Piana Victor

20/05/2025

Rapport TP 3 MV57

Table des matières

Exercice 1 — Détection de la pose caméra à l'aide d'un marqueur ArUco

(Notons que ce tp a été réalisé sur le webcam de mon pc)

1. Détection d'ArUco

Pour cette partie j'ai utilisé OpenCV et son module ArUco. J'ai utilisé un ArUco affiché sur mon téléphone. Ensuite j'ai mis en place la détection en suivant un peu ce qu'il y avait sur le lien du tuto OpenCV. En gros, on détecte le marqueur dans l'image et après on utilise estimatePoseSingleMarkers pour avoir la pose. Plutôt que d'utiliser solvePnP() de manière manuelle, j'ai utilisé estimatePoseSingleMarkers() qui encapsule ce processus et appelle solvePnP() en interne. Cela m'a permis de me concentrer sur l'objectif principal du TP : la mise en œuvre du filtre de Kalman Étendu.

J'ai récupéré les coordonnées du centre et les axes, ça m'a permis de visualiser le repère avec cv2.drawFrameAxes. J'ai aussi affiché le marqueur avec cv2.aruco.drawDetectedMarkers.

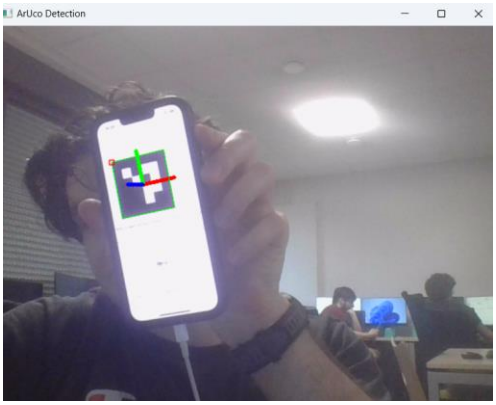
2. Le problème PnP

Le PnP c'est un problème où on veut retrouver la position et l'orientation d'une caméra par rapport à un objet 3D. En gros, on connaît les points 3D (genre les coins d'un ArUco) et les points 2D (ce qu'on voit sur l'image) et on veut savoir où est la caméra qui voit ça. On appelle ça "Perspective-n-Point".

OpenCV le fait avec la fonction solvePnP.

3. Résultat de solvePnP

solvePnP nous donne deux trucs : un vecteur de rotation (rvec) et un vecteur de translation (tvec). Le vecteur de rotation n'est pas directement utilisable, il faut le convertir en matrice ou en angles d'Euler (ou quaternion si on veut plus tard). Donc on ne peut pas utiliser directement rvec dans le Kalman ; il faut d'abord le transformer(avec cv2.Rodrigues).




```
Translation vector: [ 0.09010939 -0.00079546  0.32335539]
Rotation vector: [-3.04296483  0.02668055  0.50910489]
Translation vector: [ 0.08975105 -0.00069871  0.32251418]
Rotation vector: [-3.0680478   0.01641548  0.47729886]
Translation vector: [ 0.09008987 -0.00073846  0.32284937]
Rotation vector: [-3.0680478   0.01641548  0.47729886]
Translation vector: [ 0.09008987 -0.00073846  0.32284937]
Rotation vector: [-3.04342656  0.02654613  0.51171378]
Translation vector: [ 0.09016761 -0.00070054  0.32257331]
Rotation vector: [-3.04135161  0.01895144  0.37636103]
Translation vector: [ 0.09078935 -0.00056336  0.32382836]
```

Exercice 2 — Implémentation du filtre de Kalman Étendu

Dans cet exo, on devait faire une classe qui fait le filtre de Kalman Étendu (EKF). L'idée c'est d'avoir un truc qu'on pourra utiliser pour filtrer des données de pose plus tard, comme celles qu'on récupère avec les ArUco.

