Optimisation de la trajectoire d'un drone

MTH8408

Jouglet Nicolas, Dawut Esse, Joey Van Melle

Le contrôle de drones quadrotors est un enjeu majeur dans le domaine de la robotique autonome en raison de leur grande maniabilité, mais aussi de leur instabilité naturelle.
Ce projet vise à concevoir un **contrôleur optimal** capable de suivre une trajectoire prédéfinie tout en minimisant la consommation d'énergie.
Nous nous appuyons pour cela sur l'approche proposée par Suicmez et Kutay (2014), qui repose sur la commande optimale en temps discrèt.

Contexte du papier

- Les **quadrotors** sont des drones à 4 moteurs capables de **décoller et atterrir verticalement (VTOL)** et d'exécuter des **manœuvres agiles**.
- Ce sont des systèmes **fortement non linéaires et instables**.
- Leur principale limitation : **une consommation énergétique élevée**.

Description de la problèmatique

Le papier traité modélise un **drone quadrotor** et conçoit une commande optimale permettant de minimiser une fonction objectif quadratique prenant en compte différent critères tels que la précision du suivi d'une trajectoire, la consommation d'énergie et la position finale.

Afin d'obtenir la commande optimale, les auteurs du papier ont utilisé la méthode récursive de Riccati pour résoudre le problème d'optimisation quadratique en temps discret.

Cependant d'autres méthodes peuvent être utilisées pour determiner la commande optimale, la méthode recursive de Riccati est-elle la plus efficace ?

Description des objectifs et du plan d'action

La première objectif du projet consiste à **reproduire le modèle dynamique du drone** puis d'y appliquer **l'algorithme reccursif de Riccati** afin de reproduire les résultats obtenu dans le papier.
Ensuite, nous essaierons de résoudre le système avec **d'autres méthodes**, chaque méthode devra minimiser la fonction objectif. Les différentes méthodes utilisées sont :
- la résolution monolithique via IPOPT (optimisation non linéaire générique),
- la résolution avec COSMO (optimisation quadratique conique),
- et la résolution avec OSQP (optimisation quadratique à contraintes linéaires).

Cette comparaison permettra de mettre en évidence les avantages et inconvénients de chaque méthode pour le suivi optimal de trajectoire d'un drone quadrotor tout en déterminant laquelle est la plus efficace pour cette fonction objectif. Nous voulions aussi tester une méthode spécifique aux problèmes quadratiques présentée en cours, mais nous n'avons pas eu le temps de compléter cette objectif. Les méthodes utilisées permettent toutefois de comparer Ipopt et Riccati à des méthodes spécifiques aux problèmes quadratiques, ce qui était le but du projet.

Modélisation du système

Modèle dynamique non linéaire

Dans un premier temps, le système est modélisé de façon dynamique à partir des équations de Newton :

- **Dynamique translationnelle** : influencée par la poussée verticale U_1 et les angles ϕ, θ ;
  -Où ϕ et θ sont respectivement les angles de roulis et de tangage
- **Dynamique rotationnelle** : influencée par les couples de commande U_2, U_3, U_4.

Dans le modèle dynamique du quadrotor, les quatre entrées de commande U_1, U_2, U_3 et U_4 correspondent aux forces et couples générés par les quatre rotors :

$$U_1 = F_1 + F_2 + F_3 + F_4 \quad \text{(force totale générée par les quatre rotors, liée à la poussée verticale)} \tag{1}$$
$$U_2 = F_4 - F_2 \quad \text{(couple de roulis, contrôle l'angle de roulis } \phi\text{)} \tag{2}$$
$$U_3 = F_3 - F_1 \quad \text{(couple de tangage, contrôle l'angle de tangage } \theta\text{)} \tag{3}$$
$$U_4 = T_2 + T_4 - T_1 - T_3 \quad \text{(couple de lacet, contrôle l'angle de lacet } \psi\text{)} \tag{4}$$

où F_i représente la force générée par le i-ème rotor et T_i le couple associé.

Ainsi : - U_1 agit principalement sur la **poussée verticale** et donc l'altitude.
- U_2 contrôle le **roulis** (*roll*) autour de l'axe x.
- U_3 contrôle le **tangage** (*pitch*) autour de l'axe y.
- U_4 contrôle le **lacet** (*yaw*) autour de l'axe z.

Le modèle dynamique est constitué de deux parties :

- **Équation non linéaire pour le mouvement de translation :**

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = - \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} + LEB \begin{bmatrix} 0 \\ 0 \\ U_1/m \end{bmatrix} - \left(\frac{K_t}{m}\right) \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} \tag{1}$$

Avec

$$LEB = \begin{bmatrix} \cos(\theta)\cos(\phi) & \sin(\theta)\sin(\phi)\cos(\psi) - \cos(\theta)\sin(\psi) & \cos(\theta)\sin(\phi)\cos(\psi) + \sin(\theta)\sin(\psi) \\ \cos(\theta)\sin(\phi) & \sin(\theta)\sin(\phi)\sin(\psi) + \cos(\theta)\cos(\psi) & \cos(\theta)\sin(\phi)\sin(\psi) - \sin(\theta)\cos(\psi) \\ -\sin(\theta) & \sin(\theta)\cos(\phi) & \cos(\theta)\cos(\phi) \end{bmatrix}$$

- Équations non linéaires pour le mouvement de rotation :

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} \frac{(I_y - I_z)}{I_x} qr \\ \frac{(I_z - I_x)}{I_y} pr \\ \frac{(I_x - I_y)}{I_z} pq \end{bmatrix} + \begin{bmatrix} \frac{U_2 d}{I_x} \\ \frac{U_3 d}{I_y} \\ \frac{U_4}{I_z} \end{bmatrix} \tag{2}$$

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin(\phi)\tan(\theta) & \cos(\phi)\tan(\theta) \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi)/\cos(\theta) & \cos(\phi)/\cos(\theta) \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{3}$$

Modèle dynamique non linéaire simplifié

Si les perturbations par rapport à la condition de vol stationnaire sont faibles, on suppose que les vitesses angulaires du corps [p, q, r] et les dérivées des angles d'Euler [$\dot{\phi}$, $\dot{\theta}$, $\dot{\psi}$] peuvent être considérées comme égales.
On à alors :

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \quad ; \quad \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} \tag{4}$$

En utilisant l'équation 4 dans l'équation 2, on obtient **l'équation d'Euler** :

$$\begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} \frac{(I_y - I_z)}{I_x} \dot{\theta}\dot{\psi} \\ \frac{(I_z - I_x)}{I_y} \dot{\phi}\dot{\psi} \\ \frac{(I_x - I_y)}{I_z} \dot{\phi}\dot{\theta} \end{bmatrix} + \begin{bmatrix} \frac{U_2 d}{I_x} \\ \frac{U_3 d}{I_y} \\ \frac{U_4}{I_z} \end{bmatrix} \tag{5}$$

Linéarisation du modèle dynamique simplifié

Pour utiliser l'algorithme LQT en temps discret, il est nécessaire de linéariser le modèle autour d'une condition d'équilibre (vol stationnaire).

À ce point d'équilibre :

$$\phi = \dot{\phi} = \theta = \dot{\theta} = \psi = \dot{\psi} = x = \dot{x} = y = \dot{y} = \dot{z} = 0,$$
$$z = 1 \text{ mètre.}$$

En linéarisant LEB grâce aux hypothèses du vol stationnaire on obtient :

$$L_{EB}^{\text{lin}} \approx I_3$$

L'équation de translation (1) devient alors :

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = - \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} + I_3 \begin{bmatrix} 0 \\ 0 \\ U_1/m \end{bmatrix} \tag{6}$$

De même en vol stationnaire $\dot{\psi} = \dot{\phi} = \dot{\theta} = 0$, donc l'équation d'Euler peut s'écrire :

$$\begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} \frac{U_2 d}{I_x} \\ \frac{U_3 d}{I_y} \\ \frac{U_4}{I_z} \end{bmatrix} \tag{7}$$

Le modèle peut donc se résumer à :

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = - \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} + I_3 \begin{bmatrix} 0 \\ 0 \\ U_1/m \end{bmatrix} \quad ; \quad \begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} \frac{U_2 d}{I_x} \\ \frac{U_3 d}{I_y} \\ \frac{U_4}{I_z} \end{bmatrix} \tag{5}$$

Afin de linéariser le modèle, on introduit X le vecteur d'état, Y le vecteur de sortie et le vecteur de commande

U tels que :

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{bmatrix} = \begin{bmatrix} \phi \\ \dot{\phi} \\ \theta \\ \dot{\theta} \\ \psi \\ \dot{\psi} \\ x \\ \dot{x} \\ y \\ \dot{y} \\ z \\ \dot{z} \end{bmatrix} \quad ; \quad Y = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{bmatrix} \quad ; \quad U = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} \tag{6}$$

