabularies,
- may require sequential decoding.

Continuous Actions Advantages:

- natural representation for robot control,
- no quantization error,
- entire action vectors can be predicted directly.

Disadvantages:

- deterministic regression can average multiple valid action modes.

Neither representation is universally superior.

2.3.18 Interview Summary

VLA: A policy model that conditions on visual observations, language, and robot state to predict actions.

Observation Tokenization: Converts images, state, proprioception, and other observations into vector representations suitable for a sequence model.

Action Tokenization: Discretizes continuous robot actions so control prediction can be treated similarly to next-token prediction.

Continuous Actions: Predicts real-valued robot commands directly rather than discretizing them.

Action Chunking: Predicts several future actions in a single model inference.

Receding-Horizon Control: Predicts a long action chunk but executes only part of it before observing and replanning.

Temporal Ensembling: Combines overlapping predictions for the same future action to produce smoother control.

Hierarchical Policy: Separates slow high-level task reasoning from fast low-level motor execution.

Behavior Cloning: Learns a policy by matching demonstrated expert actions.

Covariate Shift: Deployment errors push the robot into states not represented in expert demonstrations, causing errors to compound.

2.3.19 Key Mental Model

The complete embodied-AI pipeline can be summarized as

Observation → Tokens → Multimodal Transformer → Intent → Action Chunk → Low-Level Controller

The central design questions are

How should observations be represented?
How should actions be represented?
Discrete or continuous actions?
Single-step or chunked prediction?
Flat or hierarchical policy?
How frequently should the system replan?

For interview preparation, the most important relationships to remember are

$$\boxed{(o_t, \ell) \to \pi_\theta \to a_t}$$

$$\boxed{A_t = [a_t, \ldots, a_{t+H-1}]}$$

and

$$\boxed{g_t = \pi_H(o_t, \ell), \qquad a_t = \pi_L(o_t, g_t).}$$

2.4 Diffusion Models, Flow Matching, and Generative Robot Policies

Many robot-learning problems are naturally *multimodal*. Given the same observation and task, there may be several distinct action sequences that would all lead to success. A robot navigating around an obstacle, for example, may pass on either the left or the right. Similarly, a manipulator may approach an object from several valid directions.

This creates a fundamental difficulty for deterministic regression.

Suppose two valid action trajectories are

$$A_{\text{left}}$$

and

$$A_{\text{right}}.$$

A regression model trained with mean-squared error tends to predict the conditional mean,

$$\hat{A} \approx \frac{A_{\text{left}} + A_{\text{right}}}{2}.$$

However, the average of two valid trajectories is not necessarily itself valid. If the two trajectories correspond to passing an obstacle on opposite sides, their average may pass directly through the obstacle.

This motivates a different modeling objective. Rather than predicting a single action trajectory, the model should represent a conditional distribution

$$p(A|c),$$

where A denotes an action trajectory and c denotes the conditioning information available to the policy, such as images, language instructions, proprioceptive state, or task goals.

Two important families of generative models for this purpose are *diffusion models* and *flow-matching models*. Both transform a simple random distribution into a complex data distribution, but they formulate the transformation differently.

Diffusion models learn how to reverse a gradual corruption process, while flow-matching models learn a continuous velocity field that transports probability mass from a simple base distribution toward the data distribution.

2.4.1 Generative Modeling of Robot Actions

For robot control, the object being generated is often not a single action but an *action chunk*

$$A = [a_t, a_{t+1}, \ldots, a_{t+H-1}],$$

where H is the prediction horizon.

If the condition is

$$c = (I_t, q_t, \ell),$$

then the policy may model

$$p(A|I_t, q_t, \ell),$$

where I_t is the current visual observation, q_t is the robot state, and ℓ is a language instruction.

This formulation allows the same observation to produce several possible trajectories, each corresponding to a valid mode of the action distribution.

Sampling a trajectory then becomes part of policy inference.

2.4.2 Diffusion Models

A diffusion model defines two complementary stochastic processes.

The *forward diffusion process* gradually corrupts a data sample by adding Gaussian noise.

The *reverse diffusion process* attempts to invert this corruption and transform noise back into a sample from the data distribution.

For robot policies, the clean sample may be an action trajectory

$$A_0 = [a_t, a_{t+1}, \ldots, a_{t+H-1}].$$

The goal is to learn a model capable of generating samples approximately distributed according to

$$p(A_0|c).$$

The important idea is that training is performed on partially corrupted versions of real trajectories, while generation begins with noise.

2.4.3 The Forward Diffusion Process

Let

$$x_0 \sim p_{\text{data}}$$

denote a clean data sample.

A standard Denoising Diffusion Probabilistic Model (DDPM) defines a Markovian forward process

$$q(x_t|x_{t-1}) = \mathcal{N}\left(\sqrt{1-\beta_t}\, x_{t-1}, \beta_t I\right),$$

where β_t is a small positive noise parameter.

At every diffusion step, the sample is slightly attenuated and additional Gaussian noise is introduced.

It is convenient to define

$$\alpha_t = 1 - \beta_t$$

and the cumulative product

$$\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s.$$

A particularly important result is that x_t can be sampled directly from x_0 without explicitly simulating all preceding diffusion steps:

$$\boxed{x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1-\bar{\alpha}_t}\, \epsilon}$$

where

$$\epsilon \sim \mathcal{N}(0, I).$$

This identity is central to practical diffusion training.

At small t, the coefficient

$$\sqrt{\bar{\alpha}_t}$$

is relatively large, so the sample still resembles the original data.

As t increases, the signal component decreases while the noise component increases.

At the final diffusion step,

$$x_T \approx \mathcal{N}(0, I).$$

Thus, the forward process gradually converts structured data into approximately Gaussian noise:

$$x_0 \to x_1 \to \cdots \to x_T.$$

The forward process is normally fixed mathematically rather than learned.

2.4.4 The Reverse Diffusion Process

Generation proceeds in the opposite direction.

One begins with

$$x_T \sim \mathcal{N}(0, I)$$

and attempts to construct

$$x_{T-1}, x_{T-2}, \ldots, x_0.$$

The ideal reverse conditional distribution

$$q(x_{t-1}|x_t)$$

depends on the unknown data distribution and cannot generally be evaluated directly.

A parameterized neural network is therefore used to approximate the reverse process:

$$p_\theta(x_{t-1}|x_t).$$

A common parameterization is

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}\left(\mu_\theta(x_t, t), \Sigma_t\right).$$

The network receives both the noisy sample x_t and the diffusion timestep t, because the correct denoising behavior depends strongly on how noisy the sample currently is.

Conceptually, the network must answer:

> Given this corrupted sample and its noise level, how should it be transformed toward a cleaner sample?

2.4.5 Noise Prediction

A particularly common parameterization does not predict the clean sample or the reverse mean directly. Instead, it predicts the Gaussian noise that was used to construct x_t.

From the forward relation,

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon,$$

a neural network predicts

$$\hat{\epsilon} = \epsilon_\theta(x_t, t).$$

The canonical training objective is

$$\mathcal{L}_{\text{diff}} = \mathbb{E}_{x_0,t,\epsilon}\left[\|\epsilon - \epsilon_\theta(x_t, t)\|_2^2\right].$$

Training therefore consists of repeatedly performing the following procedure:

1. sample a clean data point x_0,
2. sample a diffusion timestep t,
3. sample Gaussian noise ϵ,
4. construct x_t,
5. ask the model to predict ϵ,
6. minimize the noise-prediction error.

One appealing aspect of this formulation is that the objective is simply a regression loss even though the resulting model represents a complicated probability distribution.

2.4.6 Why Predicting Noise Is Sufficient

The connection between noise prediction and denoising follows directly from the forward equation.

Starting from

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon,$$

we can rearrange to obtain

$$x_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\epsilon}{\sqrt{\bar{\alpha}_t}}.$$

Therefore, if the model accurately estimates ϵ, it can also infer an estimate of the clean sample x_0.

The denoising network can consequently be parameterized in several equivalent or closely related ways.

It may predict the added noise,

$$\epsilon_\theta(x_t, t) \approx \epsilon,$$

the clean sample,

$$x_\theta(x_t, t) \approx x_0,$$

or a derived quantity such as a velocity parameterization,

$$v = \sqrt{\bar{\alpha}_t}\epsilon - \sqrt{1-\bar{\alpha}_t}x_0.$$

The phrase *diffusion objective* therefore does not uniquely imply noise prediction. Noise prediction is simply one particularly common formulation.

2.4.7 Conditional Diffusion

For robot learning, unconditional generation is rarely sufficient.

The model should generate actions appropriate for the current scene and task. Instead of learning

$$p(A),$$

we therefore want

$$p(A|c).$$

The conditioning variable may contain

$$c = (I_t, q_t, \ell),$$

where I_t represents visual input, q_t represents proprioceptive state, and ℓ represents a language instruction.

The denoiser becomes

$$\epsilon_\theta(x_t, t, c).$$

For an action chunk

$$A_0 = [a_t, \ldots, a_{t+H-1}],$$

training constructs

$$A_\tau = \sqrt{\bar{\alpha}_\tau} A_0 + \sqrt{1 - \bar{\alpha}_\tau}\epsilon,$$

and predicts

$$\hat{\epsilon} = \epsilon_\theta(A_\tau, \tau, c).$$

The training loss remains

$$\mathcal{L} = \|\epsilon - \hat{\epsilon}\|_2^2.$$

During inference, the model begins with

$$A_T \sim \mathcal{N}(0, I)$$

and repeatedly denoises while conditioning on c.

The final output is a trajectory consistent with the current observation and task.

This is the basic idea behind a diffusion policy.

2.4.8 Why Diffusion Is Attractive for Robot Policies

Diffusion is particularly useful when the conditional action distribution is multimodal.

Suppose the robot may validly execute either

$$A_{\text{left}}$$

or

$$A_{\text{right}}.$$

A deterministic regressor may predict an undesirable average between the two.

A diffusion model instead represents

$$p(A|c)$$

and can sample

$$A \sim p(A|c).$$

Different random initial conditions can therefore lead to different valid modes.

One sample might produce the left-side strategy, while another might produce the right-side strategy.

The model does not need to collapse both strategies into a single conditional mean.

2.4.9 How Conditioning Enters the Network

There are several ways for external context to influence a generative model.

The simplest is concatenation:

$$[x_t; c].$$

The network receives the noisy sample and conditioning variables as a joint input.

Another approach is feature-wise modulation. The condition generates scale and shift parameters,

$$h' = \gamma(c)h + \beta(c).$$

This allows the condition to modulate intermediate network features.

For Transformer-based models, cross-attention is especially natural.

Action representations may produce queries,

$$Q = X_{\text{action}} W_Q,$$

while visual or language representations produce keys and values:

$$K = X_{\text{obs}} W_K, \qquad V = X_{\text{obs}} W_V.$$

Cross-attention then computes

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V.$$

The generated trajectory can therefore selectively attend to the portions of the scene or instruction that are relevant to each action.

2.4.10 DDPM Sampling

The DDPM formulation uses a stochastic reverse process.

A conceptual reverse step is

$$x_{t-1} = \mu_\theta(x_t, t) + \sigma_t z,$$

where

$$z \sim \mathcal{N}(0, I).$$

Thus, generation proceeds through a sequence of stochastic transitions:

$$x_T \to x_{T-1} \to \cdots \to x_0.$$

Traditional DDPM formulations often use a large number of denoising steps.

Every step requires a neural-network evaluation, so generation can be computationally expensive.

