



Ingeniería Electrónica Electivo

Fundamentos de Robótica

Informe del Proyecto Final

**Diseño y control de un robot manipulador de 6
grados de libertad**

Alumnos:

Alessandro Giuffra Lovera
Joseph Matías Cherre Córdova
Mauricio Rivera
Alfieri Podestá

2021-1

1. Introducción

El presente proyecto abarca el diseño, modelado, simulación y control de un robot manipulador de 6 grados de libertad. Este tipo de robot se encuentra diseñado originalmente para ser un robot industrial que automatice las tareas de delivery en supermercados del sector retail, ya sea para alcanzar algo del estante al cliente o para colocar los insumos en su sitio correspondiente del estante. Posee una articulación prismática que le permitirá desplazarse horizontalmente y cinco articulaciones de revolución para alcanzar un rango de movimiento amplio en el espacio de operación.

Este primer enfoque del robot ahorrará el tiempo efectivo que toman los operadores para despachar productos así como reducir los riesgos de accidentes por alcanzar productos en anaquellos de gran altura. También puede tener un posible uso en la reposición y almacenamiento de stock así como en la colaboración del despacho de productos con clientes; sin embargo, el enfoque para esta tarea debería cambiarse a un tipo de robot colaborativo y tomarse restricciones adicionales que garanticen la seguridad de los clientes.

Después de realizar un primer bosquejo a mano, se modeló el robot utilizando el software CAD Fusion 3D para luego exportar el diseño a ROS utilizando la descripción URDF. Una vez obtenido el diseño del robot, se realizó el modelamiento cinemático y se verificó utilizando la herramienta de visualización que otorga ROS: RViz. Además, se realizó el control por cinemática diferencial y el control dinámico del robot. En el anexo se encuentran los códigos y funciones utilizadas.



Imagen 1.1. Renderizados del robot manipulador.



Imagen 1.2. Idea del Robot en el almacén.

Simulación en Rviz

Visualización en Ross mediante la herramienta Rviz. Para este se colocó a todas las articulaciones 'q' iguales a 0 para observar su posición inicial

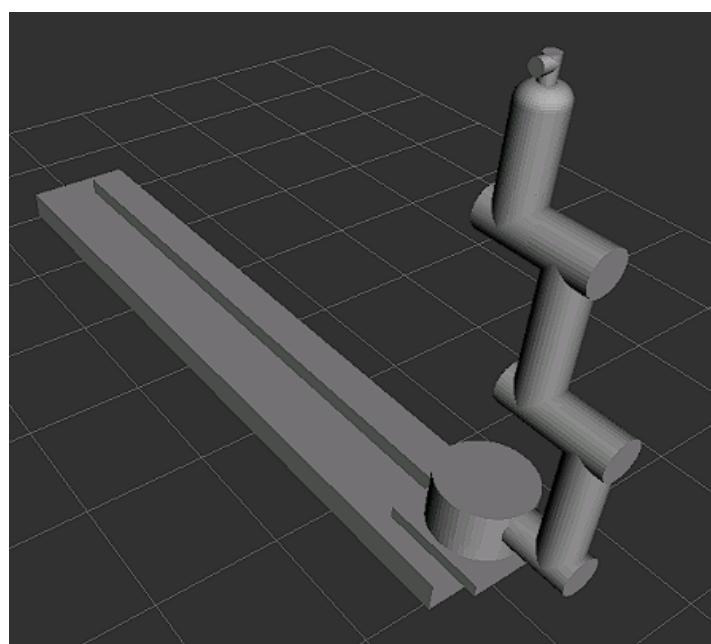


Imagen 1.3. Modelado del robot en RViz.

Simulación en Gazebo

Visualización en ROS mediante la herramienta Gazebo. Para este se colocó a todas las articulaciones 'q' iguales a 0 para observar su posición inicial

