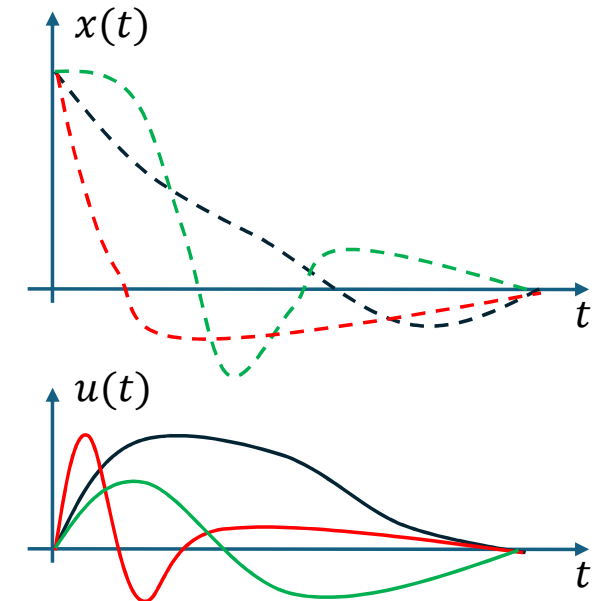
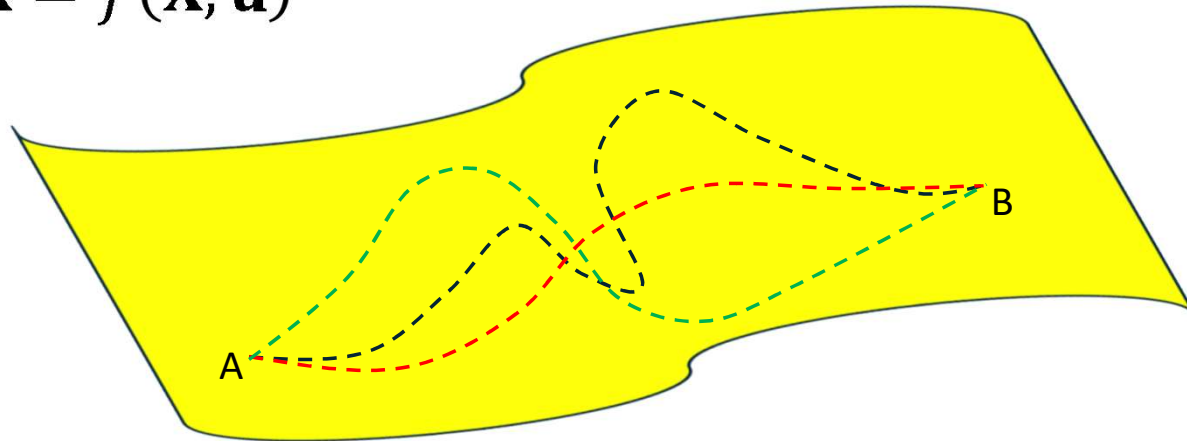


Commande Optimale

Guidage FISE 2A 2024-2025

Commande Optimale

- $\dot{\mathbf{X}} = f(\mathbf{X}, \mathbf{u})$



Faisabilité ?

Contraintes :

$$|u| < u_{max} \text{ (exemple du bras)}$$

$$x \in [x_{min}, x_{max}] - \Omega_x^{obs}$$

Performance

(butées)

Solution optimale (meilleure trajectoire)

Minimiser une fonction coût



$$J = \int_{t_0}^{t_f} \underbrace{g(x, u)}_{\text{Coût instantané}} \cdot dt + \underbrace{h(x(t_f))}_{\text{Distance finale à la cible}}$$

Commande Optimale

- $\dot{\mathbf{X}} = f(\mathbf{X}, \mathbf{u})$

Faisabilité ?

Contraintes :

$$|u| < u_{max}$$

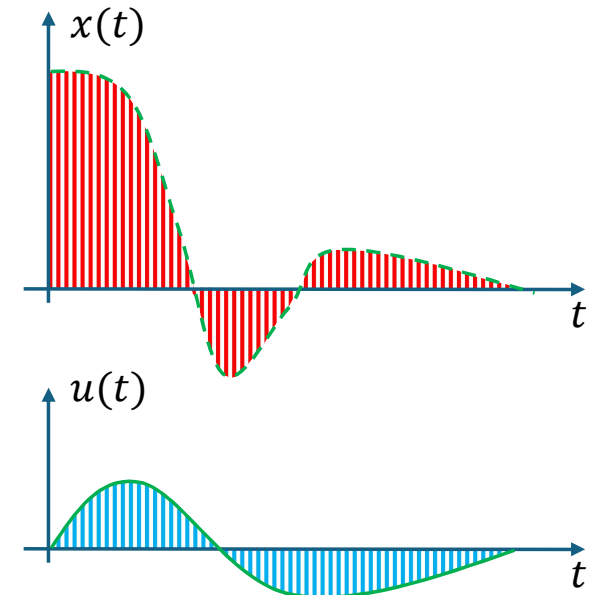
$$x \in [x_{min}, x_{max}] - \Omega_x^{obs}$$

Performance

Solution optimale (meilleure trajectoire)

Minimiser une fonction coût

$$J = \int_{t_0}^{t_f} \underbrace{g(x, u)}_{\text{Coût instantané}} \cdot dt + \underbrace{h(x(t_f))}_{\text{Distance finale à la cible}}$$



Exemples de $g(x, u)$

- Coût quadratique : $g(x, u) = \underbrace{x^T \cdot Q \cdot x}_{\text{erreur}} + \underbrace{u^T \cdot R \cdot u}_{\text{effort des actionneurs}}$

