Hands-On Control and Algorithms for Robots

Michal Chovanec, PhD.

Chapter 1

Robot Modelling

The first step in controlling a robot is to understand its mathematical model. Every movement of a robot—at every tiny time step—is governed by its equations of motion. As a simple yet powerful example, let us consider a **two-wheel differential drive robot**.

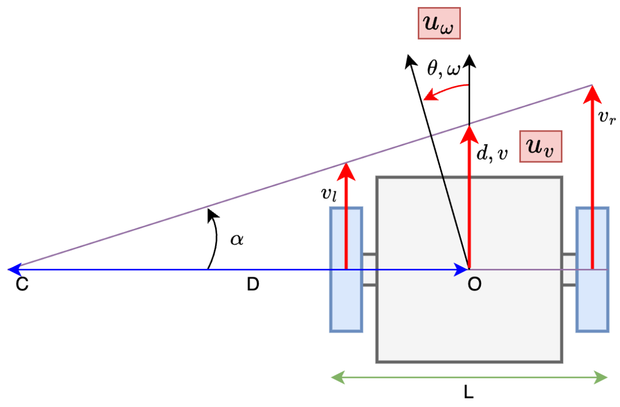


Figure 1.1: Differential drive robot geometry and notation.

Figure 1.1 shows a schematic of such a robot. The center of the robot is denoted by O, and the distance between the two driving wheels is L [m]. Each wheel produces a linear velocity, denoted v_l and v_r [m/s] for the left and right wheel respectively. If the wheels are rotating with angular velocities ω_l and ω_r [rad/s], then the corresponding linear velocities can be expressed as

$$v_l = \omega_l r, \quad v_r = \omega_r r,$$

where r [m] is the wheel radius.

In practical robotics, motor speed is often measured in revolutions per minute (rpm). The relationship between angular velocity in radians per second and rpm is given by

$$\omega = rpm \cdot \frac{2\pi}{60}.$$

Knowing the individual wheel velocities, we can express the **robot's linear velocity** v as the average of both wheels:

$$v = \frac{1}{2}(v_l + v_r).$$

The corresponding **linear displacement** d [m] evolves over time as

$$\frac{dd}{dt} = v.$$

Similarly, the **robot's angular rate** ω [rad/s] and **orientation angle** θ [rad] are related by

$$\omega = \frac{v_r - v_l}{L}, \quad \frac{d\theta}{dt} = \omega.$$

These two variables — linear and angular velocities — fully describe the planar motion of a differential drive robot.

Control Inputs

In practice, the control inputs to the robot are the individual motor commands, denoted u_{left} and u_{right}. However, it is often more convenient to work with combined control variables: the **linear velocity command** u_v and the **angular velocity command** u_ω, defined as

$$\begin{aligned} u_{\text{left}} &= u_v - u_\omega, \\ u_{\text{right}} &= u_v + u_\omega. \end{aligned} \tag{1.1}$$

This formulation allows independent control of forward motion and rotation, which greatly simplifies controller design and enables linear control strategies.

Instantaneous Center of Rotation (ICR)

In the diagram, point C represents the **instantaneous center of rotation (ICR)**. This point lies on the axis connecting the two wheels, and its distance from the robot's center O depends on the ratio between left and right wheel velocities.

The distance D from O to the ICR is given by

$$D = \frac{L}{2} \cdot \frac{v_l + v_r}{v_r - v_l}.$$

When both wheels move at the same speed ($v_l = v_r$), the denominator becomes zero, and thus $D \to \infty$ — which corresponds to straight-line motion.

Motion Formalism

The **state-space model** is commonly used to describe how a robot's trajectory evolves over time. A physical system can be represented by a continuous-time state-space model, where the state is expressed as $x(t)$. However, for our purposes — particularly for controller implementation in digital systems — a **discrete-time model** is more convenient, denoted as $x(n)$.

For a very simple case, such as a slowly moving robot with negligible inertia, the state can be represented using only two quantities: the **linear displacement** d [m] and the **orientation angle** θ [rad]:

$$x(n) = \begin{bmatrix} d(n) \\ \theta(n) \end{bmatrix}. \tag{1.2}$$

However, a real robot — especially one that moves quickly or has non-negligible inertia — cannot be accurately modeled in this minimal form. When the motors are turned off, the robot does not stop instantaneously due to its momentum. To capture this physical behavior, we extend the state vector by including the **robot's linear velocity** v [m/s] and the **robot's angular rate** ω [rad/s], leading to the following definition:

$$x(n) = \begin{bmatrix} d(n) \\ \theta(n) \\ v(n) \\ \omega(n) \end{bmatrix}. \tag{1.3}$$

In a similar way, we define the **control input vector** $u(n)$ as:

$$u(n) = \begin{bmatrix} u_v(n) \\ u_\omega(n) \end{bmatrix}, \tag{1.4}$$

where u_v and u_ω represent the commanded linear and angular control inputs, respectively. In this book, we assume that

$$u \text{ is unit-less, in the range } \langle -1, 1 \rangle, \tag{1.5}$$

meaning that these commands are normalized. They can later be mapped to real-world actuator quantities — for example, motor RPM, PWM duty cycle, or desired wheel velocity — depending on the specific hardware configuration.

Let us now describe the relationship between the robot's state x and control input u, starting with a simplified (inertia-free) model that does not include velocity terms:

$$\begin{aligned} d(n+1) &= d(n) + b_0\, u_v(n), \\ \theta(n+1) &= \theta(n) + b_1\, u_\omega(n). \end{aligned} \tag{1.6}$$

These two equations are completely independent: the first describes how the robot's position changes, while the second describes how its orientation evolves. The constant b_0 expresses how strongly the linear control u_v affects forward motion — it depends on motor speed, wheel diameter, and control step length. Similarly, b_1 expresses how efficiently the robot turns — it depends on the same motor characteristics and the wheelbase L.

Higher motor speed or larger wheel radius increases b_0, while a **larger wheelbase** decreases b_1 because the robot must rotate more slowly for the same wheel speed difference.

Practical estimation of b_0 and b_1:

If we know the maximum wheel speed at full control input $u = 1$, denoted $RPM_{\max}$, and the wheel radius r [m], along with the control sampling period ΔT [s], we can estimate the corresponding maximum linear and angular velocities as:

$$\begin{aligned} v_{\max} &= 2\pi \frac{RPM_{\max}\, r}{60}, \\ \omega_{\max} &= 4\pi \frac{RPM_{\max}\, r}{60L}. \end{aligned} \tag{1.7}$$

From these relations, we can approximate the discrete-time model parameters as:

$$\begin{aligned} b_0 &= v_{\max}\, \Delta T, \\ b_1 &= \omega_{\max}\, \Delta T. \end{aligned} \tag{1.8}$$

This allows us to estimate a simple, inertia-free model directly from basic geometric parameters and motor specifications. Although this model is only approximate, it works well for low-speed motion or when a rough dynamic approximation is sufficient.

Matrix form of the state-space model:

In conventional control theory, the discrete-time state-space model is expressed compactly as:

$$x(n+1) = Ax(n) + Bu(n). \tag{1.9}$$

Rewriting the simplified dynamics from Eq. 1.6 in this formalism yields:

$$\begin{bmatrix} d(n+1) \\ \theta(n+1) \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}}_{A} \begin{bmatrix} d(n) \\ \theta(n) \end{bmatrix} + \underbrace{\begin{bmatrix} b_0 & 0 \\ 0 & b_1 \end{bmatrix}}_{B} \begin{bmatrix} u_v(n) \\ u_\omega(n) \end{bmatrix}. \tag{1.10}$$

Commentary:

This simple kinematic model forms the foundation of robot motion control. Although it ignores acceleration dynamics, wheel slip, and mechanical delays, it is extremely useful for initial controller design, algorithm prototyping, and understanding how inputs affect motion. In later chapters, we will extend this formulation to include velocity dynamics, actuator limitations, and state estimation, which together lead to more realistic and robust control architectures. The more details how state space models works can be found in chapter the **Capturing dynamics**.

Creating a Simulator

We start by implementing a robot class that handles the robot dynamics. The implementation is intentionally simple, using the **NumPy** library for matrix operations. This choice allows us to later describe more complex dynamics (e.g., with inertia or slip effects) without changing the underlying mathematical framework.

The constructor of the **class DifferentialDriveModel** is defined as:

```
def __init__(self, b0, b1, width, height)
```

This constructor receives the parameters describing robot dynamics (b_0, b_1) and the robot's physical dimensions (`width`, `height`), useful for visualization purposes.

By calling:

```
def reset(self, initial_x, initial_y, initial_theta)
```

we can reset the robot's state vector and bring it to the desired initial position and orientation.

The main function, which is called repeatedly in the simulation loop, is:

```
def step(self, u)
```

It accepts a control input vector u and returns the current state dictionary:

```
self.robot_state = {}
self.robot_state["width"]  = self.width
self.robot_state["height"] = self.height

self.robot_state["x"]     = self.x
self.robot_state["x_pos"] = self.x_pos
self.robot_state["y_pos"] = self.y_pos
self.robot_state["theta"] = theta
```

—

The remaining question is: **How do we convert the internal robot state** (d, θ) **into Cartesian coordinates** (x_{pos}, y_{pos})**?**

The robot's state space (d, θ) is convenient for control and dynamics modeling, but it does not directly tell us where the robot is located in the plane. We therefore need a geometric transformation.

At each discrete time step, the robot travels a distance increment:

$$\Delta d(n) = d(n+1) - d(n),$$

and its current heading (orientation in the global frame) is $\theta(n)$. Thus, the displacement in Cartesian coordinates can be computed as:

$$\begin{aligned} \Delta x(n) &= \Delta d(n) \cos(\theta(n)), \\ \Delta y(n) &= \Delta d(n) \sin(\theta(n)). \end{aligned}$$

Why this works: The robot's forward motion always occurs along its heading direction θ. By decomposing the traveled distance Δd into its projections on the global x and y axes (using the cosine and sine of θ), we obtain the actual movement in Cartesian space. This is a standard transformation between the robot's local frame and the world frame.

Integrating (in discrete simulation simply summing) these increments gives the full global position:

$$\begin{aligned} \Delta d(n) &= d(n+1) - d(n), \\ \Delta x(n) &= \Delta d(n) \cos(\theta(n)), \\ \Delta y(n) &= \Delta d(n) \sin(\theta(n)), \\ x_{pos}(n+1) &= x_{pos}(n) + \Delta x(n), \\ y_{pos}(n+1) &= y_{pos}(n) + \Delta y(n). \end{aligned} \tag{1.11}$$

Here, d and θ are taken from the robot state vector x. This approach lets us keep a simple linear state-space model for control purposes, while still maintaining accurate global position information for visualization, mapping, and collision detection.

The complete Python implementation of the robot model is shown below:

```python
import numpy

class DifferentialDriveModel:
    def __init__(self, b0, b1, width, height):
        self.A = numpy.array([[1, 0],
                              [0, 1]])
        self.B = numpy.array([[b0, 0],
                              [0, b1]])
        self.width  = width
        self.height = height

        self.reset(0, 0, 0)

    def reset(self, initial_x, initial_y, initial_theta):
        # Initialize state [d, theta]
        self.x = numpy.zeros((2, 1))
        self.x[1, 0] = initial_theta
        self.x_pos = initial_x
        self.y_pos = initial_y

        self.step(numpy.zeros((2, 1)))

    def step(self, u):
        # Discrete-time dynamics update
        x_new = self.A @ self.x + self.B @ u

        # Calculate position increment in Cartesian coordinates
        dpos  = x_new[0, 0] - self.x[0, 0]
        theta = self.x[1, 0]

        dx = dpos * numpy.cos(theta)
        dy = dpos * numpy.sin(theta)

        # Update global position
        self.x_pos += dx
        self.y_pos += dy

        # Update internal state
        self.x = numpy.array(x_new)

        # Fill result values
        self.robot_state = {
            "width":  self.width,
            "height": self.height,
            "x":      self.x,
            "x_pos":  self.x_pos,
            "y_pos":  self.y_pos,
            "theta":  theta
        }

        return self.robot_state

    def get_state(self):
        return self.robot_state
```

—

Every model must be tested. It is always good practice to choose **non-trivial parameters** to avoid errors that are hidden by “nice” numbers. Here we define robot dynamics, the simulation time step, motor parameters, and robot dimensions.

From the equations for maximum velocities (Eq. 1.7) and the parameter estimation (Eq. 1.8), we compute b_0 and b_1 and pass them into the robot model:

```
# Input parameters
dt       = 0.001          # time step [s]
r_wheel = 15.0 * 0.001   # wheel radius [m]
l_wheels= 95.0 * 0.001   # wheel separation [m]
rpm_max = 750            # maximum motor speed [RPM]

width    = 2 * l_wheels
height   = l_wheels

v_max = 2.0 * numpy.pi * rpm_max * r_wheel / 60.0
w_max = 4.0 * numpy.pi * rpm_max * r_wheel / (60.0 * l_wheels)

b0 = v_max * dt
b1 = w_max * dt

# Initialize robot
robot = DifferentialDriveModel(b0, b1, width, height)
robot.reset(0, 0, 0)
```

—

Now, let the robot follow an **equilateral triangular trajectory**. This is a better test than a square path because 90-degree turns are a special case — they only affect one Cartesian coordinate at a time, while 120-degree turns give a more general motion pattern.

We define two possible control inputs:

- Full-speed forward motion: $u = (1, 0)$
- In-place rotation: $u = (0, 1)$

To move along the triangle sides, we must compute how many simulation steps correspond to a 1-meter straight motion and a 120° turn (converted to radians):

```
n_steps_forward = round(1.0 / (v_max * dt))
angle = 120.0 * numpy.pi / 180.0
n_steps_turn = round(angle / (w_max * dt))
```

We now execute three segments of motion (forward 1 m, turn 120°) to complete the triangle, recording the robot's positions for later plotting:

```
x_pos = []
y_pos = []

for j in range(3):
    # move robot for 1m distance
    for n in range(n_steps_forward):
        u = numpy.array([[1.0],[0.0]])

        state = robot.step(u)

        x_pos.append(state["x_pos"])
        y_pos.append(state["y_pos"])

    # turn robot at 120 degrees
    for n in range(n_steps_turn):
        u = numpy.array([[0.0],[1.0]])

        state = robot.step(u)

        x_pos.append(state["x_pos"])
        y_pos.append(state["y_pos"])

print("terminal state")
print("x_pos = ", round(state["x_pos"], 4))
print("y_pos = ", round(state["y_pos"], 4))
print("theta = ", round(state["theta"]*180.0/numpy.pi, 4))
```

At the end, the robot should return to approximately the same starting point, with its orientation close to zero (modulo 360°). In the test output, we observe:

```
terminal state
x_pos =  -0.0094
y_pos =   0.0167
theta =  356.6842
```

Explanation: This small offset from the starting position is caused by **numerical integration error** due to discretization. Because the simulation uses a finite time step $\Delta T = 1$ ms, the continuous motion is approximated by small discrete jumps. Such integration errors accumulate over time and cause the trajectory to drift slightly, especially in long simulations or when using small angular increments.

