Hands-On Control and Algorithms for Robots

Michal Chovanec, PhD.

Chapter 1

System identification

In this chapter we leverage state space models for describing mathematical model of robot. We consider differential drive robot, with two independently controller wheels. For comfort the state of robot will be its linear distance x, linear velocity v, robot angle θ and angular rate ω. This coordinate system differs from more used cartesian, however, it allow us to avoid non-linear model (containing sine, cosine rotation matrix), and further we can use fully linear state space model, and linear controller.

Note, the upprt level of controll, e.g. path planener, will probaly still needs to works within cartesian coordinates, however, this controll loop is running relatively slow, so the excesive computations doesn't matter.

Chapter 2

Linear quadratic regulator

LQR controller is standart baseline for regulating systems with multiple input and multiple outputs. Controller requires knowledge of system dynamics (state space model), and cost function weights - which are design options.

TODO : explain LQR more, and also applications, why it is used as baseline, why not PID

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

This system have two inputs, applied force $u_x(n)$, $u_y(n)$. State is 4 dimensional, position and velocity for each spatial dimension. As demonstrated on figure 2.1, the state vector is $x(n)$, we also marked velocity term as $v(n)$ (which is part of state), and control input $u(n)$, which have to act conteract the velocity and intertia, to stear system into desired state $x_r(n)$.

This is system with multiple outputs and multiple inputs :

$$x(n) = \begin{bmatrix} p_x(n) \\ p_y(n) \\ v_x(n) \\ v_y(n) \end{bmatrix} \tag{2.1}$$

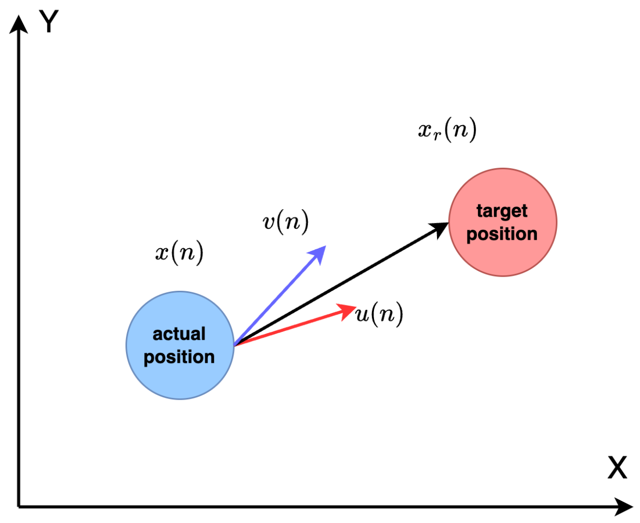

Figure 2.1: 2D position control

$$u(n) = \begin{bmatrix} u_x(n) \\ u_y(n) \end{bmatrix} \tag{2.2}$$

Note the ordering of state variables is completely arbitrary. Here we first define position and then velocity terms.

The intuition behing LQR is : can we find linear projection, a mapping from difference of current state x and desired state x_r into control vector u ? First we define difference between desired and observed state as error, $e(n) = x_r(n) - x(n)$. What is simplest operator which can do mapping, transformation from error space into control, action space ? The simplest operation is of course matrix multiplication, giving us forumula of LQR :

$$e(n) = x_r(n) - x(n) \tag{2.3}$$
$$u(n) = -Ke(n) \tag{2.4}$$

The matrix K is simple gain matrix, taking each state terms, putting some weights to them, and by linear compbinaiton of error singnal $e(n)$ mapping to output u.

2.1 Derivate of LQR

Now we have to find solution of matrix K. 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**, $M \times M$.

Controller stears system into desired state, required state, marked as $x_r(n)$, of same shape as state $x(n)$, i.e. there is minimalization of error magnitude $e(n) = x_r(n) - x(n)$.

The discrete-time linear system dynamics are

$$x(n+1) = Ax(n) + Bu(n). \tag{2.5}$$

Corresponding LQR problem is given as minimization of quadtratic loss
TODO : fix this cost function
The **quadratic cost function** is

$$\mathcal{L}(u) = x(n)\tilde{Q}x(n) + u(n)^T\tilde{R}u(n), \tag{2.6}$$

subject to the system dynamics constraint

$$x(n+1) = Ax(n) + Bu(n). \tag{2.7}$$

TODO : validate this writing, here we wanna explain in very simple what quadtratic loss is, how to imagine it.

The following figure deciphers meaning of quadratic loss 2.2. We demonstrate 4 dimensional state vector x and 2 dimensional control input u. Therefore matrix Q is 4×4, and matrix R is 2×2. In figure is captured one step of loss $\mathcal{L}(n)$. The terms $x(n)$ multiplied by itself tranposed is basically quadratic term, well known as MSE loss, same for control variable u. Matrices Q and R are weighting state elements terms. Some state variables may have bigger weight for opitmisation. Same for control variable u, if the amplitude of control signal matters more (e.g. fuel consumption, maximum motor power), terms in R are bigger, if we don't care about u amplitude, terms in R can be very small.

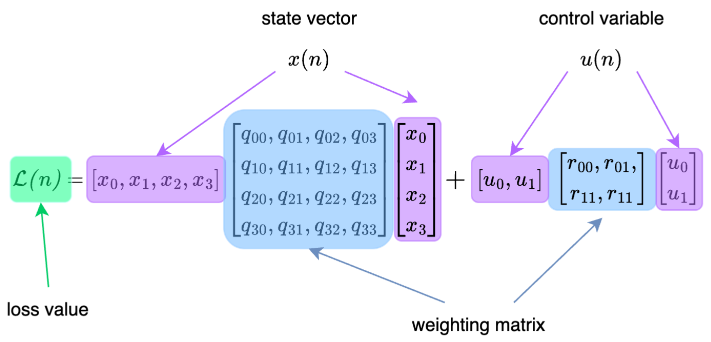
state vector
$x(n)$
control variable
$u(n)$
$\mathcal{L}(n) = [x_0, x_1, x_2, x_3] \begin{bmatrix} q_{00}, q_{01}, q_{02}, q_{03} \\ q_{10}, q_{11}, q_{12}, q_{13} \\ q_{20}, q_{21}, q_{22}, q_{23} \\ q_{30}, q_{31}, q_{32}, q_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}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

Figure 2.2: Quadratic loss meaning

Chosing of matrices Q and R are design choices, mostly those matrices are diagonal only, meaning all terms have independent weights.