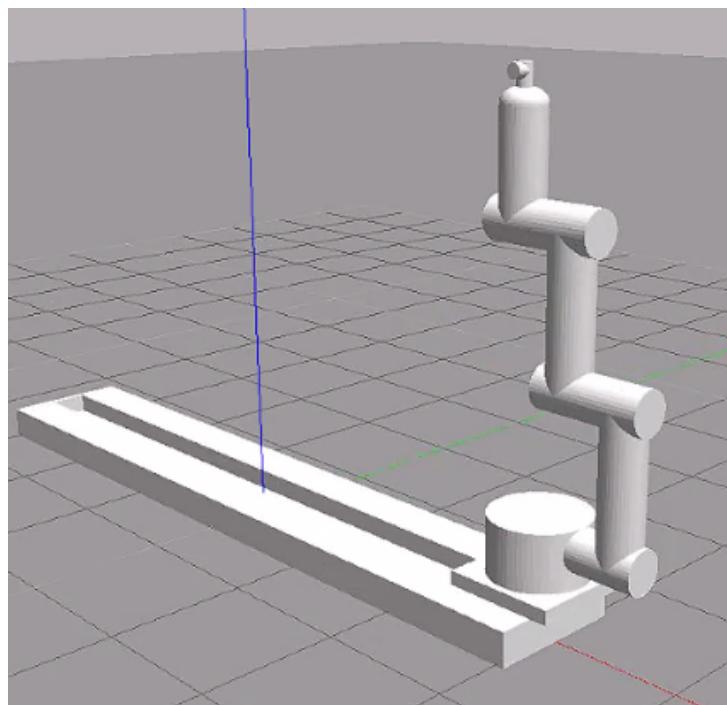


Imagen 1.4. Modelado del robot en Gazebo.

2. Componentes

Estructura mecánica:

El robot manipulador es de configuración P-R-R-R-R-R, teniendo un total de **6 grados de libertad**.

Articulación prismática en q1 y articulación rotacional en q2, q3, q4, q5 y q6

Elemento terminal:

Debido a las formas y pesos de los objetos con los que el robot manipulador interactúa es conveniente utilizar una **garra mecánica de 2 movidos por un actuador neumático lineal**, por ejemplo, podría ser de **cilindros de doble efecto**, ya que para realizar el transporte de estos elementos no se requiere de mucha precisión en el agarre de estos objetos. Entre estos tenemos botellas de plástico, latas metálicas, cajas de cartón, etc. Este necesitará un sistema compresor incluido en el robot.

Sistema de actuación:

Debido a que es un robot manipulador que no va a necesitar de mantener cargas pesadas por mucho tiempo, sería ideal el uso de **actuadores eléctricos** para las articulaciones del robot. Se estima el uso de **servomotores** debido a que estos vienen con **controlador y una caja reductora** (transmisión), de ser necesario, se puede modificar el **sistema de transmisión**. Para ello es recomendable utilizar una fuente de poder

proveniente de la toma de corriente, pasando **la corriente por transformadores y rectificadores** para su posterior uso en los **motores eléctricos** de las articulaciones.

Sistema de sensado:

Sensores propioceptivos:

- Encoders digitales de una vuelta (absolutos o relativos):
Ubicados en cada uno de los motores articulares para ver en qué posición se ubican los mismos, con un máximo 360°. Se utilizará para el control de bajo nivel.

Sensores exteroceptivos:

- Sensor de proximidad (ultrasónico):
Para la detección de la distancia de aquellos objetos que se agarraron. Este se ubicará en dirección en la que apunte el efecto final.
- Sensor de torque:
Utilizado para obtener los valores del torque ejercido sobre o por el efecto final, de manera que estos datos puedan ser utilizados en el control del robot.



Sistema de control:

Sistema de control de bajo nivel:

- ATmega328P AU

Se considera como opción correcta la elección de este microcontrolador pues lleva PWM para el accionamiento de las articulaciones incluidas en este proyecto. Su reducido tamaño es conveniente para ubicarlo y conectarlo con cada motor.

Sistema de control de alto nivel:

- Raspberry Pi Zero

Un SBC (Single Board Computer) con la suficiente capacidad para realizar el control de este robot. Esta es una opción económica, con la capacidad de procesamiento suficiente (1 GHz) y de tamaño preciso para evitar poca estética y que este no se note. Este se encargará de hacer el procesamiento general de todo el robot.

Ya que cada motor será controlado por un microcontrolador, estos se comunicarán con el SBC por medio del protocolo de comunicación I2C (o TWI por Two Wire Interface).

3. Modelo del robot

En la siguiente imagen se podrán observar las medidas del robot obtenidas a partir del uso de Fusion. Notar que las medidas que nos interesan son con respecto a la posición del origen de cada sistema.

