

MPC without MSc



Michal Chovanec, PhD.

November 9, 2025

Chapter 1

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 1.1 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 $\hat{x}(n+1), \hat{x}(n+2), \dots, \hat{x}(n+H_p)$ 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 1.2. Note in general model can be imperfect, 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.

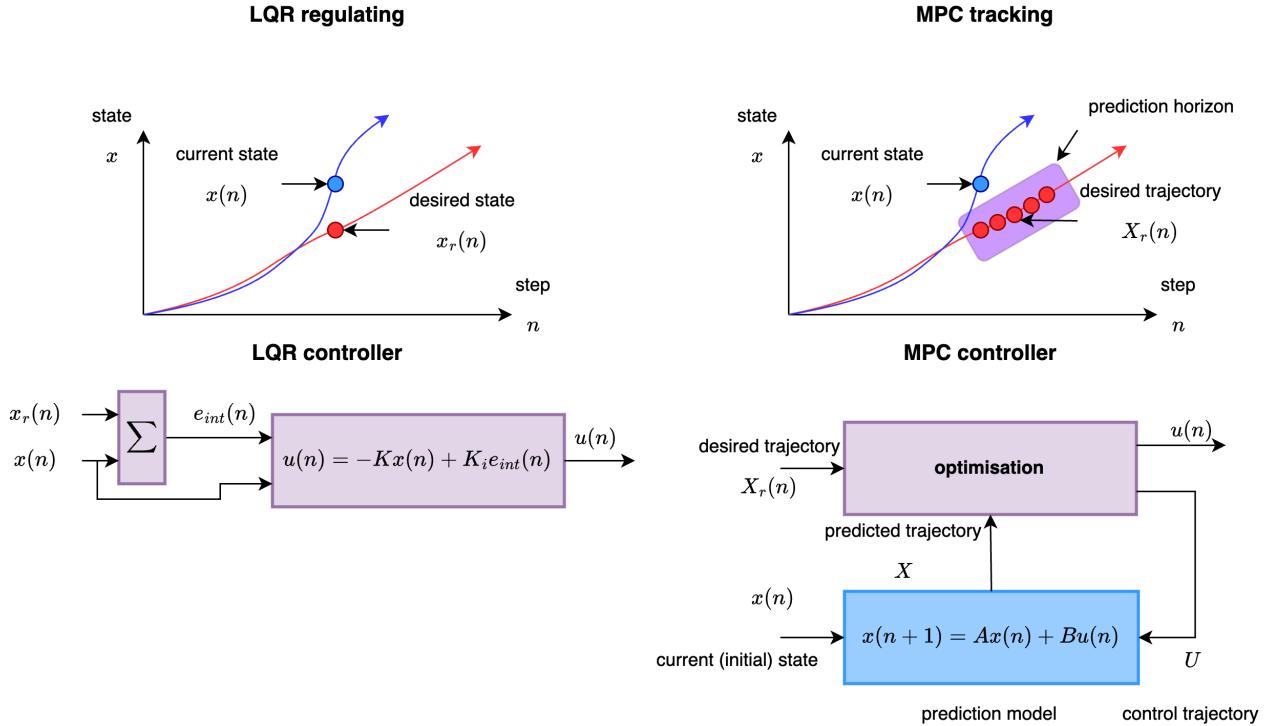


Figure 1.1: Comparison of LQR and MPC for robot trajectory tracking.

In following steps, we derive **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.