TODO : explain this more, maybe derivative

Solution of this optimisation problem is by solving discrete algebraic Riccati equation, Which leads to linear quadratic regulator, with control matrix K :

$$u(n) = -Kx(n) \tag{2.8}$$

Not this always stears system to zero state - this formulation not allows arbitrary setting of x_r, If system contains integral terms (e.g. servo, positional mechanism i.e. for zero error requiring zero control output u), the controller can be formulated as (with same matrix K)

$$u(n) = K(x_r(n) - x(n)) \tag{2.9}$$

In the figure 2.3 we presents full basic LQR algorithm. The system matrices A and B, together with weight matrices Q and R are used as input into discrete algebraic riccati equation solver. This is computed only once, giving us control matrix K. In controller loop running in real time we first compute error as $e(n) = x_r(n) - x(n)$, then we apply control law $u(n) = Ke(n)$. Note we ommits minus sign here, bacause it is laready part of error signal $e(n)$.

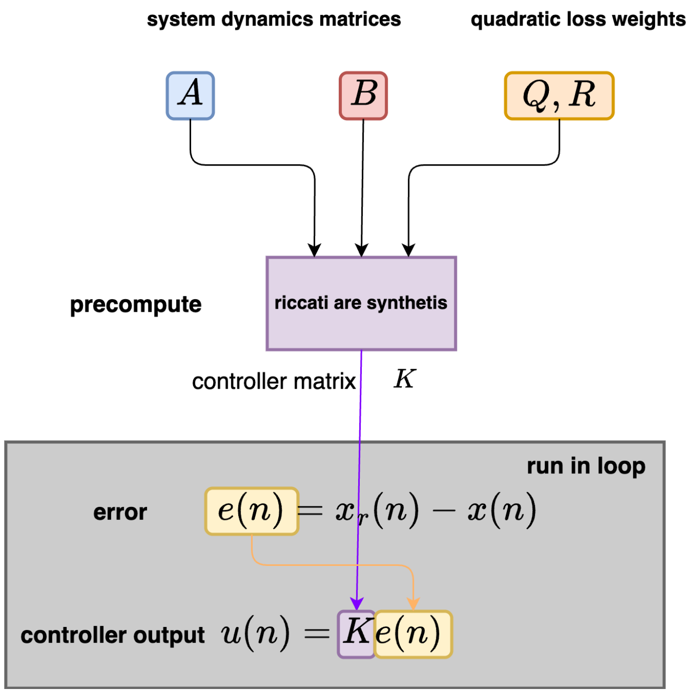

Figure 2.3: LQR algorithm

2.2 Adding integral action

Our controller is missing integral term, which is necessary to remove steady state error. This is must have, if there is constant disturbance. Also systems which doesen't have integral terms in nature.

TODO : explain, why integral terms helps to remove steady state error
That's why we augment system dynamics by integral terms :

$$\tilde{A} = \begin{bmatrix} A & 0 \\ I & I \end{bmatrix} \tag{2.10}$$

Also matrix Q should be augmented

$$\tilde{Q} = \begin{bmatrix} Q & 0 \\ 0 & \kappa Q \end{bmatrix} \tag{2.11}$$

Where κ is giving weights to integral terms.
Solution of this controller is some augmented gain, called $\tilde{K}$:

$$\tilde{K} = \begin{bmatrix} K \\ Ki \end{bmatrix} \tag{2.12}$$

Which we split into two matrices K, Ki.
The full algorithm of LQR controller is captured in the following figure 2.4.

We start with augmenting system matrix A with integral terms, and also for matrix Q. Then we apply discrete algebraic Riccati equation solver, to obtain matrices K, K_i. This is precomputed once.

First we compute error as $e(n) = x_r(n) - x(n)$. The algorithm is introducing integral action e_int as : $e_{int}(n) = e_{int}(n-1) + K_i e(n)$. This is nonconvetional compared to textbooks. This allov us much simpler antiwindup implementation in further steps. Then the control output is computed as $u(n) = Ke(n) + e_{int}(n)$. Note here we ommits minus sign, because it is already embedded in error term, state is there as $-x(n)$.

2.3 Adding antiwindup

Real actuator is always limited. It is easy to compute control output 10 000 volts, if real motor have limit 12V and battery only 16V. Our linear controller doesn't know those constrains.

