

UNIVERSIDAD POLITÉCNICA DE MADRID

ROBÓTICA
MANIPULADORES

Estudio del manipulador μArm

Autores:

Javier Alonso Silva - javier.asilva@alumnos.upm.es
Roberto Álvarez Garrido - roberto.alvarezg@alumnos.upm.es
José Alejandro Moya Blanco - alejandro.moya.blanco@alumnos.upm.es

Última modificación: 13 de octubre de 2020



Índice

1. Configuración geométrica	4
2. Cinemática directa	6
3. Cinemática inversa	8
4. Matriz Jacobiana	12
4.1. Matriz Jacobiana inversa	14
5. Planificación y descripción de una trayectoria	15
6. Estudio de diferentes casos	16
6.1. Matriz de transformación directa	16
6.2. Matriz de transformación inversa	17
6.3. Matriz Jacobiana	17
6.4. Matriz Jacobiana inversa	17
A. Código Python	19

Índice de figuras

1. zonas de actuación de los motores en el brazo [1]	2
2. rango del manipulador μ Arm [3]	2
3. configuración geométrica del robot	4
4. los grados de libertad del brazo, representados por los diferentes Z_i	4
5. longitudes del brazo robótico [1]	4
6. distribución geométrica del brazo	8
7. plano XY del brazo robótico	9
8. aproximación geométrica del μ Arm	9
9. configuración del brazo robótico donde se puede aplicar el teorema del coseno	10
10. visión del μ Arm en el plano XY	11
11. área de trabajo del μ Arm [1]	13
12. descripción de la trayectoria del brazo robótico	15

Índice de cuadros

1. ángulo de giro de los motores [2]	2
2. longitudes y desviaciones del manipulador	4
3. primera tabla de <i>Denavit–Hartenberg</i>	5
4. primera tabla de <i>Denavit–Hartenberg</i> parametrizada	5
5. segunda tabla de <i>Denavit–Hartenberg</i>	5
6. segunda tabla de <i>Denavit–Hartenberg</i> parametrizada	5
7. tabla de <i>Denavit–Hartenberg</i>	5
8. tabla de <i>Denavit–Hartenberg</i> parametrizada	5

Conocimientos previos

Antes de ponernos a hablar sobre los resultados obtenidos en la práctica, vamos a hablar sobre algunas características básicas del brazo robótico e introducirlo brevemente. El manipulador robótico μ Arm es un dispositivo creado por la empresa [UFACTORY](#) el cual cuenta con cuatro grados de libertad. De dichos grados de libertad, tres son usados para mover el brazo robótico hasta ciertas posiciones y, el último, para mantener el extremo del mismo paralelo al suelo.

El manipulador es controlado mediante cuatro motores:

El motor de la base el cual permite la rotación del manipulador.

En el brazo, el **motor que está a la derecha** (ver la figura 1), coordina el movimiento de la parte inferior del brazo (*Lower Arm* en la figura).

En esta parte del manipulador, el movimiento es como el de un flexo: la parte superior del flexo está supeditada a la parte inferior, de manera que se mantiene de forma constante la altura a la que está el extremo final del mismo.

El otro motor **izquierdo del brazo**, localizado a la izquierda de la imagen, se encarga de controlar la parte superior (*Upper Arm* en la figura) del brazo mediante una junta que controla la estructura triangular superior. Cabe destacar que dado que la articulación situada en el extremo de la parte superior del manipulador no es controlable, esta se mantiene siempre paralela al suelo y con orientación hacia delante. Debido a esta opción de diseño, el manipulador se simplifica y se reduce un grado de libertad, teniendo el manipulador tres grados de libertad (base, parte inferior y parte superior).

El motor localizado en el extremo, se encuentra incluido en el *end-effector* y se utiliza para dotar de movimiento independiente al mismo. Por ejemplo, cuando se coloca una ventosa permite rotarla, cuando se coloca la pinza, el movimiento del motor permite abrirla o cerrarla.

Para este estudio, este último motor se descartará, ya que no afecta a las posiciones accesibles por el robot.

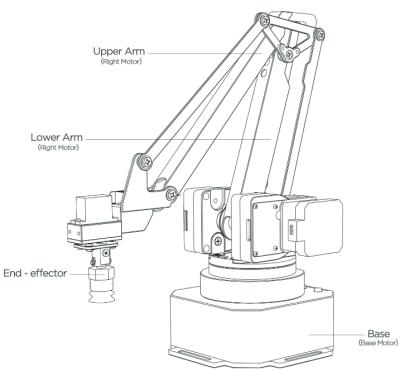


Figura 1: zonas de actuación de los motores en el brazo [1]

Motor	Rango de trabajo
Base	0° ~ 180°
Derecho	0° ~ 130°
Izquierdo	0° ~ 106°
Extremo	0° ~ 180°

Cuadro 1: ángulo de giro de los motores [2]

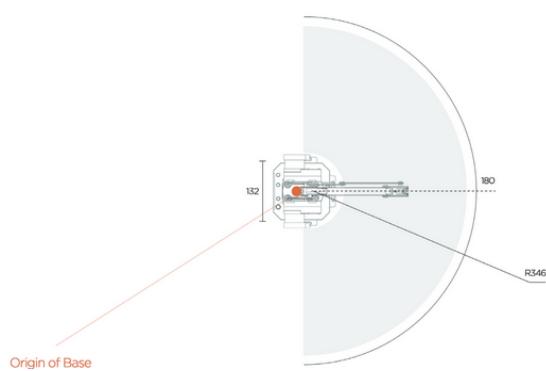


Figura 2: rango del manipulador μ Arm [3]

Toda la información relativa al desarrollo del proyecto puede ser encontrada en [GitHub - UPM Robotics \[4\]](#). Allí están detallados los distintos hitos a conseguir así como más información sobre el robot.

Además, se encuentra disponible la siguiente bibliografía:

- [Manual de usuario](#)
- [Especificaciones](#)
- [Guía del desarrollador](#)
- [Modelo en 3D](#)
- [Web de UFACTORY](#)
- [Soporte de UFACTORY](#)

1. Configuración geométrica

En esta sección vamos a describir la configuración geométrica del brazo robótico. La configuración que obtuvimos fue la siguiente:

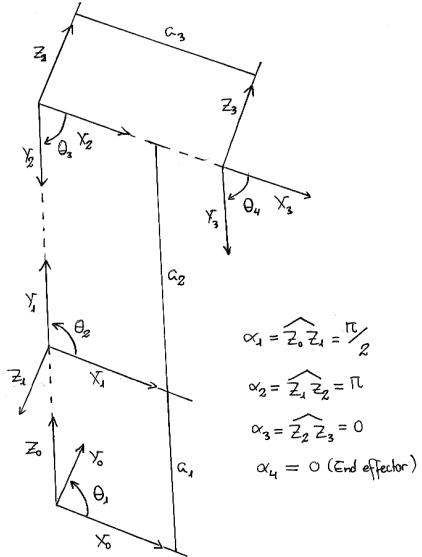


Figura 3: configuración geométrica del robot

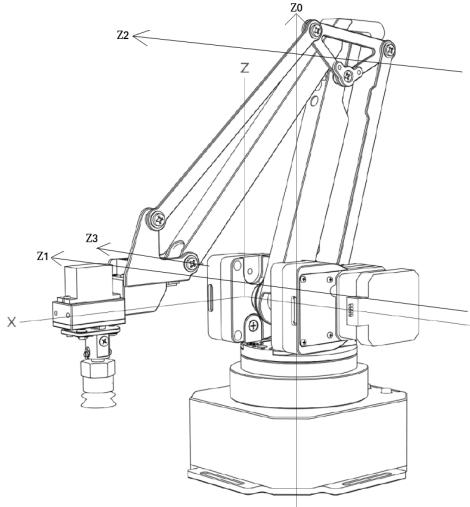


Figura 4: los grados de libertad del brazo, representados por los diferentes Z_i

Usando los datos que están presentes en la documentación al desarrollador [1], pudimos obtener los siguientes datos para los a_i (distancia entre ejes) del manipulador; además, descubrimos que hay una pequeña desviación d_i entre las articulaciones {1, 2}:

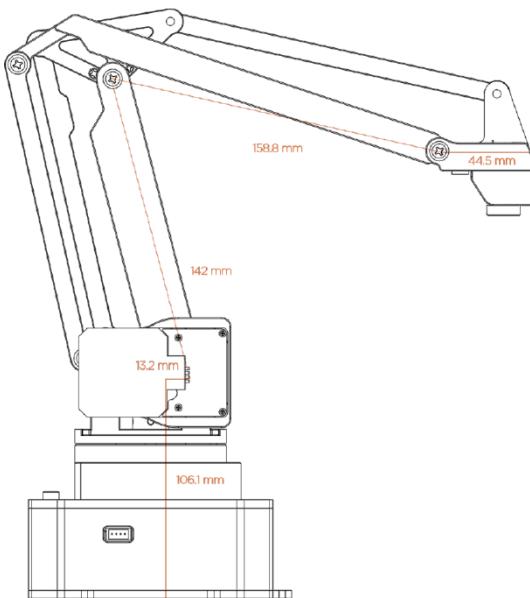


Figura 5: longitudes del brazo robótico [1]

i	a_i (mm.)	d_i (mm.)
1	106,1	13,2
2	142	0
3	158,8	0
4	44,5	0

Cuadro 2: longitudes y desviaciones del manipulador

De esta manera, con los datos obtenidos, generamos una primera matriz de *Denavit-Hartenberg*:

i	θ_i	d_i (mm.)	a_i (mm.)	α_i
1	θ_1	13,2	106,1	$\frac{\pi}{2}$
2	θ_2	0	142	π
3	θ_3	0	158,8	0
4	θ_4	0	44,5	0

Cuadro 3: primera tabla de *Denavit-Hartenberg*

i	θ_i	d_i (mm.)	a_i (mm.)	α_i
1	θ_1	d_1	a_1	$\frac{\pi}{2}$
2	θ_2	0	a_2	π
3	θ_3	0	a_3	0
4	θ_4	0	a_4	0

Cuadro 4: primera tabla de *Denavit-Hartenberg* parametrizada

La cuestión es que, al hacer los distintos modelos, descubrimos que debido a la orientación del brazo robótico, la d_i está en la posición del equivalente a_i , en particular d_1 y a_1 . Esto es debido a que, como se puede ver en la figura 4 junto con las figuras 3 y 5, el plano sobre el que está d_i es el XZ , lo cual nos obliga a cambiar las posiciones en las que existen a la vez un a_i y un d_i , siempre y cuando ambos sean constantes, que en este caso lo son. De esta forma, la tabla de *Denavit-Hartenberg* quedaría:

i	θ_i	d_i (mm.)	a_i (mm.)	α_i
1	θ_1	106,1	13,2	$\frac{\pi}{2}$
2	θ_2	0	142	π
3	θ_3	0	158,8	0
4	θ_4	0	44,5	0