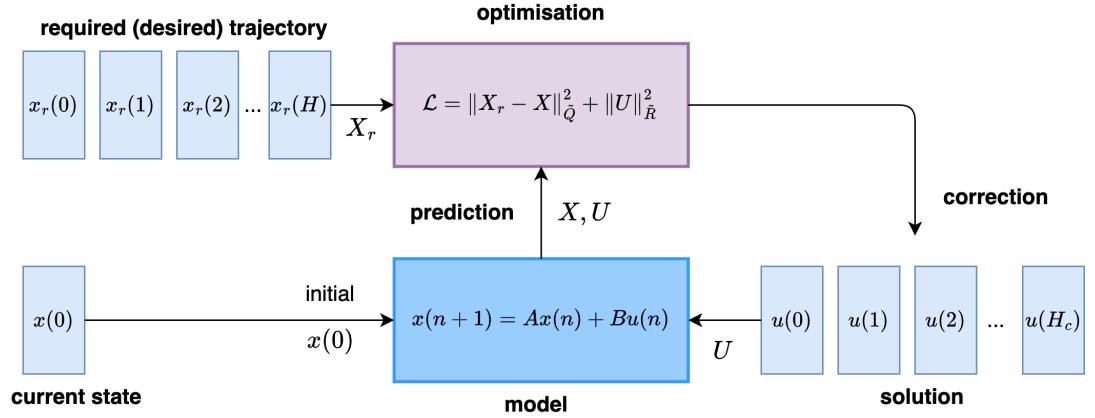


Figure 1.2: MPC algorithm working principle

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

$$x(n+1) = Ax(n) + Bu(n). \quad (1.1)$$

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 $H_p > H_c$.

1.1 Stacked system formulation

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

$$X = \begin{bmatrix} x(n+1) \\ x(n+2) \\ \vdots \\ x(n+H_p) \end{bmatrix}, \quad (1.2)$$

of total dimension $H_p N \times 1$.

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

$$U = \begin{bmatrix} u(n) \\ u(n+1) \\ \vdots \\ u(n+H_c-1) \end{bmatrix}, \quad (1.3)$$

of size $H_c M \times 1$.

The **quadratic cost function** is

$$\mathcal{L}(U) = (X_r - X)^\top \tilde{Q} (X_r - X) + U^\top \tilde{R} U, \quad (1.4)$$

subject to the system dynamics constraint

$$x(n+1) = Ax(n) + Bu(n). \quad (1.5)$$

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

$$\tilde{Q} = \begin{bmatrix} Q & & & \\ & Q & & \\ & & \ddots & \\ & & & Q \end{bmatrix}, \quad \tilde{R} = \begin{bmatrix} R & & & \\ & R & & \\ & & \ddots & \\ & & & R \end{bmatrix}. \quad (1.6)$$

Their dimensions are $\tilde{Q} \in \mathbb{R}^{H_p N \times H_p N}$ and $\tilde{R} \in \mathbb{R}^{H_c M \times H_c M}$.

1.2 Prediction model formalism

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

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

This compactly becomes

$$X = \Psi x(n) + \Theta U, \quad (1.7)$$

where

$$\Psi = \begin{bmatrix} A^1 \\ A^2 \\ \vdots \\ A^{H_p} \end{bmatrix}, \quad \Theta_{ij} = \begin{cases} A^{i-j} B, & \text{if } i \geq j, \\ 0, & \text{otherwise.} \end{cases} \quad (1.8)$$

Ψ has dimensions $H_p N \times N$ and Θ has dimensions $H_p N \times H_c M$.

1.3 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 1.3 illustrates how the matrices Ψ and Θ

