[ASPIC]

Rapport TD2 Mécatronique

MOURLANNE Axel
CAMILLE Ulrich

21 février 2024

Summary

Chapter 1

Introduction

Ce TD s'inscrit dans la continuité du TD1, nous avons commencé à nous familiariser avec Webots et le but de ce TD est d'implémenter non plus un MGD (Modele Géometrique Direct) mais un MGI (Modele Géometrique Inverse), et aussi un modèle analytique c'est-à-dire utiliser des relations géométriques du système et non simplement les matrices de transformation dépendantes des actionneurs.

Chapter 2

Robot

Robot est une classe abstraite comportant des méthodes communes aux autres robots qui vont hériter de cette classe.
La Jacobienne est une matrice qui représente la dérivée des degrés de liberté que l'on veut commander par chaque actionneur q_i, elle nécessite donc le calcul de dérivée partielle par chaque q_i que l'on retrouvera plus tard pour chaque robot. Deux méthodes à partir de la jacobienne ont été implémentées.

2.1 MGI

2.1.1 Jacobienne inversée

*

```
    def solveJacInverse(self, joints, target):
        """
        Parameters
        ----------
        joints: np.ndarray shape(n,)
            The initial position for the search in angular space
        target: np.ndarray shape(n,)
            The wished target for the tool in operational space
        """

        target_norm = 0.001
        k1 = 0.1


        while np.linalg.norm(self.computeMGD(joints) - target) > target_norm:
            print("norme diff rence =",np.linalg.norm(self.computeMGD(joints) -
target))
            epsilon = k1*np.linalg.pinv(self.computeJacobian(joints)) @ (target -
self.computeMGD(joints))
            joints += epsilon

        res = np.array([joints[0],joints[1],joints[0]])
        print("joints =",joints)
        return  joints
```

Cette fonction prend en arguments la position actuelle des actionneurs (joints) et la cible (target).

À partir de la condition actuelle (joints), on y rajoute l'inverse de la Jacobienne mutiplié par la différence entre la cible et la position actuelle de l'outil. L'inverse de la jacobienne permet de passser du

monde opérationel (x,y,z) au monde articulaire (qi actionneurs).
La condition de sortie est :

```
self.computeMGD(joints) - target) > target_norm
```

computeMGD(joints) nous redonne la position de l'outil dans l'espace opérationnel depuis l'espace articulaire, et target la cible. Ainsi, epsilon, un vecteur de faible norme, va "diriger" joints vers la solution dans l'espace articulaire tel que l'outil sera le plus proche de target.
Dans cette méthode il est aussi important de noter la présence de np.linalg.pinv car on calcule ici un pseudo inverse ; en effet la jacobienne n'est pas forcément carrée voire même inversible si le nombre de degrés de liberté que l'on souhaite contrôler est différent du nombre d'actionneurs et de leur "capacité".

2.1.2 Jacobienne Transposée

```
    def solveJacTransposed(self, joints, target):
        """
        Parameters
        ----------
        joints: np.ndarray shape(n,)
            The initial position for the search in angular space
        target: np.ndarray shape(n,)
            The wished target for the tool in operational space
        """

        def cost_function(q):
            return np.linalg.norm(target - self.computeMGD(q))**2

        def jac(q):
            return -2 * np.transpose(self.computeJacobian(q)) @ (target - self.
    computeMGD(q))

        joints = opt.minimize(cost_function, joints, jac=jac).x

        return joints
```

La Jacobienne transposée est une autre méthode utilisant non plus une descente de gradient, mais l'outil mathématique du calcul variationel qui permet de chercher une solution voire même des fonctions qui permettent de trouver la solution d'une fonction de coût que l'on cherche à minimiser.

2.1.3 Commentaire

Le temps d'exécution de la méthode inverse est plus long que celui de la méthode transposée, ce qui fait que nous préférons la deuxième solution. Cependant, notre implémentation de la méthode inverse ne semble pas fonctionner parfaitement (en général la bonne solution est trouvée, mais les bras font souvent plusieurs tours avant de trouver leur bonne position).

Concernant la méthode transposée, l'exécution est plus rapide lorsqu'on spécifie la jacobienne dans `scipy.optimize.minimize`.