Cuadro 5: segunda tabla de *Denavit-Hartenberg*

i	θ_i	d_i (mm.)	a_i (mm.)	α_i
1	θ_1	a_1	d_1	$\frac{\pi}{2}$
2	θ_2	0	a_2	π
3	θ_3	0	a_3	0
4	θ_4	0	a_4	0

Cuadro 6: segunda tabla de *Denavit-Hartenberg* parametrizada

Finalmente, el *end-effector* (véase la figura 1) siempre está perpendicular al plano del suelo, es decir, $\phi_e = \pi$. Por eso mismo, el parámetro $i = 4$ en verdad está supeditado siempre al movimiento que se realice según los ángulos θ_2 y θ_3 , así que no es necesario contemplarlo en la tabla de *Denavit-Hartenberg*:

i	θ_i	d_i (mm.)	a_i (mm.)	α_i
1	θ_1	106,1	13,2	$\frac{\pi}{2}$
2	θ_2	0	142	π
3	θ_3	0	158,8	0

Cuadro 7: tabla de *Denavit-Hartenberg*

i	θ_i	d_i (mm.)	a_i (mm.)	α_i
1	θ_1	a_1	d_1	$\frac{\pi}{2}$
2	θ_2	0	a_2	π
3	θ_3	0	a_3	0

Cuadro 8: tabla de *Denavit-Hartenberg* parametrizada

2. Cinemática directa

A continuación, con los valores obtenidos en el apartado anterior, vamos a obtener las distintas matrices de transformación de referenciales de manipuladores. Para ello, partimos de la siguiente matriz:

$$A_{i-1}^i = \begin{pmatrix} \cos \theta_i & -\cos \alpha_i \sin \theta_i & \sin \alpha_i \sin \theta_i & a_i \cos \theta_i \\ \sin \theta_i & \cos \alpha_i \cos \theta_i & -\sin \alpha_i \cos \theta_i & a_i \sin \theta_i \\ 0 & \sin \alpha_i & \cos \alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Aplicando los distintos pasos, obtenemos:

$$A_0^1 = \begin{pmatrix} \cos(\theta_1) & 0 & \sin(\theta_1) & d_1 \cos(\theta_1) \\ \sin(\theta_1) & 0 & -\cos(\theta_1) & d_1 \sin(\theta_1) \\ 0 & 1 & 0 & a_1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A_1^2 = \begin{pmatrix} \cos(\theta_2) & \sin(\theta_2) & 0 & a_2 \cos(\theta_2) \\ \sin(\theta_2) & -\cos(\theta_2) & 0 & a_2 \sin(\theta_2) \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A_2^3 = \begin{pmatrix} \cos(\theta_3) & -\sin(\theta_3) & 0 & a_3 \cos(\theta_3) \\ \sin(\theta_3) & \cos(\theta_3) & 0 & a_3 \sin(\theta_3) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Con todas las matrices ya obtenidas, podemos calcular la matriz de transformación directa del manipulador (son 4 columnas, sin embargo no cabe la matriz al completo y por eso la última columna está debajo):

$$A_0^2 = \begin{pmatrix} \cos(\theta_1) \cos(\theta_2) & \sin(\theta_2) \cos(\theta_1) & -\sin(\theta_1) & (a_2 \cos(\theta_2) + d_1) \cos(\theta_1) \\ \sin(\theta_1) \cos(\theta_2) & \sin(\theta_1) \sin(\theta_2) & \cos(\theta_1) & (a_2 \cos(\theta_2) + d_1) \sin(\theta_1) \\ \sin(\theta_2) & -\cos(\theta_2) & 0 & a_1 + a_2 \sin(\theta_2) \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A_0^3 = \begin{pmatrix} \cos(\theta_1) \cos(\theta_2 - \theta_3) & \sin(\theta_2 - \theta_3) \cos(\theta_1) & -\sin(\theta_1) & -\sin(\theta_1) \\ \sin(\theta_1) \cos(\theta_2 - \theta_3) & \sin(\theta_1) \sin(\theta_2 - \theta_3) & \cos(\theta_1) & \cos(\theta_1) \\ \sin(\theta_2 - \theta_3) & -\cos(\theta_2 - \theta_3) & 0 & 0 \\ 0 & 0 & 0 & 0 \\ (a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3) + d_1) \cos(\theta_1) & (a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3) + d_1) \sin(\theta_1) & a_1 + a_2 \sin(\theta_2) + a_3 \sin(\theta_2 - \theta_3) & 1 \end{pmatrix}$$

Las matrices están parametrizadas. Numéricamente, la matriz de transformación directa A_0^3 quedaría:

$$A_0^3 = \begin{pmatrix} \cos(\theta_1) \cos(\theta_2 - \theta_3) & \sin(\theta_2 - \theta_3) \cos(\theta_1) & -\sin(\theta_1) \\ \sin(\theta_1) \cos(\theta_2 - \theta_3) & \sin(\theta_1) \sin(\theta_2 - \theta_3) & \cos(\theta_1) \\ \sin(\theta_2 - \theta_3) & -\cos(\theta_2 - \theta_3) & 0 \\ 0 & 0 & 0 \\ (142,0 \cos(\theta_2) + 158,9 \cos(\theta_2 - \theta_3) + 13,2) \cos(\theta_1) & (142,0 \cos(\theta_2) + 158,9 \cos(\theta_2 - \theta_3) + 13,2) \sin(\theta_1) & 142,0 \sin(\theta_2) + 158,9 \sin(\theta_2 - \theta_3) + 106,1 \\ (142,0 \cos(\theta_2) + 158,9 \cos(\theta_2 - \theta_3) + 13,2) \sin(\theta_1) & 142,0 \sin(\theta_2) + 158,9 \sin(\theta_2 - \theta_3) + 106,1 & 1 \end{pmatrix}$$

Finalmente, hay que añadir una traslación¹ en el eje Z (debido a la posición del *end-effector*) y en el eje X , ya que después de la articulación θ_3 hay una extensión de 44,5 mm. (ver figura 5), para obtener así la posición final del robot ($X_e Y_e Z_e$):

$$A_0^3 = \begin{pmatrix} \cos(\theta_1) \cos(\theta_2 - \theta_3) & \sin(\theta_2 - \theta_3) \cos(\theta_1) & -\sin(\theta_1) \\ \sin(\theta_1) \cos(\theta_2 - \theta_3) & \sin(\theta_1) \sin(\theta_2 - \theta_3) & \cos(\theta_1) \\ \sin(\theta_2 - \theta_3) & -\cos(\theta_2 - \theta_3) & 0 \\ 0 & 0 & 0 \\ (a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3) + d_1) \cos(\theta_1) + T_X & (a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3) + d_1) \sin(\theta_1) & a_1 + a_2 \sin(\theta_2) + a_3 \sin(\theta_2 - \theta_3) - T_Z \\ (a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3) + d_1) \sin(\theta_1) & a_1 + a_2 \sin(\theta_2) + a_3 \sin(\theta_2 - \theta_3) - T_Z & 1 \end{pmatrix} \quad (1)$$

$$A_0^3 = \begin{pmatrix} \cos(\theta_1) \cos(\theta_2 - \theta_3) & \sin(\theta_2 - \theta_3) \cos(\theta_1) & -\sin(\theta_1) \\ \sin(\theta_1) \cos(\theta_2 - \theta_3) & \sin(\theta_1) \sin(\theta_2 - \theta_3) & \cos(\theta_1) \\ \sin(\theta_2 - \theta_3) & -\cos(\theta_2 - \theta_3) & 0 \\ 0 & 0 & 0 \\ (142,0 \cos(\theta_2) + 158,9 \cos(\theta_2 - \theta_3) + 13,2) \cos(\theta_1) + 44,5 & (142,0 \cos(\theta_2) + 158,9 \cos(\theta_2 - \theta_3) + 13,2) \sin(\theta_1) & 142,0 \sin(\theta_2) + 158,9 \sin(\theta_2 - \theta_3) + 106,1 - T_Z \\ (142,0 \cos(\theta_2) + 158,9 \cos(\theta_2 - \theta_3) + 13,2) \sin(\theta_1) & 142,0 \sin(\theta_2) + 158,9 \sin(\theta_2 - \theta_3) + 106,1 - T_Z & 1 \end{pmatrix} \quad (2)$$

De esta forma, obtendríamos las siguientes ecuaciones para ($X_e Y_e Z_e$):

$$\left. \begin{array}{l} X_e = (a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3) + d_1) \cos(\theta_1) + T_X \\ Y_e = (a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3) + d_1) \sin(\theta_1) \\ Z_e = a_1 + a_2 \sin(\theta_2) + a_3 \sin(\theta_2 - \theta_3) - T_Z \end{array} \right\} \quad (3)$$

Además, debido al diseño geométrico de este brazo robótico, la orientación del *end-effector* es siempre perpendicular al suelo ($\phi_e = \frac{\pi}{2}$), pero esto no aporta información útil al robot, ya que mantiene siempre esta orientación (véase las figuras 3 y 4). Por este motivo, durante el desarrollo posterior se asumirá que:

$$\phi_e = \phi_{23} = \theta_2 - \theta_3 \quad (4)$$

y que, por consiguiente, los puntos útiles en este brazo robótico son:

$$P = (X_e, Y_e, Z_e, \phi_{23}) \quad (5)$$

¹ T_X es la traslación en X , mientras que T_Z es la traslación en Z

3. Cinemática inversa

Una vez tenemos las ecuaciones correspondientes a $(X_e \ Y_e \ Z_e)$, podemos calcular la cinemática inversa del μ Arm. Con dicha cinemática, se pueden obtener los ángulos $(\theta_1, \theta_2, \theta_3)$ del brazo robótico.