Enfin grâce aux hypothèses de vol stationnaire on obtient le modèle linèaire suivant :

$$\dot{X} = AX + BU \quad ; \quad Y = CX \tag{7}$$

Où A et B et C valent :

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & g & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ -g & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{A}$$

$$B = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \frac{1}{I_x} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{I_y} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{I_z} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{m} & 0 & 0 & 0 \end{bmatrix} \tag{B}$$

$$C = I_{12} \tag{C}$$

Fonction Performance

La fonction performance est construite de la façon suivante :

$$J = \frac{1}{2} \sum_{k=k_0}^{k_f-1} \left(\|C_d X(k) - r(k)\|_Q^2 + \|U(k)\|_R^2 \right) + \frac{1}{2} \|C_d X(k_f) - r(k_f)\|_F^2 \tag{8}$$

où $\|x\|_M^2 = x^\top M x$ est la norme quadratique pondérée par la matrice M, et F est la matrice de coût terminal, souvent choisie égale à Q, donc $F = Q$.

$$Q = \begin{bmatrix} 100 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 50 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 100 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 100 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1000 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.1 \end{bmatrix} \tag{Q}$$

$$R = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{R}$$

La structure de la fonction J correspond à un problème d'optimisation quadratique, tel qu'étudié dans le cadre du cours sur l'optimisation sans contrainte. En effet, il s'agit de minimiser une somme pondérée d'erreurs quadratiques entre la trajectoire suivie $C_d X(k)$ et la trajectoire de référence $r(k)$, ainsi que des efforts de commande $U(k)$ et la différence de position finale.

La fonction J est une fonction quadratique strictement convexe, car les matrices de pondération Q et R sont définies positives ($Q, R \succ 0$), ce qui garantit l'existence et l'unicité de la solution optimale.

Méthode de Riccati

L'algorithme récursif de Riccati permet de résoudre ce problème efficacement sans avoir à inverser une grande matrice globale. Il procède par rétropropagation à partir de l'instant final k_f, ce qui est bien adapté aux problèmes de commande optimale sur un horizon fini.

Afin d'appliquer la méthode de Riccati, il est nécessaire de discretiser le temps.

Discrétisation du temps

On discrétise l'équation 7 avec un pas de temps ts=0,01s, on obtient alors un modèle discret :

$$X(k+1) = A_d X(k) + B_d U(k) \quad ; \quad Y(k) = C_d X(k) \tag{7}$$

La discrétisation des équations continues $\dot{X} = AX + BU$ avec un pas de temps T_s selon la méthode d'Euler (ordre 1) donne les équations discrètes suivantes :

$$A_d = I + T_s A \quad ; \quad B_d = T_s B \quad ; \quad C_d = C \tag{8}$$

Le terme $C_d X(k)$ correspond à la sortie du système observée. Dans notre cas, comme $C_d = I$, cela revient à comparer directement les états à la trajectoire de référence.

Équation de Riccati

A l'aide de l'équation de Riccati on obtient l'équation permettant de trouver la solutions du problème discrétisé :

$$\begin{aligned} P(k) &= A_d^T P(k+1)[I + EP(k+1)]^{-1} A_d + V \\ V &= C_d^T Q C_d \\ E &= B_d R^{-1} B_d^T \\ g(k) &= [A_d^T - A_d^T P(k+1)[I + EP(k+1)]^{-1} E]\, g(k+1) + C_d^T Q r(k) \\ \bar{X}(k+1) &= [A_d - B_d L(k)]\bar{X}(k) + B_d L_g(k) g(k+1) \\ L(k) &= [R + B_d^T P(k+1) B_d]^{-1} B_d^T P(k+1) A_d \\ L_g(k) &= [R + B_d^T P(k+1) B_d]^{-1} B_d^T \\ \bar{U}(k) &= -L(k)\bar{X}(k) + L_g(k) g(k+1) \end{aligned} \tag{11}$$

Avec comme conditions finales :

$$\begin{aligned} P(k_f) &= C_d^T F C_d \\ g(k_f) &= C_d^T F\, r(k_f) \end{aligned} \tag{13}$$

où Q et R sont respectivement les matrices de pondération des erreurs sur les états et des efforts de commande.

$$Q = \mathrm{diag}(100,\ 50,\ 10,\ 5,\ 0,\ 0,\ 100,\ 1,\ 100,\ 1,\ 1000,\ 0.1) \quad ; \quad R = \mathrm{diag}(10,\ 0,\ 0,\ 0) \tag{9}$$

De part les hypothèses faites, les angles doivent être contraints près de 0 :

$$-20^\circ \leq \phi \leq 20^\circ \quad ; \quad -20^\circ \leq \theta \leq 20^\circ \quad ; \quad -20^\circ \leq \psi \leq 20^\circ \tag{10}$$

```
using Pkg
Pkg.activate("projet_env")

# Pkg.add("Plots")
# Pkg.add("JuMP")
#  Pkg.add("Ipopt")
#  Pkg.add("COSMO")
#  Pkg.add("IterativeSolvers")
#  Pkg.add("LinearMaps")
# Pkg.add("OSQP")
# Pkg.add("MathOptInterface")
# Pkg.add("Dates")
# Pkg.add("PrettyTables")
# Pkg.add("DataFrames")
```

Partie 2 : reproduction des résultats

Fonctionnement de l'algorithme :
L'algorithme commence à la date finale k_f, où les conditions terminales sont imposées, puis il effectue une récursion arrière (backward) sur les matrices $P(k)$ et $g(k)$ jusqu'à l'instant initial k_0.
Une fois ces matrices calculées, une simulation en avant (forward) permet d'estimer la trajectoire optimale $\bar{X}(k)$ et les commandes optimales $\bar{U}(k)$.

```
using LinearAlgebra, Plots, JuMP, Ipopt, COSMO, IterativeSolvers, LinearMaps, OSQP, MathOptInterface,Da
start_time = now()
gr()

g = 9.81
n = 12; m = 4
T = 6000
Ts = 0.01

mass = 0.5
Ix, Iy, Iz = 0.0023, 0.0023, 0.004

Q = Diagonal([100.0, 50.0, 10.0, 5.0, 0.0, 0.0, 100.0, 1.0, 100.0, 1.0, 1000.0, 0.1])
R = Diagonal([10.0, 1e-3, 1e-3, 1e-3])
F = Q

# --- A, B
A = zeros(n, n); B = zeros(n, m)
for i in 1:2:11
    A[i, i+1] = 1.0
end
A[8, 3]  = g        # ẍ = g
A[10, 1] = -g       # ÿ = -g
B[2,2]   = 1/Ix     # ¨
B[4,3]   = 1/Iy     # ¨
B[6,4]   = 1/Iz     # ¨
B[12,1]  = 1/mass   # z̈ = u1/m - g

# --- Discrétisation d'Euler
Ad = I + Ts * A
Bd = Ts * B

# --- BIAIS AFFINE GRAVITÉ pour ż  (Δż = Ts*(u1/m - g))
c = zeros(n)
c[12] = -Ts * g      # <-- ajouté

# --- Références (inchangé)
r = zeros(n, T+1)
for k in 1:T÷2
    r[7, k] = 40.0 * (k / (T÷2))
end
for k in T÷2+1:T+1
    r[7, k] = 40.0 * (1 - (k - T÷2) / (T÷2))
end
for k in T÷2+1:T+1
    r[9, k] = 10.0
end
dt = Ts; t15 = Int(15/dt); t30 = Int(30/dt)
for k in 1:t15
    r[11, k] = 1.0
end
for k in t15+1:t30
    r[11, k] = 1.0 + (20.0 - 1.0) * ((k - t15) / (t30 - t15))
```