TODO : this is good example, but needs more grammar check

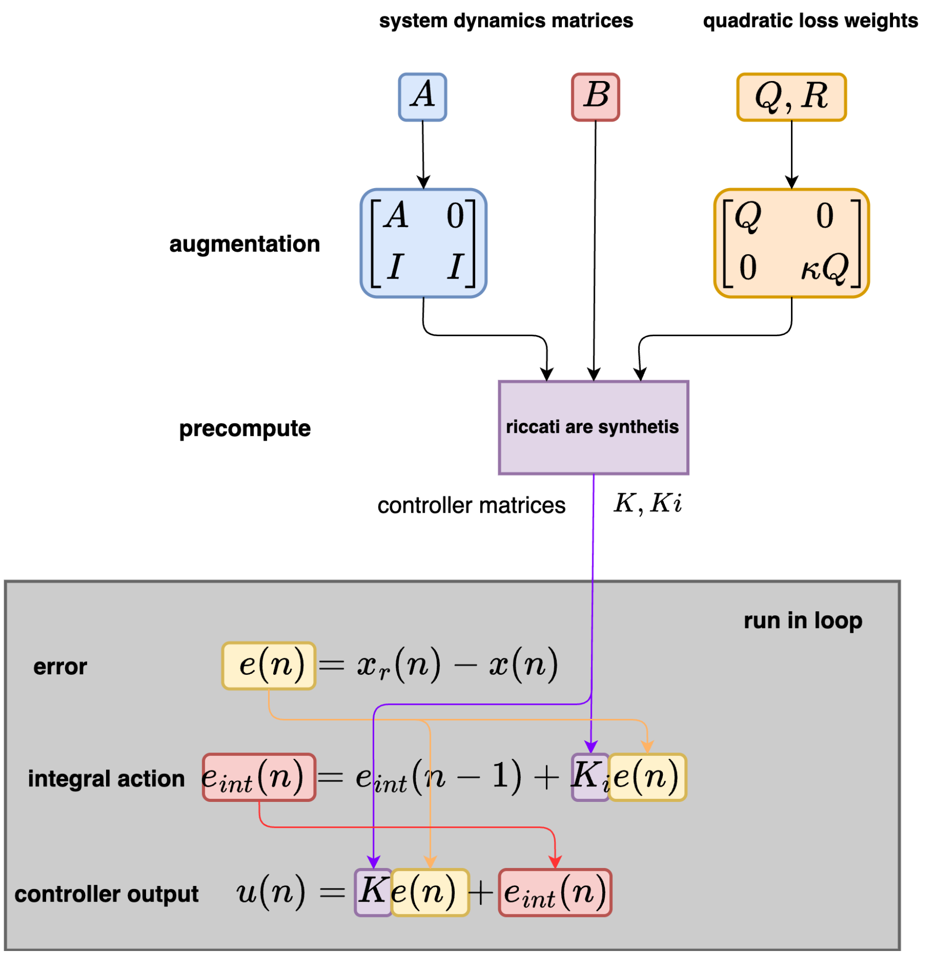

Figure 2.4: LQR algorithm with integral action

Consider our motor accepts maximum 12 volts. There is non zero error and controller computes output 20 volts. Actuator gives only 12 V (e.g. PWM saturation, physical limit ...), anyway, motor is running its best on full power. However, controller things output is 20 volts, and error is still not zero, causing error summation in integral action term. Which increaes controller output even more, e.g. into 25, 30, 40 volts ... but not effecting real output, only $e_{int}(n)$ accumulator. This is called integral windup up, a state when error is not zero, keeping accumualting integral action, but causing not real output change. As soon error reaches zero, the controller heavy overshoot output, because in integral term is accumulated huge value, error beceomes negative, and integrator start decreasing, which takes lot of time. We can observe huge overshoot, or even oscilating, and very long settle time. Not beacuse of wrong control matrices, or wrong system model, but bacause we simply ignored real actuator, which is always limited.

Solution of this problem is antiwindup. In the figure 2.5 we presents simple solution of this problem. Control matrices K and K_i are computed as in previous section.

First we compute candidate of integral error $\hat{e}_{int}(n)$, note, if actuator is non saturated, this will become equal to $e(n)$:

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

The main antwiwindup works as follow : we compute output candidate $\hat{u}$, non saturated, as in previous section : $\hat{u}(n) = Ke(n) + K_i e_{int}(n)$

then we saturate output, simply by clipping into u_{min} and u_{max} range : $u(n) = clip(\hat{u}(n), u_{min}, u_{max})$ this output is used as controller output.

And finally we have to push this information to our integrator :

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

Note the second term : if system is not in saturation (normal conditions), the second term is **zero** and integrator works same as in previous section. As soon the $u(n)$ is saturated, the second term **substract** exact the overshoting part of integral, keeping integral term in limits.

Here we can also observe benefits why we are working with wihin error integral space as $K_i e_{int}(n)$ - the space is equal to action space, and therefore, we can simply define limits u_{min}, u_{max} based on actuator limitations.

TODO : check this statament In case of textbook integral action, where they define error integral as :

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

The error is in state space, don't allowing as simple clipping - we have to use aditional inverse matrix projections.

2.4 Practical implementation

Now we have to put all togother to create single python class, implementing LQR controller.

First we precompute all control matrices in constructor, and store output min max values, here we consider symmetrical clipping. The method **solve** is augmenting matrices, by exteding with integral action terms and solving discrete riccati equation. Main controller is calling in real time controll