Antes de realizar los cálculos, se planteó el modelo geométrico del robot, para poder deducir los distintos ángulos que se podían formar y cómo estaba distribuido el brazo:

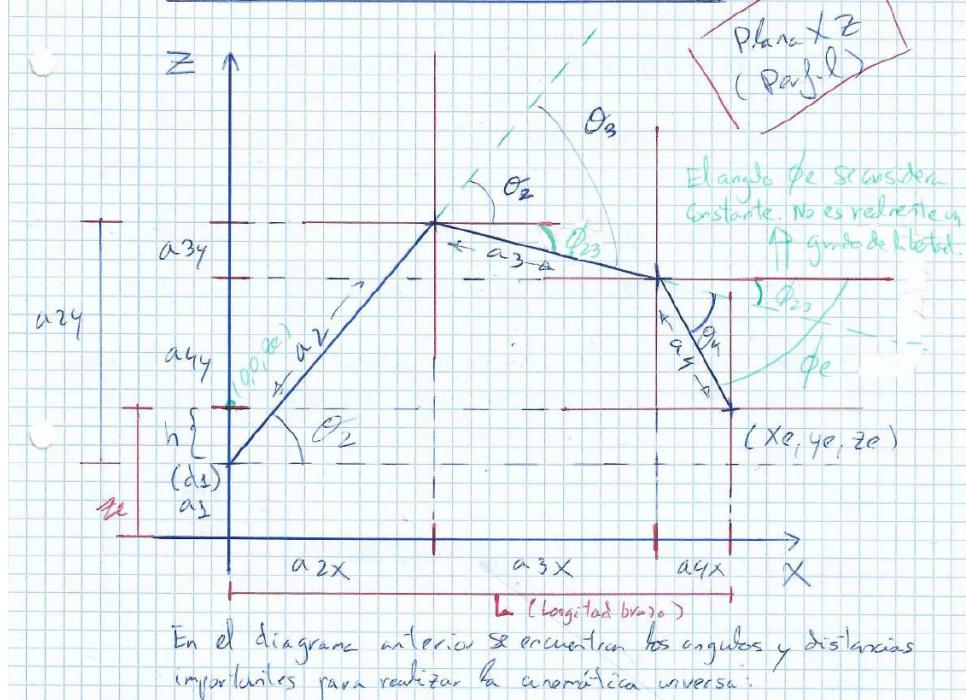


Figura 6: distribución geométrica del brazo

En la figura 6 se puede ver cómo, en el plano XZ (de perfil), se encuentra presente una altura inicial a_1 correspondiente a la base del brazo, así como una pequeña desviación en los ejes d_1 la cual afecta a la posición en Z . A continuación, cada extremidad del brazo está definida junto con la correspondiente articulación θ_i . Como se puede observar, el ángulo θ_2 es positivo pero, debido a cómo está configurado el robot geométricamente, los ángulos θ_3 y θ_4 han de ser negativos².

Como se vio anteriormente en la sección *Conocimientos previos*, en particular la figura 3, se suprimió el último parámetro de la tabla de Denavit–Hartenberg ya que dicho ángulo siempre es perpendicular al plano del suelo y, por ende, está supeditado a los ángulos anteriores θ_2 y θ_3 . En su lugar, se sustituye por una traslación en el eje Z : T_Z (véase las matrices 1 y 2 y la ecuación 3).

De esta forma, se obtuvieron las siguientes ecuaciones para el plano XZ :

$$L = \begin{cases} a_{2X} = a_2 \cdot \cos(\theta_2) \\ a_{3X} = a_3 \cdot \cos(\theta_2 + \theta_3) \\ a_{4X} = a_4 \cdot \cos(\phi_e) \end{cases} \quad (6)$$

$$H = \begin{cases} a_{2Z} = a_2 \cdot \sin(\theta_2) \\ a_{3Z} = a_3 \cdot \sin(\theta_2 + \theta_3) \\ a_{4Z} = a_4 \cdot \sin(\phi_e) \end{cases} \quad (7)$$

²véase la figura 5 para más información sobre la configuración del brazo robótico

Finalmente, se evaluó el plano XY del modelo geométrico del brazo robótico:

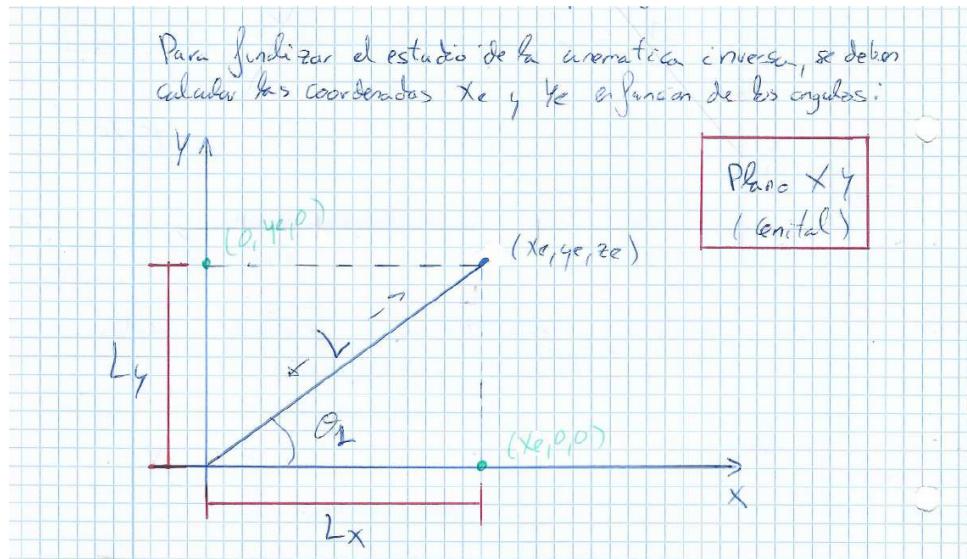


Figura 7: plano XY del brazo robótico

De esta manera, se obtuvieron las siguientes ecuaciones para L_X y L_Y :

$$\begin{cases} L_X = L \cdot \cos(\theta_1) = X_e \\ L_Y = L \cdot \sin(\theta_1) = Y_e \end{cases} \quad (8)$$

La problemática de plantear el modelo de esta forma se mostró a la hora de intentar resolver las ecuaciones de X_e , Y_e y Z_e : al sustituir en la ecuación 3, el sistema a resolver era demasiado complejo, resultando imposible la obtención de alguno de los ángulos, con lo que el sistema no tenía solución.

Como ya se comentó anteriormente, en la tabla de Denavit-Hartenberg se eliminó el último parámetro: pese a que sea un grado de libertad, al estar supeditado a los dos ángulos anteriores y ser siempre perpendicular al plano del suelo, no es necesario tenerlo en cuenta para la obtención de la inversa del robot.

Esto produjo que el último punto efectivo del robot resultara ser el extremo del brazo de la articulación θ_3 :

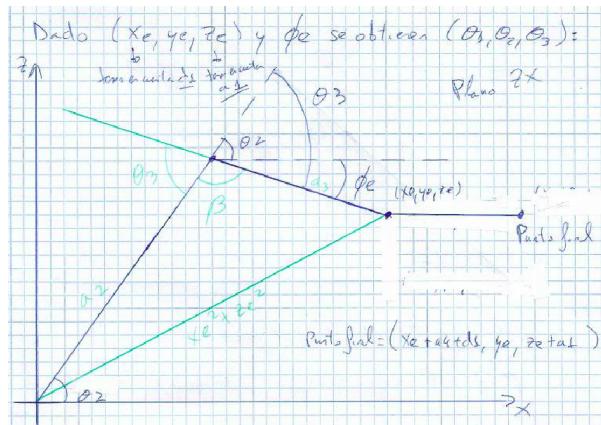


Figura 8: aproximación geométrica del μ Arm

De esta forma, la complejidad se reduce bastante, ya que ahora en el plano XZ solo se cuentan con dos grados de libertad: θ_2 y θ_3 , además de la traslación en X que se comentó anteriormente (T_X) y que se puede ver en la figura 8.

Esta nueva configuración permite aplicar el teorema del coseno³:

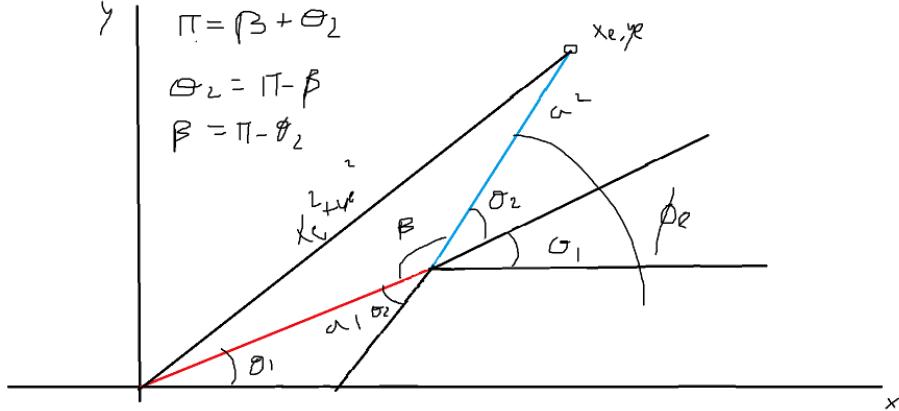


Figura 9: configuración del brazo robótico donde se puede aplicar el teorema del coseno

Así, se pueden extraer θ_2 y θ_3 :

$$\pi = \beta + \theta_3 \implies \beta = \pi - \theta_3 \quad (9)$$

$$\phi_{23} = \theta_2 - \theta_3 \quad (10)$$

En este caso, ϕ_e siempre va a ser $\frac{\pi}{2}$, por lo que no podemos obtener directamente el valor de θ_2 , ya que no se puede obtener directamente ϕ_{23} . Por este motivo, como se comentó anteriormente, el valor de $\phi_e = \phi_{23}$ (véase las ecuaciones 4 y 5).

Usando la ecuación 3, podemos despejar:

$$\begin{aligned} X_e^2 + Z_e^2 &= a_2^2 + a_3^2 - 2a_2a_3 \cos(\beta) = \\ &= a_2^2 + a_3^2 - 2a_2a_3 \cos(\pi - \theta_3) = \\ &= a_2^2 + a_3^2 + 2a_2a_3 \cos(\theta_3) \end{aligned}$$

$$\cos(\theta_3) = \frac{X_e^2 + Z_e^2 - a_2^2 - a_3^2}{2a_2a_3} \quad (11)$$

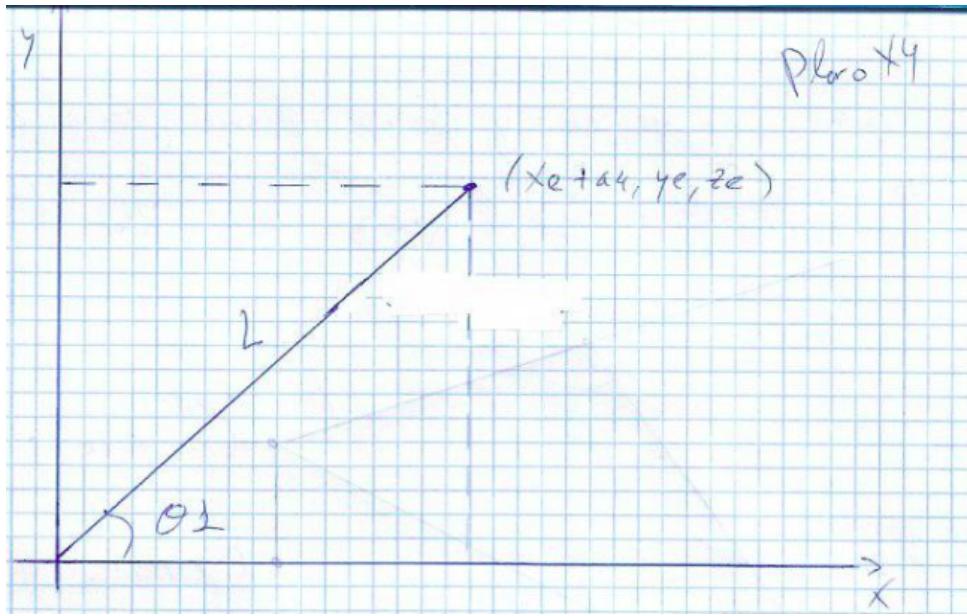
$$\sin(\theta_3) = \sqrt{1 - \cos^2(\theta_3)} \quad (12)$$

$$\implies \theta_3 = \arctan\left(\frac{\sin(\theta_3)}{\cos(\theta_3)}\right) \quad (13)$$

$$\implies \theta_2 = \phi_{23} + \theta_3 \quad (14)$$

Finalmente, en el plano XY tenemos la siguiente configuración:

³Teorema del coseno: $c^2 = a^2 + b^2 - 2ab \cos \beta$

Figura 10: visión del μ Arm en el plano XY

De este modo, aplicando el teorema de Pitágoras⁴, se obtiene:

$$\begin{aligned} L &= \pm \sqrt{(X_e + T_X)^2 + Y_e^2} \\ \cos(\theta_1) &= \frac{X_e + T_X}{\sqrt{(X_e + T_X)^2 + Y_e^2 + d_1}} \\ \sin(\theta_1) &= \frac{Y_e}{\sqrt{(X_e + T_X)^2 + Y_e^2 + d_1}} \\ \Rightarrow \tan(\theta_1) &= \frac{\sin(\theta_1)}{\cos(\theta_1)} \end{aligned}$$

$$\theta_1 = \arctan \left(\frac{\frac{Y_e}{\sqrt{(X_e + T_X)^2 + Y_e^2 + d_1}}}{\frac{X_e + T_X}{\sqrt{(X_e + T_X)^2 + Y_e^2 + d_1}}} \right) \quad (15)$$

$$= \arctan \left(\frac{Y_e}{X_e + T_X + d_1} \right) \quad (16)$$

De esta forma, un algoritmo de obtención de la inversa sería:

- Obtener θ_1 a partir del punto $(X_e + T_X, Y_e, Z_e - T_Z)$, usando la ecuación 16.
- Obtener $(X_e Y_e Z_e)$ quitando las traslaciones T_X y T_Z : $X_e = X_e - T_X$; $Z_e = Z_e + T_Z$.
- Obtener $\cos(\theta_3)$ con la ecuación 11 y el $\sin(\theta_3)$ con la ecuación 12.
- Calcular θ_3 usando la ecuación 13.
- Calcular θ_2 usando la ecuación 14.

⁴Teorema de Pitágoras: $h^2 = c_1^2 + c_2^2$

4. Matriz Jacobiana

La matriz Jacobiana nos permite obtener el movimiento \vec{x} , correspondiente a las velocidades en el extremo del robot dadas unas velocidades de las articulaciones \vec{q} [5].

La matriz Jacobiana es una matriz de derivadas, esto es, varía según el tiempo. De esta forma, se puede definir la matriz Jacobiana de un manipulador como:

$$\vec{x} = J \cdot \vec{q} \begin{cases} \vec{q} = (\theta_1, \dots, \theta_n, d_1, \dots, d_n) \\ \vec{x} = (X_e, Y_e, Z_e, \phi_e) \\ J \equiv \text{matriz Jacobiana} \end{cases} \quad (17)$$

$$J_1(\vec{q}) = \begin{pmatrix} \frac{\partial X_e}{\partial q_0} & \frac{\partial X_e}{\partial q_1} & \dots & \frac{\partial X_e}{\partial q_n} & \frac{\partial X_e}{\partial d_1} & \dots & \frac{\partial X_e}{\partial d_n} \\ \frac{\partial Y_e}{\partial q_0} & \frac{\partial Y_e}{\partial q_1} & \dots & \frac{\partial Y_e}{\partial q_n} & \frac{\partial Y_e}{\partial d_1} & \dots & \frac{\partial Y_e}{\partial d_n} \\ \frac{\partial Z_e}{\partial q_0} & \frac{\partial Z_e}{\partial q_1} & \dots & \frac{\partial Z_e}{\partial q_n} & \frac{\partial Z_e}{\partial d_1} & \dots & \frac{\partial Z_e}{\partial d_n} \end{pmatrix} \quad (18)$$

$$J_2(\vec{q}) = \begin{pmatrix} \frac{\partial \phi_X}{\partial q_0} & \frac{\partial \phi_X}{\partial q_1} & \dots & \frac{\partial \phi_X}{\partial q_n} & \frac{\partial \phi_X}{\partial d_1} & \dots & \frac{\partial \phi_X}{\partial d_n} \\ \frac{\partial \phi_Y}{\partial q_0} & \frac{\partial \phi_Y}{\partial q_1} & \dots & \frac{\partial \phi_Y}{\partial q_n} & \frac{\partial \phi_Y}{\partial d_1} & \dots & \frac{\partial \phi_Y}{\partial d_n} \\ \frac{\partial \phi_Z}{\partial q_0} & \frac{\partial \phi_Z}{\partial q_1} & \dots & \frac{\partial \phi_Z}{\partial q_n} & \frac{\partial \phi_Z}{\partial d_1} & \dots & \frac{\partial \phi_Z}{\partial d_n} \end{pmatrix} \quad (19)$$

$$J(\vec{q}) = \begin{pmatrix} J_1(\vec{q}) \\ J_2(\vec{q}) \end{pmatrix} \quad (20)$$

En nuestro caso particular, obtenemos la siguiente matriz Jacobiana:

$$J_1(\vec{q}) = \begin{pmatrix} -(a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3) + d_1) \sin(\theta_1) & (-a_2 \sin(\theta_2) - a_3 \sin(\theta_2 - \theta_3)) \cos(\theta_1) & a_3 \sin(\theta_2 - \theta_3) \cos(\theta_1) \\ (a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3) + d_1) \cos(\theta_1) & (-a_2 \sin(\theta_2) - a_3 \sin(\theta_2 - \theta_3)) \sin(\theta_1) & a_3 \sin(\theta_1) \sin(\theta_2 - \theta_3) \\ 0 & a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3) & -a_3 \cos(\theta_2 - \theta_3) \end{pmatrix} \quad (21)$$

$$J_2(\vec{q}) = \begin{pmatrix} 0 & 1 & -1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad (22)$$

$$J(\vec{q}) = \begin{pmatrix} -(a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3) + d_1) \sin(\theta_1) & (-a_2 \sin(\theta_2) - a_3 \sin(\theta_2 - \theta_3)) \cos(\theta_1) & a_3 \sin(\theta_2 - \theta_3) \cos(\theta_1) \\ (a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3) + d_1) \cos(\theta_1) & (-a_2 \sin(\theta_2) - a_3 \sin(\theta_2 - \theta_3)) \sin(\theta_1) & a_3 \sin(\theta_1) \sin(\theta_2 - \theta_3) \\ 0 & a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3) & -a_3 \cos(\theta_2 - \theta_3) \\ 0 & 1 & -1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad (23)$$

Entre otras utilidades de la matriz Jacobiana, podemos obtener diferentes ecuaciones que van relacionando el movimiento del brazo con el trabajo que hace, la fuerza que es necesaria y la potencia:

$$W = \int F^T \cdot v \, dt \quad (24)$$

$$P = \frac{W}{t} = \dots = F^T \cdot v \quad (25)$$

Debido a la equivalencia energética, el trabajo se realiza a la misma velocidad, independientemente del sistema [5], por lo que se puede escribir la potencia en términos del espacio del *end-effector*:

$$P = F_x^T \cdot \vec{x} \quad (26)$$

$$P = F_q^T \cdot \vec{q} \quad (27)$$

donde F_x es la fuerza aplicada al brazo robótico y \vec{x} la velocidad del mismo; F_q es el torque de las articulaciones y \vec{q} es la velocidad angular de las mismas.

De esta información se deducen las matrices que conforman la Jacobiana. La ecuación 18 describe la velocidad lineal del *end-effector*, y puede ser escrita como $J_w(\vec{q})$; mientras, la ecuación 19 describe la velocidad angular del *end-effector*, que puede ser escrita como $J_\omega(\vec{q})$.

Con la matriz Jacobiana ya obtenida se pueden obtener los puntos críticos. Dichos puntos permiten conocer posiciones inalcanzables por el brazo robótico, ya que el determinante en dicho punto es 0 y, por tanto, no existe la inversa.

Para este brazo robótico, el determinante es:

$$|J(\vec{q})| = -a_2 a_3 (a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3) + d_1) \sin(\theta_3) \quad (28)$$

Analíticamente, los puntos críticos del μArm son: $\theta_3 = 0$ y $\theta_3 = \pi$. Pero, al observar las tablas con los ángulos de giro de los motores (ver tabla 1), comprobamos que el brazo robótico no es capaz de llegar a los 180° y, por la configuración geométrica del mismo, tampoco puede tener 0° ⁵, lo cual nos conduce a que no hay puntos críticos en este brazo robótico:

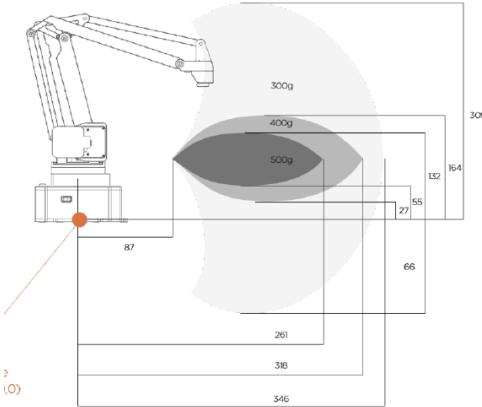


Figura 11: área de trabajo del μArm [1]

Igualmente, cuando se dan estos casos, se ha de hacer uso de o bien la matriz pseudo-inversa (J^+) de la matriz Jacobiana (si J^{-1} no existe) o bien de la matriz transpuesta (J^T):

⁵el motor tiene esos grados de libertad, pero la articulación no, y está limitada por la estructura

$$J^+ = J^T \cdot (J \cdot J^T)^{-1} \quad (29)$$

Una característica de la pseudo-inversa es que, en principio, siempre existe. Además, si la inversa de la Jacobiana J^{-1} existe, entonces la pseudo-inversa será igual a la matriz inversa:

$$J^+ = J^{-1} \Leftrightarrow \exists J^{-1} \quad (30)$$

Esto se explica con mayor detalle en el punto 4.1.

4.1. Matriz Jacobiana inversa

La matriz Jacobiana inversa permite saber qué velocidad hay que aplicar en las articulaciones \vec{q} para obtener una velocidad en el *end-effector* \vec{x} .