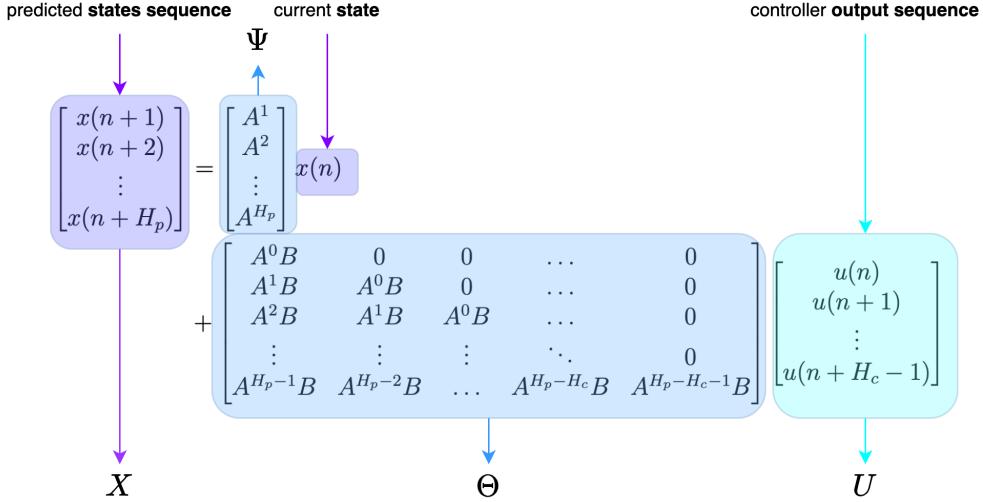


Figure 1.3: Prediction model representation.

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~1.4. 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:

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

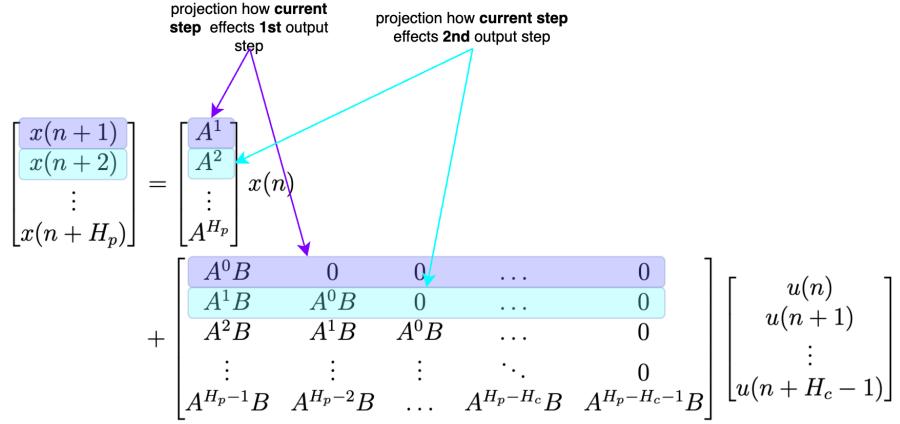


Figure 1.4: Effect of Ψ and Θ matrix terms.

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 $x_a(n+2)$:

$$\begin{aligned} x_a(n+1) &= Ax(n), \\ x_a(n+2) &= Ax_a(n+1) = A^2x(n). \end{aligned}$$

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:

$$\begin{aligned} x_b(n+1) &= Bu(n), \\ x_b(n+2) &= ABu(n) + Bu(n+1). \end{aligned}$$

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 1.5.

Consider the **first control input** $u(n)$, marked in purple in the figure. This term is projected into all future states $x(n+1), x(n+2), \dots$, 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).

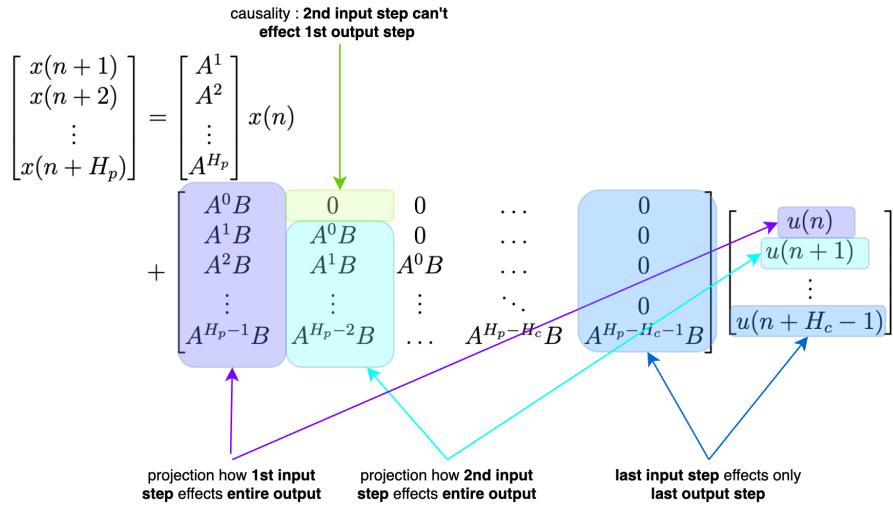


Figure 1.5: Effect of Θ matrix terms on predicted states.