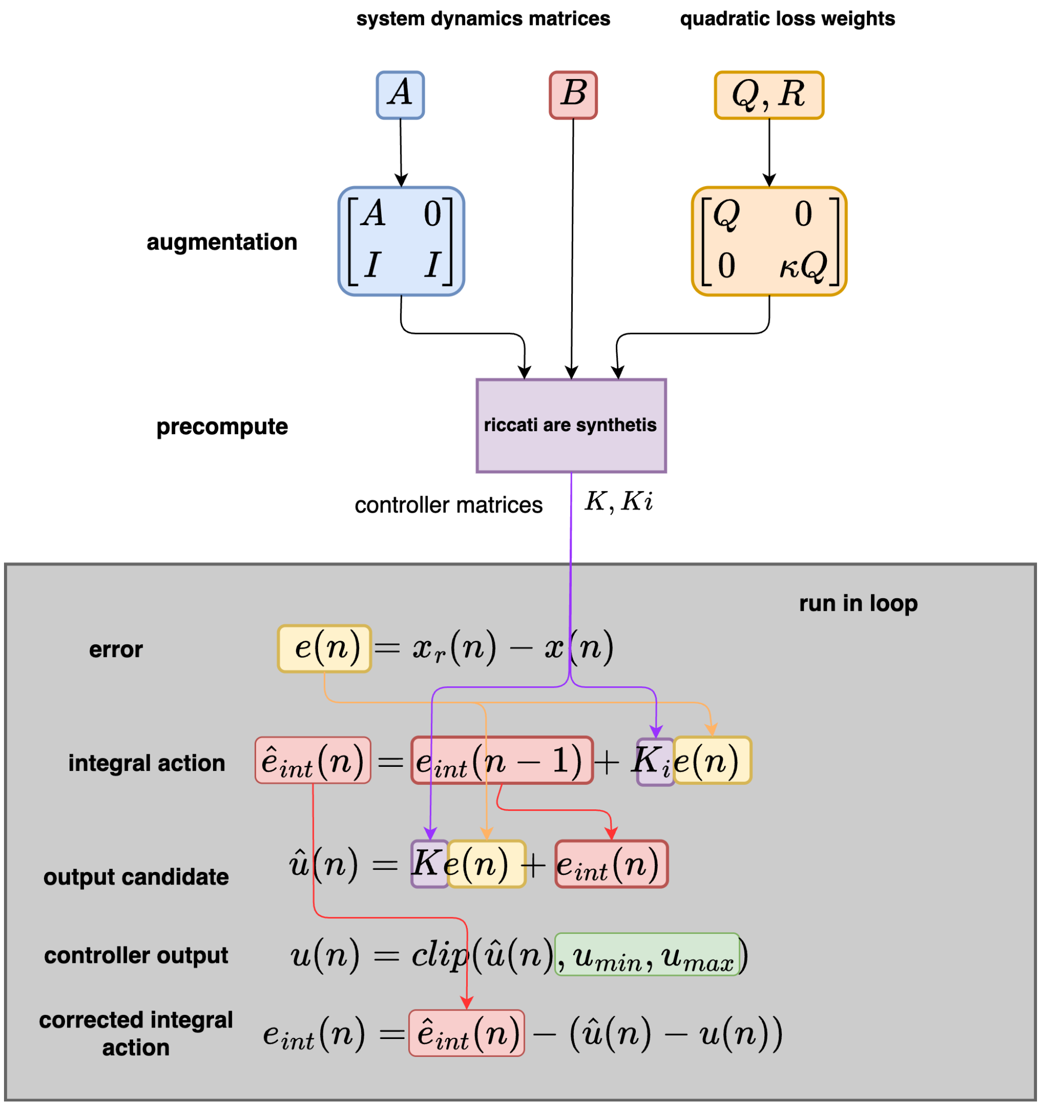
system dynamics matrices
quadratic loss weights
A
B
Q, R
augmentation
$\begin{bmatrix} A & 0 \\ I & I \end{bmatrix}$
$\begin{bmatrix} Q & 0 \\ 0 & \kappa Q \end{bmatrix}$
precompute
riccati are synthetis
controller matrices
K, Ki
run in loop
error
$e(n) = x_r(n) - x(n)$
integral action
$\hat{e}_{int}(n) = e_{int}(n-1) + K_i e(n)$
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 2.5: LQR algorithm with integral action and antiwindup

loop by calling **forward**, which requires desired state x_r current state x, each of shape $(n, 1)$ and current state of integrator, of shape $(m, 1)$. Method returns control output u and new integrator state. The output u is of shape $(m, 1)$.

```
import numpy
import scipy

```

```python
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 controll 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
```

Chapter 3

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 3.1 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 3.2. 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.

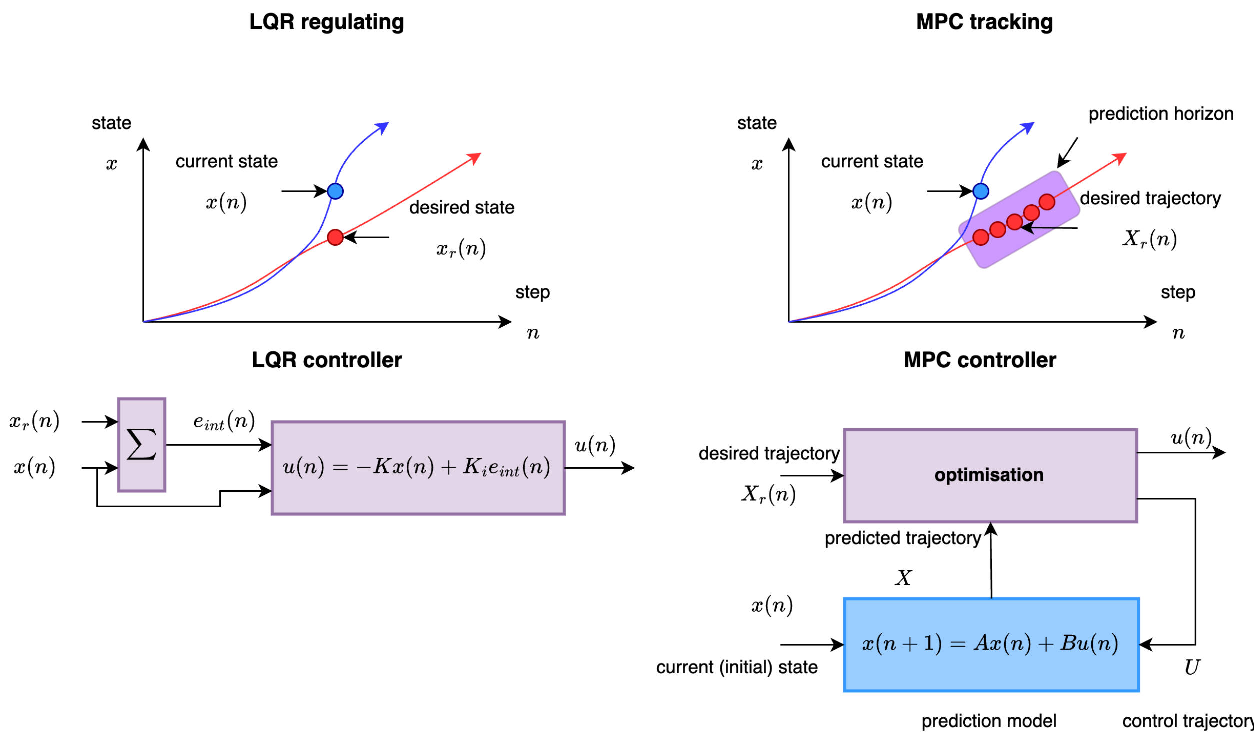

Figure 3.1: Comparison of LQR and MPC for robot trajectory tracking.