Le Kalman Étendu, c'est un peu comme le Kalman normal mais pour les systèmes non linéaires (genre rotations, 3D, etc). Du coup, au lieu d'avoir des équations toutes simples, on doit utiliser les jacobiennes des fonctions pour faire les prédictions et les mises à jour.

En gros, le filtre fait deux choses à chaque fois :

1.a. Prédiction :

- On essaie de prédire ce que devrait être le prochain état du système (x), en fonction de l'état actuel.
- On met aussi à jour la précision de notre estimation (la matrice P), en tenant compte du bruit (Q).

1.b. Mise à jour :

Ensuite, on compare avec ce qu'on observe vraiment (z), et on regarde la différence.

- On ajuste notre estimation de l'état avec le gain de Kalman, qui donne un poids à l'observation par rapport à la prédiction.
- Et on met à jour aussi la matrice de covariance P.

Les équations qu'on applique sont donc :

- Prédiction :
- $x' = f(x_{k-1}, u_{k-1})$
- $P' = A_k \cdot P_{k-1} \cdot A_k^T + W_k \cdot Q_{k-1} \cdot W_k^T$
- Mise à jour :
- $y = z_k - h(x')$
- $S = H_k \cdot P' \cdot H_k^T + V_k \cdot R_k \cdot V_k^T$
- $K = P' \cdot H_k^T \cdot S^{-1}$
- $x = x' + K \cdot y$
- $P = (I - K \cdot H_k) \cdot P'$

2. Ce qu'on a codé

J'ai fait une classe qui s'appelle ExtendedKalmanFilter. Elle prend en entrée :

- les fonctions f (prédiction) et h (mesure),
- leurs jacobiennes respectives,
- les matrices Q (bruit de prédiction) et R (bruit de mesure),
- l'état initial x et la matrice de covariance P.

Il y a deux fonctions dedans :

- predict() qui fait la prédiction.
- update() qui fait la correction.

Conclusion

C'est une version assez générale du filtre. Pour l'instant, on ne l'utilise pas encore vraiment (on verra ça dans l'exo 3), mais elle est prête pour fonctionner avec différents types de modèles.

Exercice 3 — Test du filtre de Kalman Étendu avec un modèle de vélocité + angle d'Euler

Dans cet exercice, j'ai testé le filtre de Kalman Étendu que j'ai codé avant, en l'appliquant à la position de la caméra qu'on récupère avec un marqueur ArUco.

Le but c'est de voir si on peut lisser un peu les valeurs qui bougent tout le temps, même quand on bouge presque pas.

1. Équations de changement d'état

Dans cet exercice, on suit la pose de la caméra (sa position et son orientation) à travers le temps.

On suppose que la caméra peut se déplacer et tourner, donc on modélise son état complet avec 12 variables :

x = [x, y, z, vx, vy, vz, rx, ry, rz, wx, wy, wz]

- x, y, z : position de la caméra dans l'espace
- vx, vy, vz : vitesse linéaire
- rx, ry, rz : angles d'Euler (c'est-à-dire la rotation selon chaque axe)
- wx, wy, wz : vitesse de rotation (appelée aussi vitesse angulaire)

À chaque instant, on suppose que la position change selon la vitesse, et que les angles changent selon la vitesse de rotation.

Donc les équations de prédiction sont :

position :

- x = x + vx * dt
- y = y + vy * dt
- z = z + vz * dt

rotation :

- rx = rx + wx * dt
- ry = ry + wy * dt
- rz = rz + wz * dt

On ne fait pas évoluer les vitesses ici, on suppose qu'elles restent constantes entre deux images (c'est une hypothèse simple, mais suffisante dans notre cas).

```python
# fonction de prédiction de l'état (on ajoute aussi 
def f(x, u=None):
    x_new = x.copy()
    # position = position + vitesse * dt
    x_new[0:3] += x[3:6] * dt
    # angles = angles + vitesses angulaires * dt
    x_new[6:9] += x[9:12] * dt
    return x_new
```

2. Jacobienne de la fonction de prédiction (matrice A)

La matrice A correspond à la dérivée des équations précédentes. En gros, elle dit comment chaque variable de l'état influence les autres après un petit pas de temps. Dans notre cas, seules les positions dépendent des vitesses, et les angles dépendent des vitesses angulaires. Donc A est une matrice 12×12, avec des 1 sur la diagonale (car chaque variable garde sa valeur de base), et des dt pour dire que certaines variables en influencent d'autres.