This cost is especially important in robotics, where action-generation latency may directly limit control frequency.

2.4.11 DDIM Sampling

The Denoising Diffusion Implicit Model (DDIM) provides a different sampling formulation that can reuse the same underlying denoising network.

DDIM allows the reverse process to become deterministic by setting the stochastic component to zero:

$$\sigma_t = 0.$$

A reverse update then has the form

$$x_{t-1} = f\left(x_t, \epsilon_\theta(x_t, t)\right).$$

For a fixed initial noise sample and timestep schedule, the resulting trajectory can therefore be deterministic.

DDIM also makes it possible to use a reduced sequence of denoising steps.

The key distinction is not that DDPM and DDIM require completely different learned networks. Rather, DDIM primarily changes the sampling trajectory through the diffusion process.

Thus, at a high level,

DDPM

is associated with stochastic reverse sampling, while

DDIM

allows deterministic or less stochastic sampling and often supports fewer network evaluations.

2.4.12 Diffusion Transformers

The diffusion formulation specifies a generative objective and sampling process, but it does not dictate the architecture of the denoising network.

Historically, U-Net-style architectures have been common diffusion backbones.

A *Diffusion Transformer* (DiT) instead uses a Transformer as the network responsible for predicting the denoising quantity.

Conceptually,

$$x_t \to \text{tokenization} \to \text{Transformer} \to \hat{\epsilon}.$$

For robot action generation, suppose

$$A_\tau \in \mathbb{R}^{H \times d_a}.$$

Each action timestep can be embedded as a token:

$$a_i \to e_i.$$

The Transformer can then process the sequence of action tokens along with additional information such as

$$\text{generative timestep},$$

$$\text{trajectory position},$$

$$\text{visual context},$$

$$\text{robot state},$$

and

$$\text{language instructions}.$$

If E denotes the action-token sequence, a simplified model may be written as

$$H = \text{Transformer}(E, c, \tau),$$

followed by

$$\hat{\epsilon} = WH.$$

The important distinction is that DiT is an *architecture*, not a training objective.

A Transformer can be used inside a diffusion model, a flow-matching model, or another generative framework.

2.4.13 Why Transformers Are Natural for Action Trajectories

Robot action chunks are inherently sequential:

$$a_t, a_{t+1}, \ldots, a_{t+H-1}.$$

Actions at different positions in the sequence are correlated.

For example, an early reaching motion affects the appropriate orientation and gripper behavior later in the trajectory.

Self-attention gives the model a direct mechanism for representing such dependencies.

Cross-attention also provides a convenient way to condition action generation on visual or language tokens.

For this reason, Transformer-based generative models fit naturally with trajectory prediction.

2.4.14 Flow Matching

Flow matching provides another approach to generative modeling.

Rather than defining a stochastic noising process and learning how to reverse it, flow matching directly learns a *velocity field* that transports samples from a simple base distribution to the data distribution.

Let

$$x_0 \sim p_0$$

denote a sample from a simple distribution, typically Gaussian noise, and let

$$x_1 \sim p_{\text{data}}$$

denote a sample from the data distribution.

We define a continuous path

$$x_\tau, \qquad \tau \in [0, 1],$$

that connects the two.

The simplest example is linear interpolation:

$$x_\tau = (1 - \tau)x_0 + \tau x_1.$$

At

$$\tau = 0,$$

we have

$$x_\tau = x_0,$$

while at

$$\tau = 1,$$

we have

$$x_\tau = x_1.$$

2.4.15 The Velocity Field

Differentiating the linear path gives

$$\frac{dx_\tau}{d\tau} = x_1 - x_0.$$

The desired velocity is therefore

$$u_\tau = x_1 - x_0$$

for this simple interpolation.

A neural network

$$v_\theta(x_\tau, \tau)$$

is trained to predict the target velocity.

A simple flow-matching objective is

$$\mathcal{L}_{\mathrm{FM}} = \mathbb{E}\left[\|v_\theta(x_\tau, \tau) - u_\tau\|_2^2\right].$$

The model is therefore not directly predicting noise or a clean sample. It predicts how the current sample should move through the generative state space.

2.4.16 Generation with Flow Matching

Once the velocity field has been learned, generation begins with

$$x(0) \sim p_0.$$

The generated sample evolves according to the ordinary differential equation

$$\frac{dx}{d\tau} = v_\theta(x, \tau).$$

Integrating from

$$\tau = 0$$

to

$$\tau = 1$$

transports a noise sample toward the learned data distribution.

The model can be interpreted as answering the following question:

> Given the current sample and generative time, in what direction and at what speed should the sample move?

Generation therefore corresponds to following a learned vector field.

2.4.17 Conditional Flow Matching for Robot Control

As with diffusion, robot policies require conditional generation.

Let

$$A = [a_t, \ldots, a_{t+H-1}]$$

denote an action chunk, and let

$$c = (I_t, q_t, \ell)$$

denote the current observation and task condition.

Generation begins with a random action trajectory

$$A(0) \sim \mathcal{N}(0, I).$$

The model defines a conditional velocity field

$$v_\theta(A, \tau, c).$$

Inference then solves

$$\frac{dA}{d\tau} = v_\theta(A, \tau, c).$$

At

$$\tau = 1,$$

the trajectory

$$A(1)$$

should correspond to a plausible action sequence for the given condition.

Different initial noise samples can produce different trajectories, allowing the model to represent multimodal behavior.

2.4.18 Diffusion and Flow Matching

Diffusion and flow matching are conceptually different but mathematically related.

Diffusion is often introduced through a stochastic differential equation,

$$dx = f(x, t)\, dt + g(t)\, dW_t,$$

where dW_t represents Brownian noise.

The forward process gradually destroys structure, and the learned reverse dynamics reconstruct the data distribution.

Flow matching instead directly learns an ODE,

$$\frac{dx}{dt} = v_\theta(x, t).$$

At a conceptual level,

diffusion

asks how to remove corruption, whereas

flow matching

asks how samples should move continuously through probability space.

The distinction should not be overstated. Diffusion models can be associated with probability-flow ODEs that reproduce the same marginal distributions as the corresponding stochastic process.

A more precise description is therefore:

Diffusion typically begins from a stochastic corruption process and learns reverse dynamics, whereas flow matching directly trains a vector field that transports probability mass between distributions.

2.4.19 Generation Cost and Control Latency

Generative policies are computationally more expensive than direct regression because they typically require several neural-network evaluations for every generated action sequence.

Suppose a robot replans at

$$20 \text{ Hz}.$$

The available time for generating the next action chunk is then roughly

$$50 \text{ ms}$$

before accounting for perception, communication, and low-level control overhead.

If trajectory generation requires hundreds of network evaluations, real-time deployment may be impractical.

Sampling efficiency is therefore not merely a computational concern in robotics. It directly affects achievable control frequency.

This is one reason reduced-step diffusion samplers and flow-based methods are attractive for embodied systems.

2.4.20 Generative Time and Physical Time

Robot generative policies involve two distinct time coordinates that should not be confused.

Physical control time is indexed by t. An action trajectory is

$$A = [a_t, a_{t+1}, \ldots, a_{t+H-1}].$$

These indices correspond to actions that will be executed at successive robot-control timesteps.

The generative process introduces a second time variable, which is useful to denote by τ.

For example,

$$A_{\tau=0}$$

may correspond to a random trajectory, while

$$A_{\tau=1}$$

corresponds to the generated clean action trajectory.

Thus,

$$t = \text{physical robot time},$$

while

$$\tau = \text{generative time}.$$

A diffusion or flow model therefore transforms an entire trajectory through generative time while the positions within that trajectory refer to physical time.

This distinction is especially important when reading robot diffusion-policy and flow-matching equations.

2.4.21 Direct Regression and Generative Policies

It is useful to compare generative action models with direct regression.

A deterministic regressor computes

$$\hat{A} = f_\theta(c).$$

Generation requires only one forward pass, making this approach computationally attractive.

However, if the conditional distribution

$$p(A|c)$$

has several modes, deterministic regression tends to produce a central tendency such as the mean.

Diffusion and flow-matching models instead introduce a random variable at inference time.

For diffusion,

$$A_T \sim \mathcal{N}(0, I),$$

and repeated denoising transforms this noise into a trajectory.

For flow matching,

$$A(0) \sim \mathcal{N}(0, I),$$

and ODE integration transports the random trajectory toward the data distribution.

Different random initializations may therefore produce different valid behaviors.

2.4.22 A Robot Manipulation Example

Suppose a robot receives the condition

$$c = (I_t, q_t, \text{“pick up the cup”})$$

and must generate a 32-step action trajectory with seven action dimensions per timestep:

$$A \in \mathbb{R}^{32\times 7}.$$

The seven action components might represent six-dimensional end-effector motion together with a gripper command.

In a diffusion policy, inference begins with

$$A_T \sim \mathcal{N}(0, I).$$

At every generative step, the model evaluates

$$\hat{\epsilon} = \epsilon_\theta(A_\tau, \tau, c)$$

and updates the current trajectory toward a less noisy version.

After repeated updates,

$$A_T \to \cdots \to A_0,$$

where A_0 is a coherent action sequence.

In a flow-matching policy, inference instead begins with

$$A(0) \sim \mathcal{N}(0, I)$$

and integrates

$$\frac{dA}{d\tau} = v_\theta(A, \tau, c)$$

until

$$\tau = 1.$$

The result

$$A(1)$$

is the generated trajectory.

Both models can therefore map the same observation into different plausible action sequences depending on the random initial condition.

2.4.23 Choosing a Generative Formulation

The different approaches can be understood in terms of what the network predicts and how generation proceeds.

A direct regression model predicts the action trajectory itself:

$$\hat{A} = f_\theta(c).$$

A DDPM-style diffusion model usually predicts a denoising quantity such as

$$\epsilon, \qquad x_0,$$

or a related parameterization and generates samples through iterative denoising.

A DDIM sampler uses the same general denoiser but follows a different reverse trajectory that may be deterministic and use fewer steps.

A flow-matching model predicts

$$v_\theta(x, \tau),$$

a time-dependent velocity field, and generates samples by integrating an ODE.

A DiT describes the architecture used to implement the neural network rather than the generative objective itself.

It is therefore possible to combine different design choices. For example,

$$\text{Transformer architecture} + \text{diffusion objective}$$

and

$$\text{Transformer architecture} + \text{flow-matching objective}$$

are both valid model designs.

2.4.24 The Broader Role of Generative Policies

Generative robot policies are useful because robot behavior is rarely described by a unique correct action.

A task may permit several grasp poses, several collision-free trajectories, or several different high-level strategies.

A useful policy should therefore often represent

$$p(A|c)$$

rather than only its conditional mean.

This perspective unifies several modern action-generation methods.

Diffusion models construct the action distribution through iterative denoising.

Flow-matching models construct it through continuous probability transport.

Autoregressive models factor the action distribution into a sequence of conditional predictions.

Mixture models explicitly represent several modes.

Although these approaches differ mathematically, they address the same fundamental issue: uncertainty and multimodality in future robot behavior.

2.4.25 Summary

Generative robot policies model distributions over actions or trajectories rather than predicting a single deterministic control vector. This is useful because many robot tasks admit several different successful behaviors, and averaging those behaviors can produce an invalid solution.

Diffusion models introduce a forward process that gradually converts data into Gaussian noise,

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon,$$

and learn a reverse process that reconstructs data from noisy samples.