The resulting trajectory plot is shown in Figure 1.2.

If we visualize the trajectory history (e.g., using OpenCV), we can see how the position gradually drifts — a natural effect of discrete simulation and rounding errors, as illustrated in Figure 1.3. The complete implementation including visualization is available at: `examples/robot_simulator_01`.

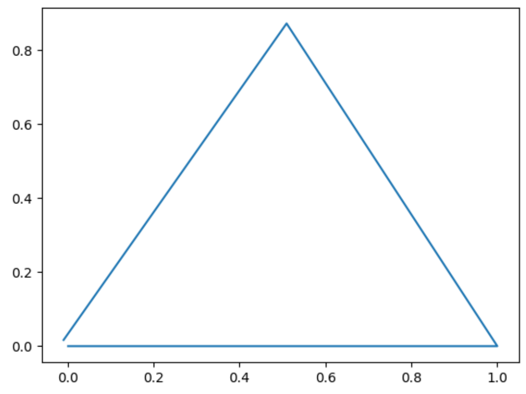


Figure 1.2: Triangular trajectory plot

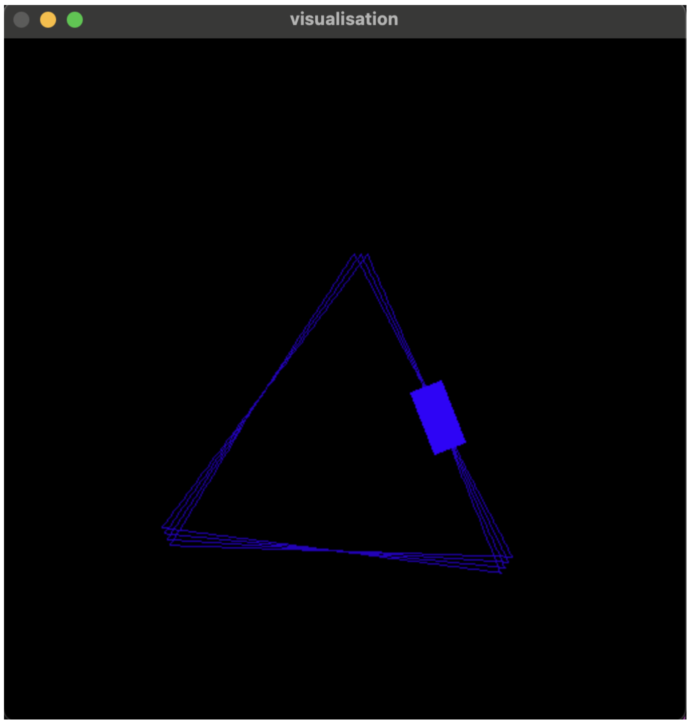


Figure 1.3: Simulator visualization

Special cases of motion control

It is useful to analyze how the robot behaves under specific control inputs. By assigning different values to the control vector $u = [u_v, u_\omega]^T$, we can immediately observe fundamental motion primitives that compose any possible trajectory:

- **Straight-line motion:** $u = (1, 0)$
  Both wheels receive identical input, hence they rotate at the same speed. There is no angular component ($u_\omega = 0$), so the robot's heading θ remains constant. As a result, the robot moves in a straight line in the direction it currently faces.

- **In-place rotation:** $u = (0, 1)$
  In this case, both wheels rotate in opposite directions with equal magnitude. This creates a pure rotational motion around the robot's center point, without any translation. The linear velocity component is zero ($u_v = 0$), and the heading θ changes at a constant rate.

- **Circular motion:** $u = (1, 0.1)$
  Here, both translational and rotational components are active. The robot moves forward while simultaneously turning with a constant angular rate. The resulting trajectory is a perfect circle, where the radius of the circle is given by:

  $$R = \frac{v}{\omega} = \frac{u_v \, v_{\max}}{u_\omega \, \omega_{\max}}.$$

  The larger the ratio u_v/u_ω, the wider the circle. When u_ω is small compared to u_v, the robot moves almost straight; when u_ω is large, it spins tightly around a small radius.

This elegant behavior results from the **decoupled control formulation** in which the input vector u directly represents the *linear* and *angular* control actions. Because the left and right motor signals are internally computed as:

$$\begin{aligned} u_{\text{left}} &= u_v - u_\omega, \\ u_{\text{right}} &= u_v + u_\omega, \end{aligned}$$

the two control modes — translation and rotation — are cleanly separated. This makes both mathematical modeling and controller design straightforward, since any arbitrary trajectory can be expressed as a combination of linear motion and circular motion segments.

In fact, **the entire movement of a differential-drive robot is a continuous composition of these two basic primitives** — straight motion and circular arcs. This geometric interpretation is extremely valuable in higher-level path planning, as it allows complex paths to be represented using simple building blocks such as arcs and lines.

An example of such combined motion is shown in Figure 1.4, where the robot follows a smooth circular trajectory as a result of constant-speed turning.

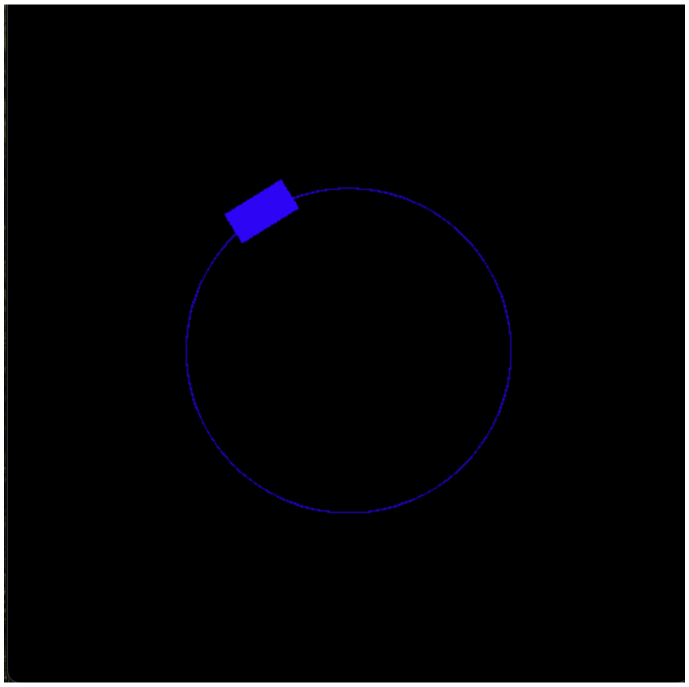


Figure 1.4: Simulator visualization of circular motion

Summary: In this section, we created a simple but complete simulation environment for a differential-drive robot. The reader learned how to:

- Represent the robot dynamics in discrete state-space form.
- Transform robot motion from state variables (d, θ) into global Cartesian coordinates (x, y).
- Implement and simulate the model in Python using NumPy.
- Test the model on a non-trivial trajectory and interpret integration errors.

Although the model ignores inertia and wheel slip, it provides a solid base for more advanced topics such as velocity control, model predictive control (MPC), and extended state estimation in later chapters.

Chapter 2

Capturing dynamics

2.1 State space models

A convenient mathematical framework for describing the dynamics of a system is the **state space model**. Such a model captures how the system's internal state evolves over time in response to inputs, and how these inputs affect the outputs.

In other words, the state space model represents the relationship between:

- the **state vector** $x(n)$ (which describes the internal condition of the system),
- the **input vector** $u(n)$ (the control or excitation applied to the system),
- and optionally, the **output vector** $y(n)$ (the measured quantities).

System State

The system state is represented as a column vector with N elements:

$$x(n) = \begin{bmatrix} x_0(n) \\ x_1(n) \\ \vdots \\ x_{N-1}(n) \end{bmatrix} \tag{2.1}$$

The number of rows in this vector, N, is called the **system order**.

Examples of System States

Single-State Example: For a simple first-order system, such as a motor where we only consider its angular velocity $\omega(n)$, the state vector contains a single element:

$$x(n) = \begin{bmatrix}\omega(n)\end{bmatrix} \tag{2.2}$$

Second-Order Example: If we model a servo motor with rotational inertia, the state vector might include both the motor's velocity and position:

- Motor shaft velocity $\omega(n)$ [rad/s]
- Motor shaft angle $\theta(n)$ [rad]

$$x(n) = \begin{bmatrix}\omega(n) \\ \theta(n)\end{bmatrix} \tag{2.3}$$

Extended Example: A more accurate servo model might also include the motor current $i(n)$:

$$x(n) = \begin{bmatrix}i(n) \\ \omega(n) \\ \theta(n)\end{bmatrix} \tag{2.4}$$

Mobile Robot Example: For a robot moving in a 2D plane, the state could describe its position and orientation:

$$x(n) = \begin{bmatrix}x'(n) \\ y'(n) \\ \theta(n)\end{bmatrix} \tag{2.5}$$

System Inputs

The system is controlled by M inputs, grouped into the **input vector** $u(n)$:

$$u(n) = \begin{bmatrix}u_0(n) \\ u_1(n) \\ \vdots \\ u_{M-1}(n)\end{bmatrix} \tag{2.6}$$

For a single-input system, such as a current-controlled motor:

$$u(n) = \begin{bmatrix} i(n) \end{bmatrix} \tag{2.7}$$

A differential-drive robot with two independently driven wheels would have two inputs:

$$u(n) = \begin{bmatrix} i_{\text{left}}(n) \\ i_{\text{right}}(n) \end{bmatrix} \tag{2.8}$$

Discrete-Time State Space Model

The relationship between the current state $x(n)$, the control input $u(n)$, and the next state $x(n+1)$ can be expressed as a **linear discrete-time state space model**:

$$x(n+1) = A\,x(n) + B\,u(n) \tag{2.9}$$

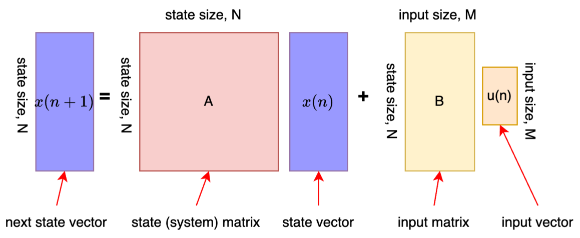


Figure 2.1: Matrix dimensions in a discrete-time state space model.

This form can describe the dynamics of any **linear system**. In some cases, the full state $x(n)$ cannot be measured directly. Instead, we observe only the system output:

$$y(n) = C\,x(n) \tag{2.10}$$

where matrix C has K rows and N columns, mapping the internal state to observable outputs. In this section, we assume full-state measurements are available.

Continuous-Time Model and Discretization

Physical systems are often described by **continuous-time state space equations**:

$$\frac{dx(t)}{dt} = \bar{A}\,x(t) + \bar{B}\,u(t) \tag{2.11}$$

The matrices $\bar{A}$ and $\bar{B}$ differ from their discrete-time counterparts A and B. They can be related through a **bilinear (Tustin) transformation**:

$$S_a = \left(I - \frac{\Delta T}{2}\bar{A}\right)^{-1} \tag{2.12}$$

$$S_b = I + \frac{\Delta T}{2}\bar{A} \tag{2.13}$$

$$A = S_a\,S_b \tag{2.14}$$

$$B = S_a\,\bar{B}\,\Delta T \tag{2.15}$$

Here, ΔT is the sampling period. This method converts a continuous (physical) model into a discrete form suitable for digital control.

The following Python code implements this discretization process:

```python
def c2d(a, b, c, dt):
    i = numpy.eye(a.shape[0])

    tmp_a = numpy.linalg.inv(i - (0.5*dt)*a)
    tmp_b = i + (0.5*dt)*a

    a_disc  = tmp_a@tmp_b
    b_disc  = (tmp_a*dt)@b

    return a_disc, b_disc, c
```

2.2 System Examples

First-Order Dynamical System

In this section, we demonstrate a simple **first-order dynamical system**. Such systems appear frequently in both mechanical and electrical domains. Common real-world examples include:

- the angular velocity dynamics of a DC motor,

- the voltage response of an RC circuit,
- the thermal response of a heating element,
- or the water level in a tank with a constant inflow and proportional outflow.

System Definition

A first-order system can be described by two key parameters:

- the **time constant** τ, which determines how quickly the system responds (its settling time),
- the **gain** k, which defines how strongly the input u affects the output (or state) x.

In this case, both the input u and the state x are single scalar quantities. The continuous-time model is given by:

$$\frac{dx(t)}{dt} = -\frac{1}{\tau}x(t) + \frac{k}{\tau}u(t) \tag{2.16}$$

Here, the system matrices have dimensions 1×1:

$$\bar{A} = \left[-\tfrac{1}{\tau}\right], \qquad \bar{B} = \left[\tfrac{k}{\tau}\right] \tag{2.17}$$

Simulation Setup

To visualize how this system behaves, we can simulate its response to a **unit step input**, where:

$$u(t) = 1 \tag{2.18}$$

We simulate for 1000 steps using a discretization step of $\Delta t = 0.01$ seconds, which corresponds to a total simulation time of 10 seconds. The initial condition is set to $x(0) = 0$.

A simple (naive) implementation in Python looks like this:

```python
x = numpy.zeros((1, 1))   # initial state
u = numpy.ones((1, 1))    # constant input = 1

for n in range(num_steps):
    dx = A @ x + B @ u    # update step
    x  = x + dx * dt      # integration
```

Numerical Integration Methods

The update step $x = x + dx \cdot dt$ performs **Euler integration**, also known as the **forward Euler method**. This approach is conceptually simple but can accumulate significant numerical error, especially for stiff or higher-order systems. To achieve stable and accurate results, Δt must be very small — which increases the number of simulation steps dramatically.