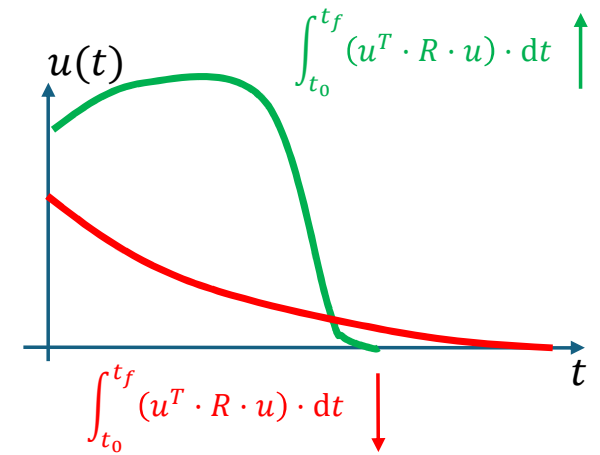
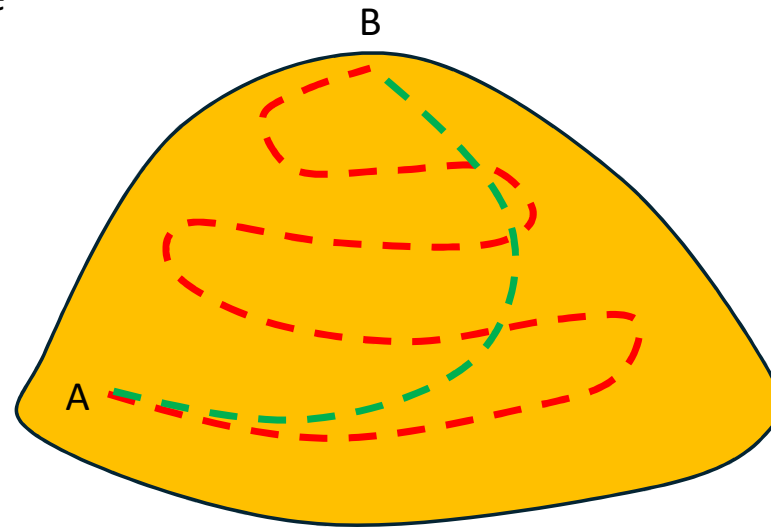
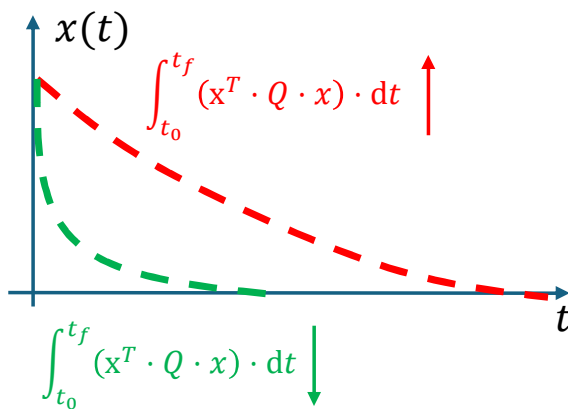
Commande Optimale

- $\dot{\mathbf{X}} = f(\mathbf{X}, \mathbf{u})$

$$J = \int_{t_0}^{t_f} \underbrace{g(x, u)}_{\text{Coût instantané}} \cdot dt + \underbrace{h(x(t_f))}_{\text{Distance finale à la cible}}$$

Exemples de $g(x, u)$

- Coût quadratique : $g(x, u) = \underbrace{x^T \cdot Q \cdot x}_{\text{erreur}} + \underbrace{u^T \cdot R \cdot u}_{\text{effort des actionneurs}}$



Commande Optimale

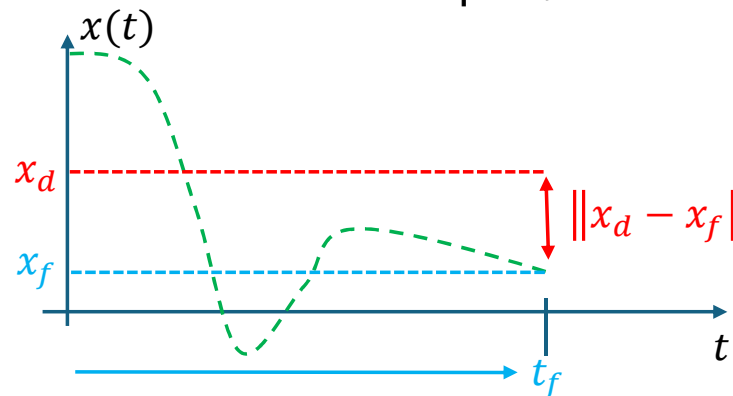
- $\dot{\mathbf{X}} = f(\mathbf{X}, \mathbf{u})$

$$J = \int_{t_0}^{t_f} \underbrace{g(x, u)}_{\text{Coût instantané}} \cdot dt + \underbrace{h(x(t_f))}_{\text{Distance finale à la cible}}$$

Exemples de $g(x, u)$

- Coût quadratique : $g(x, u) = \mathbf{x}^T \cdot \mathbf{Q} \cdot \mathbf{x} + u^T \cdot \mathbf{R} \cdot u$

- Temps : $\left| \begin{array}{l} g(x, u) = 1 \rightarrow \int_{t_0}^{t_f} g(x, u) \cdot dt = t_f \quad \text{temps final} \\ h(x_f) = \lambda \cdot (\mathbf{x}_f - \mathbf{x}_d)^T \cdot (\mathbf{x}_f - \mathbf{x}_d) \quad \text{erreur finale} \end{array} \right.$



Commande Optimale

- Méthodes
 - *Linear Quadratic Regulator* : LQR
 - Systèmes linéaires (ou linéarisés)
 - Pas de considération des contraintes
 - Solution analytique du type $\mathbf{u} = -\mathbf{K} \cdot \mathbf{x}$
 - Programmation Dynamique
 - Systèmes nonlinéaires
 - Discrétisation nécessaire
 - Considération explicite des contraintes
 - Optimisation de trajectoire
 - Shooting, Direct collocation
 - *Model Predictive Control* : MPC
 - Optimisation temps réel sur horizon temporel donné
 - On recalcule la trajectoire à chaque instant, on applique la commande courante

Commande Optimale

- Préliminaires: Stabilité de Lyapunov des systèmes linéaires

- Soit le système linéaire suivant:

$$S : \dot{x} = A \cdot x$$

- **Théorème** : *le système linéaire S est asymptotiquement stable (i.e. les valeurs propres de A sont à partie réelles négatives) **si et seulement si**, pour toute matrice symétrique définie positive Q , il existe une matrice P définie positive (symétrique) satisfaisant l'équation de Lyapunov suivante :*

$$A^T \cdot P + P \cdot A + Q = 0 \quad (1)$$

Démonstration de la condition suffisante :

Soit la fonction de *Lyapunov candidate* $V(x) = x^T \cdot P \cdot x$.