Mais dans tous les cas, ce sont les méthodes analytiques qui sont les plus rapides, et elles permettent aussi de calculer le nombre de solutions ce qui n'est pas le cas en utilisant la jacobienne. En contrepartie, il est plus difficile de les implémenter, et cette implémentation est très variable en fonction de chaque robot.

Chapter 3

RobotRT

3.1 Limite

Les limites de robotRT sont celles d'un anneau de cercle intérieur r1 = L2 et de cercle extérieur r2 = sqrt($L1^2 + L1^2$). Cependant, nous allons l'englober dans un hyper-rectangle avec des bornes en [-r2 ; r2] sur x et y. ce qui n'est pas une defition exacte de nos limites.

```
    def getOperationalDimensionLimits(self):
        r = math.sqrt(self.L1**2 + self.L2**2)
        return np.array([[-r, r],  [-r, r]])
```

3.2 MGD

```
    def getBaseFromToolTransform(self, joints):
        T_0_1 = self.T_0_1 @ ht.rot_z(joints[0])
        T_1_2 = self.T_1_2 @ ht.translation(joints[1] * np.array([1, 0, 0]))
        return T_0_1 @ T_1_2 @ self.T_2_E

    def computeMGD(self, q):
        tool_pos = self.getBaseFromToolTransform(q) @ np.array([0, 0, 0, 1])
        return tool_pos[:2]
```

La MGD est la composition de toutes les transformations dépendantes de q_i

3.3 Jacobienne

```
    def computeJacobian(self, joints):
        TR_0_1 = self.T_0_1 @ ht.rot_z(joints[0])
        TR_1_2 = self.T_1_2 @ ht.translation(joints[1] * np.array([1, 0, 0]))

        d_TR_0_1 = self.T_0_1 @ ht.d_rot_z(joints[0])
        d_TR_1_2 = self.T_1_2 @ ht.d_translation(joints[1] * np.array([1, 0, 0]))

        d_q1 = d_TR_0_1 @ TR_1_2 @ self.T_2_E
        d_q2 = TR_0_1 @ d_TR_1_2 @ self.T_2_E

        J = np.array([
            [d_q1[0, 3], d_q2[0, 3]],
            [d_q1[1, 3], d_q2[1, 3]]
```

```
        ])

        return J
```

3.4 Analytique

```
    def analyticalMGI(self, target):
        target_dist = np.linalg.norm(target)
        min_dist = np.sqrt(self.L1*self.L1 + self.L2*self.L2)
        max_dist = np.sqrt((self.L1 + self.max_q1)**2 + self.L2*self.L2)

        if target_dist < min_dist or target_dist > max_dist:
            return 0, None

        # On a un triangle O_C_q1 rectangle en q1 (avec O l'origine et C la cible)
        O_q1 = np.sqrt(target_dist*target_dist - self.L2*self.L2)
        q1 = O_q1 - self.L1

        angle_xOC = np.arctan2(target[1], target[0])
        angle_COq1 = np.arctan2(self.L2, O_q1)
        q0 = angle_xOC + angle_COq1

        return 1, [q0, q1]
```

Dans RobotRT, le contrôle de la position se fait sur le plan (x,y) et aura soit une seule solution, soit aucune (dans le cas ou la norme de la cible est supérieur à max_dist).

Chapter 4

RobotRRR

4.1 Limite

Pour les limites du RobotRRR, l'espace opérationel peut être modélisé par un thor, un anneau car le bras rouge tourne autour de l'axe Z et les autres bras restent dans le même plan et peuvent au maximum créer un cercle ; donc un cercle intégré par la rotation en Z donne un thor.
L'équation d'un thor n'étant pas si simple et surtout une fonction paramétrique, nous allons simplement l'englober dans un hyper-rectangle comportant le thor, d'où :

```
        xy_range = self.L1 + self.L2 + self.L3
        return np.array([
            [-xy_range, xy_range],
            [-xy_range, xy_range],
            [self.L0 - self.L2 - self.L3, self.L0 + self.L2 + self.L3]
        ])
```