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.

1.4 Optimization problem

Substitute the predicted state into the cost function:

$$\mathcal{L}(U) = (X_r - \Psi x(n) - \Theta U)^\top \tilde{Q} (X_r - \Psi x(n) - \Theta U) + U^\top \tilde{R} U. \quad (1.9)$$

Define the state-reference residual

$$S = X_r - \Psi x(n), \quad (1.10)$$

to obtain

$$\mathcal{L}(U) = U^\top \tilde{R} U + (S - \Theta U)^\top \tilde{Q} (S - \Theta U). \quad (1.11)$$

Expanding and collecting terms:

$$\mathcal{L}(U) = U^\top (\tilde{R} + \Theta^\top \tilde{Q} \Theta) U - 2U^\top \Theta^\top \tilde{Q} S + S^\top \tilde{Q} S. \quad (1.12)$$

The derivative terms are:

$$\frac{\partial \mathcal{L}}{\partial U} : \quad (1.13)$$

$$\begin{aligned} \frac{\partial U^\top \tilde{R} U}{\partial U} &= 2\tilde{R} U \\ \frac{\partial U^\top \Theta^\top \tilde{Q} \Theta U}{\partial U} &= 2\Theta^\top \tilde{Q} \Theta U \\ \frac{\partial -2U^\top \Theta^\top \tilde{Q} S}{\partial U} &= -2\Theta^\top \tilde{Q} S \\ \frac{\partial S^\top \tilde{Q} S}{\partial U} &= 0 \end{aligned} \quad (1.14)$$

1.5 Analytical solution

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

$$\frac{\partial \mathcal{L}}{\partial U} = 2(\tilde{R} + \Theta^\top \tilde{Q} \Theta) U - 2\Theta^\top \tilde{Q} S = 0. \quad (1.15)$$

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

$$U^* = (\tilde{R} + \Theta^\top \tilde{Q} \Theta)^{-1} \Theta^\top \tilde{Q} S, \quad (1.16)$$

which is equivalent to

$$U^* = (\tilde{R} + \Theta^\top \tilde{Q} \Theta)^{-1} \Theta^\top \tilde{Q} (X_r - \Psi x(n)). \quad (1.17)$$

1.6 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.
-

1.7 Algorithm implementation

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

$$\Sigma = (\tilde{R} + \Theta^\top \tilde{Q} \Theta)^{-1} \Theta^\top \tilde{Q}. \quad (1.18)$$

At each control step:

$$E(n) = X_r(n) - \Psi x(n), \quad (1.19)$$

$$U(n) = \Sigma E(n), \quad (1.20)$$

$$u(n) = \text{first } M \text{ elements of } U(n). \quad (1.21)$$

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~1.6.