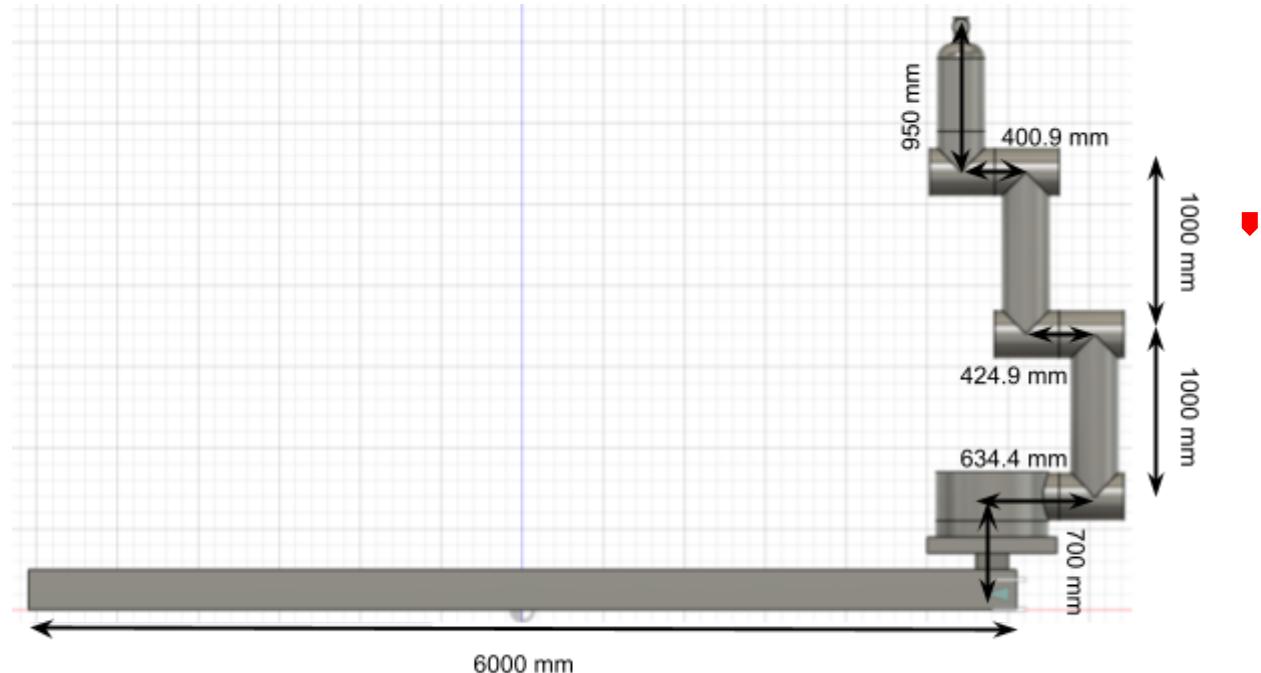


Imagen 3.1. Renderizado en Fusion (Dimensiones).

4. Cinemática directa e inversa

Análisis del robot:

Se obtuvieron las direcciones de los vectores z,x a partir de la convención de DH, además que se colocó el origen de coordenadas de cada sistema en el lugar correspondiente



Imagen 4.1. Sistemas del robot según convención DH.

X → Rojo

Z → Azul

Obtención de parámetros de DH del robot (en mm):

i	d i	theta i	a i	alpha i
1	2900-q1	180°	0	90°
2	700	180°+q2	0	90°
3	634.3965	-90°+q3	-1000	180°
4	424.906	0°+q4	-1000	0°
5	400.922	-90°+q5	0	90°
6	950	-90°+q6	0	0°