A =

[1 0 0 dt 0 0 0 0 0 0 0 0]

[0 1 0 0 dt 0 0 0 0 0 0 0]

[0 0 1 0 0 dt 0 0 0 0 0 0]

[0 0 0 1 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 1 0 0 0 0 0 0]

[0 0 0 0 0 0 1 0 0 dt 0 0]

[0 0 0 0 0 0 0 1 0 0 dt 0]

[0 0 0 0 0 0 0 0 1 0 0 dt]

[0 0 0 0 0 0 0 0 0 1 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 1]

```python
# jacobienne de la fonction f par rapport à x
def jacobian_f(x, u=None):
    A = np.eye(12)
    A[0, 3] = dt
    A[1, 4] = dt
    A[2, 5] = dt
    A[6, 9] = dt
    A[7, 10] = dt
    A[8, 11] = dt
    return A
```

Par exemple, la case A[0,3] = dt signifie que x dépend de vx, la vitesse sur x.

3. Fonction de mesure h(x)

Notre webcam nous donne :

- la position de la caméra (x, y, z)
- son orientation sous forme de vecteurs de rotation (rvec)

On transforme ces rvec en angles d'Euler avec OpenCV, ce qui nous donne rx, ry, rz.

Donc au final, la mesure qu'on observe est composée de :

h(x) = [x, y, z, rx, ry, rz]

Ce sont les 6 seules choses qu'on est capable de mesurer.

```python
# fonction de mesure h(x) -> on récupère position + angles
def h(x):
    return x[[0, 1, 2, 6, 7, 8]]  # on récupère [x, y, z, rx, ry, rz]
```

4. Jacobienne de la mesure (matrice H)

La matrice H indique quelles parties de l'état x on observe.

Ici, on observe :

- les 3 premières composantes (position)
- les composantes 6, 7, 8 (angles d'Euler)

Du coup H est une matrice 6×12, avec des 1 aux bonnes positions :

H =

[1 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 1 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 1 0 0 0 0]

[0 0 0 0 0 0 0 0 1 0 0 0]

```python
# jacobienne de la mesure h(x)
def jacobian_h(x):
    H = np.zeros((6, 12))
    H[0, 0] = 1  # x
    H[1, 1] = 1  # y
    H[2, 2] = 1  # z
    H[3, 6] = 1  # angle x
    H[4, 7] = 1  # angle y
    H[5, 8] = 1  # angle z
    return H
```

5. Matrices de bruit Q

La matrice Q représente l'incertitude sur notre modèle (les équations de f).

Même si on suppose que tout se passe bien, on met un petit bruit pour tenir compte des imprécisions ou des accélérations non modélisées.

J'ai mis :

Q = 0.01 × identité 12×12

6. Matrices de bruit R

La matrice R représente le bruit sur les mesures fournies par ArUco (donc sur la position et les angles).

Comme ArUco peut être un peu bruité (surtout les rotations), j'ai mis un peu plus de marge :

R = 0.05 × identité 6×6

```python
Q = np.eye(12) * 0.01  # bruit modèle
R = np.eye(6) * 0.05   # bruit sur mesure : position + angles
```

7. Fonctionnement du filtre et résultats

Voici comment ça se passe dans la boucle principale :

On détecte un marqueur ArUco dans l'image

On récupère :

- tvec : la position brute [x, y, z]
- rvec : le vecteur de rotation qu'on convertit en matrice 3x3

À partir de cette matrice, on calcule les angles d'Euler (rx, ry, rz) en utilisant la formule d'arctangente.