A common diffusion objective predicts the added noise:

$$\mathcal{L}_{\text{diff}} = \mathbb{E}\left[\|\epsilon - \epsilon_\theta(x_t, t, c)\|_2^2\right].$$

For robotics, conditioning allows the generated trajectory to depend on current observations, language, and robot state.

DDPM and DDIM primarily differ in the reverse sampling procedure. DDPM uses stochastic reverse sampling, whereas DDIM permits deterministic or reduced-step trajectories using the same general denoising model.

A Diffusion Transformer uses a Transformer as the denoising backbone. DiT is therefore an architectural choice rather than a distinct generative objective.

Flow matching takes a different perspective. It learns a vector field

$$v_\theta(x, \tau)$$

such that generation can be performed by integrating

$$\frac{dx}{d\tau} = v_\theta(x, \tau).$$

Conditional flow matching applies the same idea to robot actions:

$$\frac{dA}{d\tau} = v_\theta(A, \tau, c).$$

Diffusion and flow matching are mathematically related approaches to transporting a simple base distribution into a complex data distribution.

For robot control, the central practical considerations are multimodality, conditioning, trajectory coherence, and inference latency. A generative policy is valuable when multiple action sequences may be correct, but its sampling procedure must be efficient enough to satisfy the robot's control-rate requirements.

The central distinction can therefore be summarized as

$$\boxed{\text{Diffusion: learn to reverse corruption}}$$

and

$$\boxed{\text{Flow matching: learn the velocity of probability transport.}}$$

Both ultimately seek to model

$$\boxed{p(A|c),}$$

the distribution of valid action trajectories conditioned on the robot's current context.

2.5 Behavior Cloning, DAgger, and Trajectory Modeling

Imitation learning attempts to construct a policy by learning from demonstrations of desired behavior. In robotics, this is attractive because specifying a reward function for a complicated manipulation or navigation task can be difficult, whereas it may be comparatively easy for a human operator or an existing controller to demonstrate the desired behavior.

The simplest form of imitation learning is *behavior cloning*. Behavior cloning treats policy learning as a supervised learning problem: observations are used as inputs and demonstrated actions are used as targets.

This formulation is conceptually simple, but sequential decision-making introduces a difficulty that does not normally appear in static supervised learning. A prediction made by a control policy changes the physical system and therefore changes the inputs that the policy will receive in the future. Even a small error can move the robot into states that were rarely represented in the expert demonstrations. Subsequent predictions may then become less reliable, causing errors to accumulate over time.

This phenomenon connects several important concepts:

$$\text{behavior cloning} \rightarrow \text{covariate shift} \rightarrow \text{compounding error.}$$

The DAgger algorithm addresses this problem by collecting training examples from states visited by the learned policy itself rather than only from states visited by the expert.

A separate but complementary issue concerns the temporal representation of actions. A policy can predict one action at a time, or it can predict an entire short trajectory. The latter approach,

known as *action chunking*, allows the model to capture temporal structure and multiple possible short-horizon behaviors.

These ideas can be organized into two related themes:

$$\text{behavior cloning} \rightarrow \text{distribution shift} \rightarrow \text{DAgger},$$

and

$$\text{trajectory distributions} \rightarrow \text{action chunking} \rightarrow \text{short-horizon trajectory modeling}.$$

2.5.1 Behavior Cloning

Suppose an expert generates a trajectory

$$\tau = (o_1, a_1, o_2, a_2, \ldots, o_T, a_T),$$

where o_t denotes the observation available to the policy and a_t denotes the expert action.

From one or more trajectories, we construct a dataset of observation–action pairs,

$$\mathcal{D} = \{(o_i, a_i)\}_{i=1}^{N}.$$

A parameterized policy

$$\pi_\theta(a|o)$$

is trained to reproduce the demonstrated action.

For a deterministic continuous-action policy, a common objective is mean-squared error:

$$\mathcal{L}_{\mathrm{BC}}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \|a_i - \pi_\theta(o_i)\|_2^2 .$$

For discrete or stochastic actions, the corresponding maximum-likelihood objective is

$$\mathcal{L}_{\mathrm{BC}}(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \log \pi_\theta(a_i|o_i).$$

Behavior cloning is therefore ordinary supervised learning applied to expert demonstrations.

Its main attraction is simplicity. The learner does not need an explicitly designed reward function, online exploration, or reinforcement learning. If a sufficiently large collection of high-quality demonstrations is available, useful policies can often be learned directly from them.

The central limitation is that the supervised-learning objective is evaluated on observations from the demonstration dataset, while the learned policy is eventually evaluated on observations produced by its own actions.

2.5.2 Why Sequential Prediction Is Different from Static Prediction

Consider a conventional image-classification problem. If a classifier makes an incorrect prediction for one image, the mistake does not normally change the next image presented to the model.

Control is fundamentally different.

The action produced at time t changes the physical system,

$$s_t \to a_t \to s_{t+1},$$

and therefore changes the observation available to the policy at the next timestep.

Consequently,

$$\text{the policy's output at time } t$$

influences

$$\text{the policy's input at time } t+1.$$

This feedback loop is the reason small prediction errors can have effects far beyond a single timestep.

Suppose an expert policy keeps a system near a nominal condition

$$x \approx 0.$$

The training dataset therefore contains many examples near this region.

Now suppose the learned policy makes a small error and moves the system to

$$x = 0.05.$$

If this state is uncommon in the demonstrations, the prediction made there may be less accurate. The next state might become

$$x = 0.10,$$

followed by

$$x = 0.20,$$

and so forth.

The important point is that errors are not independent. One error changes the distribution of future observations and can make later errors more likely.

2.5.3 Covariate Shift in Imitation Learning

Let π_E denote the expert policy.

When the expert generates demonstrations, it induces a distribution over states,

$$s \sim d_{\pi_E}(s).$$

Behavior cloning is trained primarily using samples from this distribution.

At deployment, however, the learned policy π_θ generates its own trajectories and therefore induces a different state distribution,

$$s \sim d_{\pi_\theta}(s).$$

In general,

$$d_{\pi_\theta}(s) \neq d_{\pi_E}(s).$$

This discrepancy is the central distribution-shift problem in behavior cloning.

It may also be written as

$$p_{\text{train}}(s) \neq p_{\text{deployment}}(s).$$

The important feature of this shift is that it is *policy-induced*. The learner is not merely tested on a randomly different dataset. Its own decisions determine which future states it will encounter.

This creates the feedback process

$$\text{prediction error} \rightarrow \text{new state} \rightarrow \text{less familiar observation} \rightarrow \text{larger prediction error}.$$

2.5.4 Compounding Error

This phenomenon is often described as *compounding error*.

Suppose a learned policy has a small probability ϵ of making a poor decision at each timestep. If errors were independent and did not affect future states, one might expect error to accumulate approximately linearly with the horizon.

In a sequential control problem, however, the first error changes the state distribution. The policy may then encounter observations for which its prediction error is larger than it was on the expert distribution.

Therefore, the consequences of errors can grow more rapidly than a simple sum of independent mistakes would suggest.

The essential distinction is

$$\text{static supervised learning:} \quad \text{prediction does not change the next input},$$

whereas

$$\text{control:} \quad \text{prediction changes the next input.}$$

This is one of the fundamental reasons imitation learning is harder than ordinary supervised learning.

2.5.5 Dataset Aggregation

DAgger, short for *Dataset Aggregation*, was introduced to address the state-distribution mismatch created by behavior cloning.

The central idea is simple: instead of training only on states visited by the expert, the learner should also be trained on states that it visits itself.

Suppose the initial expert dataset is

$$\mathcal{D}_0.$$

A behavior-cloned policy

$$\pi_1$$

is trained from this dataset.

The policy is then deployed and generates states

$$s_1, s_2, \ldots$$

according to its own state-visitation distribution.

For each visited state s_t, the expert is asked what action should have been taken:

$$a_t^* = \pi_E(s_t).$$

These newly labeled examples are added to the dataset:

$$\mathcal{D}_1 = \mathcal{D}_0 \cup \{(s_t, \pi_E(s_t))\}.$$

A new policy π_2 is then trained on the enlarged dataset.

The process is repeated:

$$\pi_k \to \text{run learner} \to \text{collect visited states} \to \text{obtain expert labels} \to \mathcal{D}_{k+1} \to \pi_{k+1}.$$

The important change is that the training distribution gradually incorporates states drawn from

$$d_{\pi_\theta},$$

which is the distribution the learner must actually handle at deployment.

2.5.6 Why DAgger Improves Robustness

Consider a lane-following system.

An expert may provide demonstrations in which the vehicle is almost always centered in the lane. The resulting dataset therefore contains many examples of nominal driving and very few examples of substantial lateral error.

A learned policy may eventually drift into states such as

$$\text{20 cm right of center},$$

$$\text{40 cm right of center},$$

or

$$\text{rotated toward the road edge.}$$

A behavior-cloned policy may have little experience with these states.

DAgger deliberately collects them and asks the expert to specify corrective behavior, for example

$$\text{steer left},$$

$$\text{steer left more aggressively},$$

or

$$\text{reduce speed.}$$

The resulting dataset contains not only nominal demonstrations but also recovery behavior.

This distinction is important. A robust controller must often know how to recover from deviations rather than merely imitate trajectories that never deviate.

2.5.7 Expert Labeling and Expert Control

The expert does not necessarily have to control the system continuously during DAgger data collection.

The learned policy can generate the trajectory,

$$s \sim d_{\pi_\theta},$$

while the expert provides only the target action

$$\pi_E(s)$$

for each encountered state.

This separation is useful because the goal of DAgger is specifically to obtain expert supervision on states distributed according to the learner.

In practice, however, safety may require the expert to retain some control. One possible approach is to use a mixture policy

$$\pi_{\text{mix}} = \beta\pi_E + (1-\beta)\pi_\theta,$$

where β controls the contribution of expert behavior.

As training progresses and the learner becomes more reliable, β can be reduced.

2.5.8 Behavior Cloning and DAgger

Behavior cloning and DAgger solve related but distinct versions of the imitation-learning problem.

Behavior cloning trains on a fixed collection of expert demonstrations. It is simple, efficient, and often provides a strong initial policy.

DAgger introduces an iterative data-collection loop in which learner-visited states are labeled by the expert.

The advantage is that the dataset becomes better matched to the policy's deployment distribution.

The disadvantage is additional operational cost. The expert must remain available during training, and repeatedly running the policy on a physical robot can be expensive.

The distinction can be summarized as

$$\text{behavior cloning:} \quad \text{learn from expert-visited states,}$$

versus

$$\text{DAgger:} \quad \text{learn from learner-visited states labeled by the expert.}$$

2.5.9 Trajectory Distributions

The distribution-shift problem becomes clearer when imitation learning is described in terms of complete trajectories.

A trajectory can be written as

$$\tau = (s_0, a_0, s_1, a_1, \ldots, s_T).$$

Suppose the initial state follows

$$p(s_0),$$

the policy is

$$\pi(a_t|s_t),$$

and the environment dynamics are

$$p(s_{t+1}|s_t, a_t).$$

The policy and environment together induce the trajectory distribution

$$p_\pi(\tau) = p(s_0) \prod_{t=0}^{T-1} \pi(a_t|s_t) p(s_{t+1}|s_t, a_t).$$

This equation reveals three sources of variation in a trajectory:

1. the initial state,
2. the actions selected by the policy,
3. the environment's response to those actions.

Thus, changing the policy changes not merely the action distribution but the distribution of entire future rollouts.