En el caso del μArm , la Jacobiana inversa resulta:

$$J^{-1} = \begin{pmatrix} -\frac{\sin(\theta_1)}{a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3) + d_1} & \frac{\cos(\theta_1)}{a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3) + d_1} & 0 \\ -\frac{\cos(\theta_1) \cos(\theta_2 - \theta_3)}{a_2 \sin(\theta_3)} & -\frac{\sin(\theta_1) \cos(\theta_2 - \theta_3)}{a_2 \sin(\theta_3)} & -\frac{\sin(\theta_2 - \theta_3)}{a_2 \sin(\theta_3)} \\ -\frac{(a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3)) \cos(\theta_1)}{a_2 a_3 \sin(\theta_3)} & -\frac{(a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3)) \sin(\theta_1)}{a_2 a_3 \sin(\theta_3)} & -\frac{a_2 \sin(\theta_2) + a_3 \sin(\theta_2 - \theta_3)}{a_2 a_3 \sin(\theta_3)} \end{pmatrix} \quad (31)$$

Como se observa en la ecuación 28, el determinante del brazo robótico se hace cero cuando el ángulo $\theta_3 = 0$ o bien cuando el mismo ángulo es π . Pero, debido a la configuración geométrica del robot (ver figura 11), dichas configuraciones son inalcanzables.

Igualmente, para tratar configuraciones singulares, se utiliza la pseudo-inversa de *Moore-Penrose*. Dicha matriz existe y es única por cada matriz A de entrada, y se define según las ecuaciones 29 y 30.

Para el caso particular del μArm , la pseudo-inversa es:

$$J^+ = \begin{pmatrix} -\frac{\sin(\theta_1)}{a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3) + d_1} & \frac{\cos(\theta_1)}{a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3) + d_1} & 0 \\ -\frac{\cos(\theta_1) \cos(\theta_2 - \theta_3)}{a_2 \sin(\theta_3)} & -\frac{\sin(\theta_1) \cos(\theta_2 - \theta_3)}{a_2 \sin(\theta_3)} & -\frac{\sin(\theta_2 - \theta_3)}{a_2 \sin(\theta_3)} \\ -\frac{(a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3)) \cos(\theta_1)}{a_2 a_3 \sin(\theta_3)} & -\frac{(a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3)) \sin(\theta_1)}{a_2 a_3 \sin(\theta_3)} & -\frac{a_2 \sin(\theta_2) + a_3 \sin(\theta_2 - \theta_3)}{a_2 a_3 \sin(\theta_3)} \end{pmatrix} \quad (32)$$

Como se puede apreciar, la pseudo-inversa es exactamente igual que la inversa. Esto es debido a que, como se muestra en la ecuación 30, la pseudo-inversa es igual a la inversa siempre y cuando la inversa exista:

$$J^+ = J^{-1} \Leftrightarrow \exists J^{-1}$$

Esto se cumple ya que es una de las propiedades de la inversa de *Moore-Penrose*:

- "if A is invertible, its pseudo-inverse is its inverse. That is, $A^+ = A^{-1}$ " [6].

5. Planificación y descripción de una trayectoria

Una trayectoria es el recorrido que debe describir un manipulador para cambiar su posición de un punto a otro. La trayectoria que describe un manipulador está formada por un conjunto puntos cartesianos en el espacio, es decir, puntos con tres coordenadas espaciales.

Para este caso, la trayectoria que va a describir el manipulador se define como:

$$T = \{P_1, P_2, \dots, P_n\} \setminus P_k \in \mathbb{R}^3, \forall k \in \{0, \dots, n\} \quad (33)$$

Un caso de interés a la hora de definir una trayectoria es en el que los puntos del recorrido están definidos por una función matemática $f(x)$ de dos dimensiones. Dado que esta trayectoria se define en un plano, una de las coordenadas se fija a cero. En el caso de usar $y = f_y(x)$, el valor de la coordenada z se fijaría a cero y por lo tanto la trayectoria estaría contenida en el plano XY . De igual forma, en el caso de usar $z = f_z(x)$, el valor de la coordenada y se fijaría a cero y por lo tanto la trayectoria estará contenida en el plano XZ .

En este caso y para simplificar la situación, la trayectoria se va describir en el plano XY , siendo también posible aplicar este razonamiento para los planos XZ o YZ . Para obtener los puntos de la trayectoria, se debe muestrear la función $y = f_y(x)$ para ciertos valores de x y, de esta forma, se obtendrían los puntos $P_i = (x, f_y(x), 0)$.

Dado que el μ Arm tiene una precisión mínima de movimiento de 0,2 mm. (en su *end-effector*), la función $f_y(x)$ se va a muestrear para valores de x siguiendo la expresión $x_i = x_{i-1} + 0,2 \cdot k$ con $k \in \mathbb{N}$. De esta manera, para cada valor de x se obtiene un valor de y y dado que z es siempre nulo, se obtienen todos los puntos de la trayectoria.

Una vez definida, el manipulador debe de moverse de un punto a otro siguiendo todos los puntos de la misma, para así poder describir la función matemática. Para que esto ocurra, se debe determinar las coordenadas articulares de los ejes de giro para todos y cada uno de los puntos y, de esta forma, se podrán controlar los motores adecuadamente.

Para este manipulador, se han obtenido las expresiones explícitas de la cinemática inversa (*IK*)⁶ y, por lo tanto, para cada punto del espacio y su orientación, se conoce exactamente las coordenadas articulares del mismo.

En el gráfico siguiente se representa cómo, dados dos puntos en el espacio, se puede calcular la variación de las coordenadas articulares necesaria para que el manipulador se desplace de un punto al otro:

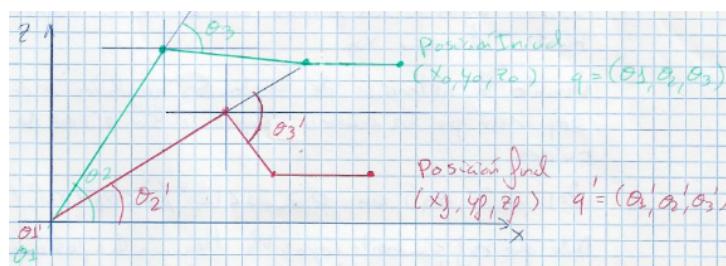


Figura 12: descripción de la trayectoria del brazo robótico

Dado el punto inicial (P_o) y punto final (P_f) de un desplazamiento, si se quiere calcular

⁶matrices de la cinemática inversa representadas en el punto 3, en particular las ecuaciones 16, 14 y 13

la variación que hay que aplicar sobre las articulaciones para realizar dicho desplazamiento, se realizan los siguientes cálculos:

- Se aplican las expresiones de cinemática inversa sobre el punto inicial $P_o = (x_o, y_o, z_o)$ y punto final $P_f = (x_f, y_f, z_f)$ del desplazamiento, de forma que se obtienen las coordenadas articulares correspondientes a cada punto $q_o = (\theta_{1o}, \theta_{2o}, \theta_{3o})$ y $q_f = (\theta_{1f}, \theta_{2f}, \theta_{3f})$. Para cada uno de los puntos, inicial y final se debe conocer también la orientación del punto final del manipulador.
- Una vez se han calculado las coordenadas articulares de cada uno de los puntos de la trayectoria, se deben tener en cuenta numerosos factores que afectan al desplazamiento del manipulador, como por ejemplo son la velocidad del *end-effector*, las posiciones singulares, la aceleración, etc. Todos estos factores definirán que tipo de trayectoria se emplea y el modo de obtenerla.
- Existen diversos métodos de planificación de trayectorias, sin embargo todos ellos se fundamentan en la cinemática inversa y directa, en el uso de la matriz Jacobiana y su inversa, y por último en el uso de métodos iterativos que aplican estas herramientas para cada punto de la trayectoria.

Durante la planificación de la trayectoria, se deben tener en cuenta las limitaciones físicas de diseño del manipulador, como pueden ser los rangos máximos de giro de los motores, la velocidad angular máxima de los mismos o el alcance máximo del manipulador.

El proceso descrito anteriormente también se puede aplicar a funciones de tres dimensiones, así como para cualquier trayectoria particular que se quiera describir.

6. Estudio de diferentes casos

6.1. Matriz de transformación directa

Usando la matriz que se ha obtenido en el punto 1, en particular la matriz 2, se puede comprobar su funcionamiento introduciendo distintas configuraciones en los ángulos que conforman las articulaciones. De los ángulos q_i se obtienen (X_e, Y_e, Z_e, ϕ_e) :

$$\begin{aligned}
 q &= (0, 0, 0) \\
 &\rightarrow (358,6, 0, 92,9, 0) \\
 q &= \left(0, -\frac{\pi}{2}, \frac{\pi}{2} + 10\right) \\
 &\rightarrow (57,7 - 158,9\cos(10), 0, 158,9\sin(10) - 49,1, -10 - \pi) \\
 q &= \left(-\frac{\pi}{2}, 0, 0\right) \\
 &\rightarrow (44,5, -314,1, 92,9, 0) \\
 q &= \left(\pi, \frac{\pi}{2}, \frac{\pi}{4}\right) \\
 &\rightarrow (31,3 - 79,45\sqrt{2}, 0, 79,45\sqrt{2} + 234,9, \frac{\pi}{4})
 \end{aligned}$$

6.2. Matriz de transformación inversa

Pese a contar con la matriz de transformación inversa (véase las ecuaciones 16, 14 y 13), en el estudio no pudimos comprobar los casos de prueba, ya que al no saber la configuración inicial del robot no se pudo saber qué angulos eran del brazo robótico (cuando no los puede realizar, el resultado es un valor imaginario - en la prueba del código fuente se puede comprobar).

6.3. Matriz Jacobiana

$$\begin{aligned}
 q &= (0, 0, 0) \\
 &\quad \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \\
 q &= \left(0, -\frac{\pi}{2}, \frac{\pi}{2} + 10\right) \\
 &\quad \begin{pmatrix} 158,9 \cdot \left(\frac{\pi}{2} + 10\right) \cdot \sin(10) - \pi \cdot \frac{142,0 - 158,9 \cdot \sin(10)}{2} \\ 0 \\ 158,9 \cdot \left(\frac{\pi}{2} + 10\right) \cdot \cos(10) + 79,45 \cdot \pi \cdot \cos(10) \\ -10 - \pi \\ 0 \\ 0 \end{pmatrix} \\
 q &= \left(-\frac{\pi}{2}, 0, 0\right) \\
 &\quad \begin{pmatrix} 72,85\pi \\ 71\pi \\ 158,9\pi \\ -\pi \\ 0 \\ -\frac{\pi}{2} \end{pmatrix} \\
 q &= (\pi, 0, 0) \\
 &\quad \begin{pmatrix} 0 \\ -314,1\pi \\ 0 \\ 0 \\ 0 \\ \pi \end{pmatrix}
 \end{aligned}$$

6.4. Matriz Jacobiana inversa

Al igual que sucedía en el caso anterior de la inversa, al no conocer los ángulos del robot salen valores anómalos que no existen y no se obtienen los resultados apropiados.