4.2 MGD

```
    def computeJacobian(self, joints):
        TR_0_1 = self.T_0_1 @ ht.rot_z(joints[0])
        TR_1_2 = self.T_1_2 @ ht.rot_x(joints[1])
        TR_2_3 = self.T_2_3 @ ht.rot_x(joints[2])

        d_TR_0_1 = self.T_0_1 @ ht.d_rot_z(joints[0])
        d_TR_1_2 = self.T_1_2 @ ht.d_rot_x(joints[1])
        d_TR_2_3 = self.T_2_3 @ ht.d_rot_x(joints[2])

        d_q1 = d_TR_0_1 @ TR_1_2 @ TR_2_3 @ self.T_3_E
        d_q2 = TR_0_1 @ d_TR_1_2 @ TR_2_3 @ self.T_3_E
        d_q3 = TR_0_1 @ TR_1_2 @ d_TR_2_3 @ self.T_3_E

        J = np.array([
            [d_q1[0, 3], d_q2[0, 3], d_q3[0, 3]],
            [d_q1[1, 3], d_q2[1, 3], d_q3[1, 3]],
            [d_q1[2, 3], d_q2[2, 3], d_q3[2, 3]],
        ])

        return J
```

Tout comme les autres Robot, on a ici la MGD qui est la composition de toutes les transformations dépendantes de q_i.

4.3 Jacobienne

```
    def getBaseFromToolTransform(self, joints):
        T_0_1 = self.T_0_1 @ ht.rot_z(joints[0])
        T_1_2 = self.T_1_2 @ ht.rot_x(joints[1])
        T_2_3 = self.T_2_3 @ ht.rot_x(joints[2])
        return T_0_1 @ T_1_2 @ T_2_3 @ self.T_3_E

    def computeMGD(self, q):
        tool_pos = self.getBaseFromToolTransform(q) @ np.array([0, 0, 0, 1])
        return tool_pos[:3]
```

4.4 Analytique

```
      solutions = []

        q0 = []
        # si target_x = 0 et target_y = 0, alors infinit  de solutions pour q0
        # sinon, q0 peut  tre  soit dans la direction de la cible soit dans la
    direction oppos e
        is_q0_infinity = (target[0] == 0 and target[1] == 0)
        if is_q0_infinity:
            q0 = [0]
        else:
            angle = np.arctan2(target[1], target[0]) - np.pi/2
            q0 = [angle, angle + np.pi]

        for q0_solution in q0:
            # calcul de la position du bout du bras rouge
            x = np.cos(q0_solution + np.pi/2) * self.L1
            y = np.sin(q0_solution + np.pi/2) * self.L1

            leg_dist = np.sqrt(x*x + y*y)
            target_dist = np.sqrt(target[0]*target[0] + target[1]*target[1])

            # on passe    un rep re local 2D
            local_target_x = np.sqrt((target[0]-x)**2 + (target[1]-y)**2)
            if leg_dist > target_dist: local_target_x *= -1
            local_target_y = target[2] - self.L0
            dist = np.linalg.norm([local_target_x, local_target_y])

            # si la cible est inaccessible depuis le bout du bras rouge, alors pas
    de solutions avec q0_solution
            if dist > self.L2 + self.L3:
                continue

            # loi des cosinus
            alpha = np.arccos((self.L2*self.L2 + dist*dist - self.L3*self.L3) / (2*
    self.L2*dist))
            beta = np.arccos((self.L2*self.L2 + self.L3*self.L3 - dist*dist) / (2*
    self.L2*self.L3))

            phi = np.arctan2(local_target_y, local_target_x)

            # calcul des solutions
```

```
            q1 = phi - alpha
            q2 = np.pi - beta
            solutions.append([q0_solution, q1, q2])
            q1 = phi + alpha
            q2 = beta - np.pi
            solutions.append([q0_solution, q1, q2])

        if not solutions:
            return 0, None
        if is_q0_infinity:
            return -1, solutions[0]
        return len(solutions), solutions[0]
```

Chapter 5

LegRobot

5.1 Limite

Un raisonnement similaire à RobotRRR avec une petite différence en z qui va avoir un offset de L4.

```
        xy_range = self.L1 + self.L2 + self.L3 + self.L4
        return np.array([
            [-xy_range, xy_range],
            [-xy_range, xy_range],
            [self.L0 - self.L2 - self.L3 - self.L4, self.L0 + self.L2 + self.L3 +
 self.L4]
        ])
```

La présence de L4 dans les limites de Z se comprend par la condition d'avoir le dernier bras toujours orienté vers le sol, et donc créer une sorte d'offset sur Z.