Considérant l'expression de S , il vient $\dot{V}(x) = x^T \cdot$

$(A^T \cdot P + P \cdot A) \cdot x$. Si P est solution (positive) de (1),

alors :

$$\begin{cases} V(x) = x^T \cdot P \cdot x > 0 \quad \forall x \neq 0 \\ \dot{V}(x) = -x^T \cdot Q \cdot x < 0 \quad \forall x \neq 0 \end{cases} \rightarrow \begin{array}{l} \text{Le système } S \text{ est} \\ \text{asymptotiquement} \\ \text{stable} \end{array}$$

Démonstration de la condition nécessaire :

Si A est stable, alors (1) admet une solution unique :

$$P = \int_0^\infty e^{A^T t} \cdot Q \cdot e^{At} dt$$

En effet :

$$\begin{aligned} A^T \cdot P + P \cdot A &= \int_0^\infty A^T \cdot e^{A^T t} \cdot Q \cdot e^{At} dt + \int_0^\infty e^{A^T t} \cdot Q \cdot e^{At} \cdot A dt \\ &= \int_0^\infty \frac{d}{dt} (e^{A^T t} \cdot Q \cdot e^{At}) dt = e^{A^T t} \cdot Q \cdot e^{At} \Big|_0^\infty = -Q \end{aligned}$$

puisque $\lim_{t \rightarrow \infty} e^{At} = 0$ si A est stable.

Commande Optimale

- Commande LQR

- Soit le système linéaire suivant, $S : \begin{cases} \dot{x} = A \cdot x + B \cdot u \\ y = C \cdot x \end{cases}$

- Hypothèse : S est contrôlable

- i.e. $\text{rank}(Q_0 = [A, A \cdot B, A^2 \cdot B, \dots, A^{n-1} \cdot B]) = n$

- Le critère quadratique:

- $J = \int_{t_0}^{t_f} (x^T \cdot Q \cdot x + u^T \cdot R \cdot u) \cdot dt$
- ou $J = \int_{t_0}^{t_f} (z^T \cdot Q_z \cdot z + u^T \cdot R \cdot u) \cdot dt$ avec $Q = C^T \cdot Q_z \cdot C$

- La commande :

- $u = -K_c \cdot x$, avec $K_c = R^{-1} \cdot B^T \cdot P_c$, où P_c est la solution >0 (symétrique) de l'éq. de Riccati :

$$P_c \cdot A + A^T \cdot P_c - P_c \cdot B \cdot R^{-1} \cdot B^T \cdot P_c + C^T \cdot Q \cdot C = 0$$

Commande Optimale

- Éléments de preuve:

La commande $u = -K_c \cdot x \rightarrow \dot{x} = (A - B \cdot K) \cdot x = A_f \cdot x$ exprime la dynamique de la boucle fermée. La réponse s'écrit alors $x(t) = x_0 \cdot e^{A_f t}$. Le critère $J = \int_{t_0}^{t_f} (x^T \cdot Q \cdot x + u^T \cdot R \cdot u) \cdot dt = x_0^T \cdot P \cdot x_0$ avec

$$P = \int_0^\infty e^{A_f^T t} \cdot (Q + K^T \cdot R \cdot K) \cdot e^{A_f t} dt$$

Si A_f est stable, alors P vérifie (1) : $A_f^T \cdot P + P \cdot A_f + (Q + K^T \cdot R \cdot K) = 0$. Par ailleurs, $P > 0$ (Pos. Def.) puisque $J(= x_0^T \cdot P \cdot x_0) > 0$. Soit K_c la valeur optimale de K qui minimise J , et P_c la solution correspondante de (1) :

$$(A - B \cdot K_c)^T \cdot P_c + P_c \cdot (A - B \cdot K_c) + (Q + K_c^T \cdot R \cdot K_c) = 0 \quad (2)$$

Toute variation Δ_K autour de K_c entraîne une variation Δ_P autour de P_c . Soit $K = \Delta_K + K_c$ et $P = \Delta_P + P_c$. Il vient :

$$(A - B \cdot (\Delta_K + K_c))^T \cdot (\Delta_P + P_c) + (\Delta_P + P_c) \cdot (A - B \cdot (\Delta_K + K_c)) + (Q + (\Delta_K + K_c)^T \cdot R \cdot (\Delta_K + K_c)) = 0 \quad (3)$$

K_c est la valeur optimale au sens du critère J , ssi J augmente pour toute variation ΔK autour de K_c , $\rightarrow \Delta P > 0$. (3)-(2) donne :

$$(A - B \cdot K)^T \cdot \Delta_P + \Delta_P \cdot (A - B \cdot K) + \Delta_K^T \cdot (R \cdot K_c - B^T \cdot P_c) + (R \cdot K_c - B^T \cdot P_c)^T \cdot \Delta_K + \Delta_K^T \cdot R \cdot \Delta_K = 0$$

Qui a la structure (1). Ainsi, puisque $(A - B \cdot K)$ est stable, $\Delta_P > 0$ ssi :

$$\Delta_K^T \cdot (R \cdot K_c - B^T \cdot P_c) + (R \cdot K_c - B^T \cdot P_c)^T \cdot \Delta_K + \Delta_K^T \cdot R \cdot \Delta_K > 0$$

Et puisque $R > 0 \rightarrow \Delta_K^T \cdot R \cdot \Delta_K > 0, \forall \Delta_K$, on choisit : $K_c = R^{-1} \cdot B^T \cdot P_c$ qui, injecté dans (2) donne l'équation de Riccati :

$$P_c \cdot A + A^T \cdot P_c - P_c \cdot B \cdot R^{-1} \cdot B^T \cdot P_c + C^T \cdot Q \cdot C = 0$$

Commande Optimale

- Résolution de l'équation de Riccati
 - Méthode directe (solveurs numériques)
 - Pour des matrices de taille raisonnable, on utilise des algorithmes numériques implémentés dans des bibliothèques comme :

```
import numpy as np
from scipy.linalg import solve_continuous_are

# Définition des matrices
A = np.array([[1, 2], [3, 4]])
B = np.array([[1], [0]])
Q = np.array([[1, 0], [0, 1]])
R = np.array([[1]])

# Résolution de l'équation de Riccati
X = solve_continuous_are(A, B, Q, R)

# Affichage du résultat
print("Solution X de l'équation de Riccati :\n", X)
```

```
% Définition des matrices
A = [1 2; 3 4];
B = [1; 0];
Q = [1 0; 0 1];
R = 1;

% Résolution de l'équation de Riccati
[X, L, G] = care(A, B, Q, R);

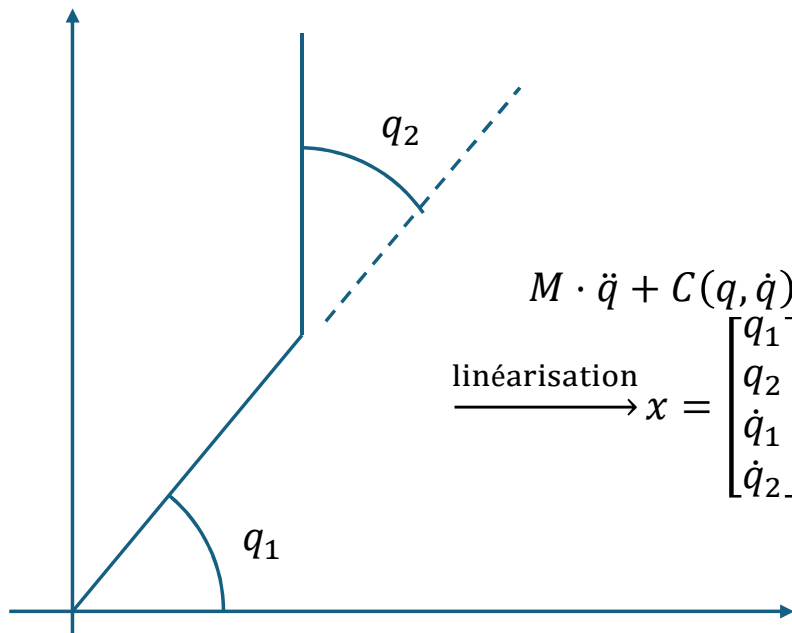
% Affichage du résultat
disp('Solution X de l''équation de Riccati :');
disp(X);
```