Referencias

- [1] *uArm Swift Pro _ Developer Guide v1.0.6.pdf*, en, 2019. dirección: http://download.ufactory.cc/docs/en/uArm%20Swift%20Pro_Developer%20Guide%20v1.0.6.pdf (visitado 02-11-2019).
- [2] *uArm-Swift-Specifications-171012.pdf*, en, 2019. dirección: <http://download.ufactory.cc/docs/en/uArm-Swift-Specifications-171012.pdf> (visitado 02-11-2019).
- [3] *uArm pro User Manual v1.1.0.pdf*, en, 2019. dirección: <http://download.ufactory.cc/docs/en/uArm%20pro%20User%20Manual%20v1.1.0.pdf> (visitado 02-11-2019).
- [4] *UPM-Robotics/uarm*, original-date: 2019-11-01T11:13:54Z, nov. de 2019. dirección: <https://github.com/UPM-Robotics/uarm> (visitado 02-11-2019).
- [5] travisdewolf, *Robot control part 2: Jacobians, velocity, and force*, en, sep. de 2013. dirección: <https://studywolf.wordpress.com/2013/09/02/robot-control-jacobians-velocity-and-force/> (visitado 17-11-2019).
- [6] *Moore–Penrose inverse*, en, Page Version ID: 926414640, nov. de 2019. dirección: https://en.wikipedia.org/w/index.php?title=Moore%20Penrose_inverse&oldid=926414640 (visitado 17-11-2019).

A. Código Python

Listing 1: manipulator.py

```

1 # manipulator
2 # Copyright (C) 2019 - Javinator9889
3 #
4 # This program is free software: you can redistribute it and/or modify
5 # it under the terms of the GNU General Public License as published by
6 # the Free Software Foundation, either version 3 of the License, or
7 # (at your option) any later version.
8 #
9 # This program is distributed in the hope that it will be useful,
10 # but WITHOUT ANY WARRANTY; without even the implied warranty of
11 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 # GNU General Public License for more details.
13 #
14 # You should have received a copy of the GNU General Public License
15 # along with this program. If not, see <http://www.gnu.org/licenses/>.
16 from typing import Union
17 from typing import Tuple
18 from typing import Dict
19 from typing import Any
20
21 from sympy import Matrix
22 from sympy import symbols
23 from sympy import simplify
24
25 from numbers import Number
26
27 from . import sin
28 from . import cos
29 from . import sqrt
30 from . import atan2
31 from . import Symbol
32 from . import DHTable
33 from . import to_latrix
34
35
36 class ForwardKinematics:
37     """
38         Container for the Forward Kinematics (FK) for an arbitrary
39             manipulator.
40             By using the Denavit-Hartenberg table, generates the required
41                 matrices
42                 in order to use them later.
43                 Refer to:
44                     https://en.wikipedia.org/wiki/Denavit%E2%80%93Hartenberg\_parameters
45                     for more information.
46                     The accessible params are:
47                         - params: DHTable.

```

```

45     - transformation_matrices: dict with the forward
        transformation matrices.
46     - phi_e: expression for phi_e.
47     Matrices are accessible by using square brackets: fk["A03"].
48     """
49
50     def __init__(self, params: DHTable, optimize: bool = True):
51         """
52             Generates a new instance for the class. It calculates the
53             forward
54             transformation matrices (symbolically) in order to use
55             them later
56             and not calculating them every time they are needed.
57             :param params: the Denavit-Hartenberg params.
58             :param optimize: whether to optimize or not the matrices
59                 - requires more
60                 computation time - default: True
61             """
62
63         self.params = params
64         self.transformation_matrices: Dict[str, Matrix] = {}
65         self._calc_matrices(optimize)
66         self.phi_e = None
67
68     def _calc_matrices(self, optimize: bool):
69         """
70             Internal function which iteratively calculates the
71             required transformation
72             matrices.
73             :param optimize: whether to optimize or not the matrices.
74             """
75
76         for i, theta, d, a, alpha in self.params:
77             self.transformation_matrices[f"A{i+1}{i}"] = \
78                 self._matrix(theta, d, a, alpha)
79         for i in range(2, self.params.max + 1):
80             self.transformation_matrices[f"A0{i}"] = \
81                 self.transformation_matrices[f"A0{i-1}"] * \
82                 self.transformation_matrices[f"A{i-1}{i}"]
83             if optimize:
84                 self.transformation_matrices[f"A0{i}"].simplify()
85             self.transformation_matrices[f"A0{self.params.max}"][0,
86                 3] += self.params.Tx
87             self.transformation_matrices[f"A0{self.params.max}"][1,
88                 3] += self.params.Ty
89             self.transformation_matrices[f"A0{self.params.max}"][2,
90                 3] += self.params.Tz
91
92     def set_phi(self, expression: Union[Symbol, Number]):
93         """
94             Sets the phi_e expression.
95             :param expression: expression - can be a Symbol or a
96                 number.

```

```

87         self.phi_e = expression
88
89     def point(self,
90               subs: Dict[Symbol, Any],
91               matrix_index: str = None) -> Tuple[Number, Number,
92                                           Number, Any]:
93         """
94         Obtain the (X, Y, Z, Phi) coordinates by changing the
95         articulations.
96         :param subs: the articulations' values.
97         :param matrix_index: the transformation matrix in which
98             apply the values.
99         By default, it is the forward transformation matrix.
100        :return: (X, Y, Z, Phi) as a tuple.
101        """
102        if matrix_index is None:
103            matrix_index = f"A{self.params.max}"
104        return
105            self.transformation_matrices[matrix_index].subs(subs)[0,
106            3], \
107            self.transformation_matrices[matrix_index].subs(subs)[1,
108            3], \
109            self.transformation_matrices[matrix_index].subs(subs)[2,
110            3], \
111            self.phi_e.subs(subs) if self.phi_e is not None
112            else None
113
114    def __getitem__(self, item):
115        return self.transformation_matrices.get(item)
116
117    @staticmethod
118    def _matrix(theta: Union[Symbol, float],
119                d: Union[Symbol, float],
120                a: Union[Symbol, float],
121                alpha: Union[Symbol, float]) -> Matrix:
122        """
123        Forward transformation matrix template.
124        :param theta: "theta" param.
125        :param d: 'd' param.
126        :param a: 'a' param.
127        :param alpha: "alpha" param.
128        :return: the forward transformation matrix.
129        """
130
131        return Matrix(
132            [[cos(theta), -cos(alpha) * sin(theta), sin(alpha) *
133              sin(theta),
134              a * cos(theta)],
135            [sin(theta), cos(alpha) * cos(theta), -sin(alpha) *
136              cos(theta),
137              a * sin(theta)],
138            [0, 0, 1]])

```

```

128         [0, sin(alpha), cos(alpha), d],
129         [0, 0, 0, 1]])
130
131
132 class InverseKinematics:
133     """
134     Container for the Inverse Kinematics (IK) for an arbitrary
135     manipulator.
136     By using the Forward Kinematics for that manipulator,
137     generates and
138     calculates the Jacobian matrix that can be used for both
139     direct
140     manipulation and inverse manipulation, relating linear speed
141     (end-effector)
142     and angular speed (joints).
143
144     The accessible params are:
145     - params: the DHTable params.
146     - Xe: expression for X.
147     - Ye: expression for Y.
148     - Ze: expression for Z.
149     - det: the determinant of the Jacobian.
150     - upper_jacobian: the first part of the Jacobian matrix
151     (linear velocity).
152     - lower_jacobian: the lower part of the Jacobian matrix
153     (angular velocity).
154     - m_jacobian: Jacobian matrix.
155     - i_jacobian: inverse Jacobian.
156     - pinv_jacobian: pseudo-inverse Jacobian.
157
158     For accessing the inverse matrix, it is better to use the
159     "inverse" property,
160     as it will return the pseudo-inverse or the inverse, in case
161     the latest one
162     does not exists.
163     """
164
165     def __init__(self, forward_kinematics: ForwardKinematics,
166                  phi_e: dict = None):
167         """
168         Generates a new instance for the inverse kinematics class.
169         :param forward_kinematics: the forward kinematics for the
170             manipulator.
171         :param phi_e: the Phi_e dict which relates the 'x', 'y'
172             and 'z' expressions.
173         """
174         self._end_effector_matrix = forward_kinematics[
175             f"A0{forward_kinematics.params.max}"]
176         self._phi_e = phi_e if phi_e is not None else dict()
177         self.params = forward_kinematics.params
178         self.Xe = self._end_effector_matrix[0, 3]

```

```

168     self.Ye = self._end_effector_matrix[1, 3]
169     self.Ze = self._end_effector_matrix[2, 3]
170     self.det = None
171     self.upper_jacobian = None
172     self.lower_jacobian = None
173     self.m_jacobian = None
174     self.i_jacobian = None
175     self.pinv_jacobian = None
176
177     def set_phi(self, xyz: str, expression: Union[Symbol,
178                                                 Number]):
178         """
179             Sets the  $\Phi_e$  expression, which relates the angle to an
180             axis.
181             :param xyz: the axis in which update the expression -
182                 must be: {'x', 'y', 'z'}.
183             :param expression: the expression for the  $\Phi$  - can be a
184                 number or an expression.
185             :raises AttributeError when xyz not in 'x', 'y', 'z'.
186         """
187
188         if xyz.lower() not in ['x', 'y', 'z']:
189             raise AttributeError("xyz attribute must be ['x', "
190                                 "'y', 'z']")
191         self._phi_e[xyz.lower()] = expression
192
193     def jacobian(self, subs: list = None) -> Matrix:
194         """
195             Calculates the Jacobian matrix. If the determinant is
196             '0', then it
197             calculates the pseudo-inverse.
198             :param subs: list of symbols that will be used for
199                 calculating the
200                 difference for the Jacobian.
201             :return: the Jacobian matrix.
202         """
203
204         smatrix = Matrix([self.Xe,
205                           self.Ye,
206                           self.Ze,
207                           self._phi_e['x'],
208                           self._phi_e['y'],
209                           self._phi_e['z']])
210
211         if subs is None:
212             subs = self.params.symbols
213         self.m_jacobian = smatrix.jacobian(subs)
214         self.upper_jacobian = self.m_jacobian[:3, :]
215         self.lower_jacobian = self.m_jacobian[3:, :]
216         self.det = self.upper_jacobian.det().simplify()
217         if self.det != 0:
218             self.i_jacobian = simplify(self.upper_jacobian ** -1)
219         else:
220             self.pinv_jacobian = self.upper_jacobian.pinv()

```

```

212     return self.m_jacobian
213
214     @property
215     def inverse(self):
216         """
217         :return: the inverse Jacobian.
218         """
219         return self.pinv_jacobian if self.i_jacobian is None else
220             self.i_jacobian
221
222 class UArmInverseKinematics:
223     """
224     uArm Inverse Kinematics class wrapper for the uArm robotic
225     arm.
226     Accessible values are:
227     - X_e: X value.
228     - Y_e: Y value.
229     - Z_e: Z value.
230     - phi: phi value.
231     - theta_1: expression for theta_1.
232     - theta_2: expression for theta_2.
233     - theta_3: expression for theta_3.
234     """
235
236     def __init__(self, params: DHTable):
237         """
238         Generates a new instance for the uArmInverseKinematics
239         class.
240         :param params: the Denavit-Hartenberg params.
241         """
242         self.X_e, self.Y_e, self.Z_e, self.phi = symbols("X_e Y_e"
243                                                 "Z_e phi_e")
244         cos_t3 = (
245             (self.X_e ** 2) + (self.Z_e ** 2) -
246             (params[1]['a'] ** 2) - (params[2]['a'] ** 2)
247             /
248             2 * params[1]['a'] * params[2]['a']
249         )
250         sin_t3 = (
251             sqrt(1 - (cos_t3 ** 2))
252         )
253         self.theta_1 = atan2(self.Y_e, self.X_e + params.Tx +
254             params[0]['d'])
255         self.theta_3 = atan2(sin_t3, cos_t3)
256         self.theta_2 = self.phi + self.theta_3
257
258     def eval(self,
259             Xe: Union[Symbol, Number],
260             Ye: Union[Symbol, Number],
261             Ze: Union[Symbol, Number],