Il existera quand même des cas particuliers selon les valeurs de L0,L1,L2... où le thor sera "fermé", c'est-à-dire que x,y = (0,0) sera disponible.

5.2 MGD

```
    def getBaseFromToolTransform(self, joints):
        T_0_1 = self.T_0_1 @ ht.rot_z(joints[0])
        T_1_2 = self.T_1_2 @ ht.rot_x(joints[1])
        T_2_3 = self.T_2_3 @ ht.rot_x(joints[2])
        T_3_4 = self.T_3_4 @ ht.rot_x(joints[3])
        return T_0_1 @ T_1_2 @ T_2_3 @ T_3_4 @ self.T_4_E

    def extractMGD(self, T):
        """
        T : np.arraynd shape(4,4)
           An homogeneous transformation matrix
        """
        return np.array([T[0,3], T[1,3], T[2,3], T[2,1]])

    def computeMGD(self, joints):
        tool_pos = self.extractMGD(self.getBaseFromToolTransform(joints))
        return tool_pos
```

On est toujours dans le même raisonnement, à l'exception de extractMGD qui va chercher des valeurs un peu plus particulières car on contrôle aussi la rotation.

5.3 Jacobienne

```
    def computeJacobian(self, joints):
        TR_0_1 = self.T_0_1 @ ht.rot_z(joints[0])
        TR_1_2 = self.T_1_2 @ ht.rot_x(joints[1])
        TR_2_3 = self.T_2_3 @ ht.rot_x(joints[2])
        TR_3_4 = self.T_3_4 @ ht.rot_x(joints[3])

        d_TR_0_1 = self.T_0_1 @ ht.d_rot_z(joints[0])
        d_TR_1_2 = self.T_1_2 @ ht.d_rot_x(joints[1])
        d_TR_2_3 = self.T_2_3 @ ht.d_rot_x(joints[2])
        d_TR_3_4 = self.T_3_4 @ ht.d_rot_x(joints[3])

        d_q1 = self.extractMGD(d_TR_0_1  @  TR_1_2    @  TR_2_3    @  TR_3_4    @
    self.T_4_E)
        d_q2 = self.extractMGD(TR_0_1    @  d_TR_1_2  @  TR_2_3    @  TR_3_4    @
    self.T_4_E)
        d_q3 = self.extractMGD(TR_0_1    @  TR_1_2    @  d_TR_2_3  @  TR_3_4    @
    self.T_4_E)
        d_q4 = self.extractMGD(TR_0_1    @  TR_1_2    @  TR_2_3    @  d_TR_3_4  @
    self.T_4_E)

        J = np.array([
            [d_q1[0], d_q2[0], d_q3[0], d_q4[0]],
            [d_q1[1], d_q2[1], d_q3[1], d_q4[1]],
            [d_q1[2], d_q2[2], d_q3[2], d_q4[2]],
            [d_q1[3], d_q2[3], d_q3[3], d_q4[3]],
        ])

        return J
```

5.4 Analytique

```
    def analyticalMGI(self, target):

        woffset = np.array([self.W ,0,0])

        solutions = []
        # angle    enlever pour prendre en compte l'offset en W
        d = np.sqrt(target[0]*target[0] + target[1]*target[1])
        offset_angle = np.arctan2(self.W , d)

        angle = np.arctan2(target[1],  target[0] ) - np.pi/2 + offset_angle
        q0 = [angle, angle + np.pi]

        for q0_solution in q0:
            # calcul de la position du bout du bras rouge
            x = np.cos(q0_solution + np.pi/2) * self.L1
            y = np.sin(q0_solution + np.pi/2) * self.L1

            leg_dist = np.sqrt(x*x + y*y)
            target_dist = np.sqrt(target[0]*target[0] + target[1]*target[1])

            # on passe    un rep re local 2D
            local_target_x = np.sqrt((target[0]-x)**2 + (target[1]-y)**2)
```

```
            if leg_dist > target_dist: local_target_x *= -1
            local_target_y = target[2] - self.L0
            dist = np.linalg.norm([local_target_x, local_target_y])

            # on calcule la 'pre_target', c'est-  -dire la cible que doit atteindre
     l'avant-dernier bras (dans le rep re local)
            dist_y = target[3] * self.L4
            dist_x = np.sqrt(self.L4*self.L4 - dist_y*dist_y)
            pre_target_x = local_target_x - dist_x
            pre_target_y = local_target_y - dist_y

            pre_dist = np.linalg.norm([pre_target_x, pre_target_y])

            # si la 'pr  -cible' est inaccessible depuis le bout du bras rouge,
    alors pas de solutions avec q0_solution
            if pre_dist > self.L2 + self.L3:
                continue

            # loi des cosinus
            alpha = np.arccos((self.L2*self.L2 + pre_dist*pre_dist - self.L3*self.
    L3) / (2*self.L2*pre_dist))
            beta = np.arccos((self.L2*self.L2 + self.L3*self.L3 - pre_dist*pre_dist
    ) / (2*self.L2*self.L3))

            phi = np.arctan2(pre_target_y, pre_target_x)

            # calcul des solutions
            q1 = phi - alpha
            q2 = np.pi - beta
            q3 = np.arcsin(target[3]) - q1 - q2
            solutions.append([q0_solution, q1, q2, q3])
            q1 = phi + alpha
            q2 = beta - np.pi
            q3 = np.arcsin(target[3]) - q1 - q2
            solutions.append([q0_solution, q1, q2, q3])

        if not solutions:
            return 0, None
        # if is_q0_infinity:
        #     return -1, solutions[0]
        return len(solutions), solutions[0]
```