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.

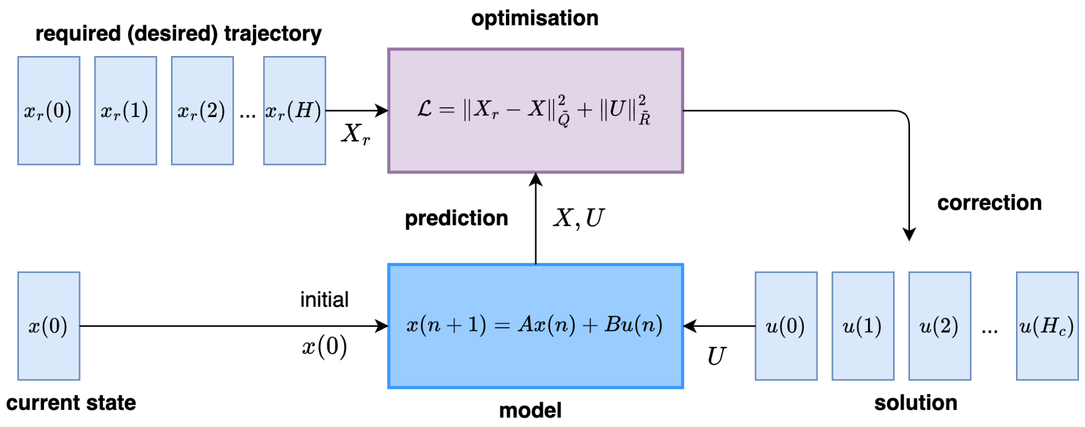

Figure 3.2: MPC algorithm working principle

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

$$x(n+1) = Ax(n) + Bu(n). \tag{3.1}$$

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 $H_p > H_c$.

—

3.1 Stacked system formulation

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

$$X = \begin{bmatrix} x(n+1) \\ x(n+2) \\ \vdots \\ x(n+H_p) \end{bmatrix}, \tag{3.2}$$

of total dimension $H_pN \times 1$.

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

$$U = \begin{bmatrix} u(n) \\ u(n+1) \\ \vdots \\ u(n+H_c-1) \end{bmatrix}, \tag{3.3}$$

of size $H_cM \times 1$.

The **quadratic cost function** is

$$\mathcal{L}(U) = (X_r - X)^\top \tilde{Q}(X_r - X) + U^\top \tilde{R} U, \tag{3.4}$$

subject to the system dynamics constraint

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

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

$$\tilde{Q} = \begin{bmatrix} Q & & & \\ & Q & & \\ & & \ddots & \\ & & & Q \end{bmatrix}, \quad \tilde{R} = \begin{bmatrix} R & & & \\ & R & & \\ & & \ddots & \\ & & & R \end{bmatrix}. \tag{3.6}$$

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

—

3.2 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

$$X = \Psi x(n) + \Theta U, \tag{3.7}$$

where

$$\Psi = \begin{bmatrix} A^1 \\ A^2 \\ \vdots \\ A^{H_p} \end{bmatrix}, \quad \Theta_{ij} = \begin{cases} A^{i-j}B, & \text{if } i \geq j, \\ 0, & \text{otherwise.} \end{cases} \tag{3.8}$$

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

—

3.3 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 3.3 illustrates how the matrices Ψ and Θ

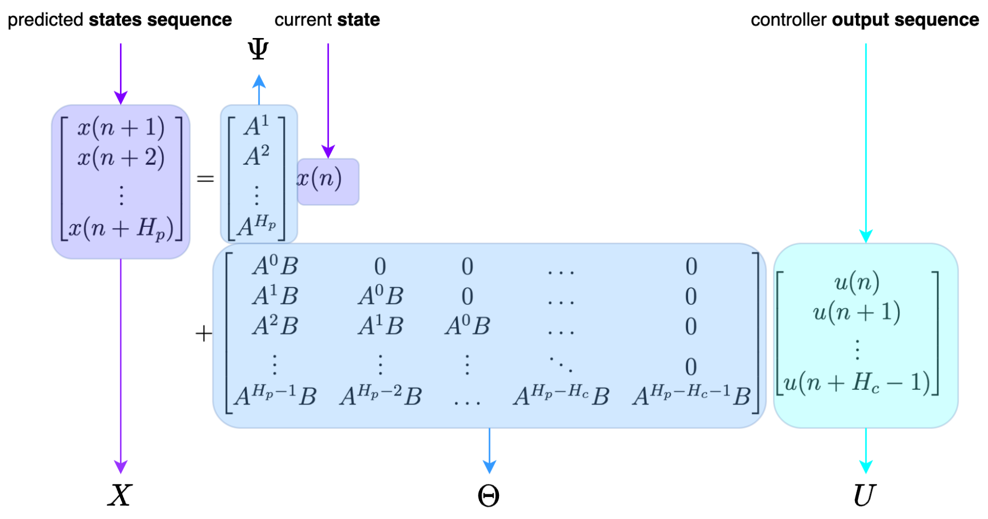
predicted **states sequence**
current **state**
controller **output sequence**
Ψ
X
Θ
U
$$\begin{bmatrix} x(n+1) \\ x(n+2) \\ \vdots \\ x(n+H_p) \end{bmatrix} = \begin{bmatrix} A^1 \\ A^2 \\ \vdots \\ A^{H_p} \end{bmatrix} x(n) + \begin{bmatrix} A^0B & 0 & 0 & \dots & 0 \\ A^1B & A^0B & 0 & \dots & 0 \\ A^2B & A^1B & A^0B & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ A^{H_p-1}B & A^{H_p-2}B & \dots & A^{H_p-H_c}B & A^{H_p-H_c-1}B \end{bmatrix} \begin{bmatrix} u(n) \\ u(n+1) \\ \vdots \\ u(n+H_c-1) \end{bmatrix}$$