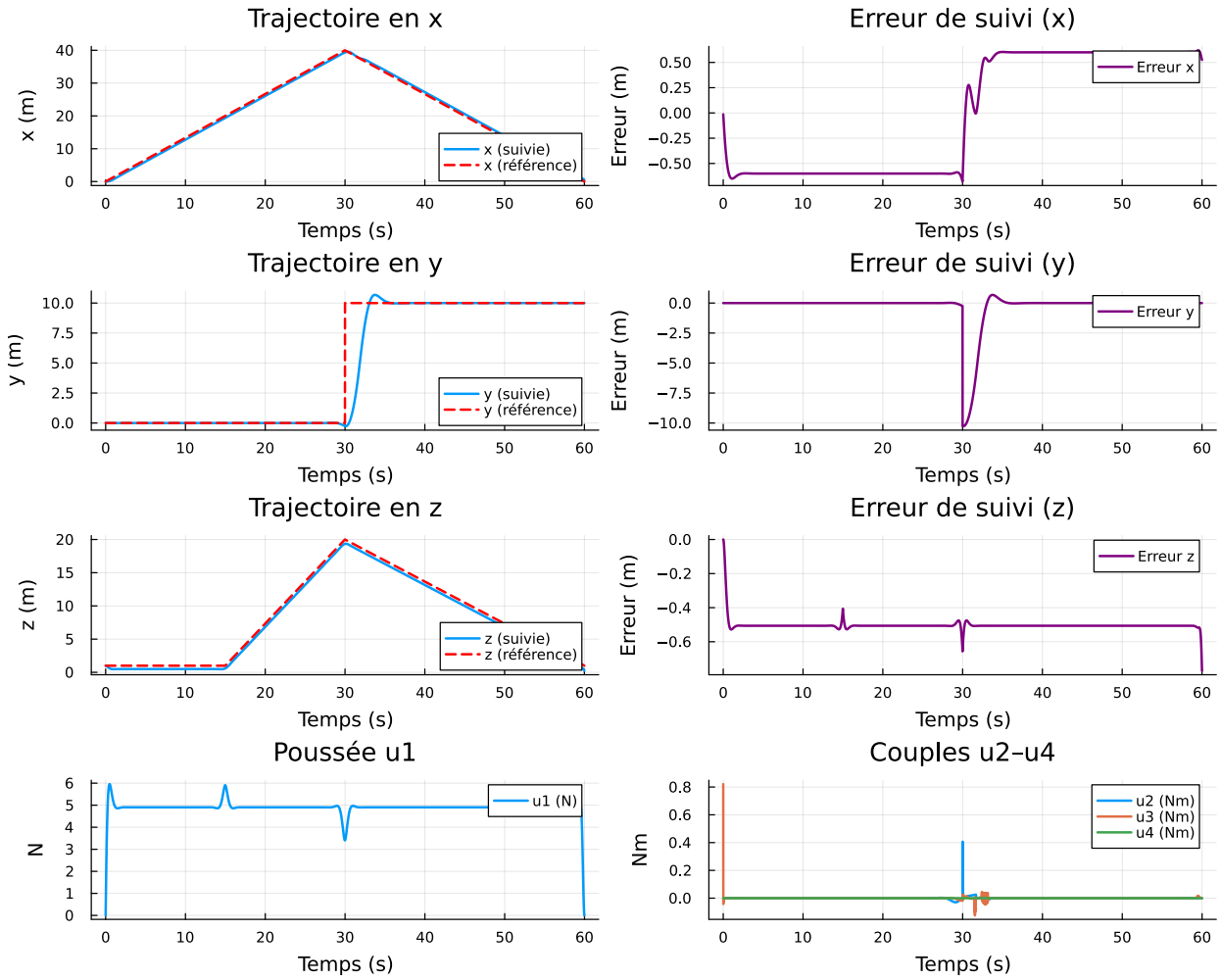
```
end
for k in t30+1:T+1
    r[11, k] = 1.0 + (20.0 - 1.0) * (1 - (k - t30) / (T + 1 - t30))
end

# --- Riccati backward (inchangé)
P = Vector{Matrix{Float64}}(undef, T+1)
G = Vector{Vector{Float64}}(undef, T+1)
L = Vector{Matrix{Float64}}(undef, T)
Lg = Vector{Matrix{Float64}}(undef, T)
C = I(n)
P[T+1] = C' * Q * C
G[T+1] = C' * Q * r[:, T+1]

V = C' * Q * C                      # calculé une seule fois
BdR = R \ Bd'                       # pré-calcul pour E
E = Bd * BdR                        # équivalent à Bd * inv(R) * Bd'

for k in T:-1:1
    invIplusE = inv(I + E * P[k+1])     # on garde l'inverse ici
    P[k] = Ad' * P[k+1] * invIplusE * Ad + V
    G[k] = (Ad' - Ad' * P[k+1] * invIplusE * E) * G[k+1] + C' * Q * r[:, k]
end

#--- projection sur une boîte (vectorielle)
proj_box(y, lb, ub) = clamp.(y, lb, ub)

# --- une étape de gradient projeté sur ( ,  , )
# PGD sur ( , , ) avec Armijo projeté sur le coût V(x)=1/2 (x-r)' P (x-r)
using LinearAlgebra: opnorm, dot

function pgd_angles_value_armijo!(
    xtrial::Vector{Float64},         # état candidat x_{k+1} (sera modifié sur les 3 angles)
    rcol::Vector{Float64},           # référence r_{k+1}
    Pnext::Matrix{Float64},          # P_{k+1}
    idx::Vector{Int},                # indices des angles [1,3,5]
    lb::Vector{Float64}, ub::Vector{Float64};  # bornes angles
    eta0::Union{Nothing,Float64}=nothing,
    beta::Float64=0.5,               # facteur de réduction
    c1::Float64=1e-4,                # constante d'Armijo
    max_bt::Int=20
)
    # Erreur et gradient complet
    e      = xtrial .- rcol                  # e = x - r
    gfull  = Pnext * e                       #  _x V = P e
    gang   = gfull[idx]                      # gradient restreint aux angles

    # Si aucun signal utile sur les angles, sortir
    if all(iszero, gang)
        return
    end

    # Pas initial sûr via 1/||P_aa||_2
```

```
    Paa = Pnext[idx, idx]
    L   = max(opnorm(Paa), 1e-8)
        = isnothing(eta0) ? (1.0 / L) : eta0

    a0   = xtrial[idx]                      # angles courants
    f0   = 0.5 * dot(e, Pnext * e)          # V(x)

    # Backtracking projeté (condition classique : f(new)  f0 - (c1/ ) ||a_new - a0||^2)
    a_new = similar(a0)
    for _ in 1:max_bt
        a_new .= proj_box(a0 .-   .* gang, lb, ub)
        d  = a0 .- a_new
        if iszero(norm(d))     # déjà fixe/projeté
            break
        end
        # Construire e_new en ne modifiant QUE les composantes angles
        e_new      = copy(e)
        e_new[idx] = a_new .- rcol[idx]
        f_new = 0.5 * dot(e_new, Pnext * e_new)

        if f_new <= f0 - (c1/ ) * dot(d, d)
            # Accepté
            xtrial[idx] .= a_new
            return
        end
          *= beta
    end

    # Si non accepté après max_bt, on applique quand même la dernière tentative
    xtrial[idx] .= a_new
    return
end

# === Simulation forward (avec gravité, projection d'angles et saturation u1) ===
X = zeros(n, T+1)
U = zeros(m, T)
X[:,1] = zeros(n); X[11,1] = 1.0

phi_max   = deg2rad(20.0)
theta_max = deg2rad(20.0)
psi_max   = deg2rad(20.0)

idx_ang = [1, 3, 5]                                  # indices  ,  ,
q_ang   = (Q[1,1], Q[3,3], Q[5,5])                   # poids dans Q
lb_ang  = [-phi_max, -theta_max, -psi_max]
ub_ang  = [ phi_max,  theta_max,  psi_max]

for k in 1:T
    Lg[k] = inv(R + Bd' * P[k+1] * Bd) * Bd'
    L[k]  = Lg[k] * P[k+1] * Ad