- Méthode de Schur (Décomposition de Hamiltonienne)

$$P_c \cdot A + A^T \cdot P_c - P_c \cdot B \cdot R^{-1} \cdot B^T \cdot P_c + C^T \cdot Q \cdot C = 0 \rightarrow \begin{bmatrix} P_c & -I_d^n \end{bmatrix} \cdot \overbrace{\begin{bmatrix} A & -B \cdot R^{-1} \cdot B^T \\ -Q & -A^T \end{bmatrix}}^H \cdot \begin{bmatrix} I_d^n \\ P_c \end{bmatrix} = 0$$

Commande Optimale

- LQR, exemple : robot manipulateur 2 axes



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

linéarisation $\rightarrow x = \begin{bmatrix} q_1 \\ q_2 \\ \dot{q}_1 \\ \dot{q}_2 \end{bmatrix}, \begin{cases} \dot{x} = A \cdot x + B \cdot u \\ y = I_d \cdot x \end{cases}$

```
...
%% Calcul du contrôleur LQR
K = lqr(A, B, Q, R);
...
```

```
clc; clear; close all;

%% Paramètres du robot
m1 = 1; m2 = 1; % Masses (kg)
l1 = 1; l2 = 1; % Longueurs des bras (m)
I1 = 0.2; I2 = 0.2; % Inerties (kg.m²)
g = 9.81; % Gravité

%% Matrices d'état (linéarisées autour de q1 = q2 = 0)
A = [0 0 1 0;
     0 0 0 1;
     0 0 0 0;
     0 0 0 0];

B = [0 0;
     0 0;
     1/l1 0;
     0 1/l2];

C = eye(4); % Sortie = tous les états
D = zeros(4,2); % Pas d'action directe de u sur y

%% Pondération pour LQR (J = ∫(x'Qx + u'Ru) dt)
Q = diag([100, 100, 10, 10]); % Pénalité sur erreur angulaire et vitesse
R = diag([1, 1]); % Effort de commande minimum

%% Calcul du contrôleur LQR
K = lqr(A, B, Q, R);

%% Simulation
dt = 0.01; % Pas de temps
T = 5; % Temps total
t = 0:dt:T;

% Conditions initiales (perturbation initiale)
x = [0.1; -0.1; 0; 0]; % (q1, q2, dq1, dq2)
x_hist = zeros(4, length(t));

for k = 1:length(t)
    % Commande optimale LQR
    u = -K * x;
    % Dynamique du système (Euler)
    dx = A*x + B*u;
    x = x + dx * dt;
    % Stockage des données
    x_hist(:,k) = x;
    U_Stock(:,k) = u;
end

%% Affichage des résultats
```

Commande Optimale

- LQR, exemple : robot manipulateur 2 axes

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

linéarisation $\rightarrow x = \begin{bmatrix} q_1 \\ q_2 \\ \dot{q}_1 \\ \dot{q}_2 \end{bmatrix}, \begin{cases} \dot{x} = A \cdot x + B \cdot u \\ y = I_d \cdot x \end{cases}$

```
...
P = scipy.linalg.solve_continuous_are(A, B, Q, R) # Résolution de l'équation de Riccati
K = np.linalg.inv(R) @ B.T @ P # Gain LQR
...
```

```
import numpy as np
import scipy.linalg
import matplotlib.pyplot as plt

# Paramètres du robot
m1, m2 = 1, 1 # Masses des segments (kg)
l1, l2 = 1, 1 # Longueurs des bras (m)
I1, I2 = 0.2, 0.2 # Moments d'inertie (kg.m²)
g = 9.81 # Gravité (m/s²)

# Matrices d'état (linéarisées autour de q1 = q2 = 0)
A = np.array([[0, 0, 1, 0],
              [0, 0, 0, 1],
              [0, 0, 0, 0],
              [0, 0, 0, 0]])

B = np.array([[0, 0],
              [0, 0],
              [1/I1, 0],
              [0, 1/I2]])

# Pondération pour la fonction de coût LQR (J = ∫(x'Qx + u'Ru) dt)
Q = np.diag([100, 100, 10, 10]) # Pénalité sur l'erreur des angles et vitesses
R = np.diag([1, 1]) # Pénalité sur l'effort de commande

# Calcul du gain optimal LQR
P = scipy.linalg.solve_continuous_are(A, B, Q, R) # Résolution de l'équation de Riccati
K = np.linalg.inv(R) @ B.T @ P # Gain LQR

# Simulation de la dynamique du robot avec LQR
dt = 0.01 # Pas de temps
T = 5 # Temps total
t = np.arange(0, T, dt)

# Conditions initiales (perturbation initiale)
x = np.array([0.1, -0.1, 0, 0]) # [q1, q2, dq1, dq2]
x_hist = np.zeros((4, len(t)))

for k in range(len(t)):
    # Commande optimale LQR
    u = -K @ x

    # Dynamique du système (Euler)
    dx = A @ x + B @ u
    x = x + dx * dt

    # Stockage des données
    x_hist[:, k] = x

# Affichage des résultats
plt.figure(figsize=(10, 5))

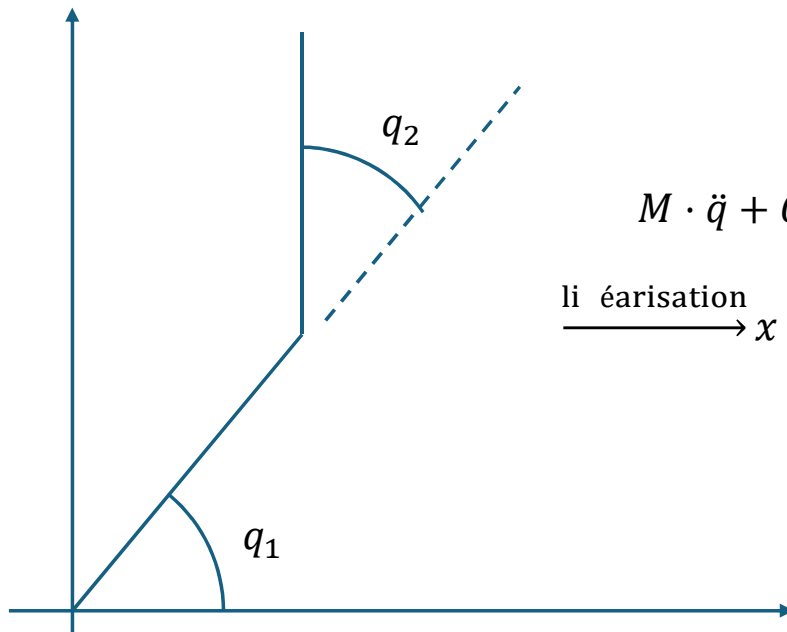
plt.subplot(2, 1, 1)
plt.plot(t, x_hist[0, :], 'b', label=r'$q_1$')
plt.plot(t, x_hist[1, :], 'r', label=r'$q_2$')
plt.ylabel('Angles (rad)')
plt.legend()
plt.grid()
plt.title("Réduction des erreurs d'angles avec LQR")

plt.subplot(2, 1, 2)
plt.plot(t, x_hist[2, :], 'b', label=r'$\dot{q}_1$')
plt.plot(t, x_hist[3, :], 'r', label=r'$\dot{q}_2$')
plt.ylabel('Vitesses angulaires (rad/s)')
plt.xlabel('Temps (s)')
plt.legend()
plt.grid()

plt.show()
```