More advanced numerical solvers, such as the **Runge–Kutta methods** (commonly RK4), provide much better stability and accuracy for the same time step size. These solvers are standard in modern simulation environments and control design tools.

If we define **system forward func** as :

```python
def forward_func(x, u):
    dx = A@x + B@u
    y  = C@x

    return dx, y
```

The ODE solvers can be implemented straightforward. The naive Euler method is :

```python
def ODESolverEuler(forward_func, x, u, dt):
    dx, y  = forward_func(x, u)
    return x + dx*dt, y
```

And commonly more used (and prefered for ballance between accuracy and speed), Runge Kutta RK4 method is :

```python
def ODESolverRK4(forward_func, x, u, dt):
    k1, y1  = forward_func(x, u)
    k1      = k1*dt

    k2, y2  = forward_func(x + 0.5*k1, u + 0.5*dt)
    k2      = k2*dt

    k3, y3  = forward_func(x + 0.5*k2, u + 0.5*dt)
    k3      = k3*dt

    k4, y4  = forward_func(x + k3, u + dt)
    k4      = k4*dt

    x_new   = x + (1.0/6.0)*(1.0*k1 + 2.0*k2 + 2.0*k3 + 1.0*k4)
    y       = (1.0/6.0)*(1.0*y1 + 2.0*y2 + 2.0*y3 + 1.0*y4)

    return x_new, y
```

The entire simulation is then single for loop, calling one step of dynamical system inside solver. Every loop iteration is one small discrete simulation step.

```python

def simulate(n, m, dt, n_steps):
    # input and
    u = numpy.zeros((m, 1))
    x = numpy.zeros((n, 1))

    # log results
    t_result = []
    u_result = []
    x_result = []
    y_result = []

    for n in range(n_steps):
        # here the input into system is obtained
        # e.g. from controller, path planner, user input
        u = obtain_controll()

        # solver step
        x, y = ODESolverRK4(forward_func, x, u, dt)

        # optional visualisation
        visualise(n, u, x, y)

        # log results
        t_result.append(n*dt)
        u_result.append(u[:, 0])
        x_result.append(x[:, 0])
        y_result.append(y[:, 0])

    t_result = numpy.array(t_result)
    u_result = numpy.array(u_result)
    x_result = numpy.array(x_result)
    y_result = numpy.array(y_result)

    return t_result, u_result, x_result, y_result
```

Response Analysis

Figure 2.2 shows several step responses for different values of the time constant τ and gain k.

From these plots, we can observe that:

- Increasing τ makes the system respond more slowly — it takes longer to reach the steady state. This behavior can be interpreted as the system having more inertia or lag.
- Increasing k scales the steady-state value of $x(t)$. For a step input of $u = 1$, the steady-state output is $x_{\text{ss}} = k \cdot u = k$.

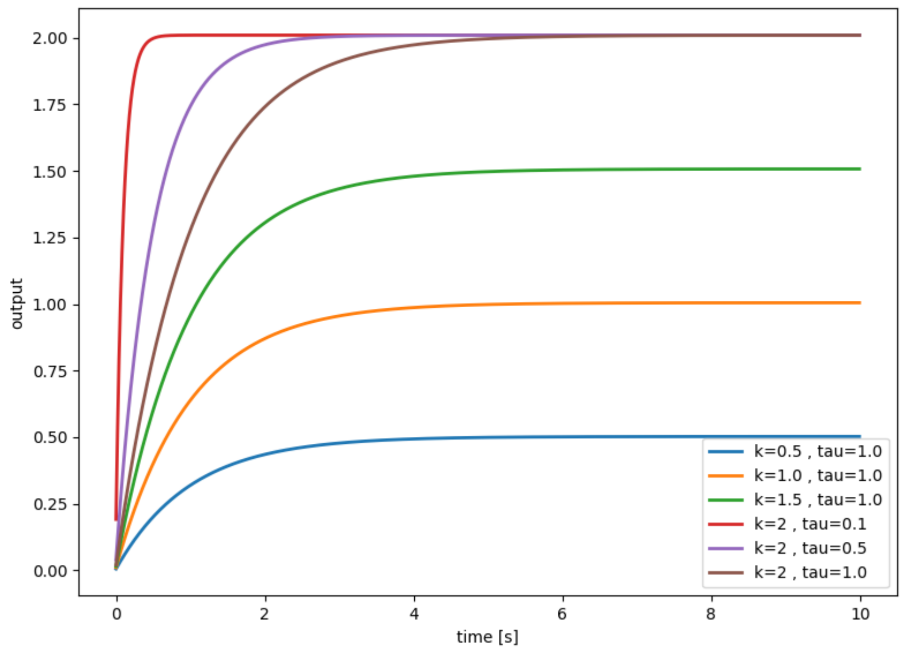


Figure 2.2: Step responses of a first-order system for different values of τ and k.

- The system always approaches the steady-state value **exponentially**, following the general solution:

$$x(t) = k\,u\left(1 - e^{-t/\tau}\right) \tag{2.19}$$

Thus, the parameters τ and k completely determine how the system responds to changes in input: τ controls the speed, while k controls the magnitude.

Practical Insight

Understanding this basic model is important because many complex systems can be approximated by combinations of such first-order dynamics. For example, the motor's current loop, the temperature of an actuator, or even the airflow in a ventilation system can each be modeled as a first-order process. This simplicity makes first-order models a cornerstone of control theory and system identification. The example of this 1st order system is located at examples/01_dynamical_system/first_order.py

2.3 First Order Identification

The process of system identification involves finding the matrices A, B (and optionally C). From this model, a controller can be synthesized, or the system can be used for multi-step prediction and planning.

The goal in this section is to identify the parameters of a DC motor. The motor velocity model can be approximated as a first-order linear system, although the real system is, of course, nonlinear. A photo of the example setup is shown in Fig. 2.3.

The control loop hardware consists of a motor driver (H-bridge DRV8212), a DC motor (Pololu HP 1:30 gear), and a quadrature magnetic encoder (2× DRV5013), as illustrated in Fig. 2.4.

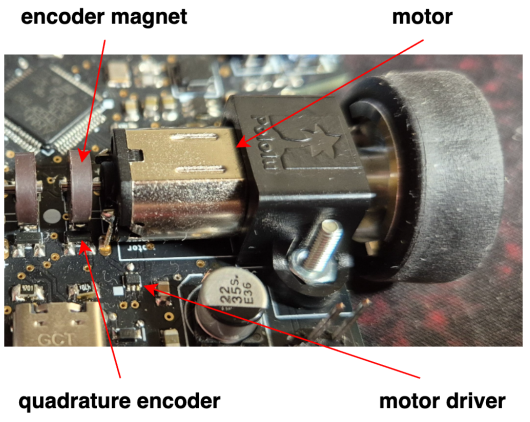


Figure 2.3: Wheel drive hardware detail.

The input signal $u(n)$ can be generated arbitrarily. A square wave is often used to capture higher-frequency dynamics. The identification algorithm estimates the model parameters based on the observed motor velocity $x(n)$ and the known input $u(n)$.

Before testing the real motor, we first demonstrate the approach using a simulation example. The **simulated motor** has the following parameters - chosen arbitrarily :

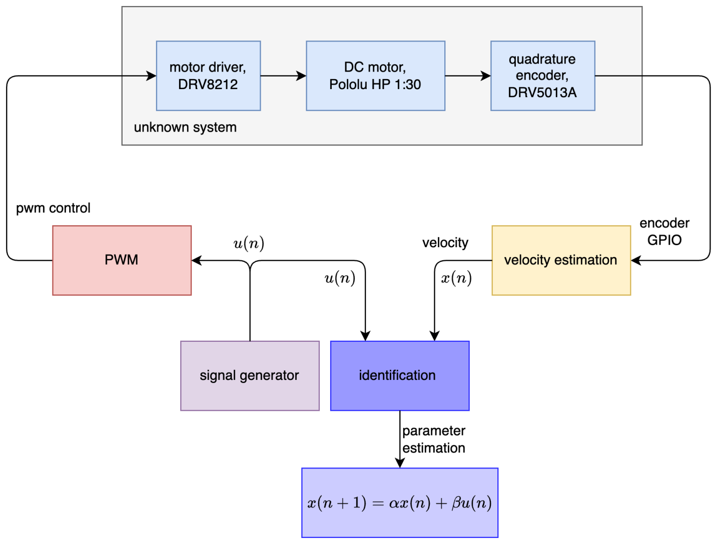


Figure 2.4: Motor identification process.

- sampling frequency: 4 kHz, $\Delta T = 1/4000$
- maximum control input: $u_{\max} = 2$
- motor constant: $k = 17$
- motor time constant: $\tau = 29$ milliseconds

The motor state is a single variable—the angular velocity $\omega(t)$. The model has two parameters, α and β. The first-order continuous model is defined as:

$$\frac{d\omega(t)}{dt} = \alpha\,\omega(t) + \beta\,u(t) \tag{2.20}$$

$$\alpha = -\frac{1}{\tau} \tag{2.21}$$

$$\beta = \frac{k}{\tau} \tag{2.22}$$

With the given parameters k and τ, we obtain $\alpha = -34.48$ and $\beta = 586.2$.

The unit step response is obtained by setting $u(n) = 1$ and solving the differential equation using any ODE solver. The simplest method is the Euler approach, which works well for small ΔT and low-order systems. For higher accuracy, the commonly used fourth-order Runge–Kutta (RK4) method is preferred.

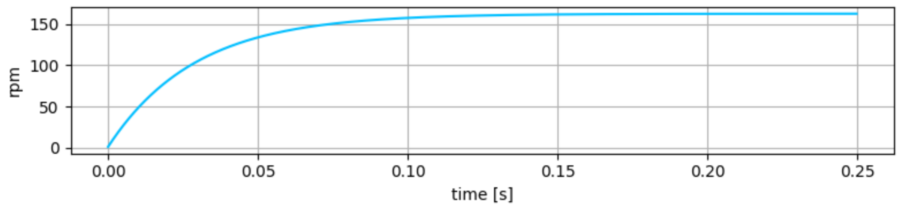


Figure 2.5: Motor step response.

Our goal is to generate input $u(t)$ and, by observing the system output $x(t)$, estimate the parameters k and τ.

We start with the simplest identification method, suitable for first-order, non-oscillating systems. This method has two main steps:

1. steady-state value estimation
2. time constant estimation

Steady-State Estimation

We drive the motor with different constant input levels, ranging from 10% to 100%. By observing the steady-state velocity, the motor constant k can be estimated.

Algorithm for estimating the motor constant:

1. choose a constant motor input $u \in (0, u_{\max}]$
2. run the motor and wait until steady-state velocity is reached
3. measure the velocity
4. estimate $\hat{k} = x/u$
5. repeat and average the resulting $\hat{k}$

In the following code, we use 10 different input levels. After setting each input, we wait 500 steps to reach steady state, measure the velocity, and then compute the average.

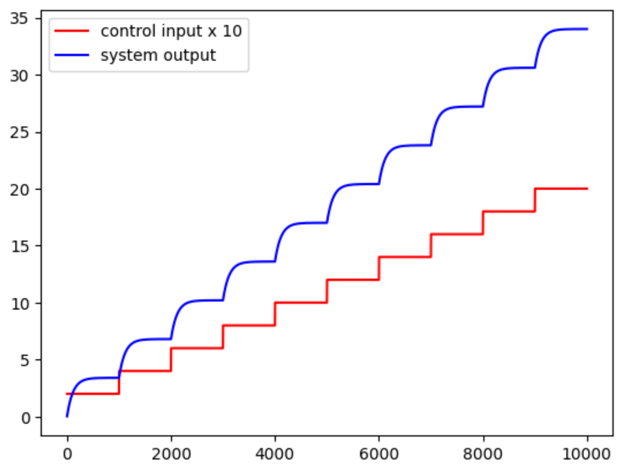


Figure 2.6: Motor constant estimation.

```python
# input levels
u_values = u_max*numpy.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])

# reset dynamical system into zero state
ds.reset()

for j in range(len(u_values)):

    x_mean = []
    for i in range(1000):

        # convert scalar u to column vector
        u    = u_values[j]
        u    = numpy.array([[u]])

        # compute dynamical system one step
        x, _ = ds.forward_state(u)

        # after steady state, store x
        if i > 500:
            x_mean.append(x[0][0])

    x_mean = numpy.array(x_mean)
    x_mean = x_mean.mean()

    k_est = x_mean/u_values[j]

    print(u_values[j], k_est)

k_mean = k_est.mean()

# average k estimate
print("k_mean = ", k_mean)
```

In our example, the resulting $\hat{k}$ is 16.995, which is very close to the true value 17. Accuracy depends on encoder noise and the number of samples used for averaging.

Time Constant Estimation

The second parameter is the time constant τ. Most textbooks estimate τ by measuring the time at which the system reaches 63.2% of its final value $x_{\max}$. Here, $x_{\max}$ is the steady-state velocity for a given u. However, this method is sensitive to noise and precise timing.

We present a more accurate approach, which excites the system into a natural oscillatory response. The algorithm is as follows:

1. choose u_{set} and set $u_{\text{in}} = u_{\text{set}}$
2. repeat for n_{steps}, each step of duration ΔT
   a) if $u_{\text{in}} > 0$ and $x(n) > 0.632\,k\,u_{\text{in}}$:
   increment $n_{\text{HalfPeriods}}$ and set $u_{\text{in}} = -u_{\text{set}}$
   b) else if $u_{\text{in}} < 0$ and $x(n) < 0.632\,k\,u_{\text{in}}$:
   increment $n_{\text{HalfPeriods}}$ and set $u_{\text{in}} = u_{\text{set}}$

The time constant $\hat{\tau}$ is then estimated as:

$$\hat{\tau} = \frac{2}{\pi} \frac{n_{\text{steps}}}{n_{\text{HalfPeriods}}} \Delta T \tag{2.23}$$

Here, ΔT is the sampling period, and the ratio $n_{\text{steps}}/n_{\text{HalfPeriods}}$ estimates the system's oscillation period. Essentially, this method performs period-duration estimation. The resulting time plot is shown in Fig. 2.7. For our simulated motor, the result is $\hat{\tau} = 27.44$ ms, which is a good estimate of the real value, 29 ms.