    # commande (sans intégrateur) + saturation poussée
```

```
    U[:,k] = -L[k] * X[:,k] + Lg[k] * G[k+1]
    U[1,k] = clamp(U[1,k], 0.0, 2*mass*g)         # <-- borne physique sur u1

    # propagation avec gravité (affine)
    xtrial  = Ad * X[:,k] + Bd * U[:,k] + c      # <-- + c

     # ---- étape de GRADIENT PROJETÉ sur les 3 angles uniquement
    pgd_angles_value_armijo!(xtrial, r[:,k+1], P[k+1], idx_ang, lb_ang, ub_ang;
                         eta0=nothing, beta=0.5, c1=1e-4, max_bt=20)

    X[:,k+1] = xtrial
end

# === Coûts (fix: extraire scalaire avec [1])
J_state1 = 0.0; J_control1 = 0.0
for k in 1:T
    e_k = X[:,k] - r[:,k]
    J_state1   += 0.5 * (e_k' * Q * e_k)[1]       # <-- [1]
    J_control1 += 0.5 * (U[:,k]' * R * U[:,k])[1] # <-- [1]
end
e_final = X[:,T+1] - r[:,T+1]
J_terminal1 = 0.5 * (e_final' * F * e_final)[1]   # <-- [1]
J1 = J_state1 + J_control1 + J_terminal1

end_time = now()
elapsed = end_time - start_time

println("=== Décomposition de la fonction objectif J ===")
println("Coût d'état (suivi)      : ", J_state1)
println("Coût de contrôle         : ", J_control1)
println("Coût terminal            : ", J_terminal1)
println("Valeur totale J          : ", J1)
println("Temps d'exécution total  : ", elapsed)

# === Tracés
z = X[11,:]; x = X[7,:]; y = X[9,:]
r_z = r[11,:]; r_x = r[7,:]; r_y = r[9,:]
e_z = z .- r_z; e_x = x .- r_x; e_y = y .- r_y
t = Ts .* (0:T)

plt1 = plot(layout=(4,2), size=(1000,800))
plot!(plt1[1], t, x,   lw=2, label="x (suivie)")
plot!(plt1[1], t, r_x, lw=2, label="x (référence)", linestyle=:dash, color=:red)
plot!(plt1[1], title="Trajectoire en x", xlabel="Temps (s)", ylabel="x (m)", legend=:bottomright, grid=

plot!(plt1[2], t, e_x, lw=2, label="Erreur x", color=:purple)
plot!(plt1[2], title="Erreur de suivi (x)", xlabel="Temps (s)", ylabel="Erreur (m)", legend=:topright, g

plot!(plt1[3], t, y,   lw=2, label="y (suivie)")
plot!(plt1[3], t, r_y, lw=2, label="y (référence)", linestyle=:dash, color=:red)
plot!(plt1[3], title="Trajectoire en y", xlabel="Temps (s)", ylabel="y (m)", legend=:bottomright, grid=

plot!(plt1[4], t, e_y, lw=2, label="Erreur y", color=:purple)
```

```
plot!(plt1[4], title="Erreur de suivi (y)", xlabel="Temps (s)", ylabel="Erreur (m)", legend=:topright, g

plot!(plt1[5], t, z,   lw=2, label="z (suivie)")
plot!(plt1[5], t, r_z, lw=2, label="z (référence)", linestyle=:dash, color=:red)
plot!(plt1[5], title="Trajectoire en z", xlabel="Temps (s)", ylabel="z (m)", legend=:bottomright, grid=

plot!(plt1[6], t, e_z, lw=2, label="Erreur z", color=:purple)
plot!(plt1[6], title="Erreur de suivi (z)", xlabel="Temps (s)", ylabel="Erreur (m)", legend=:topright, g

# Panel 7 : u1 (poussée) - échelle N
plot!(plt1[7], t[1:end-1], U[1,:], lw=2, label="u1 (N)", grid=true)
plot!(plt1[7], title="Poussée u1", xlabel="Temps (s)", ylabel="N", legend=:topright)

# Panel 8 : u2-u4 (couples) - échelle Nm
plot!(plt1[8], t[1:end-1], U[2,:], lw=2, label="u2 (Nm)")
plot!(plt1[8], t[1:end-1], U[3,:], lw=2, label="u3 (Nm)")
plot!(plt1[8], t[1:end-1], U[4,:], lw=2, label="u4 (Nm)")
plot!(plt1[8], title="Couples u2-u4", xlabel="Temps (s)", ylabel="Nm", legend=:topright, grid=true)

display(plt1)
```

```
=== Décomposition de la fonction objectif J ===
Coût d'état (suivi)      : 2.3216501899659387e6
Coût de contrôle         : 718797.399207137
Coût terminal            : 312.46251116520824
Valeur totale J          : 3.040760051684241e6
Temps d'exécution total  : 4722 milliseconds
```