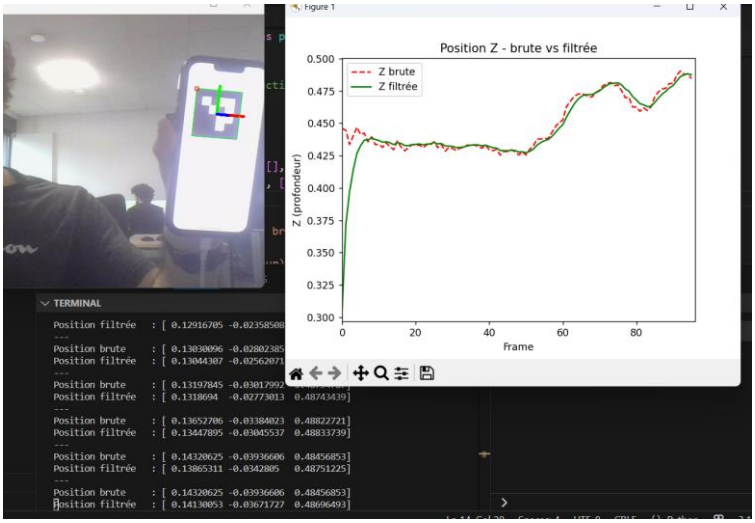
On assemble tout ça dans une mesure z = [x, y, z, rx, ry, rz]

On passe cette mesure au filtre :

- d'abord predict() pour faire une estimation de la pose actuelle
- puis update(z) pour corriger avec ce qu'on observe

```
# === EKF ===
ekf.predict()  # prédiction
ekf.update(z)  # correction avec mesure
```

Exemple de résultat dans le terminal



Au début, le filtre part de zéro donc la position est éloignée. Mais très vite il se cale, et suit la position brute sans suivre tous les tremblements. Les angles sont aussi lissés.

J'ai aussi ajouté un petit affichage graphique avec matplotlib qui montre :

- en rouge pointillés la coordonnée Z brute
- en vert plein la coordonnée Z filtrée

La courbe filtrée est beaucoup plus régulière, et on voit qu'elle suit bien la tendance sans sauter partout.

7.bis Est-ce que le résultat est proche de celui obtenu avec le filtre de Kalman d'OpenCV ?

Avec le filtre de Kalman d'OpenCV (cv2.KalmanFilter), on peut suivre une position simple (par exemple x, y, z + vitesses). Mais il est limité à des modèles linéaires simples, et il ne prend pas en compte directement les rotations (rvecs ou angles d'Euler).

Dans notre implémentation, on utilise un filtre de Kalman Étendu (EKF) codé à la main, qui est capable de filtrer à la fois :

- la position (x, y, z),
- la vitesse linéaire (vx, vy, vz),
- les angles d'Euler (rx, ry, rz),
- les vitesses angulaires (wx, wy, wz).

Donc notre EKF est plus complet, et permet de lisser aussi l'orientation de la caméra, ce que le Kalman d'OpenCV ne fait pas par défaut.

Visuellement, les trajectoires lissées peuvent sembler proches sur l'axe Z, mais notre version gère bien plus de dimensions, ce qui fait une différence en pratique.

Conclusion

Je comprends maintenant mieux comment marche un filtre de Kalman Étendu. Même si les équations qu'on utilise ici sont linéaires, j'ai utilisé l'EKF avec ses fonctions et ses jacobiennes. Le filtre améliore la stabilité de la position détectée, et il est prêt à être utilisé dans d'autres situations plus complexes (ex : rotation avec quaternions dans l'exercice suivant).