```

```

258     phi: Union[Symbol, Number]) -> Tuple[Union[Symbol,
259         Number],
260             Union[Symbol,
261                 Number],
262                 Union[Symbol,
263                     Number]]:
264
265     """
266     With a given point, returns the joints at which the
267     robotic arm achieves
268     that position.
269     :param Xe: X position.
270     :param Ye: Y position.
271     :param Ze: Z position.
272     :param phi: phi value.
273     :return: (theta_1, theta_2, theta_3) as a tuple.
274     """
275
276
277 class Manipulator:
278
279     """
280     Wrapper class for working with the manipulator.
281     Accessible params are:
282     - direct_kinematics: the direct kinematics for the DHTable.
283     - inverse_kinematics: the inverse kinematics for the DHTable.
284     - uarm_ik: the uArm inverse kinematics.
285     """
286
287     def __init__(self, params: DHTable, optimize: bool = True):
288         self.params = params
289         self.direct_kinematics = ForwardKinematics(params,
290             optimize)
291         self.inverse_kinematics =
292             InverseKinematics(self.direct_kinematics)
293         self.uarm_ik = UArmInverseKinematics(params)
294
295     def point(self, subs: Dict[Symbol, Any],
296             matrix_index: str = None) -> Tuple[Number, Number,
297                 Number, Any]:
298
299     """
300     Obtain the (X, Y, Z, Phi) coordinates by changing the
301     articulations.
302     :param subs: the articulations' values.
303     :param matrix_index: the transformation matrix in which
304         apply the values.
305     By default, it is the forward transformation matrix.

```

```

299         :return: (X, Y, Z, Phi) as a tuple.
300         """
301         return self.direct_kinematics.point(subs, matrix_index)
302
303     def set_phi(self, xyz: str, expression: Union[Symbol,
304                 Number]):
305         """
306         Sets the  $\Phi_e$  expression, which relates the angle to an
307         axis.
308         :param xyz: the axis in which update the expression -
309                     must be: {'x', 'y', 'z'}.
310         :param expression: the expression for the  $\Phi$  - can be a
311                     number or an expression.
312         :raises AttributeError when xyz not in 'x', 'y', 'z'.
313         """
314         self.inverse_kinematics.set_phi(xyz, expression)
315
316     def jacobian(self, subs: list = None) -> Matrix:
317         """
318         Calculates the Jacobian matrix. If the determinant is
319         '0', then it
320         calculates the pseudo-inverse.
321         :param subs: list of symbols that will be used for
322                     calculating the
323                     difference for the Jacobian.
324         :return: the Jacobian matrix.
325         """
326
327         return self.inverse_kinematics.jacobian(subs)
328
329     @property
330     def inverse(self):
331         """
332         :return: the inverse Jacobian.
333         """
334
335         return self.inverse_kinematics.inverse
336
337     def eval(self,
338             Xe: Union[Symbol, Number],
339             Ye: Union[Symbol, Number],
340             Ze: Union[Symbol, Number],
341             phi: Union[Symbol, Number]) -> Tuple[Union[Symbol,
342                                         Number],
343                                         Union[Symbol,
344                                         Number],
345                                         Union[Symbol,
346                                         Number]]:
347
348         """
349         With a given point, returns the joints at which the
350         robotic arm achieves
351         that position.
352         :param Xe: X position.

```

```

340     :param Ye: Y position.
341     :param Ze: Z position.
342     :param phi: phi value.
343     :return: (theta_1, theta_2, theta_3) as a tuple.
344     """
345     return self.uarm_ik.eval(Xe, Ye, Ze, phi)
346
347     def to_latrix(self, matrix_type: str, matrix_index: str) ->
348         str:
349             """
350                 With a given Matrix, obtain its representation as a LaTeX
351                 matrix.
352                 :param matrix_type: the type of the matrix. Possible
353                 values can be:
354                 [‘b’, ‘p’, ‘v’, ‘V’, ‘’]. View
355                 https://en.wikibooks.org/wiki/LaTeX/Mathematics#Matrices\_and\_arrays
356                 for more information.
357                 :param matrix: the Matrix from which obtain the
358                     representation.
359                 :return: the LaTeX string representation.
360                 """
361
362     return to_latrix(matrix_type,
363                      self.direct_kinematics[matrix_index])

```

Listing 2: dh_table.py

```

1 # manipulator
2 # Copyright (C) 2019 - Javinator9889
3 #
4 # This program is free software: you can redistribute it and/or modify
5 # it under the terms of the GNU General Public License as published by
6 # the Free Software Foundation, either version 3 of the License, or
7 # (at your option) any later version.
8 #
9 # This program is distributed in the hope that it will be useful,
10 # but WITHOUT ANY WARRANTY; without even the implied warranty of
11 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 # GNU General Public License for more details.
13 #
14 # You should have received a copy of the GNU General Public License
15 # along with this program. If not, see <http://www.gnu.org/licenses/>.
16 from typing import List
17 from typing import Union
18 from .symbols import Symbol
19
20
21 class DHTable:
22     """
23         Container class for the Denavit-Hartenberg table.
24         Creates a data structure containing the necessary data for
25             constructing

```

```

25     the required matrix.
26     """
27
28     def __init__(self, table: List[dict] = None, check: bool =
29         True):
30         """
31             Creates a new instance for the class. If any argument is
32             provided, then
33             it checks if the dict is OK.
34             :param table: a dictionary containing the DH table. It
35                 must have the following
36                 structure:
37                 {
38                     'a': length,
39                     'd': distance,
40                     'alpha': angle with i + 1,
41                     'theta': arm angle (symbol)
42                 }, ...
43
44             :param check: skip the checking of the structure of the
45                 dictionary.
46             """
47
48         if table is None:
49             table = list()
50             check = False
51
52         if check:
53             for value in table:
54                 assert len(value.keys()) == 4
55
56         self.__table = table
57         self.symbols = list()
58
59         """
60             List of symbols used in DH-Table
61             """
62
63         self.max = 0
64
65         """
66             How many items does the class have
67             """
68
69         self.Tx: float = 0.
70
71         """
72             Translation in 'X' axis
73             """
74
75         self.Ty: float = 0.
76
77         """
78             Translation in 'Y' axis
79             """
80
81         self.Tz: float = 0.
82
83         """
84             Translation in 'Z' axis
85             """
86
87         self.__lengths = [set() for _ in range(4)]

```

```

72
73     @staticmethod
74     def _check_errors(theta: Union[Symbol, float],
75                         d: Union[Symbol, float],
76                         a: Union[Symbol, float],
77                         alpha: Union[Symbol, float]) -> bool:
78         return (type(theta) is Symbol and (type(d) is Symbol or
79                 type(a) is Symbol or
80                         type(alpha) is Symbol)
81                 or type(d) is Symbol and (
82                     type(theta) is Symbol or type(a) is
83                         Symbol or
84                         type(alpha) is Symbol)
85                 or type(a) is Symbol and (type(d) is Symbol or
86                         type(theta) is
87                             Symbol or
88                         type(alpha) is Symbol)
89                 or type(alpha) is Symbol and (
90                     type(d) is Symbol or type(a) is Symbol or
91                         type(theta) is Symbol))
92
93
94     def add(self,
95             theta: Union[Symbol, float],
96             d: Union[Symbol, float],
97             a: Union[Symbol, float],
98             alpha: Union[Symbol, float],
99             checkAttrs: bool = True) -> 'DHTable':
100        """
101        Add new params to the Denavit-Hartenberg table, in order.
102        This method can safely
103        be called by using the "Builder" structure
104        (.add(...).add(...)...).
105        By default, all params can be both "Symbol" or "float"
106        but, if "checkAttrs" is
107        changed, only one can be a "Symbol". If not, it raises
108        "AttributeError".
109
110        :param theta: the parameter theta.
111        :param d: the distance (elevation) between axes.
112        :param a: the length of the segment.
113        :param alpha: the angle between Zi and Zi+1 (radians).
114        :param checkAttrs: whether to perform a check or not -
115                           default: True
116        :return: the class itself.
117        :raises AttributeError when there is two or more params
118            whose type is Symbol.
119        Disable "checkAttrs" for not throwing any exception.
120
121        """
122        if checkAttrs:
123            if self._check_errors(theta, d, a, alpha):
124                raise AttributeError("Only one param can be a")

```

```

                                Symbol")
114
115     self._table.append({
116         'a': a,
117         'd': d,
118         "alpha": alpha,
119         "theta": theta
120     })
121     self.max += 1
122     if type(theta) is Symbol:
123         self.symbols.append(theta)
124     if type(d) is Symbol:
125         self.symbols.append(d)
126     if type(a) is Symbol:
127         self.symbols.append(a)
128     if type(alpha) is Symbol:
129         self.symbols.append(alpha)
130     self._lengths[0].add(len(str(theta)))
131     self._lengths[1].add(len(str(d)))
132     self._lengths[2].add(len(str(a)))
133     self._lengths[3].add(len(str(alpha)))
134     return self
135
136 def change(self, i: int, **kwargs):
137     """
138     Changes an existing param in the Denavit-Hartenber
139     tables. 'i' (index) must
140     exist.
141
142     :param i: the table index (from 1 to n).
143     :param kwargs: the keys to modify - [theta, d, a, alpha]
144     :raises IndexError when the 'i' does not exist.
145     """
146     i -= 1
147     for key, value in kwargs.items():
148         if key not in self._table[i].keys():
149             raise KeyError(f"The key '{key}' is not a valid"
150                             " entry - it must be: "
151                             f"[theta, d, a, alpha]")
152             old_value = self._table[i][key]
153             self._table[i][key] = value
154             if type(value) is Symbol:
155                 if type(old_value) is Symbol:
156                     self.symbols.insert(self.symbols.index(old_value),
157                                         value)
158
159     def remove(self, i: int) -> dict:
160         """
161         Removes an item from the Denavit-Hartenberg table.
162         :param i: the item to remove (from 1 to n).
163         :return: the removed item.