2.5.10 Local Imitation Error Versus Rollout Quality

Behavior cloning optimizes a local objective such as

$$\pi_\theta(a|s) \approx \pi_E(a|s)$$

on states sampled from the expert dataset.

However, the quantity that ultimately matters in deployment is the quality of complete trajectories generated by the policy.

Ideally, one would like

$$p_{\pi_\theta}(\tau) \approx p_{\pi_E}(\tau).$$

These two requirements are not equivalent.

A policy may achieve very small behavior-cloning loss,

$$\mathcal{L}_{\text{BC}} \approx 0,$$

on expert states while still producing trajectories that differ substantially from expert behavior.

The reason is again feedback. A small action difference changes the next state, which changes the next policy input, and the trajectories gradually diverge.

2.5.11 State-Visitation Distributions

The trajectory distribution induces a state-visitation distribution

$$d_\pi(s).$$

Conceptually,

$$\text{policy} \rightarrow \text{trajectory distribution} \rightarrow \text{state-visitation distribution}.$$

This connects trajectory modeling directly to covariate shift.

The expert induces

$$p_{\pi_E}(\tau)$$

and therefore

$$d_{\pi_E}(s),$$

while the learned policy induces

$$p_{\pi_\theta}(\tau)$$

and therefore

$$d_{\pi_\theta}(s).$$

Behavior cloning becomes fragile when these state distributions diverge.

DAgger attempts to reduce the practical consequences of this divergence by adding expert supervision to states drawn from the learner's own distribution.

2.5.12 Multimodal Trajectory Distributions

A second difficulty in robot learning is that many tasks have more than one valid solution.

Consider a robot attempting to reach and grasp an object. It may approach the object from the left,

$$\tau_1 = \text{left-side approach},$$

from the right,

$$\tau_2 = \text{right-side approach},$$

or by first lifting above an obstacle,

$$\tau_3 = \text{overhead approach.}$$

All three trajectories may successfully accomplish the task.

The conditional trajectory distribution

$$p(\tau|o)$$

can therefore be multimodal.

This creates difficulty for simple deterministic regression.

Suppose two equally valid actions are

$$a_L = -1$$

and

$$a_R = +1.$$

If a deterministic policy is trained using mean-squared error, the optimal prediction under equal probability may be

$$\hat{a} = 0.$$

Yet 0 may correspond to a behavior that is invalid.

For instance, the two valid actions might represent moving around an obstacle from the left or right, while their average corresponds to driving directly into it.

Therefore,

$$\text{average of valid behaviors} \neq \text{necessarily a valid behavior.}$$

This problem motivates policies that represent full action distributions rather than only their conditional mean.

Examples include mixture models, autoregressive action models, diffusion policies, and flow-matching policies.

2.5.13 One-Step Action Prediction

A conventional policy predicts one action at each timestep:

$$a_t = \pi(o_t).$$

Probabilistically, it models a conditional distribution such as

$$p(a_t|o_t).$$

The policy is then queried again at the next timestep using the newly observed state.

This gives the controller frequent feedback and allows rapid reaction to environmental changes.

However, a one-step representation contains little explicit information about the temporal structure of future actions.

Many robot behaviors are naturally defined as coordinated sequences rather than isolated commands.

2.5.14 Action Chunking

Action chunking changes the policy output from one action to a sequence of future actions:

$$A_t = [a_t, a_{t+1}, \ldots, a_{t+H-1}],$$

where H denotes the chunk horizon.

If an individual action has dimension d_a, then

$$A_t \in \mathbb{R}^{H \times d_a}.$$

For example, suppose a manipulator uses a seven-dimensional action,

$$d_a = 7,$$

consisting of six end-effector motion components and a gripper command.

If the model predicts

$$H = 16$$

future actions, then each inference produces

$$16 \times 7 = 112$$

continuous values.

The policy is no longer modeling only

$$p(a_t|o_t),$$

but a short-horizon distribution more like

$$p(a_{t:t+H-1}|o_t).$$

2.5.15 Temporal Structure in Action Chunks

Action chunking allows the policy to model correlations between future control commands.

Consider a grasping motion. A reasonable action sequence may be

$$\text{approach} \rightarrow \text{slow down} \rightarrow \text{close gripper} \rightarrow \text{lift}.$$

These actions are not independent. The closing action should occur only after the hand reaches an appropriate pose, and lifting should occur only after the grasp has been established.

Predicting the sequence jointly allows the model to represent these dependencies directly.

A one-step policy, by contrast, repeatedly solves a local decision problem. Temporal consistency must emerge implicitly through the closed-loop interaction between successive predictions.

2.5.16 Action Chunking as Local Trajectory Modeling

An action chunk can be regarded as a short trajectory.

Instead of modeling

$$p(a_t|o_t),$$

the policy models

$$p(A_t|o_t).$$

This perspective becomes particularly important for generative action policies.

For the same observation o_t, a model may generate one valid chunk

$$A^{(1)} = \text{left-side grasp},$$

or another

$$A^{(2)} = \text{right-side grasp}.$$

Thus, action chunking creates a natural representation for multimodal short-horizon behavior.

Diffusion policies and flow-matching policies often operate on exactly this type of object: a distribution over complete action chunks.

2.5.17 Reducing Myopic Behavior

One-step policies can sometimes behave myopically because they optimize only the immediate next action.

Consider navigation around an obstacle.

The locally attractive action may point directly toward the goal. Yet successful motion may first require moving away from the goal:

$$\text{move sideways} \rightarrow \text{move forward}.$$

A short action chunk can explicitly encode such temporal structure.

In this sense, action chunking introduces a limited form of trajectory-level reasoning even when the policy does not contain a full planning algorithm.

2.5.18 The Open-Loop Limitation

Longer action chunks also create a new problem.

Suppose the policy predicts

$$[a_t, \ldots, a_{t+31}]$$

and executes all 32 actions without observing the environment again.

During this interval, execution is effectively open-loop.

If an object moves, a grasp slips, or the dynamics differ from expectation, later actions in the chunk may no longer be appropriate.

Increasing chunk length therefore introduces a tradeoff.

Longer chunks can improve temporal coherence and reduce the frequency of expensive model inference, but they also reduce the frequency of closed-loop feedback.

2.5.19 Receding-Horizon Execution

A common solution is to predict a relatively long chunk but execute only part of it.

Suppose the policy predicts H actions,

$$A_t = [a_t, \ldots, a_{t+H-1}],$$

but executes only the first K, where

$$K < H.$$

The robot then acquires a new observation and predicts another chunk.

For example,

$$H = 32, \qquad K = 4.$$

The procedure becomes

$$\text{observe} \rightarrow \text{predict 32 actions} \rightarrow \text{execute 4} \rightarrow \text{observe again} \rightarrow \text{replan}.$$

This is similar in spirit to receding-horizon or model-predictive control: plan farther into the future than the controller actually commits to executing.

The result combines short-horizon trajectory coherence with feedback robustness.

2.5.20 Overlapping Chunks and Temporal Ensembling

If the model replans frequently, different action chunks overlap.

At time t, the model may predict

$$[a_t^{(t)}, a_{t+1}^{(t)}, a_{t+2}^{(t)}, \ldots],$$

while at time $t+1$ it predicts

$$[a_{t+1}^{(t+1)}, a_{t+2}^{(t+1)}, \ldots].$$

There are now multiple predictions for the same future action.

These estimates can be combined. One weighted averaging rule is

$$\hat{a}_t = \frac{\sum_{k=0}^{K} w_k a_t^{(t-k)}}{\sum_{k=0}^{K} w_k}.$$

This procedure is commonly referred to as *temporal ensembling*.

The weights w_k can be chosen so that newer predictions receive greater importance than older predictions.

Averaging overlapping predictions can reduce high-frequency jitter and make execution smoother.

2.5.21 Action Chunking and Covariate Shift

Action chunking can improve temporal consistency, but it does not remove the distribution-shift problem of imitation learning.

A chunked policy still controls the system. Its actions change future states, and those states may drift outside the distribution represented in the demonstrations.

Thus, a chunked policy may still encounter observations satisfying

$$o \not\sim d_{\pi_E}.$$

DAgger and action chunking should therefore be understood as addressing different problems.

DAgger addresses *which states the policy is trained on*.

Action chunking addresses *how future actions are represented and predicted*.

They can be used together.

For example, a DAgger-style procedure could collect learner-visited observations while an action-chunk policy is trained to predict expert short-horizon trajectories from those observations.

2.5.22 A Unified View

Suppose an expert dataset contains trajectories

$$\mathcal{D}_E = \{\tau_1, \ldots, \tau_N\}.$$

Behavior cloning trains a policy

$$\pi_\theta$$

using observation–action pairs extracted from these trajectories.

Because the learner is imperfect, deployment induces a state distribution

$$d_{\pi_\theta}$$

that may differ from

$$d_{\pi_E}.$$

This is the core covariate-shift problem.

DAgger addresses the mismatch by collecting states

$$s \sim d_{\pi_\theta}$$

and requesting the expert action

$$a^* = \pi_E(s).$$

The expanded dataset therefore contains examples from the learner's own operating distribution.

At the same time, the temporal representation of the policy can be changed from one-step prediction,

$$p(a_t|o_t),$$

to chunked prediction,

$$p(a_{t:t+H-1}|o_t).$$

This gives the model access to short-horizon trajectory structure and makes it possible to represent multiple coherent future behaviors.

The broader goal of imitation learning can therefore be viewed as making the learner's rollout distribution resemble the expert's:

$$p_{\pi_\theta}(\tau) \approx p_{\pi_E}(\tau).$$

Behavior cloning attempts to achieve this indirectly through local supervised imitation.

DAgger improves the match between training and deployment state distributions.

Action chunking improves the representation of temporal action structure.

Distributional action models address the possibility that multiple different future trajectories may all be valid.

2.5.23 Summary

Behavior cloning formulates imitation learning as supervised learning on expert observation–action pairs. It is simple and effective when training and deployment observations are similar.

The difficulty is that a control policy changes the states it encounters. Small prediction errors can push the system away from the expert state distribution, producing covariate shift and compounding error.

DAgger addresses this problem by collecting states visited by the learner and obtaining expert labels for those states. This teaches the policy how to behave not only under nominal expert conditions but also in states created by its own mistakes.

The trajectory-distribution perspective explains why local imitation error is not sufficient to characterize policy quality. Ultimately, a policy induces a distribution over complete trajectories,

$$p_\pi(\tau) = p(s_0) \prod_{t=0}^{T-1} \pi(a_t|s_t) p(s_{t+1}|s_t, a_t),$$

and the deployment behavior of interest is determined by this rollout distribution.

Robot behavior is also frequently multimodal: several distinct action sequences may successfully solve the same task. Deterministic regression can average these modes and produce an invalid action.

Action chunking addresses temporal representation by predicting

$$A_t = [a_t, \ldots, a_{t+H-1}]$$

rather than a single action. This allows the policy to model short-horizon trajectory structure, reduces myopic behavior, and provides a natural representation for generative action models.

Chunked control is commonly combined with receding-horizon execution and temporal ensembling to preserve feedback and improve smoothness.

The concepts therefore address complementary aspects of imitation learning:

$$\text{behavior cloning} \rightarrow \text{learn expert actions},$$

$$\text{DAgger} \rightarrow \text{reduce learner–expert state-distribution mismatch},$$

and

$$\text{action chunking} \rightarrow \text{represent coherent short-horizon behavior.}$$

Together, they provide a useful framework for understanding how modern robot policies move from simple supervised imitation toward robust trajectory-level behavior.

