MPC notes

November 8, 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 $\hat{x}(n+1), \hat{x}(n+2), \ldots, \hat{x}(n+H_p)$ 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 1.2. 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.

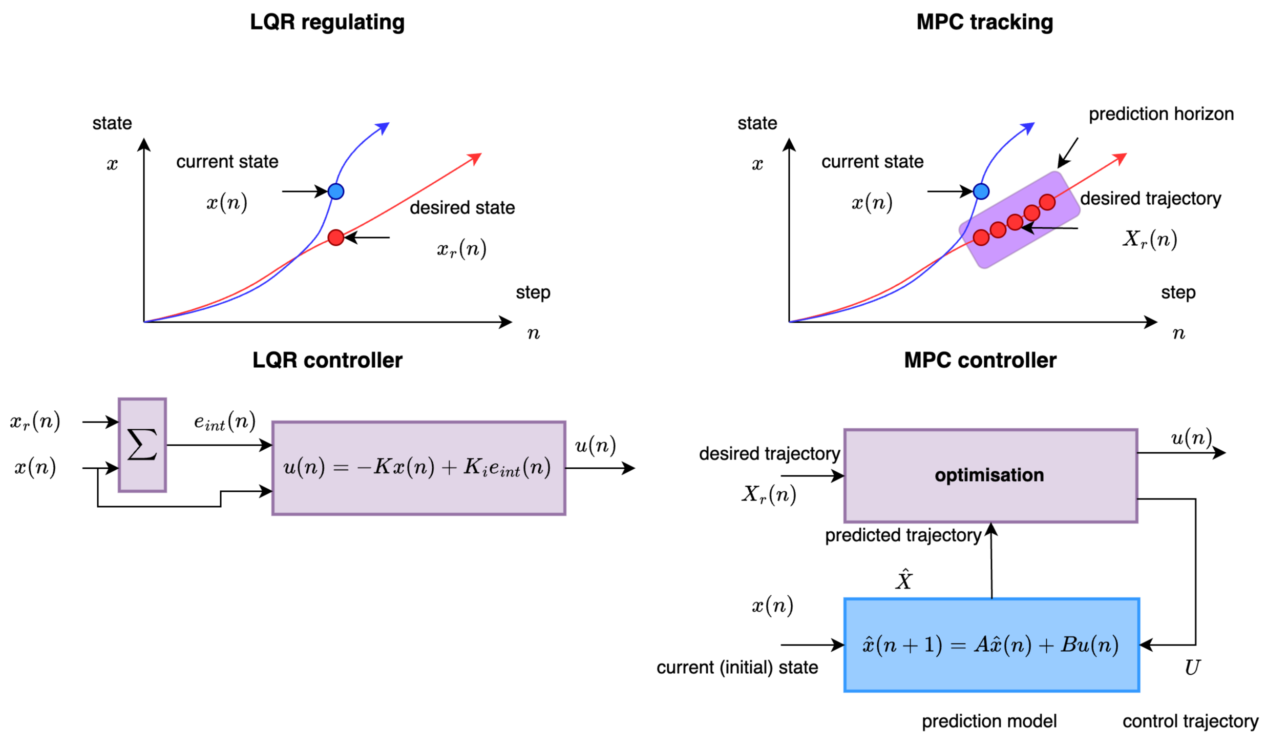


Figure 1.1: Comparison of LQR and MPC for robot trajectory tracking.

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.