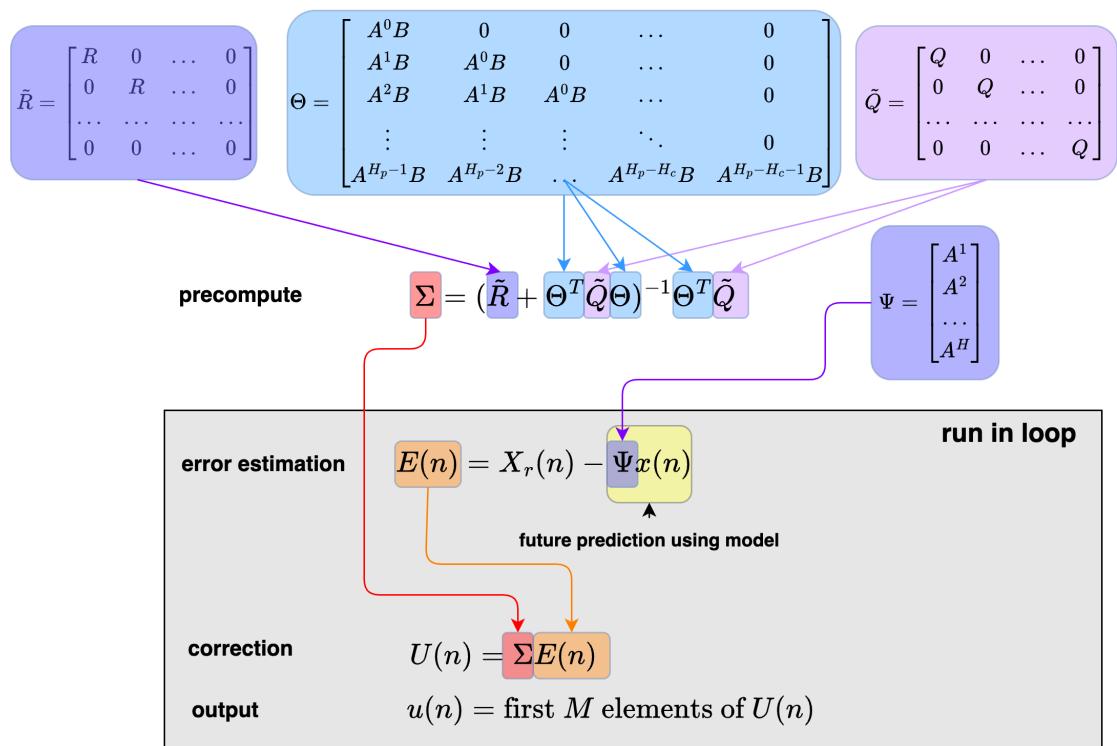


Figure 1.6: Block diagram of unconstrained MPC algorithm

Python code example

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

```

1  """
2  A: (n_x, n_x)
3  B: (n_x, n_u)
4  Q: (n_x, n_x) (state cost)
5  R: (n_u, n_u) (input cost)
6  prediction_horizon : Hp, how many future states
7  control_horizon    : Hc, how many future inputs we optimize;
8          typically <= Hp
9  """
10 def __init__(self, A, B, Q, R, prediction_horizon=16, control_horizon=4, u_max=1e10):
11
12     self.A      = A
13     self.B      = B
14     self.nx     = A.shape[0]
15     self.nu     = B.shape[1]
16     self.Hp     = prediction_horizon
17     self.Hc     = control_horizon
18     self.u_max  = u_max
19
20     # 1, build Phi and Theta
21     self.Phi, self.Theta = self._init_matrices(A, B, self.Hp, self.Hc)
22
23     # 2, build augmented tilde Q and tilde R, block-diagonal
24     self.Q_aug = numpy.kron(numpy.eye(self.Hp), Q)
25     self.R_aug = numpy.kron(numpy.eye(self.Hc), R)
26
27     # Precompute solver matrices: G and Sigma
28     G = self.Theta.T @ self.Q_aug @ self.Theta + self.R_aug
29
30     # use solve later for stability; but precompute factorization if desired
31     # here we compute Sigma by solving G Sigma^T = Theta^T Q_aug (do via solve)
32     # Sigma has shape (n_u*Hc, n_x*Hp)
33     # Solve H @ Sigma = Theta.T @ Q_aug
34     # Sigma = numpy.linalg.solve(H, Theta.T @ Q_aug)
35     self.Sigma = numpy.linalg.solve(G, self.Theta.T @ self.Q_aug)
36     self.Sigma0 = self.Sigma[:self.nu, :]
37
38
39 def _init_matrices(self, A, B, Hp, Hc):
40     nx = A.shape[0]
41     nu = B.shape[1]
42     # precompute A powers: A^0 ... A^Hp
43     A_pows = [numpy.eye(nx)]
44     for i in range(1, Hp + 1):
45         A_pows.append(A_pows[-1] @ A)
46
47     # Phi: (nx*Hp, nx) stacked [A; A^2; ...; A^Hp]
48     Phi = numpy.zeros((nx * Hp, nx))
49     for i in range(Hp):
50         Phi[i * nx:(i + 1) * nx, :] = A_pows[i + 1] # A^(i+1)
51
52     # Theta: (nx*Hp, nu*Hc) where block (i,j) is A^(i-j) B for i>=j, else 0