Dans la partie analytique géométrique, nous avons essayé de trouver des relations géométriques qui permettent de déterminer la position de tous les actionneurs.
Avec x,y,z notre cible différée par L4 sur Z car la condition d'avoir le dernier bras toujours orienté vers le sol se traduit juste par une cible avec un offset sur Z de L4.
Dans un permier temps, joints[0] est le seul actionneur qui permet de gérer la rotation sur Z ; ainsi on peut déjà facilement trouver une relation.

À partir du couple (x,y) souhaité on peut determiner dans quelle direction joints[0] doit se diriger.

```
        angle = np.arctan2(target[1]-self.W/2, target[0]-self.W/2) - np.pi/2
        q0 = [angle, angle + np.pi]
```

En ce qui concerne les autres bras, le dernier sera considéré simplement comme un offset donc la cible n'est plus x,y,z mais plutot x,y,z + offsetL4 ; ce qui nous permet d'avoir seulement deux bras cherchant une cible. Nous avons donc ainsi un cas géométrique connu qui est un système d'équations plus simple à résoudre qu'un système d'équations avec les 3 bras, sans compter les solutions qui sont quasiment

infinies dans ce cas là.

Enfin, comme le dernier bras doit toujours être orienté vers le bas on a cette relation geométrique entre tous les angles qui donne :

```
q3 = target[3] - q1 - q2
```

Il faut comprendre aussi que pour le contrôle de la rotation sur LegRobot, le choix de r32 dans la matrice de rotation n'est pas anodin. En effet le MGD est au final une compostion de plusieurs tranformations et donc matrice de rotation. Le produit de ces matrices de rotation peut être décomposé en 3 matrices de rotation des angles de rotation les axes x,y et z. le produit des 3 donnant la matrice ci-dessous. Ici on comprend bien que le terme en r32= -cos(psi)*sin(theta) est intéressant car il représente

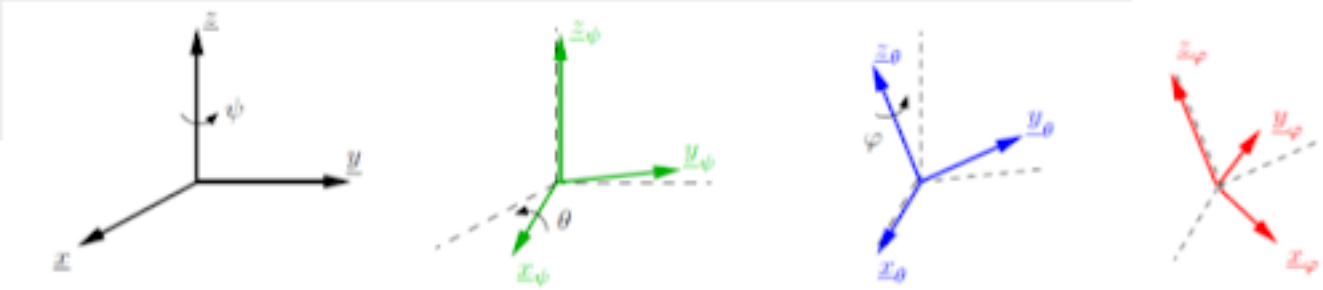

FIG. 1.6 – Rotations successives dans le paramétrage par les angles d'Euler

Sous forme développée :