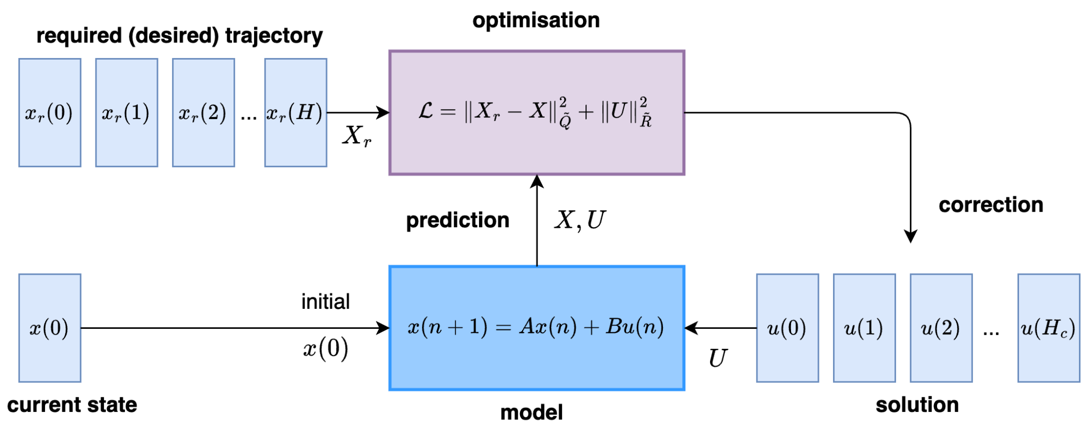


Figure 1.2: MPC algorithm working principle

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

$$x(n+1) = Ax(n) + Bu(n). \tag{1.1}$$

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**

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

of total dimension $H_pN \times 1$.

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

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

of size $H_cM \times 1$.

The **quadratic cost function** is

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

subject to the system dynamics constraint

$$x(n+1) = Ax(n) + Bu(n). \tag{1.5}$$

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

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

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

—

1.2 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

$$
X = \Psi x(n) + \Theta U, \tag{1.7}
$$

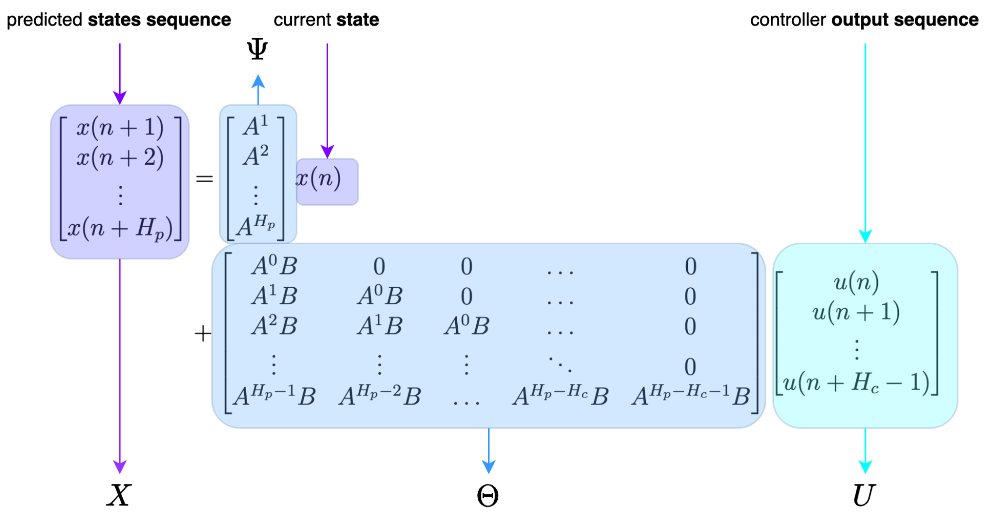
where

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

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

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



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

Figure 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:

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

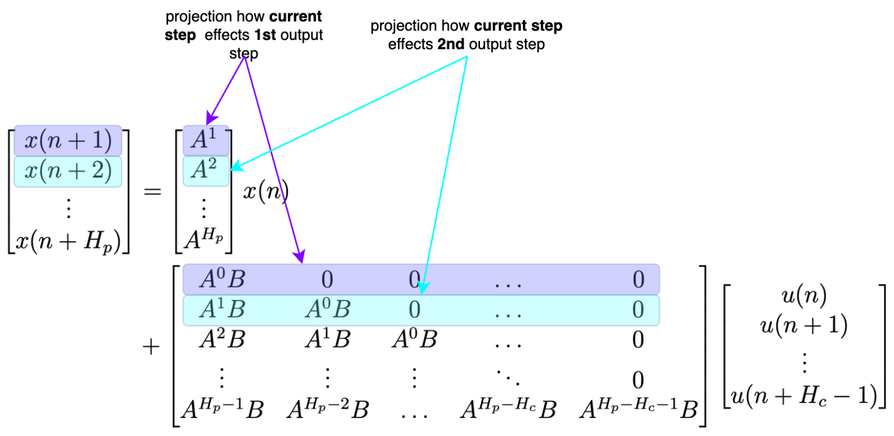

projection how **current step** effects **1st** output step
projection how **current step** effects **2nd** output step
$\begin{bmatrix} x(n+1) \\ x(n+2) \\ \vdots \\ x(n+H_p) \end{bmatrix} = \begin{bmatrix} A^1 \\ A^2 \\ \vdots \\ A^{H_p} \end{bmatrix} x(n) + \begin{bmatrix} A^0B & 0 & 0 & \dots & 0 \\ A^1B & A^0B & 0 & \dots & 0 \\ A^2B & A^1B & A^0B & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ A^{H_p-1}B & A^{H_p-2}B & \dots & A^{H_p-H_c}B & A^{H_p-H_c-1}B \end{bmatrix} \begin{bmatrix} u(n) \\ u(n+1) \\ \vdots \\ u(n+H_c-1) \end{bmatrix}$

Figure 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 $x_a(n+2)$:

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

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:

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

—

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 1.5.

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

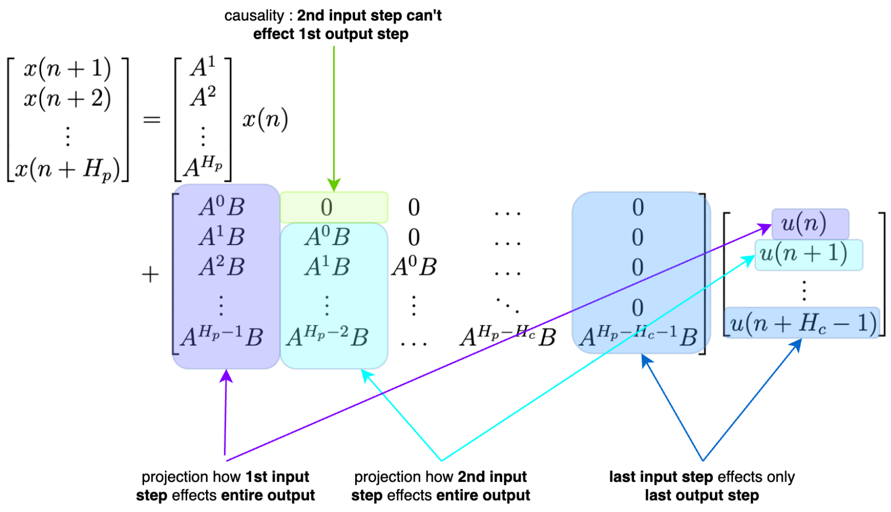

causality : 2nd input step can't effect 1st output step
$$\begin{bmatrix} x(n+1) \\ x(n+2) \\ \vdots \\ x(n+H_p) \end{bmatrix} = \begin{bmatrix} A^1 \\ A^2 \\ \vdots \\ A^{H_p} \end{bmatrix} x(n) + \begin{bmatrix} A^0B & 0 & 0 & \ldots & 0 \\ A^1B & A^0B & 0 & \ldots & 0 \\ A^2B & A^1B & A^0B & \ldots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

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

Define the state–reference residual

$$S = X_r - \Psi x(n), \tag{1.10}$$

to obtain

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

Expanding and collecting terms:

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

The derivative terms are:

$$\frac{\partial J}{\partial U} : \tag{1.13}$$

$$\begin{aligned} \frac{\partial U^T \tilde{R} U}{\partial U} &= 2\tilde{R}U \\ \frac{\partial U^T \Theta^T \tilde{Q} \Theta U}{\partial U} &= 2\Theta^T \tilde{Q} \Theta U \\ \frac{\partial {-2}U^T \Theta^T \tilde{Q} S}{\partial U} &= -2\Theta^T \tilde{Q} S \\ \frac{\partial S^T \tilde{Q} S}{\partial U} &= 0 \end{aligned} \tag{1.14}$$