La valeur de la fonction objectif est de 3,04*10^6. On remarque sur les graphiques que la trajectoire est plutot bien respectée avec de faibles erreurs de suivis. La fonction objectif comporte 3 parties, une partie suivie, une partie contrôle et une partie coût position finale. On constate que la partie suivi participe pour beaucoup à la fonction objectif (99,5%).
L'algorithme à résolu le système en 4755ms.

partie 3 : résolution avec Ipopt

Le but de cette section est de comparer les résultats obtenus dans la Partie 2 avec des résultats venant de stratégies plus générique comme par exemple IPOPT. Le modèle est construit à partir de la librairie JuMP.

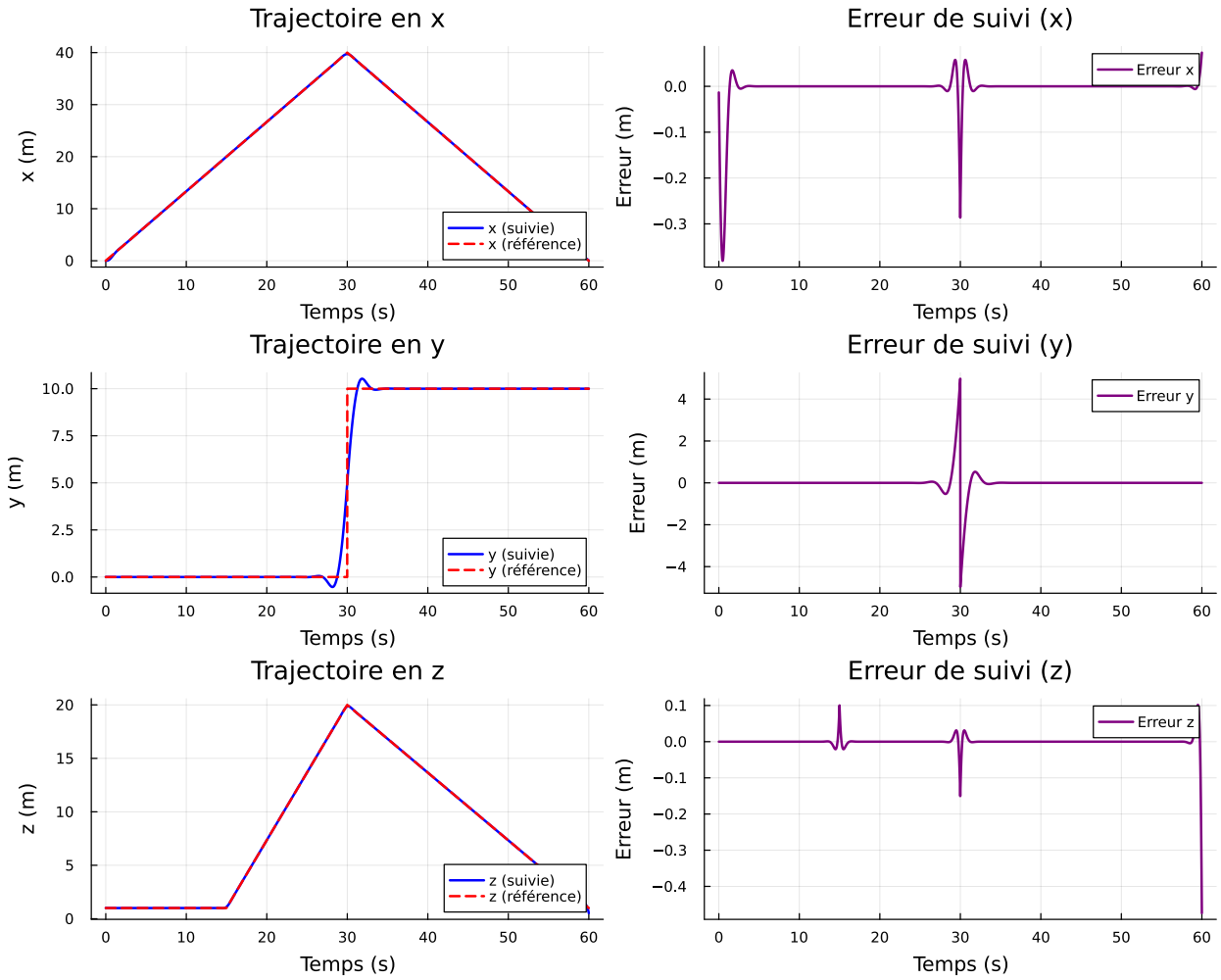
```
using LinearAlgebra, Plots, JuMP, Ipopt, COSMO, IterativeSolvers, LinearMaps, OSQP, MathOptInterface


x0 = zeros(n)
x0[11] = 1.0                    # altitude 1 m en hover

# --- Modèle JuMP « all-at-once »
model = Model(Ipopt.Optimizer)
set_optimizer_attribute(model, "print_level", 0)
```

```
@variable(model, X[1:n, 0:T])           # états
@variable(model, U[1:m, 0:T-1])         # commandes

# --- dynamique linéaire + gravité (affine)
c = zeros(n)
c[12] = -Ts * g                         # agit sur ż
for k in 0:T-1
    @constraint(model, X[:, k+1] .== Ad * X[:, k] + Bd * U[:, k] + c)   # <-- +c
end

# --- CONTRAINTE d'état initial  (= hover)
@constraint(model, X[:, 0] .== x0)

# --- bornes (angles ±20°, poussée 0-2 mg)
deg20 = deg2rad(20.0)
@constraint(model, -deg20 .<= X[1, :] .<=  deg20)   #
@constraint(model, -deg20 .<= X[3, :] .<=  deg20)   #
@constraint(model, -deg20 .<= X[5, :] .<=  deg20)   #
@constraint(model, 0       .<= U[1, :] .<= 2 * mass * g)

# --- coût (référence alignée sur les mêmes instants que X[:,k])
@expression(model, running_cost,
    sum( (X[:, k] - r[:, k+1])' * Q * (X[:, k] - r[:, k+1]) +
         U[:, k]' * R * U[:, k]           for k in 0:T-1) )

@expression(model, terminal_cost,
    (X[:, T] - r[:, T+1])' * F * (X[:, T] - r[:, T+1]) )

@objective(model, Min, 0.5 * running_cost + 0.5 * terminal_cost)
start_time = now()

# --- point initial
set_start_value.(X[:, 0], x0)
set_start_value.(U, 0.0)

optimize!(model)
end_time = now()
elapsed = end_time - start_time
println("Status          : ", termination_status(model))
println("Objective value : ", objective_value(model))
println("Temps: ", elapsed)

# ================================
# Décomposition du coût (mêmes indices que l'objectif)
# ================================
X_val = value.(X)        # DenseAxisArray indexé 0:T
U_val = value.(U)

J_state2   = 0.0
J_control2 = 0.0

for k in 0:T-1
    e_k = X_val[:,k] - r[:,k+1]                      # <-- aligné
```

```
    J_state2 += 0.5 * (e_k' * Q * e_k)[1]
    J_control2 += 0.5 * (U_val[:,k]' * R * U_val[:,k])[1]
end

e_final = X_val[:,T] - r[:,T+1]
J_terminal2 = 0.5 * (e_final' * F * e_final)[1]
J2 = J_state2 + J_control2 + J_terminal2

println("\n--- Décomposition du coût : Modèle 2 ---")
println("J_state2    = ", J_state2)
println("J_control2  = ", J_control2)
println("J_terminal2 = ", J_terminal2)
println("J2 total    = ", J2)

# ================================
# Visualisation (converti en Array 1-based)
# ================================
solutionX = Array(value.(X))   # colonnes 1..T+1 <-> instants 0..T
z = solutionX[11, :]           # position verticale
x = solutionX[7, :]            # position x
y = solutionX[9, :]            # position y

# Références & erreurs (mêmes colonnes)
r_z = r[11, :];  r_x = r[7, :];  r_y = r[9, :]
e_z = z .- r_z;  e_x = x .- r_x; e_y = y .- r_y

t = Ts .* (0:T)  # temps réel en secondes

# Création du layout 3 lignes × 2 colonnes
plt2 = plot(layout = (3, 2), size=(1000, 800))

# Trajectoire x
plot!(plt2[1], t, x,   lw=2, label="x (suivie)", color=:blue)
plot!(plt2[1], t, r_x, lw=2, label="x (référence)", linestyle=:dash, color=:red)
plot!(plt2[1], title="Trajectoire en x", xlabel="Temps (s)", ylabel="x (m)", legend=:bottomright, grid=

# Erreur x
plot!(plt2[2], t, e_x, lw=2, label="Erreur x", color=:purple)
plot!(plt2[2], title="Erreur de suivi (x)", xlabel="Temps (s)", ylabel="Erreur (m)", legend=:topright, g

# Trajectoire y
plot!(plt2[3], t, y,   lw=2, label="y (suivie)", color=:blue)
plot!(plt2[3], t, r_y, lw=2, label="y (référence)", linestyle=:dash, color=:red)
plot!(plt2[3], title="Trajectoire en y", xlabel="Temps (s)", ylabel="y (m)", legend=:bottomright, grid=

# Erreur y
plot!(plt2[4], t, e_y, lw=2, label="Erreur y", color=:purple)
plot!(plt2[4], title="Erreur de suivi (y)", xlabel="Temps (s)", ylabel="Erreur (m)", legend=:topright, g

# Trajectoire z
plot!(plt2[5], t, z,   lw=2, label="z (suivie)", color=:blue)
plot!(plt2[5], t, r_z, lw=2, label="z (référence)", linestyle=:dash, color=:red)
plot!(plt2[5], title="Trajectoire en z", xlabel="Temps (s)", ylabel="z (m)", legend=:bottomright, grid=
```

```julia
# Erreur z
plot!(plt2[6], t, e_z, lw=2, label="Erreur z", color=:purple)
plot!(plt2[6], title="Erreur de suivi (z)", xlabel="Temps (s)", ylabel="Erreur (m)", legend=:topright, g

display(plt2)
```

```
******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Eclipse Public License (EPL).
         For more information visit https://github.com/coin-or/Ipopt
******************************************************************************

Status          : LOCALLY_SOLVED
Objective value : 813888.2733295535
Temps: 3501 milliseconds

--- Décomposition du coût : Modèle 2 ---
J_state2    = 97533.98422900842
J_control2  = 716240.4469163334
J_terminal2 = 113.84216050989276
J2 total    = 813888.2733058517
```