$$R = \begin{pmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

soit finalement :

$$R = \begin{pmatrix} \cos\psi\cos\varphi - \sin\psi\cos\theta\sin\varphi & -\cos\psi\sin\varphi - \sin\psi\cos\theta\cos\varphi & \sin\psi\sin\theta \\ \sin\psi\cos\varphi + \cos\psi\cos\theta\sin\varphi & -\sin\psi\sin\varphi + \cos\psi\cos\theta\cos\varphi & -\cos\psi\sin\theta \\ \sin\theta\sin\varphi & \sin\theta\cos\varphi & \cos\theta \end{pmatrix}.$$

FIGURE 5.1 – Exemple matrice d'euler

en partie la rotation en z (psi) et la rotation en x (theta, qui nous interesse afin d'orienter le bras vers le sol). Or, la rotation en Z est nulle ; donc cos(psi) = 1, ce qui nous donne simplement -sin(theta). Ainsi si l'on veut theta orienté vers le sol, il nous faut theta = $-\pi/2$ donc r32 = 1, d'où une commande où la cible ne sera pas un angle mais directement r32 = 1.

Chapter 6

Log

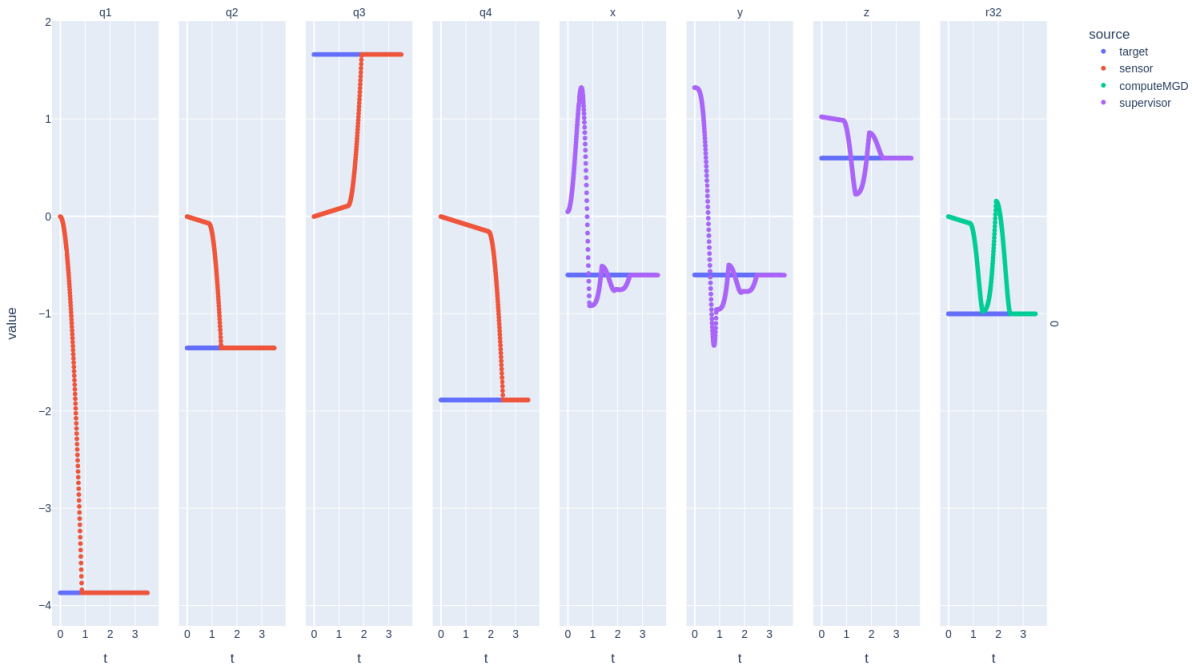

FIGURE 6.1 – Log LegRobot analytique

Dans ce log, on voit bien les valeurs dans l'espace opérationnel et aritculaire converger vers les cibles respectives.