Tabla 4.1. Parámetros de DH

Código:

```
def sTraty(ang):
    Ty = np.array([[np.cos(ang), 0, np.sin(ang), 0],
                  [0, 1, 0, 0],
                  [-np.sin(ang), 0, np.cos(ang), 0],
                  [0, 0, 0, 1]])
    return Ty

def sTrotz(ang):
    Tz = np.array([[np.cos(ang), -np.sin(ang), 0, 0],
                  [np.sin(ang), np.cos(ang), 0, 0],
                  [0, 0, 1, 0],
                  [0, 0, 0, 1]])
    return Tz

def dh(d, theta, a, alpha):
    cth=cos(theta)
    sth=sin(theta)
    ca=cos(alpha)
    sa=sin(alpha)
    T = np.array([[cth, -ca*sth, sa*sth, a*cth],
                  [sth, ca*cth, -sa*cth, a*sth],
```

```
[0, sa, ca, d],
[0,0,0,1]])

return T

def fkine_ur5(q):

    # Matrices DH (completar), emplear la funcion dh con los parametros DH
    # para cada articulacion
    T1 = dh( 3-0.1-q[0], pi, 0, pi/2)
    T2 = dh( 0.700, q[1]+pi, 0, pi/2)
    T3 = dh( 0.6343965, q[2]-pi/2, -1, pi)
    T4 = dh( 0.424906, q[3], -1, 0)
    T5 = dh( 0.400922, q[4]-pi/2, 0, pi/2)
    T6 = dh( 0.950, q[5]-pi/2, 0, 0)

    # Efector final con respecto a la base
    T =
    sTrot(y(pi/2)).dot(sTrot(z(pi/2))).dot(T1).dot(T2).dot(T3).dot(T4).dot(T5).do
    t(T6)
    return T
```

Código 4.1. Código utilizado para lograr la cinemática directa por Denavit-Hartenberg.

Comentario:

Dado que nuestro sistema 0 empezaba con z en la dirección horizontal y el de ROS estaba configurado para que empiece con z hacia arriba, fue necesario realizar una rotación en y de 90°, seguidamente de una rotación en z de 90°, este último con el fin de que la dirección del x sea coincidente con el diseño. Una vez ambos sistemas (nuestro diseño y el mostrado en ROS) estén iguales, se procedió a realizar las multiplicaciones para obtener la matriz Homogénea de 0 al efector final

Cinemática directa

Para la cinemática directa, lo que se hizo fue brindar valores articulares (q) al robot para que llegue al marker el cual tenía la posición dada por la parte de desplazamiento (x, y, z) de la Matriz de Transformación

Ejemplo 1:

Configuración deseada para la prueba de la cinemática directa

```
q = [2.4, 0, 0.2, 0.3, 1.5, 0.45]
```

Matriz de Transformación obtenida a partir de los q:

$$\begin{bmatrix} [0.9 & -0.435 & -0. & 0.31] \\ [-0.013 & -0.026 & 1. & 0.851] \\ [-0.435 & -0.9 & -0.029 & 2.647] \\ [0. & 0. & 0. & 1.] \end{bmatrix}$$

Imagen 4.2. Matriz de Transformada Homogénea resultante de la cinemática directa.

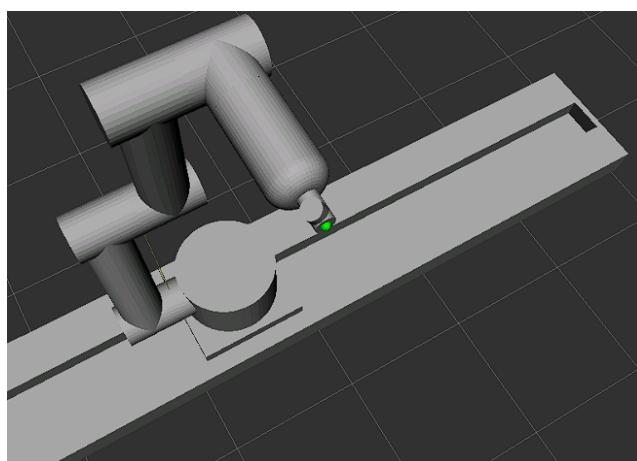


Imagen 4.3. Robot posicionado en la configuración deseada.

Ejemplo 2:

Segunda configuración deseada para la prueba de la cinemática directa

$$q = [3.2, 0, 0.1, 1.2, 1.3, 0.12]$$

Matriz de Transformación obtenida a partir de los q:

$$\begin{bmatrix} [0.993 & -0.12 & -0. & -0.49] \\ [-0.088 & -0.732 & 0.675 & 1.433] \\ [-0.081 & -0.671 & -0.737 & 1.448] \\ [0. & 0. & 0. & 1.] \end{bmatrix}$$

Imagen 4.3. Matriz de Transformada Homogénea resultante de la segunda cinemática directa.