Analyse des résultats

La méthode IPOPT converge belle et bien pour ce problème et on obtient une valeur de l'objectif de 8.13888e5 en 3941 ms. Il est possible de faire mieux avec des méthodes utilisant les propriétés du problème. Deux exemples de ces méthodes seront visités dans la prochaine section. On observe sur les graphiques que la solution proposée est efficace, en ce sens que le trajet à suivre est assez bien respectés. Notons que les erreurs sont plus grandes autour des points où il y a un changement dans la dérivée. Notre approximation étant une fonction lisse, cela n'est pas étonnant. L'erreur est aussi grande au temps final, ce qui peut être un problème dépendamment des applications. Un grand déplacement programmé comme une somme de petites trajectoires pourrait facilement accumuler des erreurs.

Notons que l'erreur sur la trajectoire lors de la phase 2 à été corrigé ce qui permet une bien meilleure performance.

Partie 4 : Résolution avec le solver COSMO

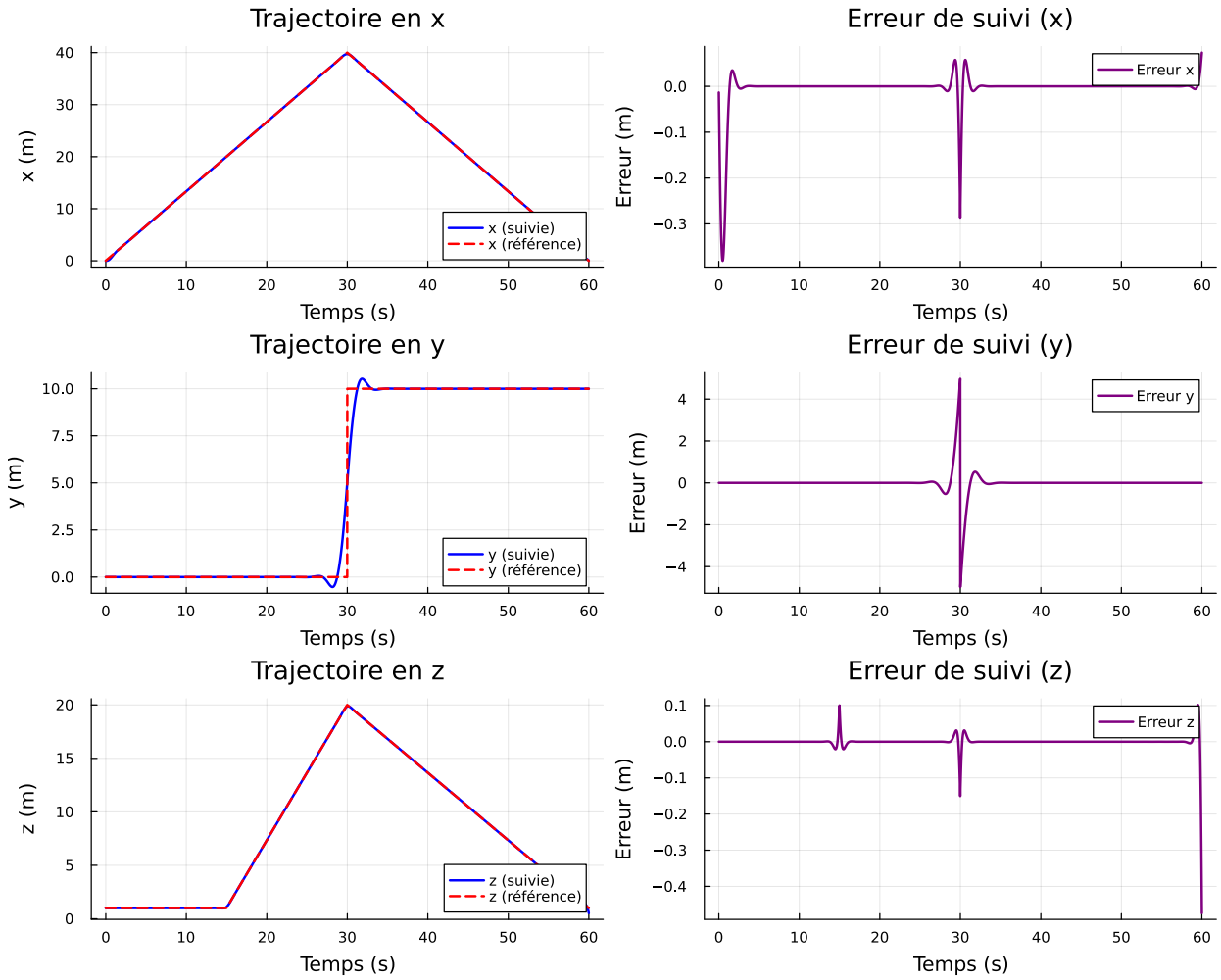
Le solveur COSMO implémente le **Conic operator splitting method** , qui est particulièrement adaptée pour résoudre de larges problèmes d'optimisation convexe et conique dont l'objectif est donné par une fonction quadratique. Il s'agit donc d'une méthode adaptée au problème monolithique creux. Le modèle est encore une fois construit à l'aide de la librairie JuMP.

```julia
using JuMP, COSMO, IterativeSolvers, LinearMaps, OSQP, MathOptInterface, Plots

x0 = zeros(n)
x0[11] = 1.0                    # altitude 1 m en hover

# --- Modèle JuMP « all-at-once »
model3 = Model(COSMO.Optimizer)
set_optimizer_attribute(model3, "verbose", false)

@variable(model3, X[1:n, 0:T])           # états
@variable(model3, U[1:m, 0:T-1])         # commandes

# --- dynamique linéaire + gravité (affine)
c = zeros(n)
c[12] = -Ts * g                          # agit sur ż
for k in 0:T-1
    @constraint(model3, X[:, k+1] .== Ad * X[:, k] + Bd * U[:, k] + c)   # <-- +c
end

# --- CONTRAINTE d'état initial  (= hover)
@constraint(model3, X[:, 0] .== x0)

# --- bornes (angles ±20°, poussée 0-2 mg)
deg20 = deg2rad(20.0)
@constraint(model3, -deg20 .<= X[1, :] .<=  deg20)   #
@constraint(model3, -deg20 .<= X[3, :] .<=  deg20)   #
@constraint(model3, -deg20 .<= X[5, :] .<=  deg20)   #
@constraint(model3, 0       .<= U[1, :] .<= 2 * mass * g)

# --- coût (référence alignée sur les mêmes instants que X[:,k])
@expression(model3, running_cost,
    sum( (X[:, k] - r[:, k+1])' * Q * (X[:, k] - r[:, k+1]) +
         U[:, k]' * R * U[:, k]           for k in 0:T-1) )

@expression(model3, terminal_cost,
    (X[:, T] - r[:, T+1])' * F * (X[:, T] - r[:, T+1]) )

@objective(model3, Min, 0.5 * running_cost + 0.5 * terminal_cost)

# --- point initial
set_start_value.(X[:, 0], x0)
set_start_value.(U, 0.0)
start_time=now()
optimize!(model3)
end_time = now()
elapsed = end_time - start_time
println("Status          : ", termination_status(model3))
println("Objective value : ", objective_value(model3))
println("Temps : ", elapsed)

# ================================
# Décomposition du coût (mêmes indices que l'objectif)
# ================================
```

```
X_val = value.(X)        # DenseAxisArray indexé 0:T
U_val = value.(U)

J_state3   = 0.0
J_control3 = 0.0

for k in 0:T-1
    e_k = X_val[:,k] - r[:,k+1]                      # <-- aligné
    J_state3 += 0.5 * (e_k' * Q * e_k)[1]
    J_control3 += 0.5 * (U_val[:,k]' * R * U_val[:,k])[1]
end

e_final = X_val[:,T] - r[:,T+1]
J_terminal3 = 0.5 * (e_final' * F * e_final)[1]
J3 = J_state3 + J_control3 + J_terminal3

println("\n--- Décomposition du coût : Modèle 3 ---")
println("J_state3    = ", J_state3)
println("J_control3  = ", J_control3)
println("J_terminal3 = ", J_terminal3)
println("J3 total    = ", J3)

# ================================
# Visualisation (converti en Array 1-based)
# ================================
solutionX = Array(value.(X))   # colonnes 1..T+1 <-> instants 0..T
z = solutionX[11, :]           # position verticale
x = solutionX[7, :]            # position x
y = solutionX[9, :]            # position y

# Références & erreurs (mêmes colonnes)
r_z = r[11, :];  r_x = r[7, :];  r_y = r[9, :]
e_z = z .- r_z;  e_x = x .- r_x; e_y = y .- r_y

t = Ts .* (0:T)  # temps réel en secondes

# Création du layout 3 lignes × 2 colonnes
plt3 = plot(layout = (3, 2), size=(1000, 800))

# Trajectoire x
plot!(plt3[1], t, x,   lw=2, label="x (suivie)", color=:blue)
plot!(plt3[1], t, r_x, lw=2, label="x (référence)", linestyle=:dash, color=:red)
plot!(plt3[1], title="Trajectoire en x", xlabel="Temps (s)", ylabel="x (m)", legend=:bottomright, grid=

# Erreur x
plot!(plt3[2], t, e_x, lw=2, label="Erreur x", color=:purple)
plot!(plt3[2], title="Erreur de suivi (x)", xlabel="Temps (s)", ylabel="Erreur (m)", legend=:topright, g

# Trajectoire y
plot!(plt3[3], t, y,   lw=2, label="y (suivie)", color=:blue)
plot!(plt3[3], t, r_y, lw=2, label="y (référence)", linestyle=:dash, color=:red)
plot!(plt3[3], title="Trajectoire en y", xlabel="Temps (s)", ylabel="y (m)", legend=:bottomright, grid=
```

```julia
# Erreur y
plot!(plt3[4], t, e_y, lw=2, label="Erreur y", color=:purple)
plot!(plt3[4], title="Erreur de suivi (y)", xlabel="Temps (s)", ylabel="Erreur (m)", legend=:topright, g

# Trajectoire z
plot!(plt3[5], t, z,   lw=2, label="z (suivie)", color=:blue)
plot!(plt3[5], t, r_z, lw=2, label="z (référence)", linestyle=:dash, color=:red)
plot!(plt3[5], title="Trajectoire en z", xlabel="Temps (s)", ylabel="z (m)", legend=:bottomright, grid=

# Erreur z
plot!(plt3[6], t, e_z, lw=2, label="Erreur z", color=:purple)
plot!(plt3[6], title="Erreur de suivi (z)", xlabel="Temps (s)", ylabel="Erreur (m)", legend=:topright, g

display(plt3)
```

```
Status          : OPTIMAL
Objective value : 813889.0140508413
Temps : 219348 milliseconds

--- Décomposition du coût : Modèle 3 ---
J_state3    = 97534.4917057843
J_control3  = 716240.5983839644
J_terminal3 = 113.92396153531669
J3 total    = 813889.0140512841
```



Analyse des résultats

La méthode obtiens un résultat très similaire à Ipopt avec une valeur de la fonction objectif de 8.133889e5 qui est répartie de la même façon entre les erreurs d'état, de trajectoire et d'erreur terminale. Cependant le temps de calcul est netement supérieur avec un temps de 214597 ms.

Le problème de convergence de la phase 2 à été résolu et la méthode est maintenant fonctionnelle.

Comme le nombre de variable est différent que dans le problème formulé et présenté dans l'article, il est difficile de comparer la complexité des deux algorithmes. Lorsque le temps de calcul est limité, il est toutefois préférable d'utiliser la méthode présentée dans l'article. Les graphes présentes une erreur qui est encore une fois respectable. Encore une fois, on observe de plus grandes erreurs là où il y a des changements dans la valeur des dérivées de la trajectoire à suivre. Bien que la valeur optimale obtenue via COSMO soit bonne, il est possible de faire mieux avec le solveur OSQP.