2.6 Reinforcement Learning Fundamentals

Reinforcement learning studies sequential decision-making problems in which an agent interacts with an environment, receives feedback in the form of rewards, and attempts to learn behavior that maximizes long-term performance. Unlike supervised learning, where the desired output is usually provided directly for each input, reinforcement learning must infer which actions are desirable from their consequences over time.

This temporal dependence is the defining difficulty of reinforcement learning. An action may produce little immediate reward but place the system in a state from which large future rewards become possible. Conversely, an action that appears beneficial in the short term may lead to poor outcomes later. Reinforcement learning therefore requires reasoning about both immediate and delayed consequences.

A useful progression is

$$\text{MDP/POMDP} \rightarrow \text{value functions} \rightarrow \text{Bellman equations} \rightarrow \text{value-based and policy-based methods} \rightarrow \text{actor–critic}$$

Algorithms such as Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC) can then be understood as particular actor–critic methods with different assumptions about data collection and optimization.

2.6.1 Markov Decision Processes

The standard mathematical model for reinforcement learning is the *Markov Decision Process* (MDP). An MDP is usually written as

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma),$$

where

- $\mathcal{S}$ is the state space,
- $\mathcal{A}$ is the action space,
- $P(s'|s, a)$ is the transition distribution,

- $R(s, a, s')$ defines the reward,
- $\gamma \in [0, 1)$ is the discount factor.

At time t, the agent observes the current state s_t, chooses an action a_t, receives reward r_t, and transitions to a new state s_{t+1}:

$$s_t \to a_t \to (r_t, s_{t+1}).$$

A stochastic policy is a conditional distribution

$$\pi(a|s),$$

and actions are sampled according to

$$a_t \sim \pi(\cdot|s_t).$$

The environment then evolves according to

$$s_{t+1} \sim P(\cdot|s_t, a_t).$$

The policy and transition dynamics together induce a distribution over complete trajectories of states and actions.

2.6.2 Return and the Reinforcement-Learning Objective

The goal of the agent is not generally to maximize only the immediate reward r_t. Instead, it attempts to maximize the cumulative discounted return

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}.$$

The discount factor γ controls the relative importance of future rewards. If γ is close to zero, the agent behaves myopically and emphasizes immediate reward. If γ is close to one, future outcomes matter strongly.

The objective of reinforcement learning is therefore commonly written as

$$J(\pi) = \mathbb{E}_\pi[G_0].$$

The expectation is necessary because trajectories may be stochastic due to randomness in the policy, the environment, or both.

2.6.3 The Markov Property

The defining assumption of an MDP is the Markov property:

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots) = P(s_{t+1}|s_t, a_t).$$

This states that the current state s_t contains all information needed to predict the distribution of the next state once the current action is known.

The Markov property does not mean that history is irrelevant in the physical system. Rather, it means that the state representation already summarizes the relevant history.

For example, consider a moving robot. If its state contains only position,

$$s_t = q_t,$$

then the future may depend on an unobserved velocity, and the representation may not be Markov. A richer state might be

$$s_t = (q_t, \dot{q}_t, \text{object poses}, \text{object velocities}).$$

Whether a representation is Markov therefore depends partly on which variables are included in the state.

2.6.4 Partially Observable Decision Processes

Physical robots rarely observe the full state of the world directly. Cameras may be occluded, object velocities may be unknown, and some relevant quantities may not be sensed at all.

Such problems are naturally modeled as *Partially Observable Markov Decision Processes* (POMDPs).

The underlying environment still has a latent state s_t, but the agent receives an observation o_t generated according to an observation model

$$o_t \sim O(\cdot|s_t).$$

A policy that acts only on the current observation has the form

$$\pi(a_t|o_t).$$

However, a single observation may not contain enough information to make a good decision. A more general policy can condition on observation and action history:

$$\pi\left(a_t \mid o_{\leq t}, a_{<t}\right).$$

This is especially important in robotics. A single image may reveal an object's position but not whether it is moving left or right. Several consecutive frames can provide the missing temporal information.

2.6.5 Belief States

In a POMDP, an ideal decision-maker maintains a *belief state*,

$$b_t(s) = P\left(s_t = s \mid o_{\leq t}, a_{<t}\right).$$

The belief state is a probability distribution over possible hidden states given the available history.

Although explicitly maintaining such a distribution can be difficult in high-dimensional systems, modern neural policies often learn approximate latent state estimators. Recurrent neural networks, Transformers, frame stacks, and latent-state models can all use history to construct an internal representation that serves a similar role.

Thus, in embodied systems, it is often useful to interpret a history encoder as a learned approximation to belief-state estimation.

2.6.6 State and Action Value Functions

A central idea in reinforcement learning is to assign numerical values to states and actions.

The *state-value function* of policy π is

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[G_t \mid s_t = s\right].$$

It represents the expected future return when the agent starts in state s and subsequently follows policy π.

The *action-value function* is

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}\left[G_t \mid s_t = s, a_t = a\right].$$

It represents the expected return after taking action a in state s and then following π.

The difference is therefore

$$V^{\pi}(s) = \text{value of being in state } s,$$

whereas

$$Q^{\pi}(s, a) = \text{value of taking action } a \text{ in state } s.$$

These functions convert the long-term sequential objective into quantities that can be estimated from local transitions.

2.6.7 Bellman Equations

Value functions satisfy recursive relationships known as Bellman equations.

For a fixed policy π,

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi,\, s'\sim P}\left[r(s,a,s') + \gamma V^{\pi}(s')\right].$$

This equation follows directly from separating the current reward from all future rewards.

The same idea for the action-value function gives

$$Q^{\pi}(s,a) = \mathbb{E}\left[r + \gamma \mathbb{E}_{a'\sim\pi} Q^{\pi}(s',a')\right].$$

The conceptual structure is

$$\text{current value} = \text{immediate reward} + \text{discounted future value}.$$

Bellman equations are fundamental because they replace a potentially long-horizon return with a one-step recurrence.

2.6.8 Bellman Optimality

The optimal state- and action-value functions are denoted by

$$V^{*}(s)$$

and

$$Q^{*}(s,a).$$

For the optimal action-value function,

$$Q^{*}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q^{*}(s',a')\right].$$

The maximization expresses the assumption that the agent behaves optimally after reaching s'.

Once Q^{*} is known, an optimal deterministic policy can be recovered using

$$\pi^{*}(s) = \arg\max_{a} Q^{*}(s,a).$$

This relationship motivates value-based reinforcement-learning methods such as Q-learning.

2.6.9 Temporal-Difference Learning

Bellman equations also motivate *temporal-difference* (TD) learning.

Suppose the agent observes a transition

$$(s_t, a_t, r_t, s_{t+1}).$$

A one-step Q-learning target is

$$y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a').$$

The temporal-difference error is

$$\delta_t = y_t - Q(s_t, a_t).$$

The current estimate is therefore compared with a bootstrapped estimate containing the immediate reward and the value of the next state.

The term *bootstrapping* refers to using one learned estimate to construct a target for another learned estimate. This makes TD methods efficient, but it can also introduce instability if the estimates are inaccurate.

2.6.10 Q-Learning

Q-learning attempts to learn the optimal action-value function directly.

In the tabular setting, its update is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right],$$

where α is the learning rate.

The term in brackets is exactly the TD error.

Once the Q-function has been learned, the policy is

$$\pi(s) = \arg\max_a Q(s, a).$$

Q-learning is an *off-policy* algorithm. The action actually taken during data collection may come from an exploratory behavior policy, while the learning target uses the greedy action

$$\max_{a'} Q(s', a').$$

The behavior policy and target policy therefore need not be the same.

This makes Q-learning compatible with replay buffers and previously collected experience.

2.6.11 Deep Q-Networks

When the state space is too large for a table, the Q-function can be approximated by a neural network:

$$Q_\theta(s, a).$$

A Deep Q-Network (DQN) minimizes a Bellman-error objective of the form

$$\mathcal{L}(\theta) = \left(Q_\theta(s_t, a_t) - y_t\right)^2,$$

where

$$y_t = r_t + \gamma \max_{a'} Q_{\theta^-}(s_{t+1}, a').$$

The parameters θ^- belong to a separate *target network* that is updated more slowly than the primary network.

The purpose of the target network is to reduce instability caused by using a rapidly changing function approximator to define its own targets.

2.6.12 The Difficulty of Continuous Actions

Value-based methods are particularly natural when the action space is small and discrete.

For example,

$$\mathcal{A} = \{\text{left}, \text{right}, \text{forward}, \text{stop}\}$$

can be searched explicitly.

Robotic control often instead requires continuous actions such as

$$a = [\tau_1, \ldots, \tau_7] \in \mathbb{R}^7.$$

In this case, evaluating

$$\max_a Q(s, a)$$

may require solving a continuous optimization problem at every decision step.

This difficulty motivates methods that represent the policy directly instead of recovering it through maximization of a value function.

2.6.13 Policy Gradient Methods

Policy-gradient methods parameterize the policy directly:

$$\pi_\theta(a|s).$$

The objective is

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[G(\tau)].$$

The policy-gradient theorem gives

$$\nabla_\theta J = \mathbb{E}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) Q^\pi(s_t, a_t)\right].$$

The term

$$\nabla_\theta \log \pi_\theta(a_t|s_t)$$

indicates how the parameters should change to make the sampled action more or less probable.

The value term determines whether that probability should increase or decrease.

Actions associated with high return receive positive reinforcement, while actions associated with poor return become less likely.

2.6.14 REINFORCE

The simplest Monte Carlo policy-gradient algorithm is REINFORCE. It uses the observed return directly:

$$\nabla_\theta J \approx \sum_t \nabla_\theta \log \pi_\theta(a_t|s_t) G_t.$$

Although conceptually simple, this estimator can have high variance.

For example, a trajectory may receive high total reward even though only a few actions were responsible for the success. Weighting every action by a noisy trajectory return makes learning statistically inefficient.

This motivates the use of baselines and learned value functions.

2.6.15 The Advantage Function

A common variance-reduction technique is to compare an action with the expected value of the state in which it was selected.

The *advantage function* is

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s).$$

It measures whether an action is better or worse than the policy's usual expected outcome from that state.

If

$$A^\pi(s, a) > 0,$$

the action performed better than expected and should become more probable.

If

$$A^\pi(s,a) < 0,$$

the action performed worse than expected and should become less probable.

The policy gradient can therefore be written as

$$\nabla_\theta J = \mathbb{E}\left[\nabla_\theta \log \pi_\theta(a|s) A^\pi(s,a)\right].$$

Subtracting a suitable baseline does not change the expected gradient, but can substantially reduce its variance.

2.6.16 Actor–Critic Methods

Actor–critic methods combine explicit policy learning with value-function estimation.

The *actor*

$$\pi_\theta(a|s)$$

represents the policy and determines which action to take.

The *critic*

$$V_\phi(s)$$

or

$$Q_\phi(s,a)$$

estimates how desirable the actor's states or actions are.

The actor can therefore be viewed as the component that makes decisions, while the critic provides a learned evaluation signal.

A typical actor update contains a term such as

$$\nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t,$$

where $\hat{A}_t$ is an estimate of the advantage.

The critic is trained by minimizing an error such as

$$\left(V_\phi(s_t) - y_t\right)^2.$$

Actor–critic methods are especially important for continuous-control problems because they avoid explicitly maximizing $Q(s,a)$ over all possible actions at every timestep.