Figure 3.3: 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~3.4. 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:

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

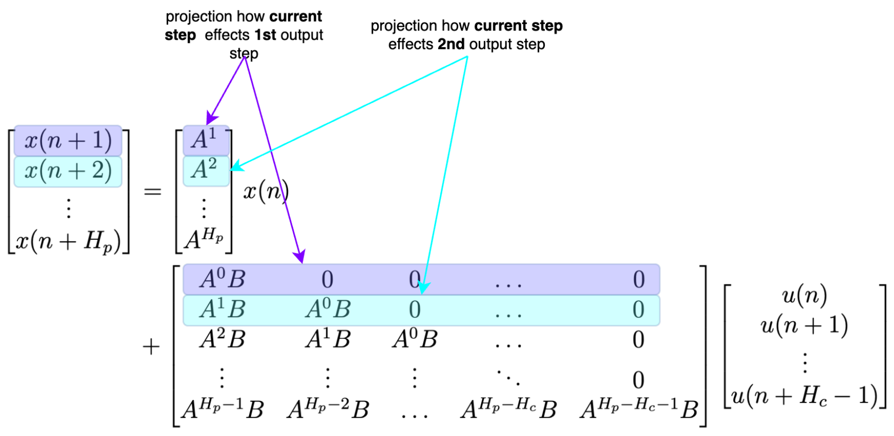
projection how **current step** effects **1st** output step
projection how **current step** effects **2nd** output step
$\begin{bmatrix} x(n+1) \\ x(n+2) \\ \vdots \\ x(n+H_p) \end{bmatrix} = \begin{bmatrix} A^1 \\ A^2 \\ \vdots \\ A^{H_p} \end{bmatrix} x(n) + \begin{bmatrix} A^0B & 0 & 0 & \dots & 0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

Figure 3.4: 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 3.5.

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).

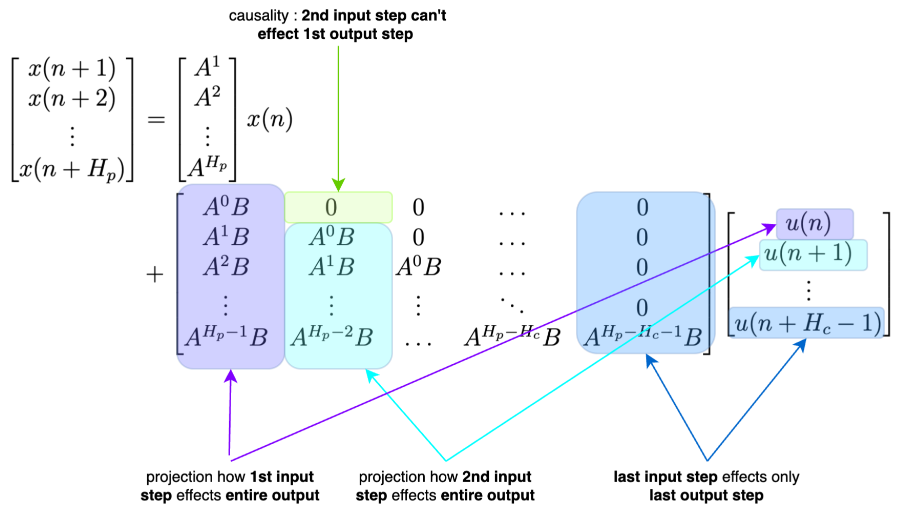
causality : 2nd input step can't effect 1st output step
$$\begin{bmatrix} x(n+1) \\ x(n+2) \\ \vdots \\ x(n+H_p) \end{bmatrix} = \begin{bmatrix} A^1 \\ A^2 \\ \vdots \\ A^{H_p} \end{bmatrix} x(n) + \begin{bmatrix} A^0B & 0 & 0 & \cdots & 0 \\ A^1B & A^0B & 0 & \cdots & 0 \\ A^2B & A^1B & A^0B & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ A^{H_p-1}B & A^{H_p-2}B & \cdots & A^{H_p-H_c}B & A^{H_p-H_c-1}B \end{bmatrix} \begin{bmatrix} u(n) \\ u(n+1) \\ \vdots \\ u(n+H_c-1) \end{bmatrix}$$
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 3.5: 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.

—

3.4 Optimization problem

Substitute the predicted state into the cost function:

$$\mathcal{L}(U) = (X_r - \Psi x(n) - \Theta U)^\top \tilde{Q}(X_r - \Psi x(n) - \Theta U) + U^\top \tilde{R} U. \tag{3.9}$$

Define the state–reference residual

$$S = X_r - \Psi x(n), \tag{3.10}$$

to obtain

$$\mathcal{L}(U) = U^\top \tilde{R} U + (S - \Theta U)^\top \tilde{Q}(S - \Theta U). \tag{3.11}$$

Expanding and collecting terms:

$$\mathcal{L}(U) = U^\top (\tilde{R} + \Theta^\top \tilde{Q} \Theta) U - 2U^\top \Theta^\top \tilde{Q} S + S^\top \tilde{Q} S. \tag{3.12}$$

The derivative terms are:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial U} &: \\ \frac{\partial U^T \tilde{R} U}{\partial U} &= 2\tilde{R}U \\ \frac{\partial U^T \Theta^T \tilde{Q} \Theta U}{\partial U} &= 2\Theta^T \tilde{Q} \Theta U \\ \frac{\partial {-2U^T \Theta^T \tilde{Q} S}}{\partial U} &= -2\Theta^T \tilde{Q} S \\ \frac{\partial S^T \tilde{Q} S}{\partial U} &= 0 \end{aligned} \tag{3.13}$$

—

3.5 Analytical solution

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

$$\frac{\partial \mathcal{L}}{\partial U} = 2(\tilde{R} + \Theta^\top \tilde{Q} \Theta) U - 2\Theta^\top \tilde{Q} S = 0. \tag{3.14}$$

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

$$U^* = (\tilde{R} + \Theta^\top \tilde{Q} \Theta)^{-1} \Theta^\top \tilde{Q} S, \tag{3.15}$$

which is equivalent to

$$U^* = (\tilde{R} + \Theta^\top \tilde{Q} \Theta)^{-1} \Theta^\top \tilde{Q} (X_r - \Psi x(n)). \tag{3.16}$$

3.6 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.