Exercice 4 — Utilisation d'un filtre de Kalman Étendu avec des quaternions

Dans ce dernier exercice, on cherche à améliorer le modèle précédent en remplaçant les angles d'Euler par des quaternions. C'est une autre manière de représenter les rotations dans l'espace 3D, qui évite les problèmes d'angles limites et de discontinuités (comme le gimbal lock). C'est un peu plus compliqué à manipuler, mais c'est plus propre sur le long terme, surtout pour des mouvements complexes.

1. Est-ce que le modèle est linéaire ?

Non, le modèle n'est plus linéaire. Avec les angles d'Euler, on pouvait faire quelque chose de simple comme :
rotation = rotation + vitesse_angulaire * dt
Mais avec un quaternion, on ne peut pas faire une addition directe. Il faut :

- convertir la vitesse angulaire [wx, wy, wz] en un quaternion de rotation q_delta, puis
- multiplier ce quaternion avec l'orientation actuelle q_old pour obtenir q_new.

La formule devient donc :
q_new = q_old ⊗ q_delta
Et comme cette multiplication n'est pas linéaire, on est obligé d'utiliser un filtre de Kalman Étendu (EKF).

2. Quel impact ce changement a-t-il sur les matrices ?

a. **Le vecteur d'état x** passe de 12 à 13 valeurs.
Avant :
x = [x, y, z, vx, vy, vz, rx, ry, rz, wx, wy, wz]
Maintenant :
x = [x, y, z, vx, vy, vz, qx, qy, qz, qw, wx, wy, wz]

(avec qw, qx, qy, qz = composantes du quaternion unitaire)

b. **La fonction de prédiction f(x)** change pour la rotation.
La position reste :
x = x + vx * dt
y = y + vy * dt
z = z + vz * dt
Mais pour la rotation, on fait :

- Calcul de l'angle parcouru : theta = ||omega|| * dt
- Normalisation de l'axe : axis = omega / ||omega||
- Construction du quaternion élémentaire :
  q_delta = [sin(theta/2) * axis_x, sin(theta/2) * axis_y, sin(theta/2) * axis_z, cos(theta/2)]
- Application de la rotation :
  q_new = q_old ⊗ q_delta
- Puis normalisation de q_new.

Tout cela est codé dans la fonction f(x) avec AngleAxisToQuat() et quaternion_multiply().

c. **La fonction de mesure h(x)** renvoie maintenant le quaternion, pas les angles.
Donc on mesure :
z = h(x) = [x, y, z, qw, qx, qy, qz]
(7 valeurs au lieu de 6 avant)

d. **La matrice de prédiction A (jacobienne de f)** devient plus compliquée.
La partie position est simple (car x dépend de vx, etc.) :

A = [

[1 0 0 dt 0 0 0 0 0 0 0 0 0]

[0 1 0 0 dt 0 0 0 0 0 0 0 0]

[0 0 1 0 0 dt 0 0 0 0 0 0 0]

[0 0 0 1 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 1   0 0 0 0 0 0 0]

[0 0 0 0 0 0 dq_dq00 dq_dq01 dq_dq02 dq_dq03 0 0 0]

[0 0 0 0 0 0 dq_dq10 dq_dq11 dq_dq12 dq_dq13 0 0 0]

[0 0 0 0 0 0 dq_dq20 dq_dq21 dq_dq22 dq_dq23 0 0 0]

[0 0 0 0 0 0 dq_dq30 dq_dq31 dq_dq32 dq_dq33 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 1 0]

[0 0 0 0 0 0   0 0 0 0 0 0 1]

]

j*'ai laissé dq_dq qui était une fonction que j'avais codé dans une première version (où j'avais oublié qu'un fichier quaternion_utils.py nous était fourni)*

Mais la partie quaternion est dérivée d'une multiplication de quaternions, donc on on utilise une jacobienne spéciale de ce produit (via PredictionJacobian() dans le fichier quaternion_utils.py).