Commande Optimale

- LQR, exemple : robot manipulateur 2 axes

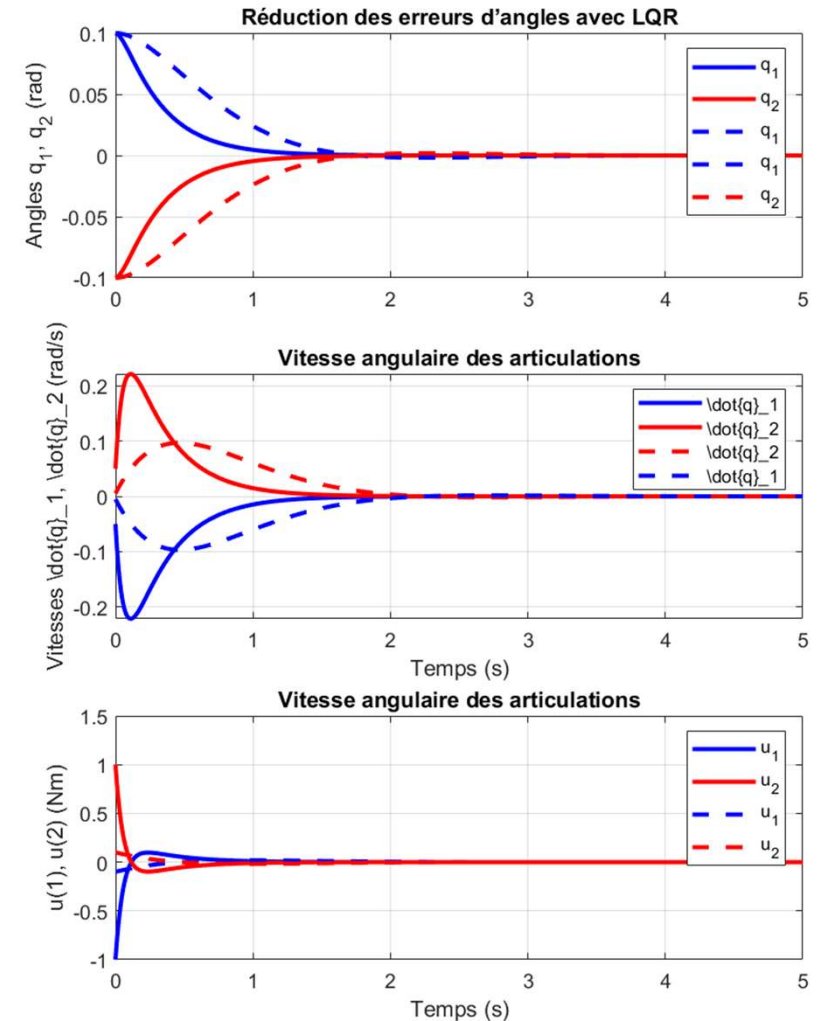


$$M \cdot \ddot{q} + C(q, \dot{q}) \cdot \dot{q} + G(q) = \tau$$

linéarisation $\rightarrow x = \begin{bmatrix} q_1 \\ q_2 \\ \dot{q}_1 \\ \dot{q}_2 \end{bmatrix}, \begin{cases} \dot{x} = A \cdot x + B \cdot u \\ y = I_d \cdot x \end{cases}$

Traits pleins : $Q = \text{diag}([100, 100, 10, 10]);$
 $R = \text{diag}([1, 1])$

Pointillés : $Q = \text{diag}([100, 100, 10, 10]);$ % Pénalité sur erreur angulaire et vitesse
 $R = \text{diag}([100, 100]);$ % Effort de commande minimum



Commande Optimale

- Programmation Dynamique
 - Programmation dynamique : « planification et ordonnancement » (1950)
 - Principe d'optimalité de Bellman : une solution optimale d'un problème s'obtient en combinant des solutions optimales à ses sous-problèmes.
 - S'applique aux systèmes discret(isé)s

$$x_{k+1} = f_k(x_k, u_k)$$

- Objectif : minimiser une fonction de coût J par le choix d'actions optimales u_k^*

$$J = \sum_{k=0}^{N-1} g(x_k, u_k) + g_N(x_n)$$

The diagram shows the cost function $J = \sum_{k=0}^{N-1} g(x_k, u_k) + g_N(x_n)$. A blue arrow points from the text 'Coût instantané' to the term $g(x_k, u_k)$ inside the summation. Another blue arrow points from the text 'Coût final' to the term $g_N(x_n)$.