```

```

53     Theta = numpy.zeros((nx * Hp, nu * Hc))
54     for i in range(Hp):
55         for j in range(Hc):
56             if i >= j:
57                 # A^{i-j} B
58                 Theta[i * nx:(i + 1) * nx, j * nu:(j + 1) * nu] = A_pows[i - j] @ B
59             else:
60                 # remains zero
61                 pass
62
63     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 $(M, 1)$:

```

1 def forward_traj(self, Xr, x):
2     # residual
3     s = Xr - self.Phi @ x
4
5     # compute only first control
6     u0 = self.Sigma0 @ s
7     u0 = numpy.clip(u0, -self.u_max, self.u_max)
8
9     return u0

```

1.8 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 1.7. We control two input forces, u_x and u_y .

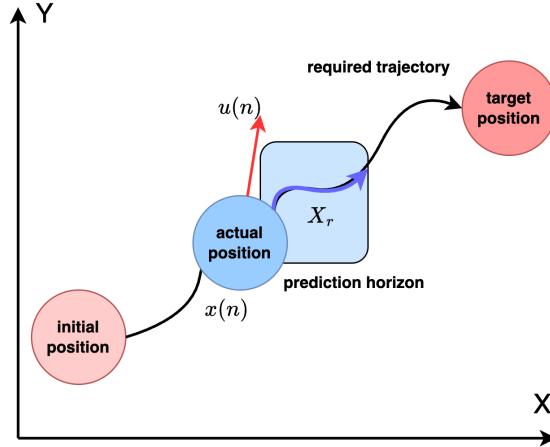


Figure 1.7: Trajectory tracking example.

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:

$$\bar{A} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\frac{1}{\tau} & 0 \\ 0 & 0 & 0 & -\frac{1}{\tau} \end{bmatrix}, \quad \bar{B} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \frac{k}{\tau} & 0 \\ 0 & \frac{k}{\tau} \end{bmatrix}. \quad (1.22)$$