e. **La matrice de mesure H** devient une matrice 7x13 La mesure z étant de taille 7 (position + quaternion), la matrice H est une matrice 7×13 :

H = [

[1 0 0 0 0 0 0 0 0 0 0 0 0], x

[0 1 0 0 0 0 0 0 0 0 0 0 0], y

[0 0 1 0 0 0 0 0 0 0 0 0 0], z

[0 0 0 0 0 0 1 0 0 0 0 0 0], qw

[0 0 0 0 0 0 0 1 0 0 0 0 0], qx

[0 0 0 0 0 0 0 0 1 0 0 0 0], qy

[0 0 0 0 0 0 0 0 0 1 0 0 0] qz]

f. **Les matrices Q et R** changent de taille.

- Q devient 13x13 (car notre état a 13 dimensions maintenant)
- R devient 7x7 (car la mesure contient 7 composantes)

Car on mesure 7 composantes au lieu de 6 (le quaternion est de taille 4)

3. Modifications dans le code pour adapter ce modèle

Voici ce qu'il y a en plus:

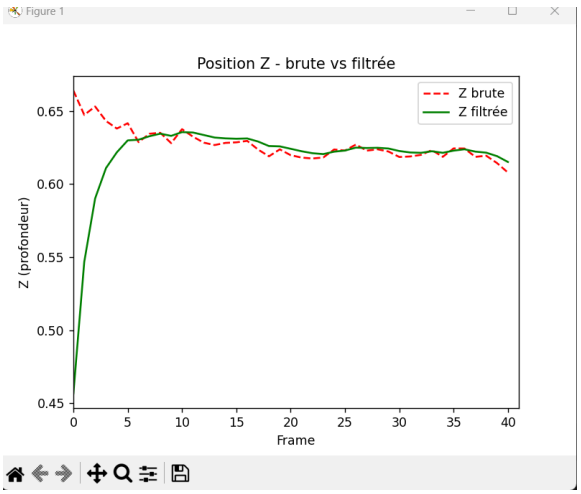
- Ajout des quaternions dans l'état (x[6:10])
- Ajout d'une fonction pour convertir omega en quaternion (AngleAxisToQuat)
- Produit de quaternions pour mettre à jour l'orientation (quaternion_multiply)
- Renormalisation du quaternion après chaque prédiction
- Changement de h(x) pour renvoyer [position, quaternion]
- Modification de la jacobienne jacobian_f() avec la version du prof (PredictionJacobian)
- Utilisation de la vitesse angulaire omega entre deux frames avec inversion du quaternion précédent (pour calculer q_delta)