Coût instantané Coût final

$$u_k^* = C_k(x_k)$$

The diagram shows the equation $u_k^* = C_k(x_k)$. Three red arrows point to the terms: 'Action optimale' points to u_k^* , 'Fonction de commande' points to C_k , and 'Etat' points to x_k .

Action optimale Fonction de commande Etat

Commande Optimale

- Programmation Dynamique

- Objectif : minimiser une fonction de coût J par le choix d'actions optimales u_k^*
- Algorithme récursif inverse (on part de la fin : *backward induction*):

$$J_k(x_k) = \min_{u_k} [g_k(x_k, u_k) + J_{k+1}(x_{k+1})]$$

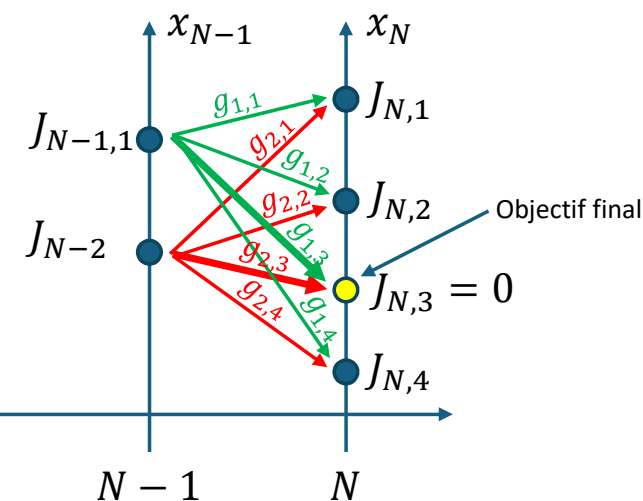
Coût instantané Coût à venir

$$C_k^*(x_k) = \arg \min_{u_k} [g_k(x_k, u_k) + J_{k+1}(x_{k+1})]$$

- 1) On commence par la fin : $J_N = g_N(x_N)$
- 2) On teste tous les u_{N-1} , on sélectionne celui qui produit

$$J_{N-1}(x_{N-1}) = \min_{u_{N-1}} [g_{N-1}(x_{N-1}, u_{N-1}) + J_N]$$

$$u_{N-1}^* = C_{N-1}^*(x_{N-1}) = \arg \min_{u_{N-1}} [g_{N-1}(x_{N-1}, u_{N-1}) + J_N]$$
- 3) ...



Commande Optimale

- Programmation Dynamique, ex Matlab :

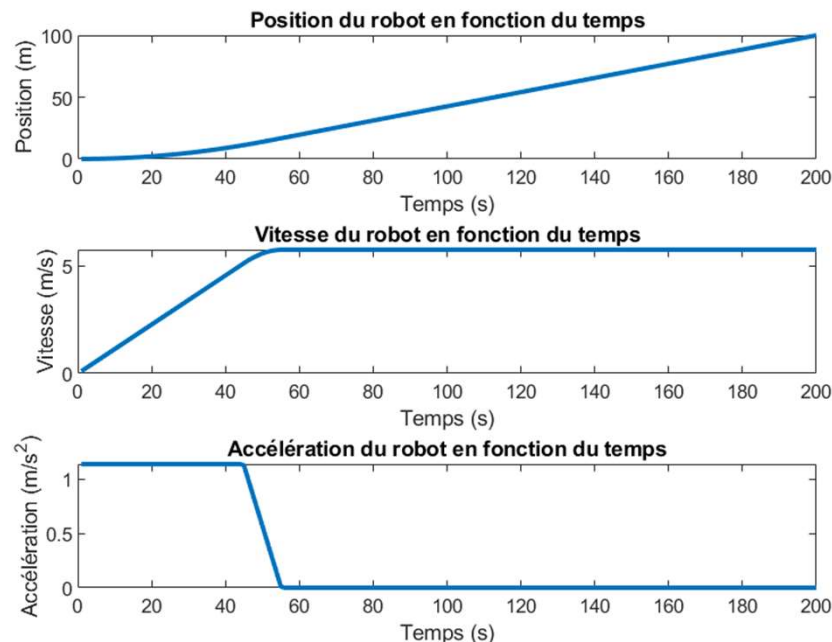


$$x_{k+1} = x_k + u_k \cdot \Delta t$$

$$g(x_k, u_k) = u_k^2$$

$$u_{min} < u_k < u_{max}$$

$$\dot{u}_{min} < \dot{u}_k < \dot{u}_{max}$$



```

x(1) = x0;           % Position initiale
v(1) = 0.1;          % Vitesse initiale faible pour initier le mouvement

% Paramètres pour la minimisation
alpha = 0.1;          % Poids pour minimiser la distance au carré (x_k - x_f)^2
beta = 1;              % Poids pour minimiser le temps
gamma = 0.1;          % Poids pour minimiser le temps de trajet

% Fonction de valeur V(x_t) - Minimisation de la distance et du temps
V = inf(1, n);         % Fonction de valeur
V(end) = 0;           % À la fin, le coût est nul (objectif atteint)

% Discrétisation des actions (accélération)
a_values = linspace(amin, amax, 21); % Accélérations possibles (-2 à 2 m/s^2)

% Récurrence de Bellman pour calculer la politique optimale
for k = n-1:-1:1
    min_cost = Inf; % Initialiser le coût minimum
    best_a = 0;    % Initialiser la meilleure accélération

    % Essayer toutes les accélérations possibles
    for i = 1:length(a_values)
        a_t = a_values(i); % Essayer l'accélération a_t
        v_t_plus_1 = v(k) + a_t * dt; % Calculer la nouvelle vitesse

        % Vérifier si la vitesse reste dans les limites
        if v_t_plus_1 >= vmin && v_t_plus_1 <= vmax
            % Calculer le coût total : distance au carré + temps
            distance_cost = alpha * (x(k) - xf)^2; % Coût de la distance au carré
            time_cost = beta * dt; % Le coût de temps est simplement dt à chaque étape
            future_cost = V(k+1); % Coût futur (valeur optimale pour le prochain état)

            total_cost = distance_cost + time_cost + future_cost; % Coût total

            % Si ce coût est le plus bas, on met à jour la meilleure accélération
            if total_cost < min_cost
                min_cost = total_cost;
                best_a = a_t;
            end
        end
    end

    % Stocker la meilleure accélération et mettre à jour la vitesse
    a(k) = best_a;
    v(k+1) = v(k) + a(k) * dt; % Mise à jour de la vitesse
    v(k+1) = min(vmax, max(vmin, v(k+1))); % Limiter la vitesse dans les bornes
    x(k+1) = x(k) + v(k) * dt; % Mise à jour de la position
end
    
```