—

3.7 Algorithm implementation

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

$$\Sigma = (\tilde{R} + \Theta^\top \tilde{Q} \Theta)^{-1} \Theta^\top \tilde{Q}. \tag{3.17}$$

At each control step:

$$E(n) = X_r(n) - \Psi x(n), \tag{3.18}$$
$$U(n) = \Sigma E(n), \tag{3.19}$$
$$u(n) = \text{first } M \text{ elements of } U(n). \tag{3.20}$$

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~3.6.

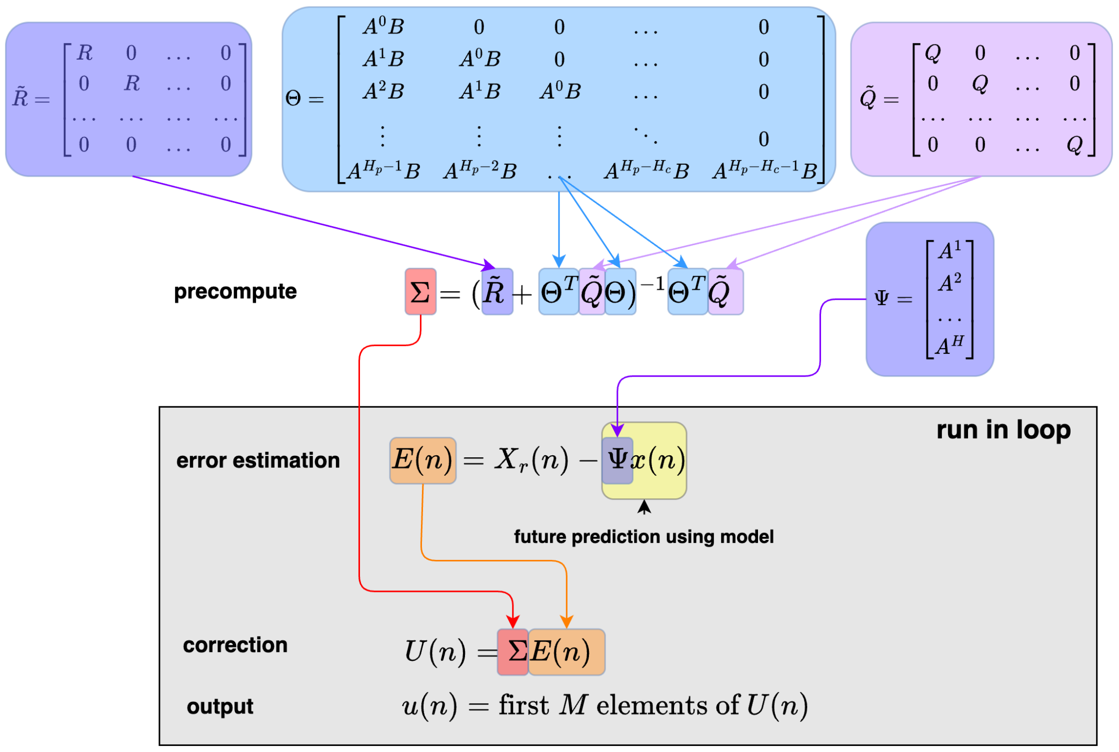
$\tilde{R} = \begin{bmatrix} R & 0 & \dots & 0 \\ 0 & R & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 0 \end{bmatrix}$
$\Theta = \begin{bmatrix} A^0B & 0 & 0 & \dots & 0 \\ A^1B & A^0B & 0 & \dots & 0 \\ A^2B & A^1B & A^0B & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ A^{H_p-1}B & A^{H_p-2}B & \dots & A^{H_p-H_c}B & A^{H_p-H_c-1}B \end{bmatrix}$
$\tilde{Q} = \begin{bmatrix} Q & 0 & \dots & 0 \\ 0 & Q & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & Q \end{bmatrix}$
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 3.6: 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
```

—

3.8 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 3.7. We control two input forces, u_x and u_y.

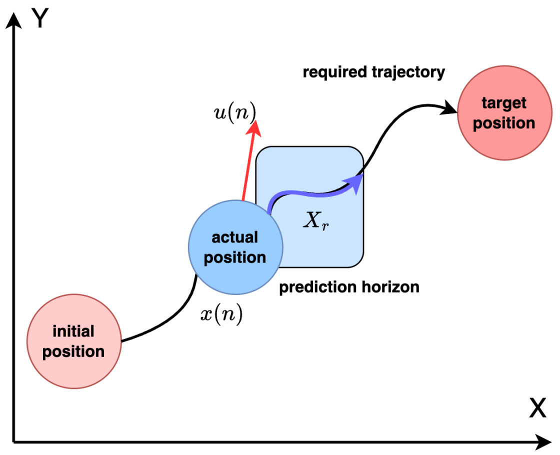

Figure 3.7: Trajectory tracking example.