2.6.17 Advantage Estimation

One simple estimate of the advantage is the one-step TD error:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t).$$

This estimator has relatively low variance but may be biased when the learned value function is inaccurate.

A widely used alternative is *Generalized Advantage Estimation* (GAE):

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \cdots .$$

Equivalently,

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}.$$

The parameter λ controls a bias–variance tradeoff.

Smaller values place more emphasis on short-horizon bootstrapped estimates, whereas values closer to one incorporate longer-horizon return

2.7 Data Sources and Dataset Design for Robot Learning

The performance of a learned robotic system depends not only on the model architecture and optimization procedure, but also on the data from which the model is trained. In robot learning, this dependence is particularly strong because the data distribution determines which states, actions, environments, and failure modes the policy has had an opportunity to observe. A highly expressive policy cannot reliably infer behavior in regions of the state space that are poorly represented in its training set.

Robot-learning datasets may be collected from several sources, including human teleoperation, autonomous robot interaction, human video, simulation, and other synthetic processes. These sources differ in what kinds of supervision they provide. Teleoperation gives direct correspondence between observations and robot actions. Human video offers large-scale information about task semantics, but usually lacks robot-specific control labels. Simulation provides inexpensive and precisely labeled experience, although the simulated distribution may differ from the physical world. Replay buffers and dataset sampling strategies further determine which of these experiences are emphasized during optimization.

Consequently, dataset design is not simply a matter of collecting as many examples as possible. It involves deciding which experiences should be collected, how they should be represented, and how frequently they should appear during training.

2.7.1 Demonstrations and Trajectories

A common unit of data in robot learning is the *trajectory*. A trajectory records the evolution of observations and actions over time and may be written as

$$\tau = (o_0, a_0, o_1, a_1, \ldots, o_T, a_T),$$

where o_t denotes the observation available to the agent at time t, and a_t denotes the corresponding action. If the complete physical state of the system is available, the trajectory may instead be written using s_t:

$$\tau = (s_0, a_0, s_1, a_1, \ldots, s_T, a_T).$$

Depending on the learning problem, additional quantities may also be associated with the trajectory. These can include rewards r_t, task instructions ℓ, success indicators, contact labels, or other auxiliary signals.

A demonstration dataset containing N trajectories can therefore be written as

$$\mathcal{D} = \{\tau_1, \tau_2, \ldots, \tau_N\}.$$

The term *demonstration* is often used when a trajectory is intended to illustrate useful or task-relevant behavior. Importantly, however, a demonstration is not necessarily optimal. Human demonstrations may contain hesitation, unnecessary motions, imperfect corrections, or complete failures. A learned policy therefore tends to reproduce the statistical structure of the demonstrated behavior rather than some abstract notion of the best possible behavior.

This distinction is important in imitation learning. If undesirable behaviors appear frequently in the dataset, a sufficiently powerful imitation model may learn them along with the desirable ones.

2.7.2 Behavior Cloning from Demonstrations

The simplest use of demonstrations is behavior cloning. In behavior cloning, a policy is trained as a supervised learning model that predicts the demonstrated action from the current observation.

Let

$$\pi_\theta(a \mid o)$$

denote a policy parameterized by θ. For deterministic continuous actions, a common training objective is

$$\mathcal{L}_{\mathrm{BC}}(\theta) = \mathbb{E}_{(o,a)\sim\mathcal{D}} \left[\|\pi_\theta(o) - a\|_2^2 \right].$$

The policy is therefore encouraged to produce an action close to the demonstrated action.

For stochastic or discrete-action policies, behavior cloning is commonly formulated as maximum likelihood:

$$\mathcal{L}_{\mathrm{BC}}(\theta) = -\mathbb{E}_{(o,a)\sim\mathcal{D}} \left[\log \pi_\theta(a \mid o)\right].$$

In both cases, the training signal comes directly from the demonstration dataset. This makes behavior cloning simple and stable, but it also means that the resulting policy is strongly shaped by the quality and coverage of that dataset.

2.7.3 Dataset Quality, Diversity, and Coverage

Three closely related properties are useful when reasoning about robot-learning datasets: *quality*, *diversity*, and *coverage*.

Dataset quality describes how desirable or correct the recorded actions are. A dataset containing smooth, successful, and deliberate demonstrations usually provides a cleaner supervision signal than one dominated by operator errors or failed trials.

Diversity describes variation across the dataset. This may include variation in objects, lighting, camera viewpoints, task instructions, initial conditions, robot poses, environments, and strategies for solving a task. Diversity is particularly important when the learned policy will be expected to generalize beyond a narrow training setup.

Coverage refers to the extent to which the relevant state and action spaces are represented. A dataset may contain a large number of examples while still having poor coverage. For example, a manipulation dataset may contain millions of observations of successful grasps but almost no examples of recovery after a missed grasp.

The three properties are related but not interchangeable. A small dataset can have high quality but insufficient coverage. Conversely, a very large dataset can have broad coverage but contain low-quality demonstrations. Generalization depends on the combination of these properties rather than on dataset size alone.

2.7.4 Teleoperation as a Source of Robot Data

Teleoperation is one of the most direct methods for collecting robot demonstrations. A human operator controls the physical or simulated robot using an interface such as a joystick, spacemouse, VR controller, motion-capture system, or leader robot. While the operator performs the task, the system records the robot's observations and actions.

For a manipulation system, the observation might include an RGB image I_t, robot joint configuration q_t, and gripper state g_t:

$$o_t = (I_t, q_t, g_t).$$

The corresponding action might be expressed as a relative end-effector command,

$$a_t = (\Delta x_t, \Delta y_t, \Delta z_t, \Delta r_t, g_{t+1}),$$

where the translational and rotational increments define the desired motion and g_{t+1} controls the gripper.

The principal advantage of teleoperation is that the robot observation and the executed action are naturally aligned in time. The resulting pairs

$$(o_t, a_t)$$

can therefore be used directly for behavior cloning.

This property distinguishes teleoperation from ordinary human video. A video of a person performing a task may reveal what happened, but it does not directly specify which actuator commands a robot should produce.

2.7.5 Noise and Bias in Teleoperation

Teleoperation should not be treated as a perfect source of expert behavior. Human operators often overshoot targets, pause unnecessarily, make corrective motions, or compensate for delays in the control interface. Some behaviors may reflect the peculiarities of the teleoperation device rather than the strategy that an autonomous robot should ideally execute.

As a result, the teleoperation action distribution

$$p_{\text{teleop}}(a \mid o)$$

may differ from the action distribution desired at deployment.

Practical datasets are therefore often processed before training. Trajectories may be segmented into individual tasks, synchronized across sensors, filtered for quality, labeled as successful or unsuccessful, or smoothed to reduce control noise. The preprocessing stage can substantially affect the final policy because it determines which parts of the recorded experience remain in the training distribution.

2.7.6 Learning from Human Video

Human video provides a very different type of supervision. Large-scale video collections may contain people interacting with objects, manipulating tools, opening doors, preparing food, and performing many other tasks relevant to robotics.

Such data contain rich semantic information. From a video of someone opening a drawer, a model may learn that the handle is the relevant interaction point, that grasping precedes pulling, and that the desired final state is an open drawer.

However, human video does not normally provide robot control labels. It does not specify the robot's joint configuration,

$$[q_1, \ldots, q_n],$$

its actuator torques,

$$[\tau_1, \ldots, \tau_n],$$

or the precise end-effector commands required for execution.

Human video is therefore more naturally interpreted as supervision about the structure and semantics of a task than as direct supervision of robot motor commands.

2.7.7 The Embodiment Gap

The difficulty of transferring behavior from humans to robots is often described as the *embodiment gap*. Humans and robots differ in morphology, kinematics, sensing, control frequency, force capability, reachability, and compliance. Even when a human and robot accomplish the same semantic task, the physical trajectories used to do so may be very different.

For example, human video may reveal that opening a drawer involves the sequence

$$\text{locate handle} \rightarrow \text{grasp handle} \rightarrow \text{pull outward}.$$

This sequence captures the task structure. It does not determine the robot-specific grasp pose, joint configuration, force profile, or motion trajectory.

It is therefore useful to distinguish two forms of knowledge:

$$\text{human data} \quad \rightarrow \quad \text{what should happen},$$

and

$$\text{robot data} \quad \rightarrow \quad \text{how this robot can make it happen}.$$

This distinction explains why large-scale human data and comparatively small robot datasets can be complementary rather than redundant. Human data can supply broad semantic priors, while robot data ground those priors in the robot's own action space.

2.7.8 Synthetic Data and Simulation

Synthetic data are generated by an artificial process rather than collected directly from real-world human or robot interaction. In robotics, physics simulation is one of the most important synthetic data sources.

A simulator can produce complete trajectories together with quantities that may be difficult or expensive to measure in the physical world. These can include RGB images, depth maps, segmentation masks, contact states, forces, rewards, and exact object poses.

For example, the simulator may directly provide the six-degree-of-freedom pose

$$T_i \in SE(3)$$

of every object in the environment. In the real world, obtaining the same quantity would require perception, motion capture, or another estimation system.

Simulation is therefore especially valuable when precise labels are required at large scale.

Another advantage is that simulated interaction can be parallelized. Thousands of simulated environments may generate experience simultaneously, allowing data collection at rates that would be impractical with physical robots. Simulation also makes it possible to create rare or dangerous scenarios without risking hardware.

2.7.9 The Sim-to-Real Gap

The usefulness of simulation is limited by the fact that simulated and physical environments are not identical.

Let

$$p_{\text{sim}}(x)$$

denote the distribution of observations and experiences produced in simulation, and

$$p_{\text{real}}(x)$$

the corresponding real-world distribution. In general,

$$p_{\text{sim}}(x) \neq p_{\text{real}}(x).$$

This discrepancy is known as the *sim-to-real gap*.

The gap can arise from inaccuracies in rendering, contact mechanics, actuator dynamics, friction, compliance, sensor noise, latency, and many other factors. Even small modeling errors can become important if a learned policy discovers behavior that exploits simulator-specific properties.

For this reason, simulation is better understood as a source of inexpensive and controllable experience than as a perfect replacement for real-world data.

2.7.10 Domain Randomization

One common method for reducing sensitivity to simulation inaccuracies is *domain randomization*. Instead of training in a single fixed simulated environment, properties of the simulation are varied across episodes.

Randomized parameters may include lighting, texture, object geometry, camera position, friction, mass, actuator parameters, sensor noise, and initial conditions.

The objective is not necessarily to make any one simulation perfectly realistic. Instead, the goal is to make

$$p_{\text{sim}}(x)$$

broad enough that the physical deployment distribution lies within, or close to, the range of variations encountered during training.

The intuition is that a policy which succeeds under many simulated variations is less likely to depend on a simulator-specific feature that disappears in the real world.

2.7.11 Other Forms of Synthetic Data

Synthetic data are not limited to physics simulation. Procedural generators can create object configurations, environments, and task variants. Generative models can produce images, language instructions, or candidate trajectories. Language data can also be augmented through paraphrasing.

For instance, an instruction such as

> Pick up the red cup.

might be augmented with semantically equivalent instructions such as

> Grab the red mug.

or

> Lift the red cup from the table.

Such augmentation changes the linguistic distribution seen during training and may improve robustness to phrasing at deployment.