Using equations 2.21 and 2.22, we can estimate $\hat{\alpha}$ and $\hat{\beta}$ as $\hat{\alpha} = -36.443$ and $\hat{\beta} = 619.372$.

Real Motor Identification

For implementation on real hardware, the code must run in real time. With a relatively high sampling frequency of 4 kHz, the software is written in C++. The interface is universal, consisting of a few key functions:

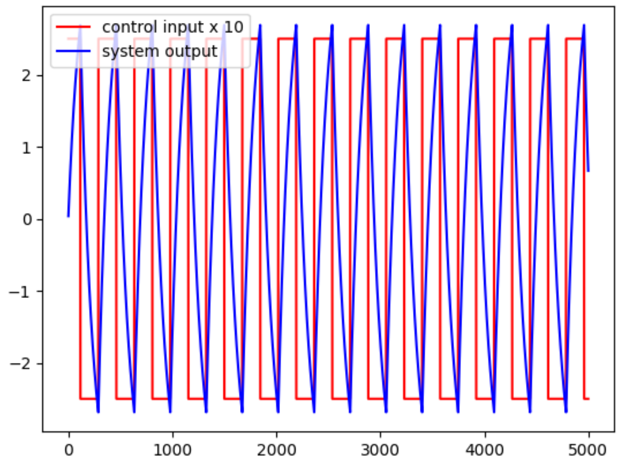


Figure 2.7: Motor time constant estimation.

- `timer.delay_ms(unsigned int time)` — waits for the given time in milliseconds
- `float x = motor_control.get_right_velocity()` — returns the measured wheel velocity
- `motor_control.set_right_torque(float x)` — sets the PWM torque command; the sign controls rotation direction

These functions must be adapted for the user's specific hardware.

First, we estimate the motor constant k:

```
//1, estimate motor constant k, on different input values
uint32_t n_steps = 500;

float u_values[10] = {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0};

float k_mean     = 0.0;
float x_var_mean = 0.0;

for (unsigned int j = 0; j < 10; j++)
{
    //run motor with desired input and wait for steady state
    float u_in = u_values[j];

    motor_control.set_right_torque(u_in*MOTOR_CONTROL_MAX_TORQUE);
    timer.delay_ms(500);

    //estimate average velocity
    float x_mean = 0.0;
    for (unsigned int i = 0; i < n_steps; i++)
    {
        float x = motor_control.get_right_velocity();
        x_mean+= x;
        timer.delay_ms(1);
    }
    x_mean = x_mean/n_steps;

    //estimate average variance - encoder noise
    float x_var = 0.0;
    for (unsigned int i = 0; i < n_steps; i++)
    {
        float x = motor_control.get_right_velocity();
        x_var+= (x - x_mean)*(x - x_mean);
        timer.delay_ms(1);
    }
    x_var = x_var/n_steps;

    float k = x_mean/u_in;
    k_mean+= x_mean/u_in;
    x_var_mean+= x_var;

    terminal << "u_in   = " << u_in << "\n";
    terminal << "k      = " << k << "\n";
    terminal << "x_mean = " << x_mean << "\n";
    terminal << "x_var  = " << x_var << "\n";
    terminal << "\n\n";
}

//print summary results
k_mean     = k_mean/10.0;
x_var_mean = x_var_mean/10.0;

terminal << "k          = " << k_mean << "\n";
terminal << "x_var_mean = " << x_var_mean << "\n";
```

Next, we estimate the motor time constant τ:

```
//2, estimate time constant by oscillating motor
float u_in = 0.25;
uint32_t periods = 0;

n_steps = 2000;

for (unsigned int i = 0; i < n_steps; i++)
{
    motor_control.set_right_torque(u_in*MOTOR_CONTROL_MAX_TORQUE);
    float x = motor_control.get_right_velocity()*60.0/(2.0*PI);

    if (u_in > 0.0)
    {
        if (x > 0.632*k_mean*u_in)
        {
            u_in = -u_in;
            periods++;
        }
    }
    else
    {
        if (x < 0.632*k_mean*u_in)
        {
            u_in = -u_in;
            periods++;
        }
    }

    timer.delay_ms(1);
}

float t_period = (2*n_steps/periods)/PI;
terminal << "periods = " << periods << "\n";
terminal << "tau     = " << t_period << "[ms]\n";
terminal << "\n\n";
```

The average motor constant is $k = 80.056$, and the time constant is $\tau = 9.230$ ms. Terminal output:

```
...

u_in   =  0.800
k      =  87.194
x_mean =  69.755
x_var  =  19.236


u_in   =  0.900
k      =  86.440
x_mean =  77.796
x_var  =  18.124


u_in   =  1.000
k      =  85.750
x_mean =  85.750
x_var  =  7.554


k          =  80.056
x_var_mean =  170.903


periods = 136
tau     =  9.230[ms]
```

```
u_in   =  0.200
k      =  295.103
rpm    = 563
x_mean =  59.020
x_var  =  21.035


u_in   =  0.300
k      =  289.379
rpm    = 829
x_mean =  86.813
x_var  =  21.276


u_in   =  0.400
k      =  268.297
rpm    = 1024
x_mean =  107.319
x_var  =  69.277


u_in   =  0.500
k      =  247.716
rpm    = 1182
x_mean =  123.858
x_var  =  55.126
```

```


u_in   =  0.600
k      =  225.210
rpm    = 1290
x_mean =  135.126
x_var  =  29.614


u_in   =  0.700
k      =  204.766
rpm    = 1368
x_mean =  143.336
x_var  =  113.756


u_in   =  0.800
k      =  187.124
rpm    = 1429
x_mean =  149.699
x_var  =  446.506


u_in   =  0.900
k      =  177.842
rpm    = 1528
x_mean =  160.058
x_var  =  76.448


u_in   =  1.000
k      =  158.996
rpm    = 1518
x_mean =  158.996
x_var  =  274.646


k          =  205.443
x_var_mean =  110.768


periods = 240
tau     =  7.957[ms]
```

Summary of Identification Results

The identification results from both simulation and real hardware show that the proposed methods provide accurate and robust estimates of the motor parameters.

The purpose of the Python simulation was to demonstrate the principle of the proposed identification method and to show how well it fits our imaginary motor model. Table 2.1 presents a direct comparison between the ground truth parameters and the estimated values obtained from the simulation.

Table 2.1: Comparison between ground truth and estimated parameters in simulation

Parameter	**Symbol**	**Ground Truth**	**Estimated**
Motor constant	k	17.000	16.995
Time constant	τ	29 ms	27.44 ms

The results clearly show that the estimation method performs very well for a simple first-order system, with minimal deviation from the true parameter values. This confirms the correctness of the identification algorithm in a noise-free simulation environment.

For the real hardware experiment, the results demonstrate consistent values of both τ and k across multiple input levels u_{in}, which indicates that the method is robust even under real-world conditions. On the physical motor, the identified parameters were $k = 80.056$ and $\tau = 9.230$ ms, which are consistent with the expected values for a small, high-speed DC drive.

This experiment demonstrates how a simple first-order model can successfully capture the **essential dynamics of a real motor** and serve as a foundation for controller design. In practice, the identification accuracy depends on factors such as encoder resolution, sampling frequency, and PWM control precision. Despite nonlinearities such as friction, saturation, and supply voltage variations, the identified model provides a reliable linear approximation—suitable for designing velocity or torque controllers in robotic drive systems.

Additionally, we also estimated the velocity variance x_{var}, which provides valuable information about measurement noise. This value will be further utilized in the **design of a Kalman filter** for sensor fusion and state estimation in the following chapter.

Chapter 3

Linear Quadratic Regulator

The LQR controller is a standard baseline for regulating systems with multiple inputs and multiple outputs. The controller requires knowledge of the system dynamics (state–space model), and two weighting matrices in the cost function, which are design parameters chosen by the engineer.

LQR gained its position as a baseline method because it is:

- mathematically well–defined with a unique optimal solution,
- robust for a wide class of linear systems,
- able to naturally balance performance vs. control effort,
- easy to tune once the engineer understands the role of Q and R,
- computationally efficient (solved offline),
- superior to PID for multivariable, coupled, high-dimensional systems.

Why not PID? PID is fantastic for **single–input single–output** systems. But as soon as the dynamics are coupled (e.g. drones, robots, manipulators, cars), hand-tuning dozens of gains quickly becomes impossible. LQR solves all gains at once optimally — which is why it is the industry default for linear robotics control.

Assume we would like to control a 2D system: a moving sphere with inertia.

This system has two inputs, applied forces $u_x(n)$ and $u_y(n)$. The state is 4-dimensional: position and velocity in both spatial dimensions. As shown in Fig. 3.1, the state vector is $x(n)$, and we explicitly mark the velocity term as $v(n)$ (which is part of the state). The control input $u(n)$ must counteract velocity and inertia to steer the system into the desired state $x_r(n)$.

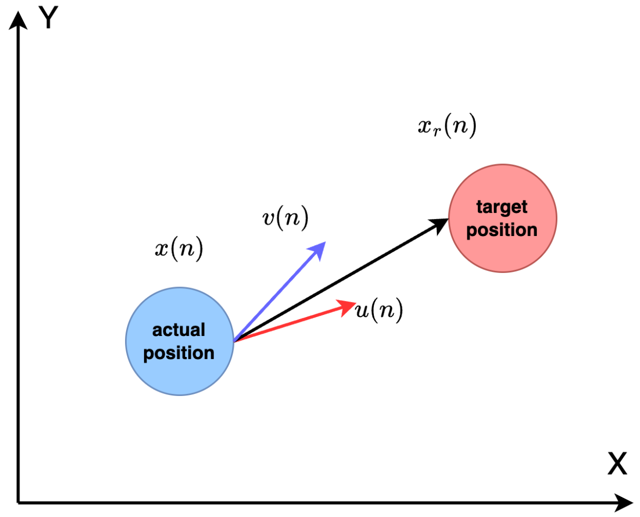


Figure 3.1: 2D position control

This is a system with multiple outputs and multiple inputs:

$$x(n) = \begin{bmatrix} p_x(n) \\ p_y(n) \\ v_x(n) \\ v_y(n) \end{bmatrix} \tag{3.1}$$

$$u(n) = \begin{bmatrix} u_x(n) \\ u_y(n) \end{bmatrix} \tag{3.2}$$

The ordering of state variables is arbitrary; here we place positions first, then velocities.

The intuition behind LQR is simple:

Can we find a linear mapping from the error between the current state and desired state into an optimal control action?

We define the tracking error:

$$e(n) = x_r(n) - x(n) \tag{3.3}$$

The simplest general mapping from error space to control space is matrix multiplication:

$$u(n) = Ke(n) \tag{3.4}$$

The matrix K is the gain matrix. It assigns weights to each state variable and uses a linear combination to compute the optimal action u.

3.1 Derivation of LQR

We now derive the matrix K. We begin by defining notation and assumptions used throughout this derivation.

- n — discrete **time step**.
- N — number of **state variables** (state dimension).
- M — number of **control inputs** (input dimension).
- $x(n)$ — **system state**, column vector of size $N \times 1$.
- $u(n)$ — **control input**, column vector of size $M \times 1$.
- A — **system matrix**, $N \times N$.
- B — **input matrix**, $N \times M$.
- Q — positive semidefinite **state weighting matrix**, $N \times N$.
- R — positive semidefinite **input weighting matrix**, $M \times M$.

The controller drives the system toward the desired state $x_r(n)$ by minimizing the error $e(n) = x_r(n) - x(n)$.

The discrete–time system dynamics are:

$$x(n+1) = Ax(n) + Bu(n) \tag{3.5}$$

Quadratic Cost Function

The standard discrete-time LQR problem minimizes the infinite-horizon quadratic cost

$$\mathcal{L} = \sum_{n=0}^{\infty} \left[x(n)^T Q x(n) + u(n)^T R u(n) \right] \tag{3.6}$$

This cost penalizes:

- **state deviation** through Q (how much we care about errors),
- **control effort** through R (how expensive the control is).

Intuitively:

- Large Q entries $\Rightarrow$ state errors are very expensive.
- Large R entries $\Rightarrow$ control effort is expensive (fuel, power, torque limits).

Fig. 3.2 interprets the quadratic loss. For a 4-D state and 2-D control input, Q is 4×4, R is 2×2. Multiplying a vector by its transpose forms a quadratic (parabolic) penalty — the same idea as MSE loss.

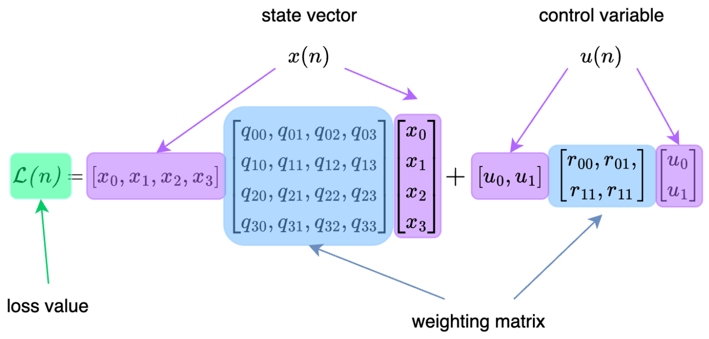

state vector
control variable
$x(n)$
$u(n)$
$\mathcal{L}(n) = [x_0, x_1, x_2, x_3]$
$q_{00}, q_{01}, q_{02}, q_{03}$
$q_{10}, q_{11}, q_{12}, q_{13}$
$q_{20}, q_{21}, q_{22}, q_{23}$
$q_{30}, q_{31}, q_{32}, q_{33}$
x_0 x_1 x_2 x_3
$+ [u_0, u_1]$
$r_{00}, r_{01}, r_{11}, r_{11}$
u_0 u_1
loss value
weighting matrix

Figure 3.2: Quadratic loss meaning

Choosing Q and R is a design choice. Most often they are diagonal, meaning each state variable is weighted independently.

Solving the LQR Problem

The optimal controller is obtained by solving the **discrete algebraic Riccati equation (DARE)**. The solution yields the optimal gain matrix:

$$u(n) = -Kx(n) \tag{3.7}$$

This steers the system to the origin $x = 0$. To track non-zero references, we rewrite:

$$u(n) = K(x_r(n) - x(n)) \tag{3.8}$$

(Here the minus sign is absorbed into the error definition.)
Fig. 3.3 shows the complete LQR algorithm.