```

```

161     :raises IndexError when the item does not exists.
162     """
163     i -= 1
164     item = self.__table.pop(i)
165     for key, value in item.items():
166         if type(item[key]) is Symbol:
167             self.symbols.remove(value)
168     return item
169
170 def get(self) -> List[dict]:
171     """
172     Obtains the table itself.
173     :return: the Denavit-Hartenberg table.
174     """
175     return self.__table
176
177 def __getitem__(self, item):
178     assert isinstance(item, int)
179     return self.__table[item]
180
181 def __iter__(self):
182     i = 0
183     for element in self.__table:
184         i += 1
185         yield i, element["theta"], element['d'],
186                     element['a'], element["alpha"]
187
188 def __str__(self):
189     row_format = "{}{}{}{}{}".format(":{>4}",
190                                         ":{>" + str(4 +
191                                         max(self.__lengths[0])) +
192                                         "}" ,
193                                         ":{>" + str(4 +
194                                         max(self.__lengths[1])) +
195                                         "}" ,
196                                         ":{>" + str(4 +
197                                         max(self.__lengths[2])) +
198                                         "}" ,
199                                         ":{>" + str(4 +
200                                         max(self.__lengths[3])) +
201                                         "}" )
202
203     result = row_format.format('i', "t", "d", "a", "alpha") +
204             "\n"
205     i = 1
206     for values in self.__table:
207         result += row_format.format(str(i),
208                                     str(values["theta"]),
209                                     str(values['d']),
210                                     str(values['a']),
211                                     str(values["alpha"])) +
212             "\n"

```

```
201     i += 1
202     return result
```

Listing 3: `__init__.py`

```
1 # manipulator
2 # Copyright (C) 2019 - Javinator9889
3 #
4 # This program is free software: you can redistribute it and/or modify
5 # it under the terms of the GNU General Public License as published by
6 # the Free Software Foundation, either version 3 of the License, or
7 # (at your option) any later version.
8 #
9 # This program is distributed in the hope that it will be useful,
10 # but WITHOUT ANY WARRANTY; without even the implied warranty of
11 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 # GNU General Public License for more details.
13 #
14 # You should have received a copy of the GNU General Public License
15 # along with this program. If not, see <http://www.gnu.org/licenses/>.
16 from .dh_table import DHTable
17
18 from .symbols import *
19
20 from .utils import to_latrix
21
22 from .manipulator import Manipulator
23 from .manipulator import UArmInverseKinematics
```

Listing 4: `__main__.py`

```
1 # src
2 # Copyright (C) 2019 - Javinator9889
3 #
4 # This program is free software: you can redistribute it and/or modify
5 # it under the terms of the GNU General Public License as published by
6 # the Free Software Foundation, either version 3 of the License, or
7 # (at your option) any later version.
8 #
9 # This program is distributed in the hope that it will be useful,
10 # but WITHOUT ANY WARRANTY; without even the implied warranty of
11 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 # GNU General Public License for more details.
13 #
14 # You should have received a copy of the GNU General Public License
15 # along with this program. If not, see <http://www.gnu.org/licenses/>.
16 from .test import main
17
18 if __name__ == '__main__':
19     main()
```

Listing 5: symbols.py

```

1 # manipulator
2 # Copyright (C) 2019 - Javinator9889
3 #
4 # This program is free software: you can redistribute it and/or modify
5 # it under the terms of the GNU General Public License as published by
6 # the Free Software Foundation, either version 3 of the License, or
7 # (at your option) any later version.
8 #
9 # This program is distributed in the hope that it will be useful,
10 # but WITHOUT ANY WARRANTY; without even the implied warranty of
11 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 # GNU General Public License for more details.
13 #
14 # You should have received a copy of the GNU General Public License
15 # along with this program. If not, see <http://www.gnu.org/licenses/>.
16 from sympy import Symbol
17 from sympy import sin
18 from sympy import cos
19 from sympy import pi
20 from sympy import atan2
21 from sympy import sqrt
22 from sympy.abc import alpha
23 from sympy.abc import theta
24 from sympy.abc import E

```

Listing 6: tesst.py

```

1 # src
2 # Copyright (C) 2019 - Javinator9889
3 #
4 # This program is free software: you can redistribute it and/or modify
5 # it under the terms of the GNU General Public License as published by
6 # the Free Software Foundation, either version 3 of the License, or
7 # (at your option) any later version.
8 #
9 # This program is distributed in the hope that it will be useful,
10 # but WITHOUT ANY WARRANTY; without even the implied warranty of
11 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 # GNU General Public License for more details.
13 #
14 # You should have received a copy of the GNU General Public License
15 # along with this program. If not, see <http://www.gnu.org/licenses/>.
16 from time import time
17
18 from . import DHTable
19 from . import pi
20 from . import Manipulator
21
22 from sympy import Matrix
23 from sympy import symbols

```

```

24 from sympy import simplify
25
26
27 def main():
28     table = DHTable()
29     t1, t2, t3 = symbols("theta_1 theta_2 theta_3")
30     table.add(theta=t1, d=106.1, a=13.2, alpha=(pi / 2)) \
31         .add(theta=t2, d=0, a=142, alpha=pi) \
32         .add(theta=t3, d=0, a=158.9, alpha=0)
33     table.Tx = 44.5
34     table.Tz = -13.2
35     print("Tabla de Denavit-Hartenberg")
36     print(table)
37     startt = time()
38     manipulator = Manipulator(params=table)
39     endt = time()
40     print("Tiempo de obtención de las matrices: {:.3f}s".format(endt - startt))
41     manipulator.direct_kinematics.set_phi(expression=t2 - t3)
42     print("Estudio de diferentes casos")
43     startt = time()
44     c1 = manipulator.point({t1: 0, t2: 0, t3: 0})
45     c2 = manipulator.point({t1: 0, t2: -pi / 2, t3: pi / 2 + 10})
46     c3 = manipulator.point({t1: -pi / 2, t2: 0, t3: 0})
47     c4 = manipulator.point({t1: pi, t2: pi / 2, t3: pi / 4})
48     endt = time()
49     print(f"q = (0, 0, 0)\n{c1}")
50     print(f"q = ({0}, {-pi/2}, {pi/2+10})\n{c2}")
51     print(f"q = (-pi/2, 0, 0)\n{c3}")
52     print(f"q = ({pi}, {pi/2}, {pi/4})\n{c4}")
53     print("Tiempo de cálculo: {:.3f}s".format(endt - startt))
54
55     print("Estudio de la inversa - si el resultado es un número imaginario, "
56           "entonces es un punto al cual el robot no puede llegar")
57     startt = time()
58     i1 = manipulator.eval(c1[0], c1[1], c1[2], c1[3])
59     i2 = manipulator.eval(c2[0], c2[1], c2[2], c2[3])
60     i3 = manipulator.eval(c3[0], c3[1], c3[2], c3[3])
61     i4 = manipulator.eval(c4[0], c4[1], c4[2], c4[3])
62     endt = time()
63     print(i1)
64     print(i2)
65     print(i3)
66     print(i4)
67     print("Tiempo de cálculo: {:.3f}s".format(endt - startt))
68
69     print("Matriz Jacobiana")
70     startt = time()
71     manipulator.set_phi('x', t2 - t3)
72     manipulator.set_phi('y', 0)

```

```

73     manipulator.set_phi('z', t1)
74     j = manipulator.jacobian(subs=[t1, t2, t3])
75     endt = time()
76     print(j)
77     print(f"Jacobiana inversa:\n{j.manipulator.inverse}")
78     print("Tiempo de cálculo: {:.3f}s".format(endt - startt))
79
80     print("Estudio de casos para la matriz Jacobiana")
81     startt = time()
82     j1 = j.subs({t1: 0, t2: 0, t3: 0}) * Matrix([0, 0, 0])
83     j2 = j.subs({t1: 0, t2: -pi / 2, t3: pi / 2 + 10}) *
84         Matrix([0, -pi / 2, pi / 2 + 10])
85     j3 = j.subs({t1: -pi / 2, t2: pi / 2, t3: pi / 2 + pi}) *
86         Matrix([-pi / 2, pi / 2,
87                     pi
88                     /
89                     2
90                     +
91                     pi])
92     j4 = j.subs({t1: pi, t2: 0, t3: 0}) * Matrix([pi, 0, 0])
93     endt = time()
94     print(f"q = (0, 0, 0)\n{j1}")
95     print(f"q = (0, -pi/2, pi/2+10)\n{j2}")
96     print(f"q = (-pi/2, 0, 0)\n{j3}")
97     print(f"q = (pi, 0, 0)\n{j4}")
98     print("Tiempo de cálculo: {:.3f}s".format(endt - startt))
99
100    print("Matriz Jacobiana inversa")
101    print(f"Determinante: {manipulator.inverse_kinematics.det}")
102
103    print(simplify(manipulator.inverse * j1[:3, :]))
104    print(simplify(manipulator.inverse * j2[:3, :]))
105    print(simplify(manipulator.inverse * j3[:3, :]))
106    print(simplify(manipulator.inverse * j4[:3, :]))

```

Listing 7: utils.py

```

1 # manipulator
2 # Copyright (C) 2019 - Javinator9889
3 #
4 # This program is free software: you can redistribute it and/or modify
5 # it under the terms of the GNU General Public License as published by
6 # the Free Software Foundation, either version 3 of the License, or
7 # (at your option) any later version.
8 #
9 # This program is distributed in the hope that it will be useful,
10 # but WITHOUT ANY WARRANTY; without even the implied warranty of
11 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 # GNU General Public License for more details.
13 #
14 # You should have received a copy of the GNU General Public License

```

```

15 # along with this program. If not, see <http://www.gnu.org/licenses/>.
16 from numpy import ndindex
17 from sympy import Matrix
18 from sympy import latex
19
20
21 def to_latex(matrix_type: str, matrix: Matrix) -> str:
22     """
23         With a given Matrix, obtain its representation as a LaTeX
24         matrix.
25         :param matrix_type: the type of the matrix. Possible values
26             can be:
27             [‘b’, ‘p’, ‘v’, ‘V’, ‘’]. View
28             https://en.wikibooks.org/wiki/LaTeX/Mathematics#Matrices\_and\_arrays
29             for more information.
30         :param matrix: the Matrix from which obtain the
31             representation.
32         :return: the LaTeX string representation.
33     """
34
35     if len(matrix.shape) > 2:
36         raise ValueError("LaTeX can display at most two dimensions")
37     if matrix_type not in ('b', 'p', 'v', 'V', ''):
38         raise ValueError("Matrix type must be: [b, p, v, V] or nothing()")
39     row_values = [r"\usepackage{amsmath}", r"\begin{" + matrix_type + "matrix}"]
40     row = ""
41     for x, y in ndindex(matrix.shape):
42         row += ' ' + latex(matrix[x, y])
43         if y < (matrix.shape[1] - 1):
44             row += ' \& '
45         else:
46             row += r"\\"
47             row_values += [row]
48             row = ""
49     row_values += [r"\end{" + matrix_type + "matrix}"]
50     return '\n'.join(row_values)

```

El código fuente al completo se puede encontrar en GitHub: <https://github.com/UPM-Robotics/uarm>.