—

1.5 Analytical solution

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

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

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

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

which is equivalent to

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

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

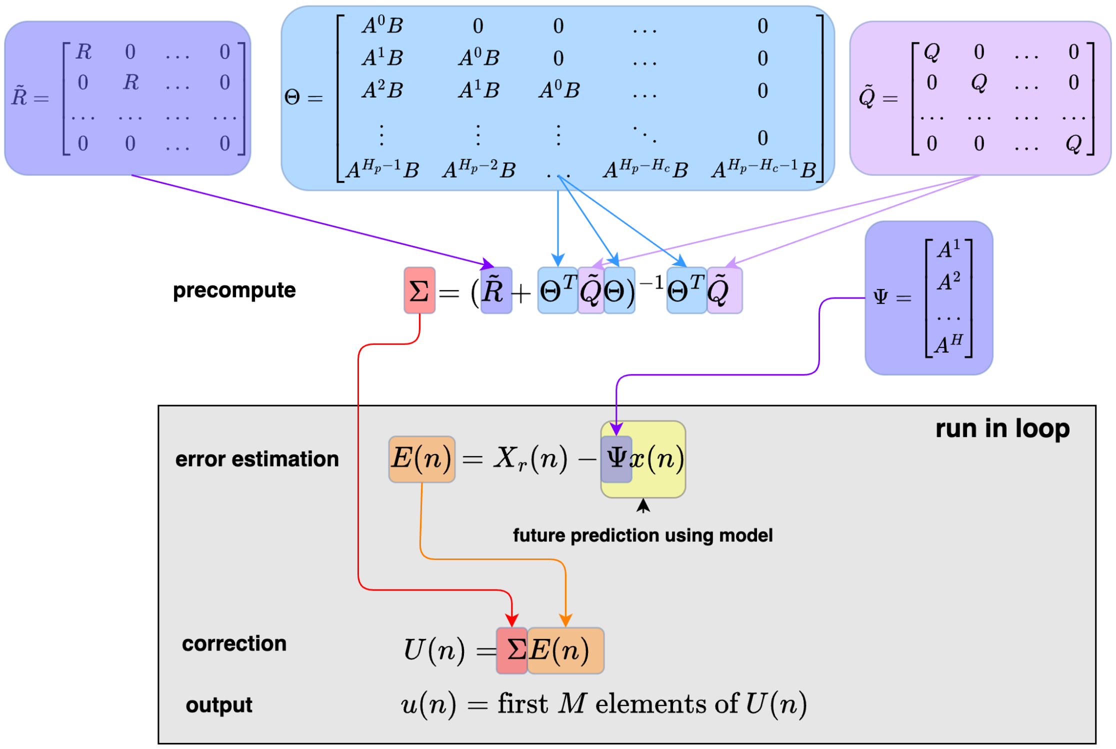
Since all matrices are constant for a given system, we can precompute

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

At each control step:

$$E(n) = X_r(n) - \Psi x(n), \tag{1.19}$$
$$U(n) = \Sigma E(n), \tag{1.20}$$
$$u(n) = \text{first } M \text{ elements of } U(n). \tag{1.21}$$

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.



$\tilde{R} = \begin{bmatrix} R & 0 & \dots & 0 \\ 0 & R & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 0 \end{bmatrix}$
$\Theta = \begin{bmatrix} A^0B & 0 & 0 & \dots & 0 \\ A^1B & A^0B & 0 & \dots & 0 \\ A^2B & A^1B & A^0B & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ A^{H_p-1}B & A^{H_p-2}B & \dots & A^{H_p-H_c}B & A^{H_p-H_c-1}B \end{bmatrix}$
$\tilde{Q} = \begin{bmatrix} Q & 0 & \dots & 0 \\ 0 & Q & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & Q \end{bmatrix}$
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 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 :

```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._build_prediction_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 _build_prediction_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
```

—

1.8 Tracking problem example

We assume simple 2D moving sphere with inertia, system is basically two servos with inertia, having two paramters, time constant τ and some amplification k. We assume continuous state space model $dx = Ax + Bu$.