1. **Offline (precompute)**:
   a) Given $A,\ B,\ Q,\ R$, solve the discrete algebraic Riccati equation (DARE) for P:

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

   b) Compute the optimal gain matrix K :

   $$K \;=\; (R + B^\top PB)^{-1}B^\top PA$$

   c) **Store** K and the nominal design matrices — these are used online.
2. **Online (every control time-step n)**:
   a) Measure or estimate current state $x(n)$ and set reference $x_r(n)$.
   b) Compute tracking error:

   $$e(n) = x_r(n) - x(n).$$

   c) Compute control action:

   $$u(n) = K\,e(n).$$

d) Apply $u(n)$ to the plant.
Notes: K is constant for linear time-invariant designs; computing it offline makes the real-time loop very cheap.

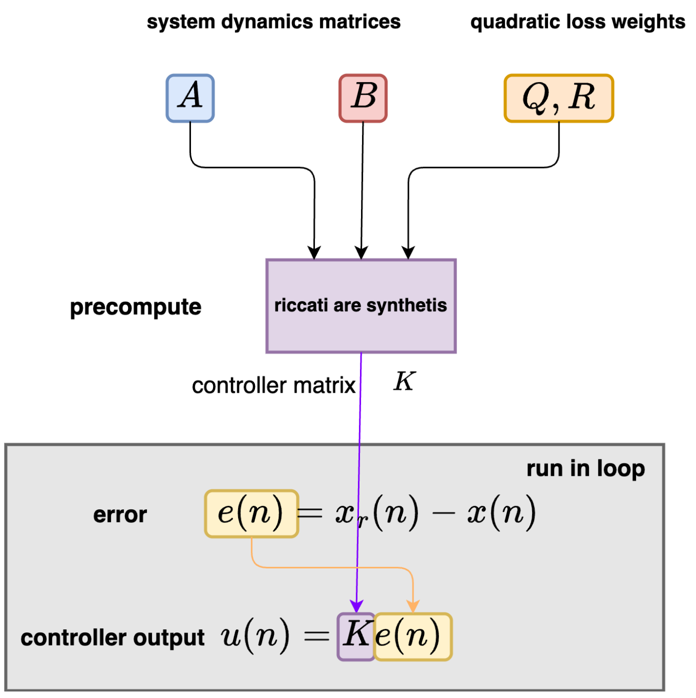


Figure 3.3: LQR algorithm

3.2 Adding Integral Action

The basic LQR has no integral term; thus it cannot remove steady-state error under constant disturbance. Integral action is necessary when:

- the plant has no natural integrator,
- external disturbances are constant or slowly varying,
- we need zero steady-state error for positions.

The integral term accumulates error over time:

if error persists, push harder

We augment the system with integral states:

$$\tilde{A} = \begin{bmatrix} A & 0 \\ I & I \end{bmatrix} \tag{3.9}$$

The cost matrix must be augmented as well:

$$\tilde{Q} = \begin{bmatrix} Q & 0 \\ 0 & \kappa Q \end{bmatrix} \tag{3.10}$$

where κ sets how expensive the integral error is.
Solving DARE for the augmented system yields:

$$\tilde{K} = \begin{bmatrix} K & K_i \end{bmatrix} \tag{3.11}$$

Fig. 3.4 shows the algorithmic structure.
The implemented integral loop:

$$e(n) = x_r(n) - x(n) \tag{3.12}$$
$$e_{int}(n) = e_{int}(n-1) + K_i e(n) \tag{3.13}$$
$$u(n) = Ke(n) + e_{int}(n) \tag{3.14}$$

Note: the minus sign is absorbed into the error state.

3.3 Adding Antiwindup

Real actuators are limited. A motor might accept only $[-12, 12]$ volts. But the controller may compute $u = 20$V because of the accumulated integral term.

What happens?

- Actuator outputs only 12V.

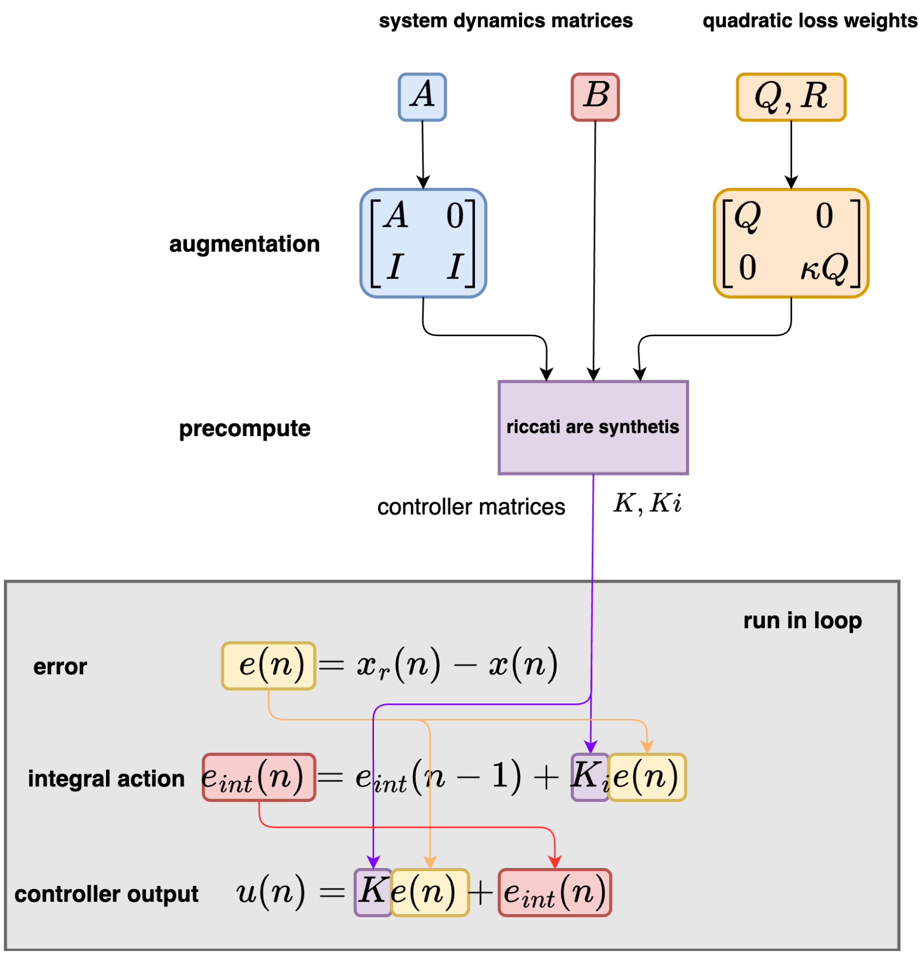


Figure 3.4: LQR algorithm with integral action

- Integrator keeps increasing (windup).
- Controller thinks it is applying 20V (internal model), but the plant sees only 12V.
- As soon as the error becomes small, the huge integrator causes overshoot.

This leads to oscillations and long settling time — not due to bad K or bad model, but due to ignoring actuator limits.

Antiwindup correction

Fig. 3.5 shows the method.

First compute the tentative integral update:

$$\hat{e}_{int}(n) = e_{int}(n-1) + K_i e(n)$$

Compute the tentative (unsaturated) control:

$$\hat{u}(n) = Ke(n) + \hat{e}_{int}(n)$$

Apply actuator saturation:

$$u(n) = \text{clip}(\hat{u}(n), u_{min}, u_{max})$$

Correct the integrator:

$$e_{int}(n) = \hat{e}_{int}(n) - (\hat{u}(n) - u(n))$$

The correction term zeroes out whenever the actuator is not saturated.

This works cleanly because the integral term lives in the **action space** ($K_i e$), so saturation and projection are consistent and simple.

3.4 Practical Implementation

Below is a complete Python implementation of the LQR controller with integral action and antiwindup.

The constructor precomputes control matrices. The method **solve** augments matrices and solves DARE. The real-time loop calls **forward**, which requires:

- desired state x_r,
- current state x,
- current integral state.

The output is control u and updated integrator.

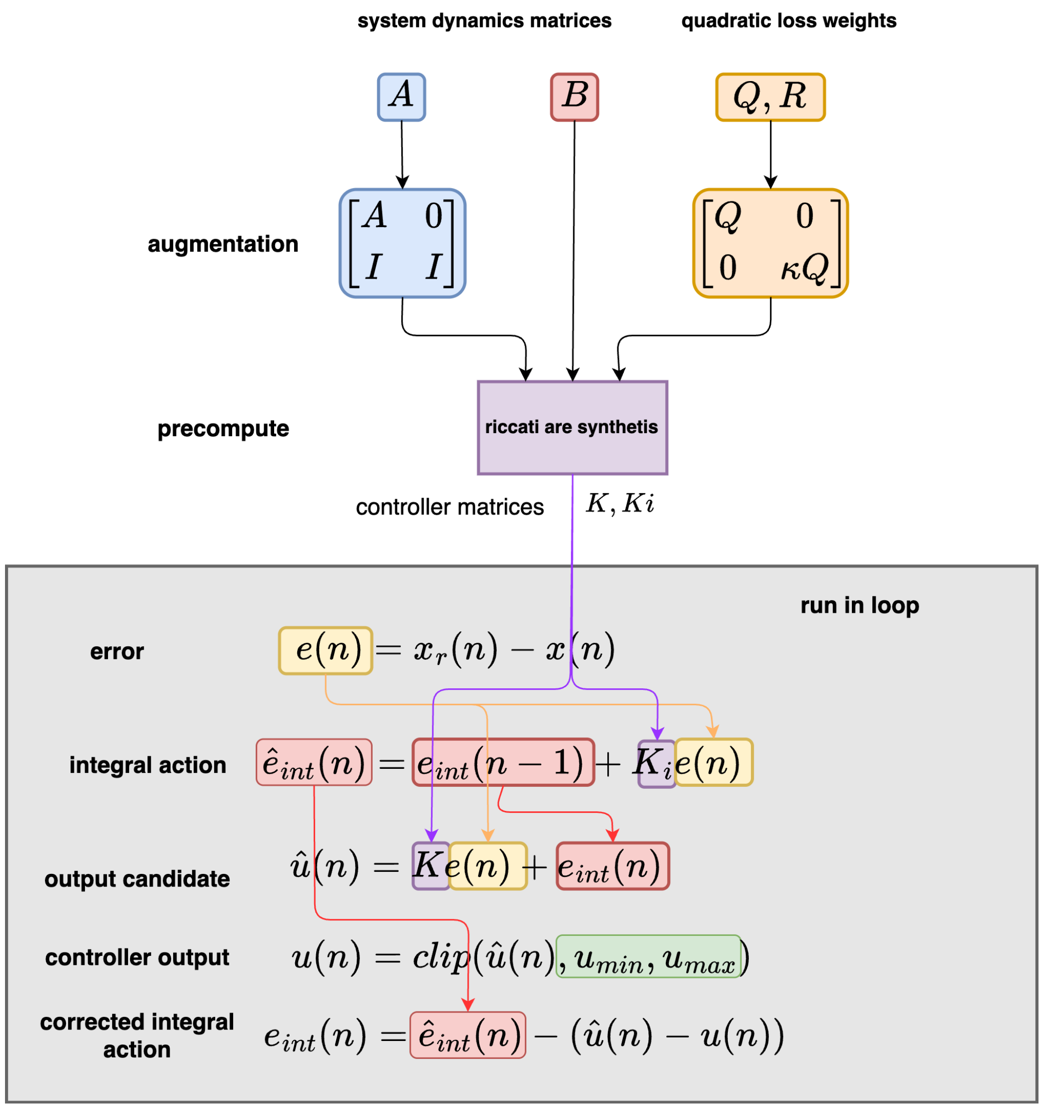

system dynamics matrices
quadratic loss weights
A
B
Q, R
augmentation
$\begin{bmatrix} A & 0 \\ I & I \end{bmatrix}$
$\begin{bmatrix} Q & 0 \\ 0 & \kappa Q \end{bmatrix}$
precompute
riccati are synthetis
controller matrices
K, Ki
run in loop
error
$e(n) = x_r(n) - x(n)$
integral action
$\hat{e}_{int}(n) = e_{int}(n-1) + K_i e(n)$
output candidate
$\hat{u}(n) = Ke(n) + e_{int}(n)$
controller output
$u(n) = clip(\hat{u}(n), u_{min}, u_{max})$
corrected integral action
$e_{int}(n) = \hat{e}_{int}(n) - (\hat{u}(n) - u(n))$

Figure 3.5: LQR algorithm with integral action and antiwindup

```
import numpy
import scipy

class LQRIDiscrete:

    def __init__(self, a, b, q, r, qi = 0.0, antiwindup = 10**10):
        self.k, self.ki = self.solve(a, b, q, r, qi)
        self.antiwindup = antiwindup

    def solve(self, a, b, q, r, qi):
        n = a.shape[0]  # system order
        m = b.shape[1]  # system inputs

        # augmented system
        a_aug = numpy.block([
            [a, numpy.zeros((n, n))],
            [numpy.eye(n), numpy.eye(n)]
        ])

        b_aug = numpy.vstack([b, numpy.zeros((n, m))])

        # augmented cost
        q_aug = numpy.block([
            [q, numpy.zeros((n, n))],
            [numpy.zeros((n, n)), qi * q]
        ])

        p = scipy.linalg.solve_discrete_are(a_aug, b_aug, q_aug, r)
        k_aug = numpy.linalg.inv(r) @ (b_aug.T @ p)

        # truncated small elements
        k_aug[numpy.abs(k_aug) < 10**-10] = 0

        k  = k_aug[:, :n]
        ki = k_aug[:, n:]

        return k, ki

    def forward(self, xr, x, integral_action):
        # integral action
        error = xr - x

        integral_action_new = integral_action + self.ki @ error

        # LQR control law
        u_new = self.k @ error + integral_action

        # conditional antiwindup
        u = numpy.clip(u_new, -self.antiwindup, self.antiwindup)
        integral_action_new = integral_action_new - (u_new - u)

        return u, integral_action_new
```

3.5 Summary

The Linear Quadratic Regulator is one of the most widely used controllers in modern control engineering. Its strength comes from combining a mathematically optimal formulation with practical usability. Once a state–space model is available, LQR provides a systematic way to compute all feedback gains at once, even for large multivariable systems.