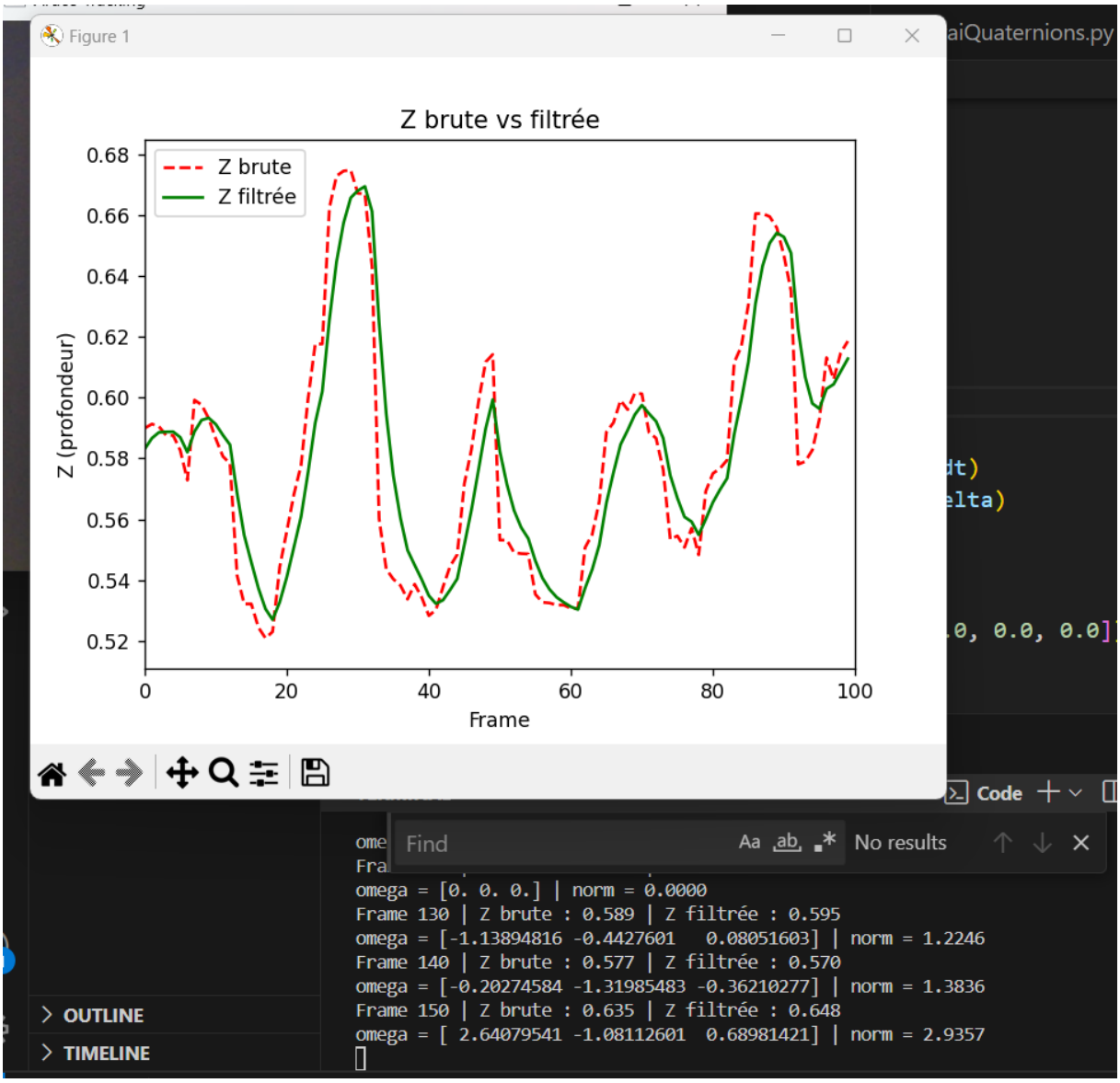
(j'ai également du modifier quelques lignes de code dans le fichier donné par le prof notamment à cause de divisions par 0 qui faisaient planter le système)

4. Quelles sont les limites de ce système ?

Même si le passage aux quaternions apporte plus de la stabilité en 3D, il y a quand même quelques points faibles :

- **C'est plus dur à coder** : il faut des fonctions spéciales pour les quaternions, des jacobiennes plus compliquées, et bien penser à renormaliser régulièrement.
- **C'est moins intuitif** : on ne peut pas visualiser facilement ce que veut dire un quaternion sans le convertir en angles. Donc pour débugger ou visualiser les erreurs, c'est moins pratique.
- **Risque d'erreurs silencieuses** : si on oublie de renormaliser un quaternion, il continue à être utilisé mais il n'est plus valide, ce qui peut faire diverger tout le système.
- **Plus de calculs** : même si ce n'est pas critique ici, les multiplications de quaternions et les calculs associés coûtent un peu plus que des additions classiques.





On remarquera que j'ai rajouté la bibliothèque matplotlib pour aider à mieux visualiser ce qu'il se passe concrètement. J'ai également affiché une frame sur 10 pour que la lecture dans le terminal soit plus facile, et j'ai également affiché omega pour vérifier la cohérence des valeurs (et pour débugué).