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:

$$\bar{A} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\frac{1}{\tau} & 0 \\ 0 & 0 & 0 & -\frac{1}{\tau} \end{bmatrix}, \qquad \bar{B} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \frac{k}{\tau} & 0 \\ 0 & \frac{k}{\tau} \end{bmatrix}. \tag{3.21}$$

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.

$$A = \begin{bmatrix} 1 & 0 & 0.00990099 & 0 \\ 0 & 1 & 0 & 0.00990099 \\ 0 & 0 & 0.98019802 & 0 \\ 0 & 0 & 0 & 0.98019802 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 5.9406 \times 10^{-3} & 0 \\ 0 & 5.9406 \times 10^{-3} \end{bmatrix}. \tag{3.22}$$

—

Simulation Setup

The complete simulation workflow is illustrated in figure 3.8.

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 : $u_{max} = 10$.

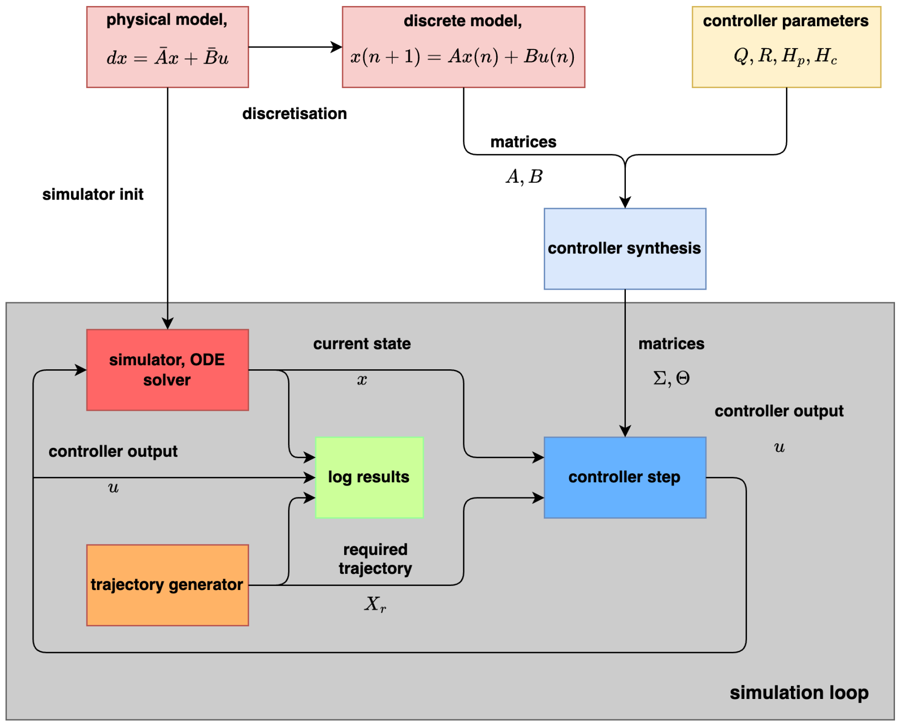

Figure 3.8: Simulation flow chart for MPC tracking.

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 3.9, 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 $x_r(n)$.

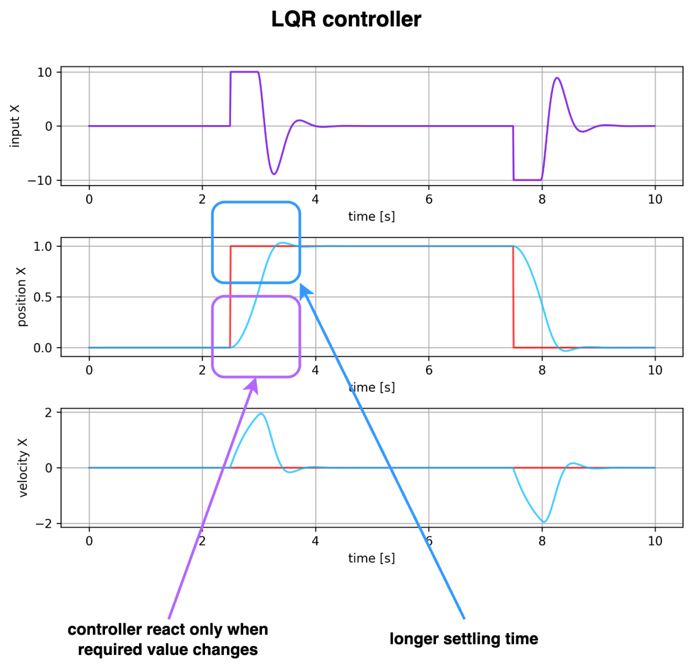

Figure 3.9: Step response for LQR control.

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

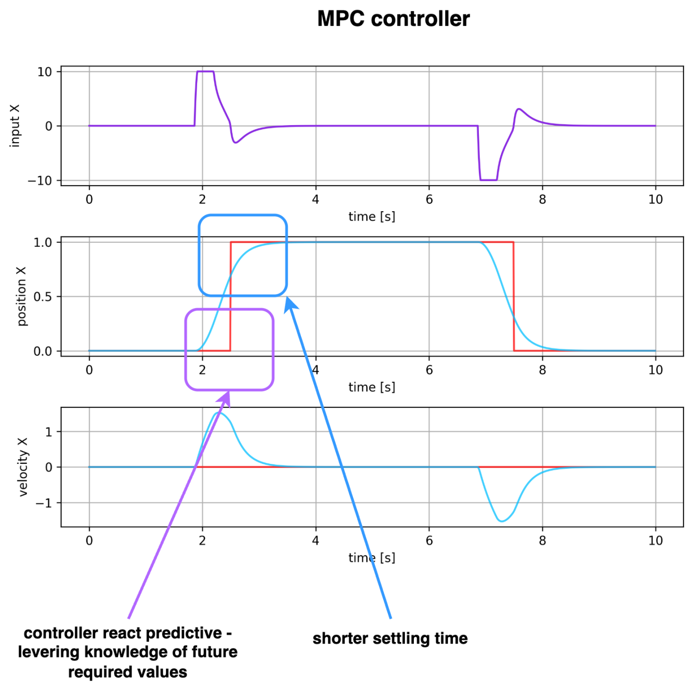

Figure 3.10: Step response for MPC control.

—

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.