Commande Optimale

- Programmation Dynamique, ex Python :

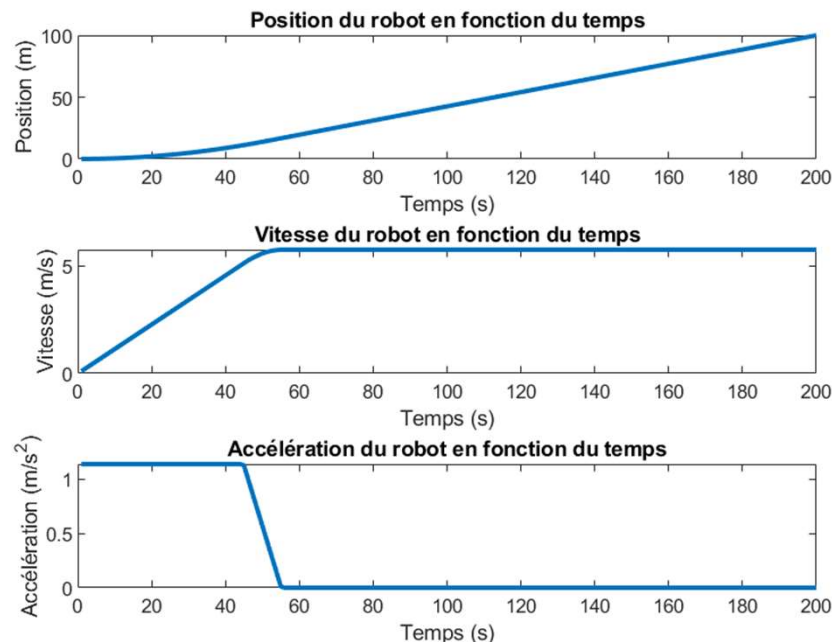


$$x_{k+1} = x_k + u_k \cdot \Delta t$$

$$g(x_k, u_k) = u_k^2$$

$$u_{\min} < u_k < u_{\max}$$

$$\dot{u}_{\min} < \dot{u}_k < \dot{u}_{\max}$$



```
x[0] = x0          # Position initiale
v[0] = 0.1         # Vitesse initiale faible pour initier le mouvement

# Paramètres pour la minimisation
alpha = 0.1        # Poids pour minimiser la distance au carré (x_k - x_f)^2
beta = 1           # Poids pour minimiser le temps
gamma = 0.1        # Poids pour minimiser le temps de trajet

# Fonction de valeur V(x_t) - Minimisation de la distance et du temps
V = np.inf * np.ones(n) # Fonction de valeur
V[-1] = 0             # À la fin, le coût est nul (objectif atteint)

# Discrétisation des actions (accélération)
a_values = np.linspace(amin, amax, 21) # Accélérations possibles (-2 à 2 m/s^2)

# Récurrence de Bellman pour calculer la politique optimale
for k in range(n-2, -1, -1): # Récurrence de Bellman, on commence de la fin
    min_cost = np.inf # Initialiser le coût minimum
    best_a = 0        # Initialiser la meilleure accélération

    # Essayer toutes les accélérations possibles
    for a_t in a_values:
        v_t_plus_1 = v[k] + a_t * dt # Calculer la nouvelle vitesse

        # Vérifier si la vitesse reste dans les limites
        if v_t_plus_1 >= vmin and v_t_plus_1 <= vmax:
            # Calculer le coût total : distance au carré + temps
            distance_cost = alpha * (x[k] - x_f) ** 2 # Coût de la distance au carré
            time_cost = beta * dt # Le coût de temps est simplement dt à chaque étape
            future_cost = V[k+1] # Coût futur (valeur optimale pour le prochain état)

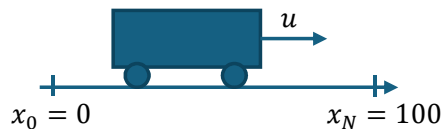
            total_cost = distance_cost + time_cost + future_cost # Coût total

            # Si ce coût est le plus bas, on met à jour la meilleure accélération
            if total_cost < min_cost:
                min_cost = total_cost
                best_a = a_t

    # Stocker la meilleure accélération et mettre à jour la vitesse
    a[k] = best_a
    v[k+1] = v[k] + a[k] * dt # Mise à jour de la vitesse
    v[k+1] = np.clip(v[k+1], vmin, vmax) # Limiter la vitesse dans les bornes
    x[k+1] = x[k] + v[k] * dt # Mise à jour de la position
```

Commande Optimale

• Programmation Dynamique, **fmincon**:

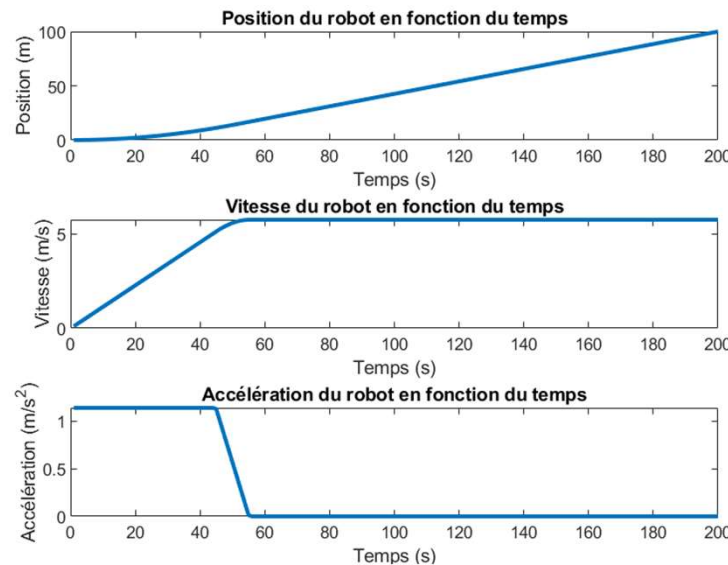


$$x_{k+1} = x_k + u_k \cdot \Delta t$$

$$g(x_k, u_k) = u_k^2$$

$$u_{\min} < u_k < u_{\max}$$

$$\dot{u}_{\min} < \dot{u}_k < \dot{u}_{\max}$$



```
# Récurrence de Bellman pour calculer la politique optimale
for k in range(n-2, -1, -1): # Récurrence de Bellman, on commence de la fin
    # Essayer de minimiser le coût en choisissant la meilleure accélération
    result = minimize(cost_function, x0=0, args=(k,), bounds=[(amin, amax)], method='L-BFGS-B')

    # Récupérer la meilleure accélération
    best_a = result.x[0]
    a[k] = best_a
    v[k+1] = v[k] + a[k] * dt # Mise à jour de la vitesse
    v[k+1] = np.clip(v[k+1], vmin, vmax) # Limiter la vitesse dans les bornes
    x[k+1] = x[k] + v[k] * dt # Mise à jour de la position
```

```
...
x(1) = x0; % Position initiale
v(1) = 0.1; % Vitesse initiale faible pour initier le mouvement

% Paramètres pour la minimisation
alpha = 0.1; % Poids pour minimiser la distance au carré (x_k - x_f)^2
beta = 1; % Poids pour minimiser le temps
gamma = 0.1; % Poids pour minimiser le temps de trajet

% Fonction de coût (à minimiser)
% La fonction de coût dépend de l'accélération à chaque instant
cost_func = @(a) calculate_cost(a, x, v, xf, alpha, beta, dt, n);

% Contraintes de vitesse et d'accélération
A = []; b = []; % Pas de contraintes linéaires
Aeq = []; beq = []; % Pas de contraintes d'égalités

% Contraintes non linéaires
nonlcon = @(a) non_linear_constraints(a, vmin, vmax, amin, amax, dt, n);

% Valeurs initiales pour l'optimisation
a0 = zeros(1, n-1); % Initialisation des accélérations à zéro

% Optimisation avec fmincon
options = optimoptions('fmincon', 'Display', 'iter', 'Algorithm', 'sqp');
[a_optimal, fval] = fmincon(cost_func, a0, A, b, Aeq, beq, [], [], nonlcon, options);

% Mise à jour des accélérations optimales
a = [a_optimal, a_optimal(end)]; % Ajouter l'accélération finale pour la continuité

% Simulation avec les accélérations optimales
for k = 1:n-1
    v(k+1) = v(k) + a(k) * dt; % Mise à jour de la vitesse
    v(k+1) = min(vmax, max(vmin, v(k+1))); % Limiter la vitesse dans les bornes
    x(k+1) = x(k) + v(k) * dt; % Mise à jour de la position
end
...
```

```
% Fonction de coût
function cost = calculate_cost(a, x, v, xf, alpha, beta, dt, n)
    % Simulation pour calculer la fonction de coût total
    x_temp = zeros(1, n);
    v_temp = zeros(1, n);
    x_temp(1) = x(1); % Position initiale
    v_temp(1) = v(1); % Vitesse initiale

    % Calculer les positions et vitesses à chaque étape
    for k = 1:n-1
        v_temp(k+1) = v_temp(k) + a(k) * dt; % Mise à jour de la vitesse
        x_temp(k+1) = x_temp(k) + v_temp(k) * dt; % Mise à jour de la position
    end

    % Calcul du coût
    distance_cost = alpha * (x_temp(end) - xf)^2; % Coût de la distance au carré
    time_cost = beta * n * dt; % Le coût du temps
    cost = distance_cost + time_cost; % Somme des coûts
end
```

```
% Fonction de contraintes non linéaires
function [c, ceq] = non_linear_constraints(a, vmin, vmax, amin, amax, dt, n)
    % Contraintes sur la vitesse et l'accélération
    c = [];
    ceq = [];
    v = zeros(1, n);
    v(1) = 0.1; % Vitesse initiale

    for k = 1:n-1
        v(k+1) = v(k) + a(k) * dt; % Mise à jour de la vitesse
        % Contraintes sur la vitesse et l'accélération
        c = [c; v(k+1) - vmax]; % Limite supérieure de la vitesse
        c = [c; vmin - v(k+1)]; % Limite inférieure de la vitesse
        c = [c; a(k) - amax]; % Limite supérieure de l'accélération
        c = [c; amin - a(k)]; % Limite inférieure de l'accélération
    end
end
```

Commande Optimale

- *Model Predictive Control* : MPC
 - Modèle dynamique du système
 - Horizon de prédiction
 - Optimisation à chaque étape, en temps réel
 - Application de la première commande

- LQR

- Système linéaire

- $X_{k+1} = A_k \cdot X_k + B_k \cdot u_k$

Evolution

- $\sum_k^N ((x^T \cdot Q \cdot x) + (u^T \cdot R \cdot u))$

Coût

```
x[0] = x0          # Position initiale
v[0] = 0.1         # Vitesse initiale faible pour initier le mouvement

# Paramètres pour la minimisation
alpha = 0.1        # Poids pour minimiser la distance au carré (x_k - x_f)^2
beta = 1           # Poids pour minimiser le temps
gamma = 0.1        # Poids pour minimiser le temps de trajet

# Fonction de valeur V(x_t) - Minimisation de la distance et du temps
V = np.inf * np.ones(n) # Fonction de valeur
V[-1] = 0              # À la fin, le coût est nul (objectif atteint)

# Discrétisation des actions (accélération)
a_values = np.linspace(amin, amax, 21) # Accélérations possibles (-2 à 2 m/s^2)

# Récurrence de Bellman pour calculer la politique optimale
for k in range(n-2, -1, -1): # Récurrence de Bellman, on commence de la fin
    min_cost = np.inf # Initialiser le coût minimum
    best_a = 0        # Initialiser la meilleure accélération

    # Essayer toutes les accélérations possibles
    for a_t in a_values:
        v_t_plus_1 = v[k] + a_t * dt # Calculer la nouvelle vitesse

        # Vérifier si la vitesse reste dans les limites
        if v_t_plus_1 >= vmin and v_t_plus_1 <= vmax:
            # Calculer le coût total : distance au carré + temps
            distance_cost = alpha * (x[k] - xf) ** 2 # Coût de la distance au carré
            time_cost = beta * dt # Le coût de temps est simplement dt à chaque étape
            future_cost = V[k+1] # Coût futur (valeur optimale pour le prochain état)

            total_cost = distance_cost + time_cost + future_cost # Coût total

            # Si ce coût est le plus bas, on met à jour la meilleure accélération
            if total_cost < min_cost:
                min_cost = total_cost
                best_a = a_t

    # Stocker la meilleure accélération et mettre à jour la vitesse
    a[k] = best_a
    v[k+1] = v[k] + a[k] * dt # Mise à jour de la vitesse
    v[k+1] = np.clip(v[k+1], vmin, vmax) # Limiter la vitesse dans les bornes
    x[k+1] = x[k] + v[k] * dt # Mise à jour de la position
```

= 0

)