State x is then basically position and velocity :

$$x = \begin{bmatrix} x_{pos} \\ y_{pos} \\ x_{vel} \\ y_{vel} \end{bmatrix} \tag{1.22}$$

System dynamics matrices A and B are :

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\frac{1}{\tau} & 0 \\ 0 & 0 & 0 & -\frac{1}{\tau} \end{bmatrix} \tag{1.23}$$

$$B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \frac{k}{\tau} & 0 \\ 0 & \frac{k}{\tau} \end{bmatrix} \tag{1.24}$$

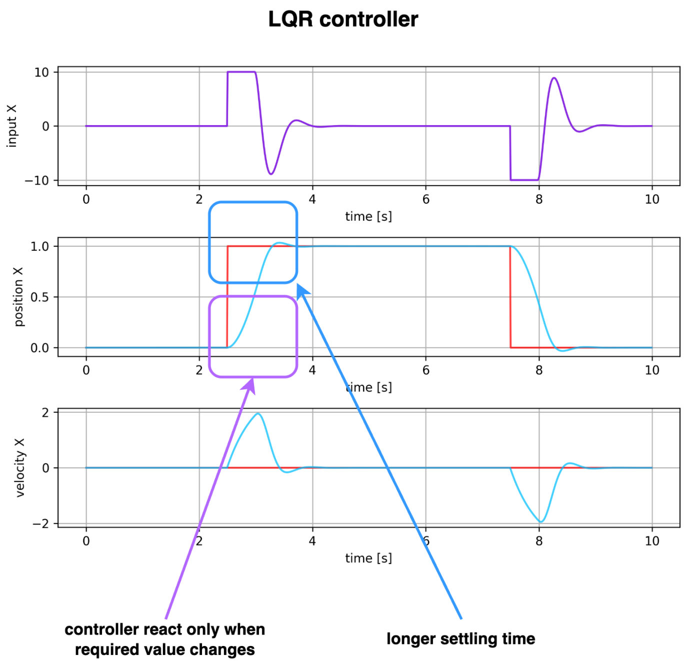


Figure 1.7: LQR response for servo control

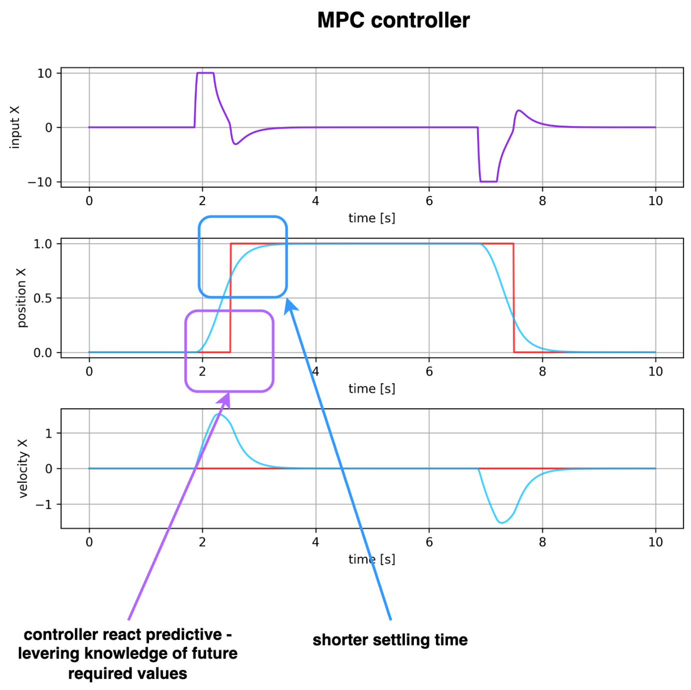


Figure 1.8: MPC response for servo control