2.7.12 Replay Buffers

In reinforcement learning, interaction data are often stored in a *replay buffer*. A basic replay buffer contains transitions of the form

$$(s_t, a_t, r_t, s_{t+1}),$$

possibly together with termination indicators, task identities, success labels, or auxiliary sensor information.

Let the replay buffer be denoted by

$$\mathcal{B}.$$

Training samples can then be drawn according to

$$(s, a, r, s') \sim \mathcal{B}.$$

The replay buffer separates data collection from parameter updates. Rather than learning only from the newest transition, the algorithm can repeatedly reuse earlier experience.

2.7.13 Why Experience Replay Is Useful

Experience replay has two important effects.

First, it improves sample efficiency. A transition obtained from an expensive physical interaction can contribute to multiple gradient updates rather than being discarded after one use.

Second, it reduces temporal correlation between training examples. Consecutive states in a trajectory are usually highly correlated:

$$s_t \approx s_{t+1}.$$

If optimization repeatedly uses consecutive transitions, successive minibatches may contain very similar information. Random sampling from a replay buffer mixes transitions collected at different times and in different episodes, producing a more diverse stochastic gradient estimate.

Replay buffers are especially natural in off-policy algorithms such as Q-learning and Soft Actor-Critic because these methods can learn from experience generated by policies other than the current policy.

2.7.14 Trajectory Replay

For sequence models, a single transition may not contain enough context. Recurrent policies, Transformers, and action-chunking policies often require a temporal window.

The replay unit may therefore be a subsequence such as

$$\tau_{t:t+H} = (o_t, a_t, \ldots, o_{t+H}, a_{t+H}).$$

Sampling entire windows preserves the temporal dependencies needed for learning motion, velocity, multi-step behaviors, and other sequential structure.

2.7.15 Replay Buffers and Offline Datasets

A replay buffer should be distinguished from an offline dataset.

A replay buffer normally changes while learning proceeds. If $\mathcal{B}_t$ denotes its contents at training time t, then

$$\mathcal{B}_t \to \mathcal{B}_{t+1}$$

as new experience is added and possibly old experience is removed.

An offline dataset, by contrast, is fixed before training begins:

$$\mathcal{D}_{\text{offline}} = \text{constant}.$$

The policy cannot collect new environment interactions to correct weaknesses in the dataset. Consequently, the quality and coverage of an offline dataset are particularly important.

2.7.16 Dataset Imbalance

Large robot datasets are rarely balanced. Some tasks may appear millions of times, whereas others may have only a few thousand examples. Similarly, long stretches of ordinary free-space motion may dominate rare events such as grasp transitions, collisions, corrective maneuvers, or recovery behavior.

Suppose task k contributes n_k examples. If individual examples are sampled uniformly from the raw dataset, then the probability that a training example belongs to task k is

$$P(k) = \frac{n_k}{\sum_j n_j}.$$

Consequently, tasks with more recorded examples contribute proportionally more gradient updates.

This is not necessarily undesirable. If the raw dataset accurately reflects the distribution encountered at deployment, such sampling may be appropriate. However, it can cause rare tasks or rare but safety-critical events to contribute very little to the training objective.

2.7.17 Task Balancing

One alternative is to sample tasks uniformly:

$$P(k) = \frac{1}{K},$$

where K is the number of tasks.

This removes the influence of dataset size, but may give excessive weight to tasks represented by very small datasets.

A useful interpolation between raw-frequency sampling and uniform task sampling is

$$P(k) \propto n_k^{\alpha}, \qquad 0 < \alpha < 1.$$

When

$$\alpha = 1,$$

sampling follows the raw dataset frequency. As

$$\alpha \to 0,$$

the sampling distribution approaches uniform weighting across tasks.

The parameter α therefore controls how aggressively large tasks are downweighted relative to small tasks.

2.7.18 Balancing Rare Events

Balancing may also be performed according to event type rather than task identity.

Consider a dataset in which 95% of episodes are successful and only 5% contain failures. If samples are drawn uniformly, the optimizer sees relatively few examples of how the robot behaves near failure states.

However, these states may be precisely the ones in which robust recovery behavior is most important.

One may therefore oversample examples containing

- failures,
- collisions,
- grasp transitions,
- contact events,
- corrective actions,
- recovery behavior.

The appropriate sampling distribution depends on the objective. Rare events need not be sampled according to their natural frequency if their consequences at deployment are disproportionately important.

2.7.19 The Effective Training Distribution

Sampling and weighting modify the distribution actually seen by the optimizer.

Suppose the collected dataset follows

$$p_{\text{data}}(x).$$

If an example x is assigned sampling weight $w(x)$, then the effective training distribution is proportional to

$$\boxed{p_{\text{train}}(x) \propto w(x) p_{\text{data}}(x).}$$

This equation is important because it shows that dataset balancing is not merely an implementation detail. It changes the statistical objective being optimized.

The model is not trained on the raw dataset distribution unless

$$w(x) = \text{constant}.$$

Every choice of oversampling, undersampling, prioritization, or task reweighting changes which regions of the experience space contribute most strongly to learning.

2.7.20 Training Distribution and Deployment Distribution

Ideally, the training data should provide sufficient support for the conditions encountered during deployment.

Let

$$x \sim p_{\text{train}}(x)$$

denote training experiences and

$$x \sim p_{\text{deploy}}(x)$$

denote deployment experiences.

If the deployment distribution contains states that are rarely or never represented under p_{train}, the policy must operate out of distribution.

This observation suggests an important principle of robot dataset design: coverage should include not only nominal task execution, but also states created by disturbances, mistakes, environmental variation, and recovery.

The desired training distribution is therefore not always identical to the naturally collected distribution. Dataset design often requires deliberately increasing representation of states that are rare in collected data but important for robust deployment.

2.7.21 Combining Data Sources

Modern robot-learning systems often combine multiple data sources because no single source provides all of the required information.

A useful conceptual decomposition is

$$\begin{aligned} \text{Human video} &\rightarrow \text{semantic and task knowledge,} \\ \text{teleoperation} &\rightarrow \text{robot-specific action supervision,} \\ \text{simulation} &\rightarrow \text{large-scale controlled variation,} \\ \text{autonomous interaction} &\rightarrow \text{states induced by deployed policies.} \end{aligned}$$

These sources may then be filtered, segmented, labeled, and sampled to construct the final training distribution.

The resulting pipeline can be written abstractly as

$$\text{Raw Experience} \rightarrow \text{Curation} \rightarrow \text{Sampling} \rightarrow p_{\text{train}} \rightarrow \text{Optimization} \rightarrow \pi_\theta.$$

This perspective highlights an important fact: the learner never sees "the dataset" in an abstract sense. It sees a sequence of minibatches generated by the collection and sampling procedure.

2.7.22 Dataset Design as Part of the Learning Algorithm

Dataset design is sometimes treated as separate from model design, but in large-scale robot learning the two are closely coupled.

Changing the model architecture changes which forms of data can be exploited. For example, temporal models can make use of trajectory windows that would be invisible to a single-frame policy. Conversely, changing the dataset distribution changes which behaviors the model is encouraged to represent.

A policy trained mainly on successful nominal behavior may become highly accurate in nominal states but fragile under disturbance. A policy trained on a more diverse collection containing corrective actions and failure recovery may have slightly more difficult optimization but greater deployment robustness.

Thus, the final learned behavior depends jointly on

$$\text{model capacity}, \qquad \text{training objective}, \qquad \text{data distribution}.$$

In practice, these should be designed together.

2.7.23 Summary

Robot-learning datasets are composed of trajectories containing observations, actions, and potentially rewards, task descriptions, and auxiliary labels. Demonstrations provide direct supervision for imitation learning, but their usefulness depends on their quality and coverage.

Teleoperation is particularly valuable because it yields robot-specific observation-action pairs, although operator noise and interface bias can enter the data. Human video provides vastly larger-scale semantic information but suffers from an embodiment gap because human actions do not directly map to robot control commands.

Simulation and other synthetic sources provide inexpensive and precisely labeled data, but introduce a sim-to-real distribution gap. Domain randomization attempts to improve transfer by deliberately broadening the simulation distribution.

Replay buffers improve sample efficiency and reduce temporal correlation by reusing previously collected interaction data. Finally, dataset balancing determines the effective distribution presented to the optimizer. If raw data follow $p_{\text{data}}(x)$ and examples are sampled with weight $w(x)$, then the learner effectively trains under

$$p_{\text{train}}(x) \propto w(x) p_{\text{data}}(x).$$

The central lesson is that the behavior of a learned robot is shaped not simply by the number of examples available, but by the distribution of experience that reaches the optimizer. Dataset quality, diversity, coverage, and sampling strategy are therefore fundamental components of robot-policy design.

2.8 Evaluation Under Distribution Shift

A learned robotic policy is useful only to the extent that it performs reliably on the distribution of situations encountered at deployment. Evaluating such a policy therefore requires more than measuring average performance on a held-out subset of the same dataset used for training. In embodied systems, changes in environment, object appearance, robot state, camera viewpoint, task specification, or dynamics can move the system away from the training distribution. A useful evaluation methodology must explicitly characterize this distribution shift and measure both nominal task performance and robustness to unfamiliar conditions.

2.8.1 Training and Test Distributions

Let the training data be drawn from a distribution

$$(x, y) \sim p_{\text{train}}(x, y),$$

where x denotes the model input and y denotes the target output. In robot learning, x may contain observations, language instructions, proprioceptive state, or complete trajectory histories, while y may represent actions, action chunks, rewards, or task outcomes.

Evaluation data are drawn from

$$(x, y) \sim p_{\text{test}}(x, y).$$

The simplest evaluation setting assumes

$$p_{\text{test}}(x, y) \approx p_{\text{train}}(x, y).$$

This is commonly referred to as an *independent and identically distributed*, or IID, evaluation setting. The test examples are different from the training examples, but they are assumed to originate from approximately the same underlying distribution.

For example, suppose a manipulation dataset contains mugs placed randomly on a table, with object position sampled according to

$$(x_{\text{mug}}, y_{\text{mug}}) \sim p_{\text{train}}.$$

A conventional test set may use previously unseen mug placements sampled from the same distribution. Such a test measures generalization to unseen examples but does not necessarily measure generalization to qualitatively new conditions.

In deployment, it is common for

$$p_{\text{deployment}} \neq p_{\text{train}}.$$

This discrepancy is called *distribution shift*. Distribution shift may occur in the visual inputs, robot dynamics, task distribution, action consequences, or combinations of these factors.

A useful distinction is between changes in the input distribution and changes in the underlying relationship between inputs and desired outputs. In classical covariate shift,

$$p_{\text{train}}(x) \neq p_{\text{test}}(x),$$

while approximately

$$p_{\text{train}}(y|x) = p_{\text{test}}(y|x).$$

For example, a robot may encounter a mug of an unfamiliar color, even though the correct grasping behavior remains essentially unchanged.

A more difficult form of distribution shift occurs when

$$p_{\text{train}}(y|x) \neq p_{\text{test}}(y|x).$$

This can occur if system dynamics or task semantics change. For instance, the same commanded gripper motion may have different consequences if the robot carries a heavier payload or if actuator dynamics differ from those present during training.

2.8.2 In-Distribution and Out-of-Distribution Evaluation

An input is generally described as *in-distribution* if it is reasonably consistent with the support of the training distribution. An *out-of-distribution* input lies outside, or in a sparsely represented region of, that distribution.