Applications

LQR is a standard tool in fields where:

- accurate state–space models exist,
- systems have multiple coupled states and inputs,
- fast, reliable, and stable control is required.

Typical applications include:

- robotics and manipulators,
- quadrotors, drones, and aerospace vehicles,
- autonomous driving and vehicle dynamics control,
- inverted pendulums and balancing systems,
- power systems and energy control,
- any linearized model in engineering simulations.

Advantages

LQR is popular because it offers:

- **Uniqueness:** the solution to the Riccati equation is unique and globally optimal for the linear-quadratic problem.
- **Multivariable optimality:** all gains are solved simultaneously, avoiding the combinatorial tuning of PID.
- **Stability guarantees:** the resulting K stabilizes the system whenever the standard LQR conditions are met.
- **Clear tuning philosophy:** Q shapes state penalties, R shapes effort penalties.

- **Computational efficiency:** Riccati equations are solved once offline; online computation is a single matrix multiplication.
- **Smooth control actions:** the quadratic penalty naturally avoids aggressive or discontinuous inputs.

Limitations

Despite its strengths, LQR has several important limitations:

- **Requires a linear state–space model.** Nonlinear systems must be linearized or approximated.
- **Sensitive to modelling errors.** Large unmodelled dynamics or unmodelled delays degrade performance.
- **No inherent integral action.** Steady-state errors require explicit integrator augmentation.
- **No built-in actuator constraints.** Saturation must be handled externally (antiwindup, MPC, etc.).
- **Not optimal for non-quadratic cost functions.** The controller's optimality holds only for quadratic penalties.

Final Remarks

LQR remains a gold standard for linear control because it is:

- theoretically elegant,
- computationally cheap,
- practically effective,
- easy to integrate with integral action and antiwindup,
- naturally suited to multivariable systems.

For systems with good linear models and strong coupling, LQR often provides a performance baseline that is difficult to surpass using manual PID tuning.

Chapter 4

Model Predictive Control

We begin by defining the control problem. The robot must navigate along a desired trajectory X_r. The robot's motion is constrained by its own dynamics, typically expressed as a discrete-time state-space model.

The Linear Quadratic Regulator (LQR) optimal control law can **only regulate** the system to a desired state x_r, which corresponds to a **single point on the trajectory**. In contrast, Model Predictive Control (MPC) leverages the knowledge of **future desired states** x_r. At its input, the controller receives the entire reference trajectory X_r, covering several future time steps defined by the prediction horizon.

Figure 4.1 illustrates the difference between LQR and MPC for a robot trajectory tracking task.

MPC consists of two main components:

- a model of the system dynamics,
- and an optimizer.

The system dynamics model uses the current state $x(n)$ to predict the future states $\hat{x}(n+1), \hat{x}(n+2), \ldots, \hat{x}(n+H_p)$ up to the prediction horizon H_p. The controller takes as input the **desired trajectory** X_r, the **current state** $x(n)$, and the **system model**, and performs **optimization** to compute an optimal sequence of control actions U. The working principle demonstrates figure 4.2. Note in general model can be inperfect, and the predicted states as well.

This optimization incrementally adjusts the control signal so that the predicted system trajectory X follows the reference X_r as closely as possible.

Such **predictive behavior** allows the controller to, for example, initiate braking or turning earlier, resulting in smoother motion and overall higher control performance.

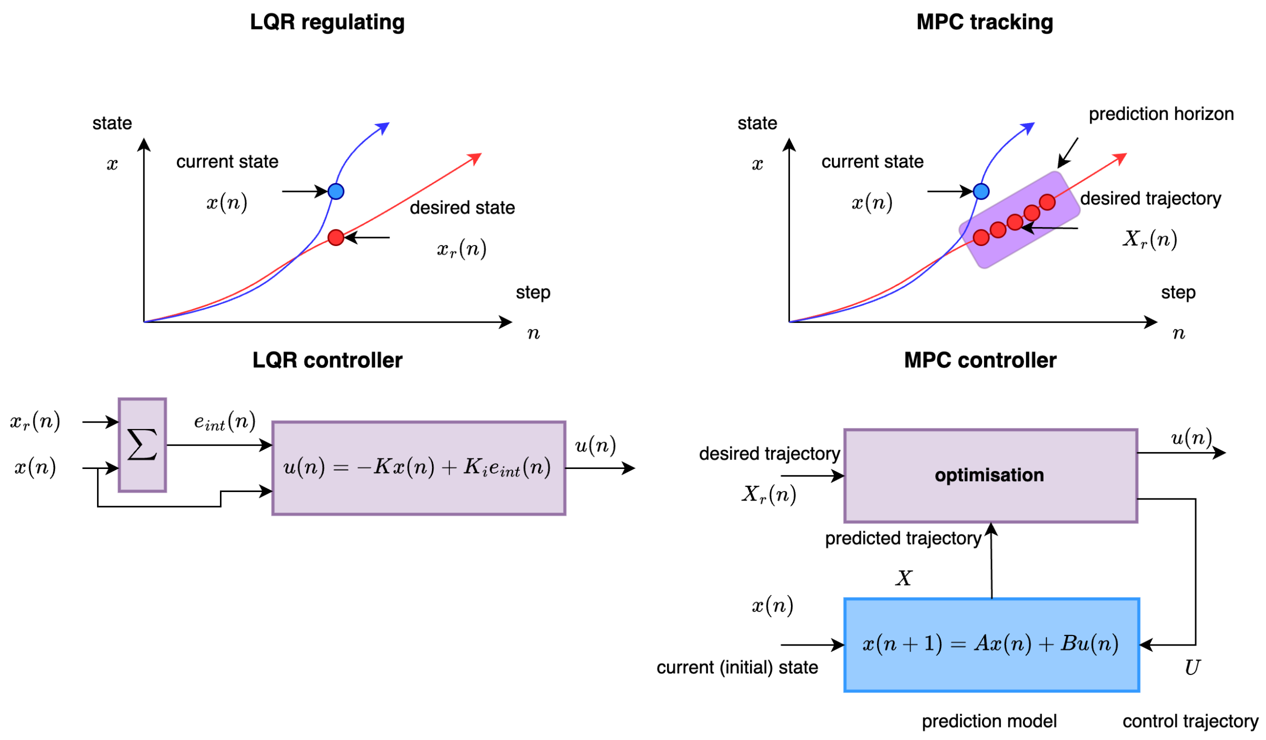


Figure 4.1: Comparison of LQR and MPC for robot trajectory tracking.

In following steps, we derivate **special case** of unconstrained MPC. This allow us to completely avoid optimizer, and give us **analytical solution**, which is capable to run in real time (less than 10ms) in almost all nowadays hardware.

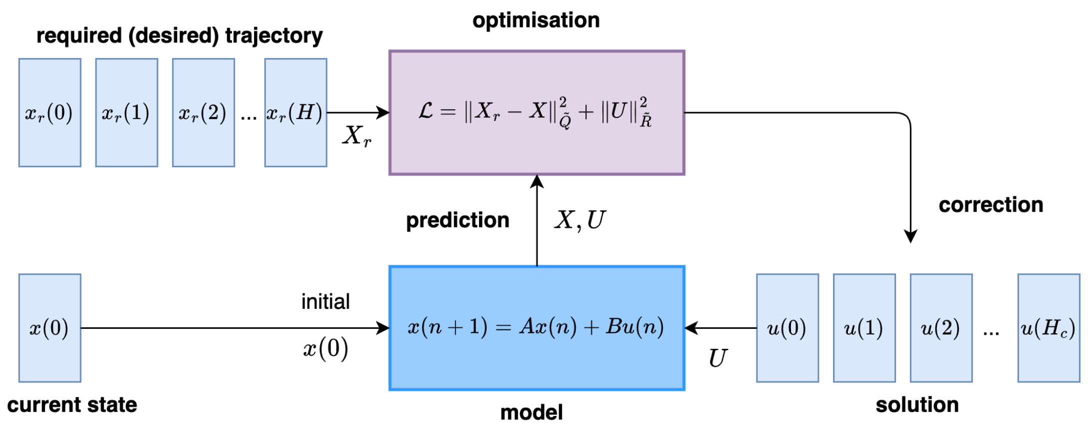


Figure 4.2: MPC algorithm working principle

We begin by defining the notation and assumptions used throughout this derivation.

- n — discrete **time step**.
- N — number of **state variables** (state dimension).
- M — number of **control inputs** (input dimension).
- $x(n)$ — **system state**, column vector of size $N \times 1$.
- $u(n)$ — **control input**, column vector of size $M \times 1$.
- A — **system matrix**, $N \times N$.
- B — **input matrix**, $N \times M$.
- Q — positive semidefinite **state weighting matrix**, $N \times N$.
- R — positive semidefinite **input weighting matrix**, $M \times M$.

The discrete-time linear system dynamics are

$$x(n+1) = Ax(n) + Bu(n). \tag{4.1}$$

For the MPC formulation, we define:

- H_p — **prediction horizon** (number of future steps predicted),
- H_c — **control horizon** (number of future control actions to optimize),

with $H_p > H_c$.

—

4.1 Stacked system formulation

We define the stacked vector of predicted future states - **states trajectory**

$$X = \begin{bmatrix} x(n+1) \\ x(n+2) \\ \vdots \\ x(n+H_p) \end{bmatrix}, \tag{4.2}$$

of total dimension $H_pN \times 1$.

Similarly, we define the stacked vector of **future control inputs**

$$U = \begin{bmatrix} u(n) \\ u(n+1) \\ \vdots \\ u(n+H_c-1) \end{bmatrix}, \tag{4.3}$$

of size $H_cM \times 1$.

The **quadratic cost function** is

$$\mathcal{L}(U) = (X_r - X)^\top \tilde{Q}(X_r - X) + U^\top \tilde{R} U, \tag{4.4}$$

subject to the system dynamics constraint

$$x(n+1) = Ax(n) + Bu(n). \tag{4.5}$$

The weighting matrices $\tilde{Q}$ and $\tilde{R}$ are block-diagonal matrices constructed from Q and R:

$$\tilde{Q} = \begin{bmatrix} Q & & & \\ & Q & & \\ & & \ddots & \\ & & & Q \end{bmatrix}, \quad \tilde{R} = \begin{bmatrix} R & & & \\ & R & & \\ & & \ddots & \\ & & & R \end{bmatrix}. \tag{4.6}$$

Their dimensions are $\tilde{Q} \in \mathbb{R}^{H_pN \times H_pN}$ and $\tilde{R} \in \mathbb{R}^{H_cM \times H_cM}$.

—

4.2 Prediction model formalism

We express the predicted future states as a function of the current state and future control sequence:

$$\begin{bmatrix} x(n+1) \\ x(n+2) \\ \vdots \\ x(n+H_p) \end{bmatrix} = \begin{bmatrix} A^1 \\ A^2 \\ \vdots \\ A^{H_p} \end{bmatrix} x(n) \\ + \begin{bmatrix} A^0B & 0 & 0 & \dots & 0 \\ A^1B & A^0B & 0 & \dots & 0 \\ A^2B & A^1B & A^0B & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ A^{H_p-1}B & A^{H_p-2}B & \dots & A^{H_p-H_c}B & A^{H_p-H_c-1}B \end{bmatrix} \begin{bmatrix} u(n) \\ u(n+1) \\ \vdots \\ u(n+H_c-1) \end{bmatrix}$$

This compactly becomes

$$X = \Psi x(n) + \Theta U, \tag{4.7}$$

where

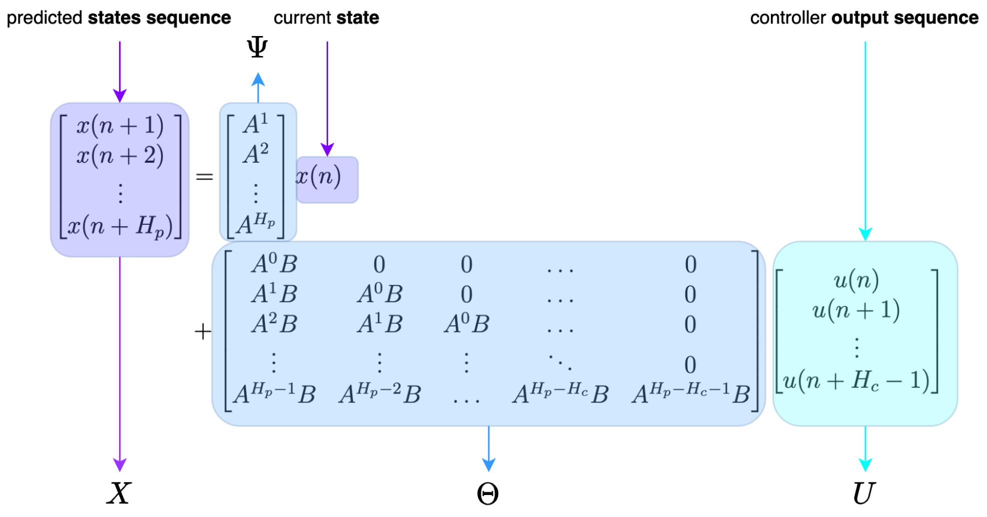
$$\Psi = \begin{bmatrix} A^1 \\ A^2 \\ \vdots \\ A^{H_p} \end{bmatrix}, \quad \Theta_{ij} = \begin{cases} A^{i-j}B, & \text{if } i \geq j, \\ 0, & \text{otherwise.} \end{cases} \tag{4.8}$$

Ψ has dimensions $H_pN \times N$ and Θ has dimensions $H_pN \times H_cM$.

—

4.3 Understanding the Prediction Model

Let us break down what the recent expressions mean on an intuitive level. **The purpose of formulating the problem in a matrix framework is to express the objective loss function in terms of the current state $x(n)$, the desired trajectory $X_r(n)$, and the sequence of control inputs U.** The following figure 4.3 illustrates how the matrices Ψ and Θ



$$\begin{bmatrix} x(n+1) \\ x(n+2) \\ \vdots \\ x(n+H_p) \end{bmatrix} = \begin{bmatrix} A^1 \\ A^2 \\ \vdots \\ A^{H_p} \end{bmatrix} x(n) + \begin{bmatrix} A^0B & 0 & 0 & \dots & 0 \\ A^1B & A^0B & 0 & \dots & 0 \\ A^2B & A^1B & A^0B & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ A^{H_p-1}B & A^{H_p-2}B & \dots & A^{H_p-H_c}B & A^{H_p-H_c-1}B \end{bmatrix} \begin{bmatrix} u(n) \\ u(n+1) \\ \vdots \\ u(n+H_c-1) \end{bmatrix}$$

Figure 4.3: Prediction model representation.

are constructed, and how they relate the current state $x(n)$ and the control sequence U to the predicted future states.