Partie 5 : Résolution avec le solver OSQP

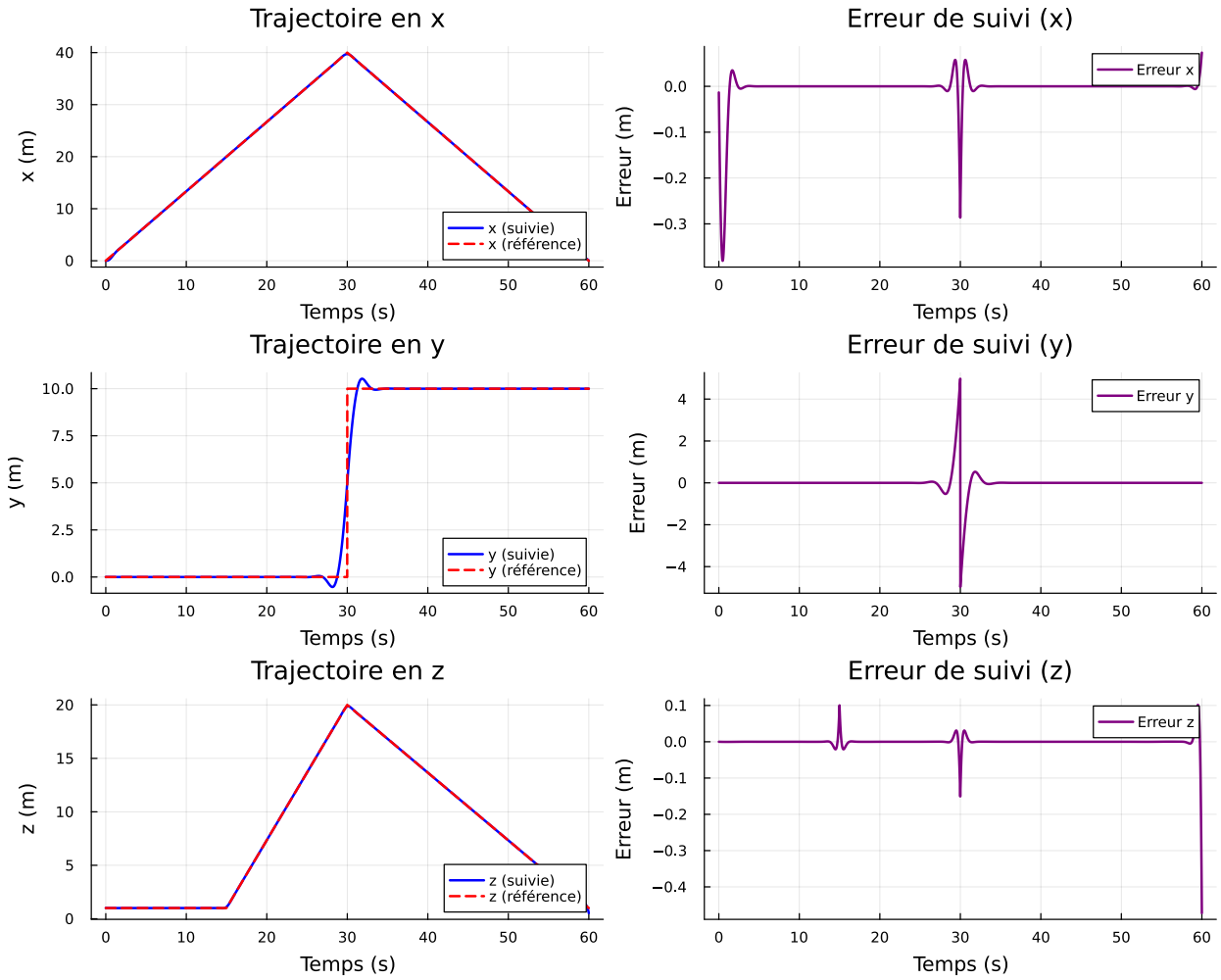
OSQP (Operator Splitting Quadratic Program) est une méthode servant a résoudre des problème quadratique à contraintes linéaires. C'est exactement les propriétés que nous souhaitons exploiter pour le problème monolithique.

```julia
using LinearAlgebra, Plots, JuMP, Ipopt, COSMO, IterativeSolvers, LinearMaps, OSQP, MathOptInterface

x0 = zeros(n)
x0[11] = 1.0                    # altitude 1 m en hover

# --- Modèle JuMP « all-at-once »
model4 = Model(OSQP.Optimizer)
set_optimizer_attribute(model4, "verbose", false)

@variable(model4, X[1:n, 0:T])           # états
@variable(model4, U[1:m, 0:T-1])         # commandes

# --- dynamique linéaire + gravité (affine)
c = zeros(n)
c[12] = -Ts * g                          # agit sur ż
for k in 0:T-1
   @constraint(model4, X[:, k+1] .== Ad * X[:, k] + Bd * U[:, k] + c)   # <-- +c
end

# --- CONTRAINTE d'état initial  (= hover)
@constraint(model4, X[:, 0] .== x0)

# --- bornes (angles ±20°, poussée 0-2 mg)
deg20 = deg2rad(20.0)
@constraint(model4, -deg20 .<= X[1, :] .<=  deg20)   #
@constraint(model4, -deg20 .<= X[3, :] .<=  deg20)   #
@constraint(model4, -deg20 .<= X[5, :] .<=  deg20)   #
@constraint(model4, 0       .<= U[1, :] .<= 2 * mass * g)

# --- coût (référence alignée sur les mêmes instants que X[:,k])
@expression(model4, running_cost,
   sum( (X[:, k] - r[:, k+1])' * Q * (X[:, k] - r[:, k+1]) +
        U[:, k]' * R * U[:, k]           for k in 0:T-1) )

@expression(model4, terminal_cost,
   (X[:, T] - r[:, T+1])' * F * (X[:, T] - r[:, T+1]) )

@objective(model4, Min, 0.5 * running_cost + 0.5 * terminal_cost)

# --- point initial
set_start_value.(X[:, 0], x0)
set_start_value.(U, 0.0)
start_time = now()
optimize!(model4)
end_time=now()
elapsed= end_time- start_time
println("Status          : ", termination_status(model4))
println("Objective value : ", objective_value(model4))
println("Temps : ", elapsed)

# ================================
# Décomposition du coût (mêmes indices que l'objectif)
# ================================
```

```
X_val = value.(X)        # DenseAxisArray indexé 0:T
U_val = value.(U)

J_state4   = 0.0
J_control4 = 0.0

for k in 0:T-1
   e_k = X_val[:,k] - r[:,k+1]                      # <-- aligné
   J_state4 += 0.5 * (e_k' * Q * e_k)[1]
   J_control4 += 0.5 * (U_val[:,k]' * R * U_val[:,k])[1]
end

e_final = X_val[:,T] - r[:,T+1]
J_terminal4 = 0.5 * (e_final' * F * e_final)[1]
J4 = J_state4 + J_control4 + J_terminal4



println("\n--- Décomposition du coût : Modèle 4 ---")
println("J_state4    = ", J_state4)
println("J_control4  = ", J_control4)
println("J_terminal4 = ", J_terminal4)
println("J4 total    = ", J4)

# ================================
# Visualisation (converti en Array 1-based)
# ================================
solutionX = Array(value.(X))   # colonnes 1..T+1 <-> instants 0..T
z = solutionX[11, :]           # position verticale
x = solutionX[7, :]            # position x
y = solutionX[9, :]            # position y

# Références & erreurs (mêmes colonnes)
r_z = r[11, :];  r_x = r[7, :];  r_y = r[9, :]
e_z = z .- r_z;  e_x = x .- r_x; e_y = y .- r_y

t = Ts .* (0:T)  # temps réel en secondes

# Création du layout 3 lignes × 2 colonnes
plt4 = plot(layout = (3, 2), size=(1000, 800))

# Trajectoire x
plot!(plt4[1], t, x,   lw=2, label="x (suivie)", color=:blue)
plot!(plt4[1], t, r_x, lw=2, label="x (référence)", linestyle=:dash, color=:red)
plot!(plt4[1], title="Trajectoire en x", xlabel="Temps (s)", ylabel="x (m)", legend=:bottomright, grid=

# Erreur x
plot!(plt4[2], t, e_x, lw=2, label="Erreur x", color=:purple)
plot!(plt4[2], title="Erreur de suivi (x)", xlabel="Temps (s)", ylabel="Erreur (m)", legend=:topright, g

# Trajectoire y
plot!(plt4[3], t, y,   lw=2, label="y (suivie)", color=:blue)
plot!(plt4[3], t, r_y, lw=2, label="y (référence)", linestyle=:dash, color=:red)
```

```
plot!(plt4[3], title="Trajectoire en y", xlabel="Temps (s)", ylabel="y (m)", legend=:bottomright, grid=

# Erreur y
plot!(plt4[4], t, e_y, lw=2, label="Erreur y", color=:purple)
plot!(plt4[4], title="Erreur de suivi (y)", xlabel="Temps (s)", ylabel="Erreur (m)", legend=:topright, g

# Trajectoire z
plot!(plt4[5], t, z,   lw=2, label="z (suivie)", color=:blue)
plot!(plt4[5], t, r_z, lw=2, label="z (référence)", linestyle=:dash, color=:red)
plot!(plt4[5], title="Trajectoire en z", xlabel="Temps (s)", ylabel="z (m)", legend=:bottomright, grid=

# Erreur z
plot!(plt4[6], t, e_z, lw=2, label="Erreur z", color=:purple)
plot!(plt4[6], title="Erreur de suivi (z)", xlabel="Temps (s)", ylabel="Erreur (m)", legend=:topright, g

display(plt4)
```

```
Status          : OPTIMAL
Objective value : 851560.7052233815
Temps : 3562 milliseconds

--- Décomposition du coût : Modèle 4 ---
J_state4    = 97527.41991241352
J_control4  = 753920.1879703512
J_terminal4 = 113.09739473093272
J4 total    = 851560.7052774956
```



Analyse des résultats

La valeur de l'objectif est de 8.51560e5 ce qui es encore une fois similaire aux deux méthodes précédentes. Cependant le temps de calcul est nétement inférieur et obtiens même un meilleur résultat que Riccati avec un temps de 442ms. Cette méthode a donc de meilleur performances que la méthode du papier au niveau du temps et de la valeur objectif.