If

$$x \sim p_{\text{train}}(x),$$

then x is nominally in-distribution. If instead

$$x \sim p_{\text{OOD}}(x),$$

where p_{OOD} differs significantly from the training distribution, the example is considered out-of-distribution.

Out-of-distribution evaluation should not be interpreted as a single test. There are many possible forms of OOD shift, and a system may generalize well to some while failing catastrophically on others.

For a robotic manipulation system, possible OOD dimensions include

- novel object instances,
- novel object categories,
- unfamiliar textures or colors,

- different illumination,
- changes in camera viewpoint,
- novel backgrounds or clutter,
- unseen object poses,
- unfamiliar language instructions,
- changes in payload or friction,
- sensor corruption,
- actuator delay,
- changes in robot embodiment,
- combinations of several simultaneous shifts.

It is therefore useful to define explicit evaluation distributions. For example,

$$p_{\text{ID}}$$

may contain familiar object categories and environments, while

$$p_{\text{OOD,obj}}$$

contains unseen object instances,

$$p_{\text{OOD,view}}$$

contains novel camera viewpoints, and

$$p_{\text{OOD,dyn}}$$

contains altered system dynamics.

This decomposition provides considerably more information than reporting a single 'OOD accuracy" or 'OOD success rate."

2.8.3 Generalization Across Increasing Distribution Shift

One useful evaluation methodology is to construct several levels of distribution shift. For example,

$$D_0 = \text{nominal test distribution},$$

$$D_1 = \text{moderate visual variation},$$

$$D_2 = \text{unseen object configurations},$$

$$D_3 = \text{combined visual and dynamics shift}.$$

The policy can then be evaluated using

$$S(D_i),$$

where S denotes task success under distribution D_i.

The degradation

$$\Delta_i = S(D_0) - S(D_i)$$

measures sensitivity to a particular degree of distribution shift.

A model with slightly lower nominal performance but smaller degradation under shift may be preferable to a model that achieves near-perfect IID performance but fails sharply outside the training distribution.

2.8.4 Task Success Metrics

For many robotic tasks, the most interpretable metric is the *task success rate*. Given N evaluation episodes, define

$$s_i = \begin{cases} 1, & \text{if episode } i \text{ succeeds}, \\ 0, & \text{otherwise}. \end{cases}$$

The empirical success rate is

$$\hat{S} = \frac{1}{N} \sum_{i=1}^{N} s_i.$$

If 83 out of 100 manipulation attempts succeed, then

$$\hat{S} = 0.83.$$

Success criteria should be defined operationally before evaluation. For example, "pick up the object" may require that the object remain above the table for at least a specified duration rather than merely making temporary contact with the gripper.

Binary task success is often insufficient by itself. Additional metrics can reveal how and why the system succeeds or fails. Depending on the task, useful quantities include

- completion time,

- number of control steps,
- path length,
- final pose error,
- grasp stability,
- collision count,
- peak contact force,
- energy consumption,
- number of replanning events,
- number of human interventions.

For a target pose $T^\star$ and achieved pose T, translational error may be measured as

$$e_p = \|p - p^\star\|_2.$$

Rotational error may be measured using the relative rotation

$$R_{\text{err}} = R^{\star T} R$$

and the corresponding angle

$$e_R = \cos^{-1}\left(\frac{\text{tr}(R_{\text{err}}) - 1}{2}\right).$$

These continuous metrics provide useful information even when the final task result is represented by a binary success variable.

2.8.5 Conditional Success Rates

An overall success rate can hide substantial differences across task categories. Suppose a policy is evaluated on several task groups c. The conditional success rate is

$$S_c = P(\text{success} \mid c).$$

For example,

$$S_{\text{pick}}, \qquad S_{\text{place}}, \qquad S_{\text{drawer}}, \qquad S_{\text{pour}}$$

may differ considerably.

The same decomposition can be performed over object categories, environments, robot configurations, or OOD conditions. A model whose average success is high may still have severe blind spots if one subgroup has significantly lower performance.

A useful aggregate metric is the worst-group success rate,

$$S_{\text{worst}} = \min_c S_c.$$

This quantity provides a simple measure of whether performance is concentrated in easy or overrepresented subsets of the evaluation distribution.

2.8.6 Robustness

Robustness refers to the degree to which performance is preserved under perturbations, uncertainty, noise, or distribution shift.

Let

$$f_\theta(x)$$

denote the learned policy and let

$$x' = x + \delta$$

represent a perturbed input. A robust policy should ideally produce behavior that remains appropriate for perturbations δ within a relevant set of disturbances.

In robotics, robustness should be evaluated at the system level rather than only through small perturbations of neural network inputs. Relevant perturbations include sensor noise, object displacement, latency, external forces, inaccurate calibration, and imperfect actuation.

For a perturbation magnitude ϵ, define an evaluation distribution

$$p_\epsilon(x)$$

containing environments with perturbations of approximately that magnitude. The resulting robustness curve can be expressed as

$$S(\epsilon) = P(\text{success} \mid \epsilon).$$

A robust system exhibits graceful degradation:

$$S(\epsilon)$$

decreases gradually as perturbations become more severe, rather than collapsing immediately after departing from nominal conditions.

A useful summary metric is the relative robustness

$$R(\epsilon) = \frac{S(\epsilon)}{S(0)}.$$

A value

$$R(\epsilon) \approx 1$$

indicates little performance degradation under the specified perturbation.

2.8.7 Perturbation and Recovery Tests

Closed-loop robotic systems should also be evaluated on their ability to recover from unexpected disturbances.

Consider a manipulation trajectory

$$\tau = (o_0, a_0, \ldots, o_T, a_T).$$

At some time t_p, an external perturbation may be introduced, such as moving the target object, perturbing the robot arm, or temporarily occluding the camera.

The evaluation then measures whether the policy can return to a successful trajectory without external assistance.

Possible recovery metrics include

$$P(\text{recovery} \mid \text{perturbation}),$$

time to recovery,

$$T_{\text{recover}},$$

and additional trajectory cost caused by the disturbance,

$$\Delta C = C_{\text{perturbed}} - C_{\text{nominal}}.$$

Recovery tests are especially valuable because a policy may perform well when initialized near expert trajectories but fail to return to the task manifold after small deviations.

2.8.8 Intervention Rate

For systems deployed with human supervision, task success alone may substantially overestimate autonomy. A robot may complete nearly every episode but require frequent human correction.

Let I_i denote the number of human interventions during episode i. A simple intervention rate per episode is

$$R_{\text{int}} = \frac{1}{N} \sum_{i=1}^{N} I_i.$$

Alternatively, if total operating time is T,

$$R_{\text{int,time}} = \frac{N_{\text{interventions}}}{T}.$$

For continuously deployed systems, one may report interventions per hour,

$$R_{\text{int,hour}} = \frac{N_{\text{interventions}}}{T_{\text{hours}}}.$$

Another useful quantity is the fraction of episodes requiring at least one intervention:

$$P_{\text{int}} = \frac{\sum_i \mathbb{I}[I_i > 0]}{N}.$$

Intervention rate is particularly important when comparing partially autonomous systems. Two policies may both achieve

$$95\%$$

task completion, but one may require a human correction every few minutes while the other runs autonomously for several hours.

2.8.9 Intervention Severity

Not all interventions are equivalent. A useful evaluation scheme may distinguish between

- minor corrections,
- trajectory resets,
- task restarts,
- safety-critical takeovers.

A weighted intervention cost can be defined as

$$C_{\text{int}} = \sum_{j=1}^{M} w_j I_j,$$

where I_j denotes the number of interventions of type j and w_j represents their severity.

For example, a brief waypoint correction may have a small cost, while an emergency stop may have a much larger cost.

This provides a better measure of operational autonomy than treating every intervention identically.

2.8.10 Success Without Intervention

For real autonomous systems, an especially informative metric is the probability of completing a task without human assistance:

$$S_{\text{autonomous}} = P(\text{success} \cap \text{no intervention}).$$

This differs from ordinary task success,

$$S = P(\text{success}),$$

because an episode recovered by human takeover may count as successful operationally but should not be counted as autonomous success.

The distinction becomes increasingly important for long-horizon tasks, where human assistance can conceal substantial weaknesses in the learned policy.

2.8.11 Long-Horizon Reliability

Per-action or per-subtask success rates can be misleading when tasks contain many sequential stages.

Suppose a system must successfully complete K independent stages, each with success probability

$$p.$$

The complete task success probability is

$$P_{\text{task}} = p^K.$$

For example, if

$$p = 0.98$$

and

$$K = 50,$$

then

$$P_{\text{task}} = 0.98^{50} \approx 0.364.$$

Thus, seemingly strong local reliability may still produce poor long-horizon autonomy.

For this reason, embodied systems should be evaluated not only on individual skills but also on complete multi-stage tasks and extended deployment periods.

A related operational metric is mean time between interventions,

$$\mathrm{MTBI} = \frac{T_{\mathrm{operation}}}{N_{\mathrm{interventions}}}.$$

Larger values indicate longer periods of sustained autonomous operation.

2.8.12 Robust Evaluation Protocols

A strong evaluation protocol should separate several distinct questions.

First, *in-distribution generalization* asks whether the model performs well on unseen examples sampled from conditions similar to training.

Second, *systematic OOD evaluation* asks how performance changes when one or more factors differ from the training distribution.

Third, *robustness evaluation* introduces controlled disturbances or noise and measures graceful degradation and recovery.

Fourth, *autonomy evaluation* measures how frequently external intervention is required during extended operation.

An evaluation suite might therefore report

$$S_{\mathrm{ID}},$$

$$S_{\mathrm{OOD,obj}},$$

$$S_{\mathrm{OOD,env}},$$

$$S_{\mathrm{OOD,dyn}},$$

$$S_{\mathrm{perturbed}},$$

along with

$$R_{\mathrm{int}}, \qquad S_{\mathrm{autonomous}}, \qquad \mathrm{MTBI}.$$

Together, these quantities provide a substantially more complete characterization of system behavior than a single average success rate.

2.8.13 Statistical Uncertainty

Reported evaluation metrics should also include uncertainty. If N independent episodes are evaluated and $\hat{S}$ is the observed success rate, an approximate standard error is

$$\mathrm{SE}(\hat{S}) = \sqrt{\frac{\hat{S}(1-\hat{S})}{N}}.$$

A model evaluated on only a small number of episodes may therefore have a highly uncertain success estimate.

For example, observing nine successes out of ten trials,

$$\hat{S} = 0.9,$$

does not provide the same statistical evidence as observing 900 successes out of 1000 trials, even though the reported success rate is identical.

This consideration is particularly important in robotics, where physical experiments are expensive and evaluation sets are often small.

2.8.14 Evaluation as Distribution Design

The central principle is that evaluation is fundamentally a problem of specifying distributions.

Training occurs under some distribution

$$p_{\text{train}},$$

while the system is ultimately required to perform under a deployment distribution

$$p_{\text{deploy}}.$$

An evaluation suite defines one or more intermediate distributions

$$p_{\text{eval}}^{(1)}, p_{\text{eval}}^{(2)}, \ldots, p_{\text{eval}}^{(K)}$$

designed to probe relevant aspects of this gap.

A useful evaluation should therefore answer not only

"What is the average success rate?"

but also

"Under which distributions does the policy succeed?"

"How rapidly does performance degrade as conditions change?"

and

"How often must an external agent intervene when the policy encounters unfamiliar states?"

For embodied systems, these questions are often more informative than nominal test-set performance because deployment necessarily exposes the policy to conditions that cannot be represented exhaustively in the training data.