Let us first question what the terms inside the Ψ and Θ matrices actually mean.

—

Projection of the Current State

Assume we select the **first row** from Ψ and Θ, as shown in figure~4.4. This single row tells us **how the current state $x(n)$ is projected into the next state $x(n+1)$**. As we can see, this follows directly from the linear dynamic model

$$x_a(n+1) = Ax(n),$$

where x_a represents the component of the next state due solely to the system dynamics.

Similarly, the input $u(n)$ is projected via the first row of the matrix Θ, as expected from the linear state-space model:

$$x_b(n+1) = Bu(n),$$

where x_b represents the component due to control inputs. Together, these two components reproduce the well-known discrete-time linear dynamical system:

$$x(n+1) = Ax(n) + Bu(n).$$

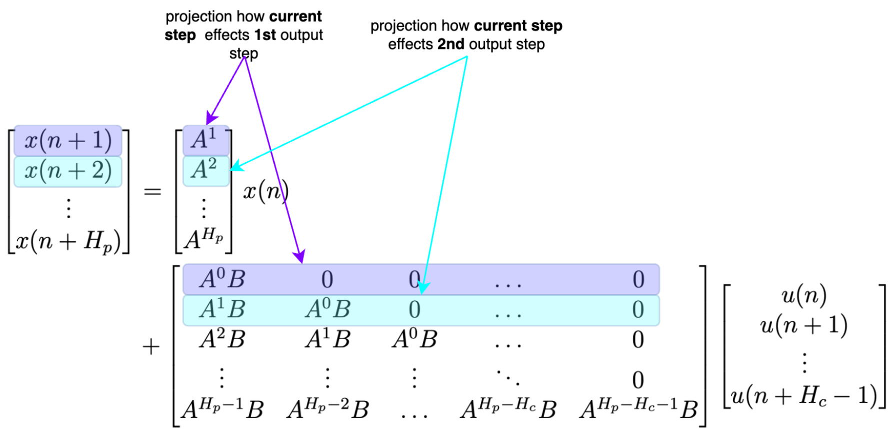

projection how **current step** effects **1st** output step
projection how **current step** effects **2nd** output step

Figure 4.4: Effect of Ψ and Θ matrix terms.

—

Extending the Prediction Horizon

Next, let us consider the **second row** of matrix Ψ, which describes how the **current state** $x(n)$ is projected further into the future, producing the A-term for $x_a(n+2)$. The current state $x(n)$ must be propagated twice to obtain $x_a(n+2)$:

$$x_a(n+1) = Ax(n),$$
$$x_a(n+2) = Ax_a(n+1) = A^2x(n).$$

This is why the **second row of matrix Ψ contains** A^2. In general, projecting the current state $x(n)$ h steps into the future is expressed as:

$$x_a(n+h) = A^h x(n).$$

—

Projection of Control Inputs

Now, let us apply the same reasoning to the **second row** of the matrix Θ and the future control input $u(n+1)$. This describes how **future control inputs** are projected into future states through the B-terms:

$$x_b(n+1) = Bu(n),$$
$$x_b(n+2) = ABu(n) + Bu(n+1).$$

This expression directly shows how to construct the matrix Θ.

In summary, Ψ **projects the current state** $x(n)$ **into future states, whereas** Θ **projects the sequence of future control inputs** U **into their effect on future states.**

—

Intuitive View of the Θ Matrix

Another way to understand Θ is to look at how each individual control input u in the sequence U affects future states. This is demonstrated in figure 4.5.

Consider the **first control input** $u(n)$, marked in purple in the figure. This term is projected into all future states $x(n+1), x(n+2), \ldots,$ which makes intuitive sense — the first control action influences the entire future trajectory.

Now look at the **second control input** $u(n+1)$, marked in cyan. In the second column of Θ, the first term is zero, reflecting causality — the input $u(n+1)$ cannot affect any past state such as $x(n+1)$.

Finally, examine the **last control input** $u(n+H_c-1)$, shown in blue. The last column of Θ contains zeros except for the last few entries, meaning that the last control input only affects the final predicted state(s).

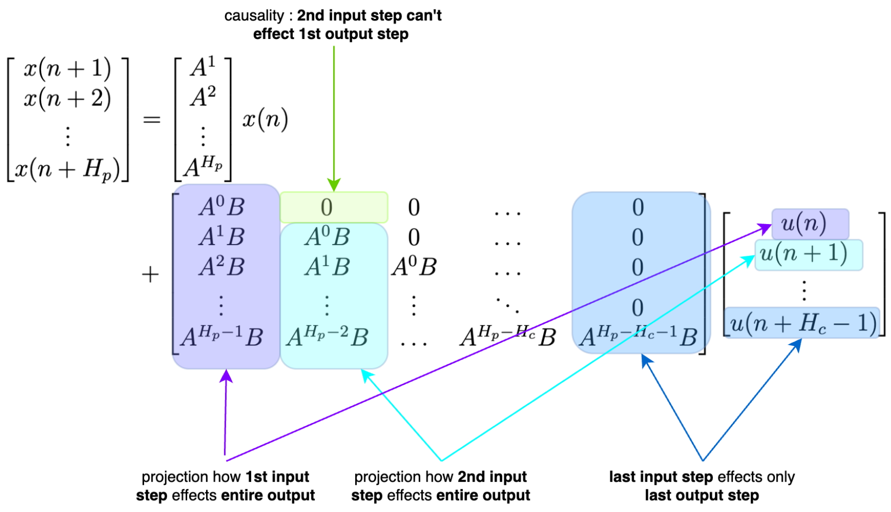

causality : 2nd input step can't effect 1st output step
$$\begin{bmatrix} x(n+1) \\ x(n+2) \\ \vdots \\ x(n+H_p) \end{bmatrix} = \begin{bmatrix} A^1 \\ A^2 \\ \vdots \\ A^{H_p} \end{bmatrix} x(n) + \begin{bmatrix} A^0B & 0 & 0 & \cdots & 0 \\ A^1B & A^0B & 0 & \cdots & 0 \\ A^2B & A^1B & A^0B & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ A^{H_p-1}B & A^{H_p-2}B & \cdots & A^{H_p-H_c}B & A^{H_p-H_c-1}B \end{bmatrix} \begin{bmatrix} u(n) \\ u(n+1) \\ \vdots \\ u(n+H_c-1) \end{bmatrix}$$
projection how 1st input step effects entire output
projection how 2nd input step effects entire output
last input step effects only last output step

Figure 4.5: Effect of Θ matrix terms on predicted states.

—

This intuitive view helps us understand how the prediction model aggregates both system dynamics and control inputs into a single compact matrix formulation, enabling MPC to efficiently predict and optimize future behavior.

—

4.4 Optimization problem

Substitute the predicted state into the cost function:

$$\mathcal{L}(U) = (X_r - \Psi x(n) - \Theta U)^\top \tilde{Q}(X_r - \Psi x(n) - \Theta U) + U^\top \tilde{R} U. \tag{4.9}$$

Define the state–reference residual

$$S = X_r - \Psi x(n), \tag{4.10}$$

to obtain

$$\mathcal{L}(U) = U^\top \tilde{R} U + (S - \Theta U)^\top \tilde{Q}(S - \Theta U). \tag{4.11}$$

Expanding and collecting terms:

$$\mathcal{L}(U) = U^\top (\tilde{R} + \Theta^\top \tilde{Q} \Theta) U - 2U^\top \Theta^\top \tilde{Q} S + S^\top \tilde{Q} S. \tag{4.12}$$

The derivative terms are:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial U} &: \\ \frac{\partial U^T \tilde{R} U}{\partial U} &= 2\tilde{R}U \\ \frac{\partial U^T \Theta^T \tilde{Q} \Theta U}{\partial U} &= 2\Theta^T \tilde{Q} \Theta U \\ \frac{\partial -2U^T \Theta^T \tilde{Q} S}{\partial U} &= -2\Theta^T \tilde{Q} S \\ \frac{\partial S^T \tilde{Q} S}{\partial U} &= 0 \end{aligned} \tag{4.13}$$

—

4.5 Analytical solution

Taking the derivative with respect to U and setting it to zero:

$$\frac{\partial \mathcal{L}}{\partial U} = 2(\tilde{R} + \Theta^\top \tilde{Q} \Theta) U - 2\Theta^\top \tilde{Q} S = 0. \tag{4.14}$$

Hence, the **optimal control sequence for the unconstrained MPC problem is**

$$U^* = (\tilde{R} + \Theta^\top \tilde{Q} \Theta)^{-1} \Theta^\top \tilde{Q} S, \tag{4.15}$$

which is equivalent to

$$U^* = (\tilde{R} + \Theta^\top \tilde{Q} \Theta)^{-1} \Theta^\top \tilde{Q} (X_r - \Psi x(n)). \tag{4.16}$$

4.6 Remarks

- The matrices $\tilde{Q}$ and $\tilde{R}$ must be symmetric and positive semidefinite to guarantee convexity of the optimization problem.
- If $H_c < H_p$, the remaining predicted inputs beyond $u(n + H_c - 1)$ are assumed to be zero.
- Actuator saturation can be applied by clipping $u(n)$ to its physical limits.
- The presented derivation corresponds to the *unconstrained* MPC case; if hard constraints on inputs or states are required, the same quadratic structure can be solved using a quadratic programming (QP) solver.

—

4.7 Algorithm implementation

Since all matrices are constant for a given system, we can precompute

$$\Sigma = (\tilde{R} + \Theta^\top \tilde{Q} \Theta)^{-1} \Theta^\top \tilde{Q}. \tag{4.17}$$

At each control step:

$$E(n) = X_r(n) - \Psi x(n), \tag{4.18}$$
$$U(n) = \Sigma E(n), \tag{4.19}$$
$$u(n) = \text{first } M \text{ elements of } U(n). \tag{4.20}$$

The control signal $u(n)$ is then applied to the plant, and the process repeats at the next sampling instant. The overall idea of algorithm is in the following figure~4.6.

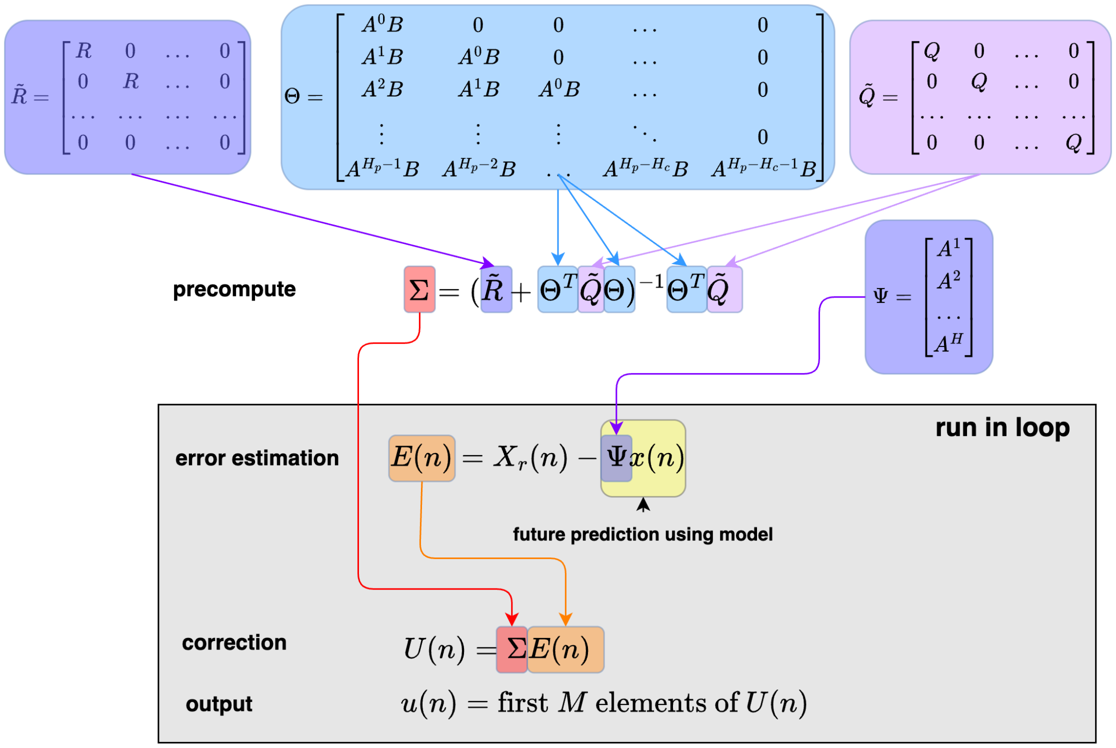

$\tilde{R} = \begin{bmatrix} R & 0 & \dots & 0 \\ 0 & R & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 0 \end{bmatrix}$
$\Theta = \begin{bmatrix} A^0B & 0 & 0 & \dots & 0 \\ A^1B & A^0B & 0 & \dots & 0 \\ A^2B & A^1B & A^0B & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ A^{H_p-1}B & A^{H_p-2}B & \dots & A^{H_p-H_c}B & A^{H_p-H_c-1}B \end{bmatrix}$
$\tilde{Q} = \begin{bmatrix} Q & 0 & \dots & 0 \\ 0 & Q & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & Q \end{bmatrix}$
precompute
$\Sigma = (\tilde{R} + \Theta^T \tilde{Q} \Theta)^{-1} \Theta^T \tilde{Q}$
$\Psi = \begin{bmatrix} A^1 \\ A^2 \\ \dots \\ A^H \end{bmatrix}$
run in loop
error estimation
$E(n) = X_r(n) - \Psi x(n)$
future prediction using model
correction
$U(n) = \Sigma E(n)$
output
$u(n) =$ first M elements of $U(n)$