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.

$$A = \begin{bmatrix} 1 & 0 & 0.00990099 & 0 \\ 0 & 1 & 0 & 0.00990099 \\ 0 & 0 & 0.98019802 & 0 \\ 0 & 0 & 0 & 0.98019802 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 5.9406 \times 10^{-3} & 0 \\ 0 & 5.9406 \times 10^{-3} \end{bmatrix}. \quad (1.23)$$

Simulation Setup

The complete simulation workflow is illustrated in figure 1.8.

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

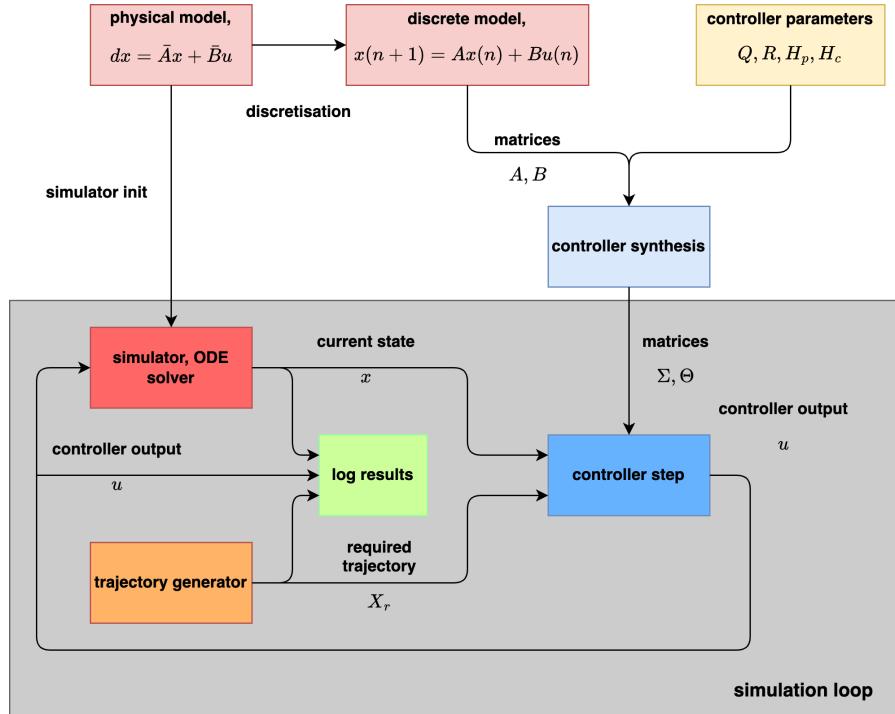


Figure 1.8: Simulation flow chart for MPC tracking.

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 1.9, 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 completeness 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 $x_r(n)$.

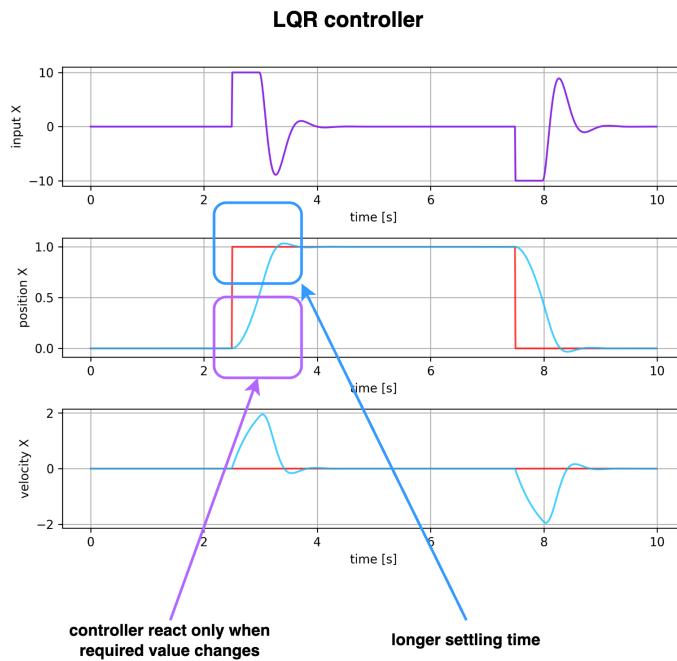


Figure 1.9: Step response for LQR control.

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 1.10, where **MPC begins acting before the reference step occurs** — the control anticipates the upcoming change.

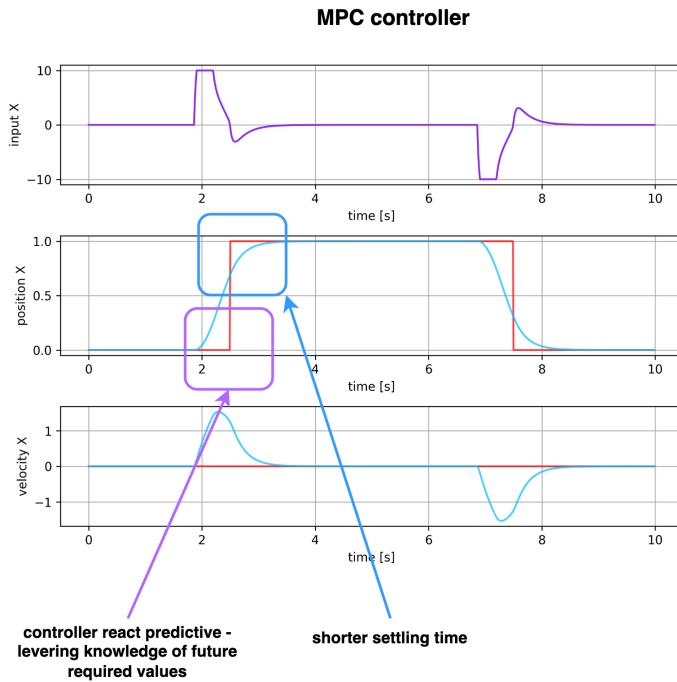


Figure 1.10: Step response for MPC control.

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.