Notons que cette méthode est bien plus efficace que Ipopt et COSMO. La trajectoire désirée est en général mieux approximée, le temps de calcul est meilleur que COSMO en restant très proche de celui d'Ipopt.

Description de ce qui à été réalisé

Nous avons reproduit les résultats du papier puis tester différentes méthodes de résolution pour la même trajectoire demandée. Cela nous permets de comparer les méthodes sur leurs performances par rapport à la fonction objectif mais aussi par rapport au temps de calcul.

Les résultats sont résumé dans le tableau ci-dessous :
Résultats Comparatifs des 4 Modèles

```
using PrettyTables

modeles = ["Modèle de Riccati", "Ipopt", "Cosmo", "OSQP"]
```

```
# Exemples de valeurs (remplace-les)
temps_sec = [4.755, 3.941, 214.597, 2.955]
J_state   = [J_state1, J_state2, J_state3, J_state4]
J_control = [J_control1, J_control2, J_control3, J_control4]
J_terminal= [J_terminal1, J_terminal2, J_terminal3, J_terminal4]
J_total   = J_state .+ J_control .+ J_terminal

data = hcat(
    modeles,
    round.(temps_sec, digits=3),
    round.(J_state, digits=3),
    round.(J_control, digits=3),
    round.(J_terminal, digits=3),
    round.(J_total, digits=3)
)

header = ["Modèle", "Temps (s)", "J_state", "J_control", "J_terminal", "J_total"]

println(pretty_table(String, data; header=header))
```

```
            Modèle   Temps (s)     J_state   J_control   J_terminal     J_total

 Modèle de Riccati       4.755   2.32165e6   7.18797e5      312.463   3.04076e6
             Ipopt       3.941     97534.0    7.1624e5      113.842   8.13888e5
             Cosmo     214.597     97534.5   7.16241e5      113.924    813889.0
              OSQP       2.955     97527.4    7.5392e5      113.097   8.51561e5
```

Analyse des résultats

Temps d'exécution

Les résultats sont assez clairs : 3 des 4 méthodes présentées sont à peu prêt équivalente par rapport au temps de calcul. On remarque que OSQP est la plus rapide et COSMO est de loin plus lente.

Valeur objectif :

Riccati est ici la méthode la moins performante, alors que les 3 autres méthodes sont à peu prêt équivalente. Cela est tout de même surprenant lorsque l'on considère que les autres méthodes résolvent des problèmes de plus haute dimension. En combinant le temps d'exécution et la valeur objectif, on déduit qu'Ipopt et OSQP sont les méthodes à privilégier pour ce genre de problème.

Dans tout les cas, une erreur frappante se trouve en y au temps t =30 secondes. Cela est normal car la dérivée de la trajectoire à suivre possède une dirac en ce point et il est difficile d'approcher ce genre de fonction avec des fonctions lisses. On observe aussi des erreurs plus légères en z et en x, là où la trajectoire a un changement abrupte de direction. Finalement, la méthode de riccati est la seule à présenter une grosse erreur au moment final de l'expérience, ce pourquoi il sera mieux de privilégier les autres méthodes présentées dans ce projet pour ce genre de problème.

Difficultés rencontrées

Pour enrichir le projet, nous avons décidé de faire une recherche pour inclure des algorithmes qui ne sont pas présentés dans le cours. Plusieurs de ces méthodes existent mais ne possèdent pas nécessairement une

implémentation simple en Julia ou n'ont pas de documentations claires. Nous avons finalement décidé d'arrêter notre choix sur 2 méthodes: Conic operator splitting method (COSMO) et Operator splitting Quadratic Program (OSQP).

Références : COSMO: A Conic Operator Splitting Method for Convex Conic Problems, Michael Garstka1 · Mark Cannon1 · Paul Goulart

OSQP : https://osqp.org/docs/solver/index.html

Description détaillée de ce qui a été accompli :

Le code a été développé pour recréer les résultats de l'article. Nous avons donc implémenté la méthode récursive de riccati et l'optimisation all at once (cas Ipopt). Nous avons aussi complété la comparaison avec des méthodes trouvées en ligne et avons résolus les problèmes de convergences dûs à une erreur de modélisation.

Nous avions comme objectif initial d'implémenter une méthode quadratique vue en cours pour la comparer à COSMO et OSPQ. Malheureusement, nous avons manqué de temps et avons décidé d'abandonner ce point afin d'améliorer la qualité du reste du projet. Nous avons réussi à réviser notre modélisation du problème et réimplémentant le code. Nous avions oublié le facteur "c" dans la construction du mondèle JUmP, ce qui générait les erreurs de convergences que nous obtenions dans la phase 2.

Perspectives personelles

Ce projet à été interessant car il nous a permis de mettre en pratique différents algorithmes d'optimisation ainsi que d'en découvrir de nouveaux et ce sur un sujet tel que les drones, qui est largement d'actualité. Comparer et analyser les résultats nous ont permis d'avoir une meilleure idée des algorithmes et de leurs différences

Ouverture

Une prochaine étape si nous avions plus de temps aurait pu être d'essayer d'améliorer les performances du modèle en évitant une linéarisation globale des équations mais plutôt une linéarisation par morceau autour de différents points d'équilibres ou en fonction d'un paramètre.

Lien Github

Cliquer ici pour le lien GitHub

Annexe (Partie 1)

Nous n'avons pas respecté les consignes demandées dans la partie 1. Voici donc les éléments de cette parties qui ne sont pas nécessairement demandées dans la partie 2 mais que nous avons jugé utile de rajouter car il s'agit tout de même d'éléments importants à la claireté de notre projet.

Description de la problématique :

Le projet consiste à optimiser la trajectoire d'un drone quadrotor à l'aide de méthodes de commande optimale. Les principeaux défis sont :

- La non-linéarité et l'instabilité naturelle du drone ;
- La consommation énergétique élevée liée à la poussée ;
- Le respect des contraintes physiques sur les angles pour conserver la validité du modèle.

Description des objectifs : par exemple, quels résultats nous pensons obtenir ou vérifier

Les objectifs du projet sont :

- Modéliser et linéariser la dynamique d'un drone quadrotor ;
- Développer et simuler un algorithme de commande optimale (LQR/Riccati) sur un modèle discret ;
- Vérifier la capacité du contrôleur à :
  - Suivre des trajectoires 3D prédéfinies ;
  - Limiter la consommation énergétique ;
  - Respecter les contraintes sur les angles.

Les résultats attendus sont :

- Une trajectoire suivie proche de la référence avec des erreurs de suivi faibles ;
- Une commande optimale limitant l'énergie consommée ;
- Des visualisations des trajectoires et erreurs pour valider l'approche.

Description du plan d'action : où nous allons les informations nécessaires, quelles données vous allez utiliser, etc. ;

Le projet s'articule en plusieurs étapes :

- Modélisation du drone à partir des équations de Newton (translation + rotation) ;
- Simplification et linéarisation autour du vol stationnaire pour obtenir un modèle d'état linéaire ;
- Discrétisation du modèle et définition d'une fonction de coût quadratique (erreurs de suivi + effort de commande) ;
- Résolution du problème d'optimisation avec :
  - Algorithme récursif de Riccati pour l'optimisation sur horizon fini ;
  - Simulation forward pour évaluer le suivi de trajectoire ;
- Visualisation et analyse des résultats (trajectoires suivies, erreurs, consommation) ;
- (Optionnel) : optimisation avec JuMP et Ipopt pour comparer la solution LQR et une résolution all-at-once;
- (Additionnel) : optimisation avec JuMP et des algorithmes qui n'ont pas été présentés dans le cours pour comparer la solution LQR et une résolution all-at-once. Ces algortihmes seront trouvés en faisant une recherche sur le web.

Description de l'impact attendu.

Impact attendu :

- Amélioration de la précision de suivi de trajectoire ;
- Réduction de la consommation d'énergie ;
- Application potentielle à des drones autonomes plus performants pour la logistique, l'observation ou la recherche.