Figure 4.6: Block diagram of unconstrained MPC algorithm

Python code example

First we construct all matrices, precompute Σ and Θ. This is costly operation, hewever it can be done once in constructor :

```python
"""
A: (n_x, n_x)
B: (n_x, n_u)
Q: (n_x, n_x) (state cost)
R: (n_u, n_u) (input cost)
prediction_horizon  : Hp, how many future states
control_horizon     : Hc, how many future inputs we optimize;
                      typically <= Hp
"""
def __init__(self, A, B, Q, R, prediction_horizon=16, control_horizon=4, u_max=1e10):

    self.A      = A
    self.B      = B
    self.nx     = A.shape[0]
    self.nu     = B.shape[1]
    self.Hp     = prediction_horizon
    self.Hc     = control_horizon
    self.u_max  = u_max

    # 1, build Phi and Theta
    self.Phi, self.Theta = self._init_matrices(A, B, self.Hp, self.Hc)

    # 2, build augmented tilde Q and tilde R, block-diagonal
    self.Q_aug = numpy.kron(numpy.eye(self.Hp), Q)
    self.R_aug = numpy.kron(numpy.eye(self.Hc), R)

    # Precompute solver matrices: G and Sigma
    G = self.Theta.T @ self.Q_aug @ self.Theta + self.R_aug

    # use solve later for stability; but precompute factorization if desired
    # here we compute Sigma by solving G Sigma^T = Theta^T Q_aug  (do via solve)
    # Sigma has shape (n_u*Hc, n_x*Hp)
    # Solve H @ Sigma = Theta.T @ Q_aug
    # Sigma = numpy.linalg.solve(H, Theta.T @ Q_aug)
    self.Sigma  = numpy.linalg.solve(G, self.Theta.T @ self.Q_aug)
    self.Sigma0 = self.Sigma[:self.nu, :]


def _init_matrices(self, A, B, Hp, Hc):
    nx = A.shape[0]
    nu = B.shape[1]
    # precompute A powers: A^0 ... A^Hp
    A_pows = [numpy.eye(nx)]
    for i in range(1, Hp + 1):
        A_pows.append(A_pows[-1] @ A)

    # Phi: (nx*Hp, nx) stacked [A; A^2; ...; A^Hp]
    Phi = numpy.zeros((nx * Hp, nx))
    for i in range(Hp):
        Phi[i * nx:(i + 1) * nx, :] = A_pows[i + 1]  # A^(i+1)

    # Theta: (nx*Hp, nu*Hc) where block (i,j) is A^(i-j) B for i>=j, else 0
    Theta = numpy.zeros((nx * Hp, nu * Hc))
    for i in range(Hp):
        for j in range(Hc):
            if i >= j:
                # A^{i-j} B
                Theta[i * nx:(i + 1) * nx, j * nu:(j + 1) * nu] = A_pows[i - j] @ B
            else:
                # remains zero
                pass

    return Phi, Theta
```

Main controll loop is straightforward. We precomputed almost everything, in fast running loop we have to perform only two matrix multiplications. The shape of Xr numpy matrix is $(NH_p, 1)$, shape of matrix x is $N, 1$, function returns column vector u of shape $(M, 1)$:

```
def forward_traj(self, Xr, x):
    # residual
    s = Xr - self.Phi @ x

    # compute only first control
    u0 = self.Sigma0 @ s
    u0 = numpy.clip(u0, -self.u_max, self.u_max)

    return u0
```

—

4.8 Tracking Problem Example

We consider a simple 2D moving sphere with inertia. The goal is to follow a given reference trajectory X_r, as shown in figure 4.7. We control two input forces, u_x and u_y.

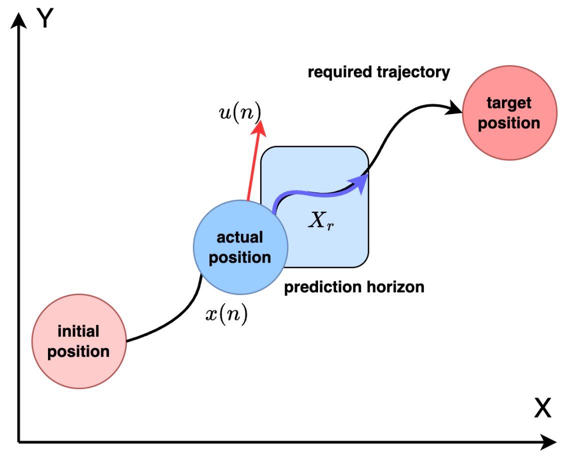


Figure 4.7: Trajectory tracking example.

The system behaves as **two independent servos with first-order inertia**, characterized by a time constant τ and an amplification factor k. We assume a continuous state-space model $\dot{x} = \bar{A}x + \bar{B}u$, which we later discretize to design a controller. This allows us to demonstrate the complete process, from a physical model to a working MPC controller.

We arbitrarily set the parameters as:

- discretization step: $dt = 0.01$ s,
- time constant: $\tau = 0.5$ s,
- amplification: $k = 0.3$.

The state vector x contains position and velocity in both axes:

$$x = \begin{bmatrix} x_{\text{pos}} \\ y_{\text{pos}} \\ x_{\text{vel}} \\ y_{\text{vel}} \end{bmatrix}.$$

The continuous system matrices $\bar{A}$ and $\bar{B}$ are:

$$\bar{A} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\frac{1}{\tau} & 0 \\ 0 & 0 & 0 & -\frac{1}{\tau} \end{bmatrix}, \qquad \bar{B} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \frac{k}{\tau} & 0 \\ 0 & \frac{k}{\tau} \end{bmatrix}. \tag{4.21}$$

After discretization for time step dt, we obtain discrete-time matrices A and B, which are used only for controller synthesis. The physical system itself is simulated in continuous time using a Runge–Kutta 4 (RK4) ODE solver, to stay close to realistic behavior.

$$A = \begin{bmatrix} 1 & 0 & 0.00990099 & 0 \\ 0 & 1 & 0 & 0.00990099 \\ 0 & 0 & 0.98019802 & 0 \\ 0 & 0 & 0 & 0.98019802 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 5.9406 \times 10^{-3} & 0 \\ 0 & 5.9406 \times 10^{-3} \end{bmatrix}. \tag{4.22}$$

—

Simulation Setup

The complete simulation workflow is illustrated in figure 4.8.

We start with a continuous-time linear dynamic model, used directly by the simulator including visualization and the RK4 solver. For controller synthesis, the model is discretized using a bilinear (Tustin) transformation for time step dt.

The controller requires weighting matrices Q and R, and the prediction and control horizons H_p and H_c. For realism, we also include a maximum control limit $u_{\max}$ to represent actuator saturation — since **real actuators are always limited**.

In our example we set:

- state weighting: $Q = \mathrm{diag}(10000, 10000, 0, 0)$,
- control weighting: $R = \mathrm{diag}(1, 1)$,
- prediction horizon: $H_p = 64$,
- control horizon: $H_c = 4$,
- output clamping : $u_{max} = 10$.

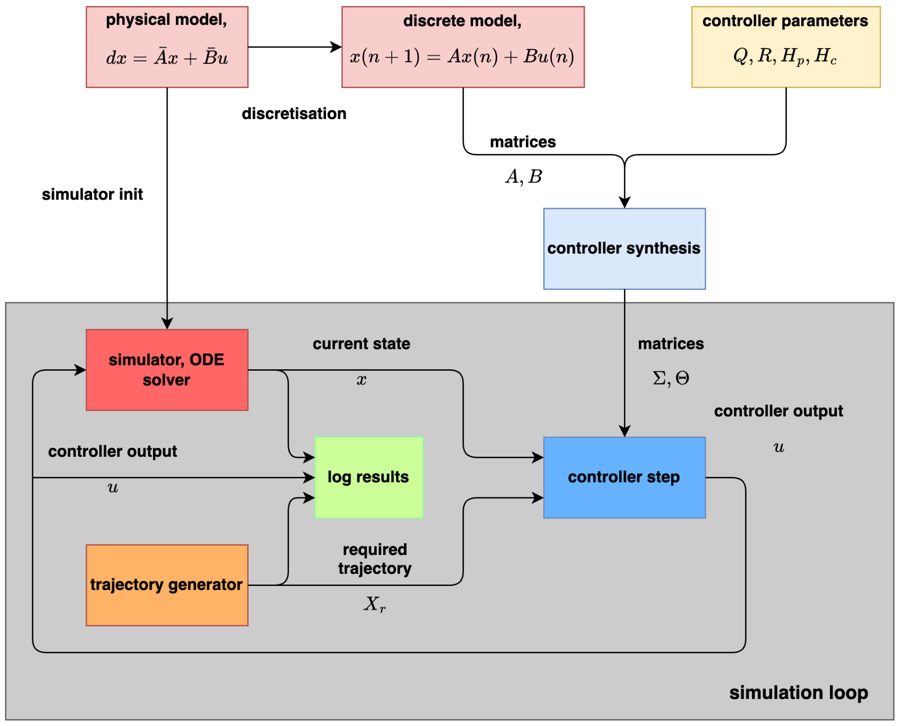


Figure 4.8: Simulation flow chart for MPC tracking.

These parameters fully define the MPC controller.

—

Main Simulation Loop

The main loop executes the simulation in the following steps:

1. The required trajectory X_r is generated procedurally, assuming the current time step and a total horizon length H_p.
2. The current state $x(n)$ is read from the simulator.
3. The controller computes the optimal control input $u(n)$.
4. The physical simulator performs one RK4 integration step, producing the next state $x(n+1)$.
5. Data for X_r, x, and u are stored for later analysis and plotting.

—

Simulation Results

The first result, shown in figure 4.9, presents a step response for an LQR controller used as a baseline. The controller output is plotted on first chart, labeled as **input X**. Note the output is saturated, and clamped into u_{max} value. In middle chart is plotted servo position labeled as **position X**, red color is plotted required value x_r, cyan color is plotted real observed system output. For completness we plot also servo velocity on bottom chart, labeled as **velocity X**. This velocity is not controlled - matrix Q is giving zero weights for velocity terms, however, natural behaviour is, as soon position is at setpoint, velocity is zero. The controller begins acting only after the reference x_r changes from 0 to 1. LQR control is purely reactive — it acts based on the current error between $x(n)$ and the instantaneous reference $x_r(n)$.

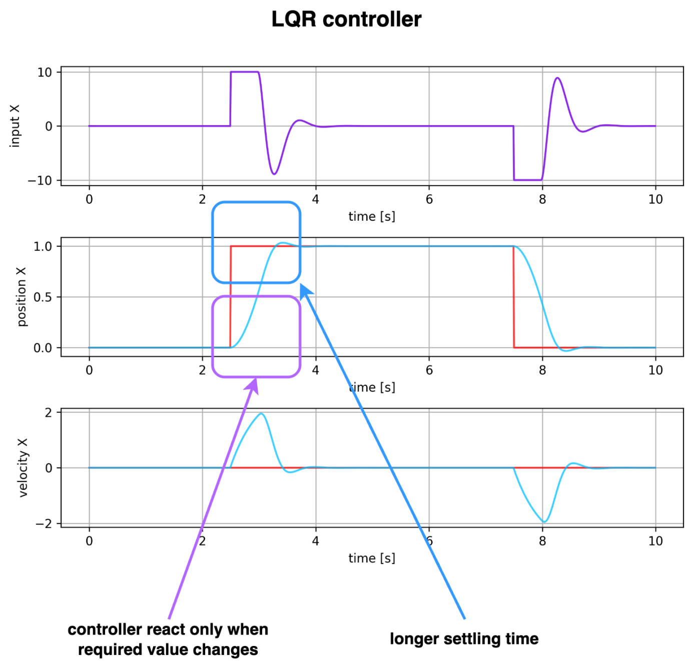


Figure 4.9: Step response for LQR control.

The MPC controller, on the other hand, receives not only the current reference but the **entire sequence** of future reference states x_r over the prediction horizon $H_p = 64$. This allows MPC to **anticipate** future motion and adjust its control signal proactively rather than reactively. The result is smoother control effort, reduced overshoot, and more accurate trajectory tracking.

This behavior is shown in figure 4.10, where **MPC begins acting before the reference step occurs** — the control anticipates the upcoming change.

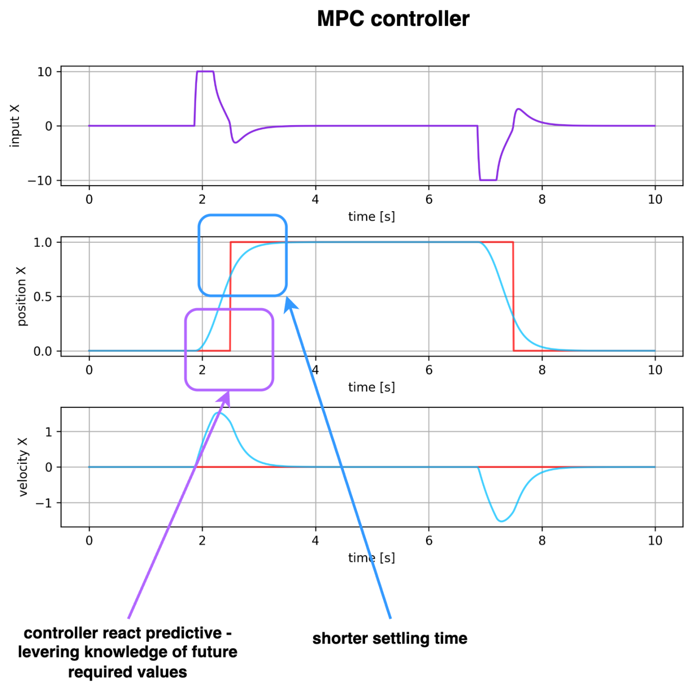


Figure 4.10: Step response for MPC control.

—

Discussion and Benefits

In summary, the example highlights several advantages of Model Predictive Control:

- **Predictive behavior:** MPC optimizes future control inputs using knowledge of the reference trajectory, rather than reacting only to current error.

- **Smooth actuation:** Because control inputs are planned ahead, actuator signals are smoother and exhibit less oscillation.

- **Constraint handling:** MPC naturally incorporates input or state constraints (such as actuator limits or safety zones) into the optimization.
- **Improved tracking performance:** MPC reduces steady-state error and overshoot while maintaining robustness against model imperfections.

Thus, even for a simple system, MPC demonstrates superior performance and provides a solid foundation for more complex multi-variable or nonlinear systems.