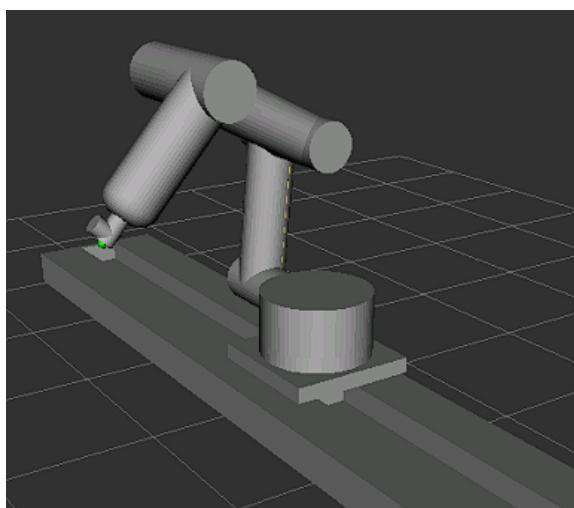


Imagen 4.4. Robot posicionado en la segunda configuración deseada.

Ejemplo 3:

Tercera configuración deseada para la prueba de la cinemática directa

$$q = [1.2, 0.2, 1.4, 0.8, 1.5, 1.1]$$

Matriz de Transformación obtenida a partir de los q:

$$\begin{bmatrix} [0.334 & -0.929 & -0.156 & 1.675] \\ [0.633 & 0.099 & 0.768 & -0.837] \\ [-0.698 & -0.355 & 0.622 & 2.286] \\ [0. & 0. & 0. & 1.] \end{bmatrix}$$

Imagen 4.5. Matriz de Transformada Homogénea resultante de la tercera cinemática directa.

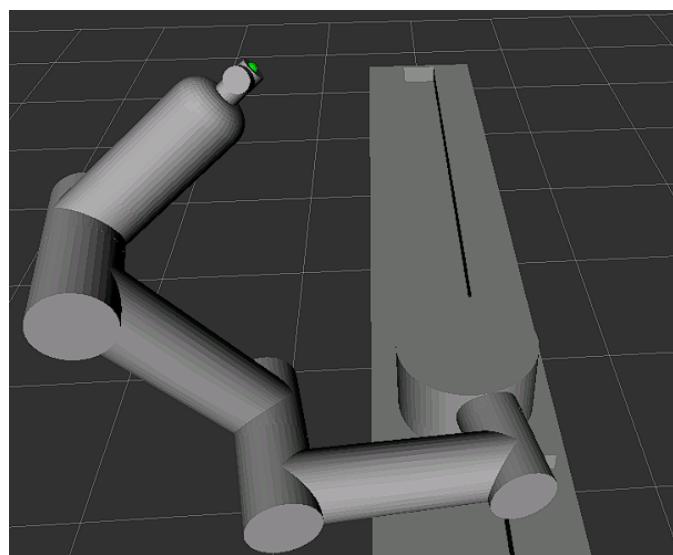


Imagen 4.6. Robot posicionado en la segunda configuración deseada.

Comentario:

Como se pudo observar en los 3 ejemplos mostrados, el efecto final del robot coincidió con la posición colocada para el marker. Esto significa que nuestra tabla de DH está bien construida. Para estos ejemplos se procuró dar valores aleatorios y diferentes de 0 a todas las articulaciones para verificar que todos los cálculos hayan sido correctos y así no tener problemas con los ejercicios mostrados más adelante

Cinemática inversa

Para la cinemática inversa, lo que se hizo fue brindar valores cartesianos(x,y,z) lo cual sería la posición deseada a la cual el robot debía llegar, esta se marcó con un marker verde. Mientras que el robot, realizando un cálculo mediante el Método de Newton, obtuvo el valor de 'q' necesario para llegar a la posición deseada y con ello se halló la posición cartesiana deseada. Cabe resaltar que la posición actual del efecto final del robot estuvo marcado con un marker rojo, por lo que cuando estos dos markers estén juntos significa que el proceso funcionó.

Ejemplo 1:

Coordenadas establecidas para la prueba de cinemática inversa

```
# Desired position
xd = np.array([1.5, 1.5, 1.5])
# Initial configuration
q0 = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
```

Matriz de Transformación obtenida a partir de los q:

```
('Obtained value:\n', array([[ 0.978, -0.021,   0.206,  1.5  ],
   [-0.207, -0.1   ,  0.973,  1.5  ],
   [-0.    , -0.995, -0.102,  1.499],
   [ 0.    ,  0.    ,  0.    ,  1.    ]]))
('q:\n', array([ 1.522, -0.209, -1.562,  4.242,  2.153,  0.    ]))
```

Imagen 4.7. Matriz de Transformada Homogénea resultante de la cinemática inversa.



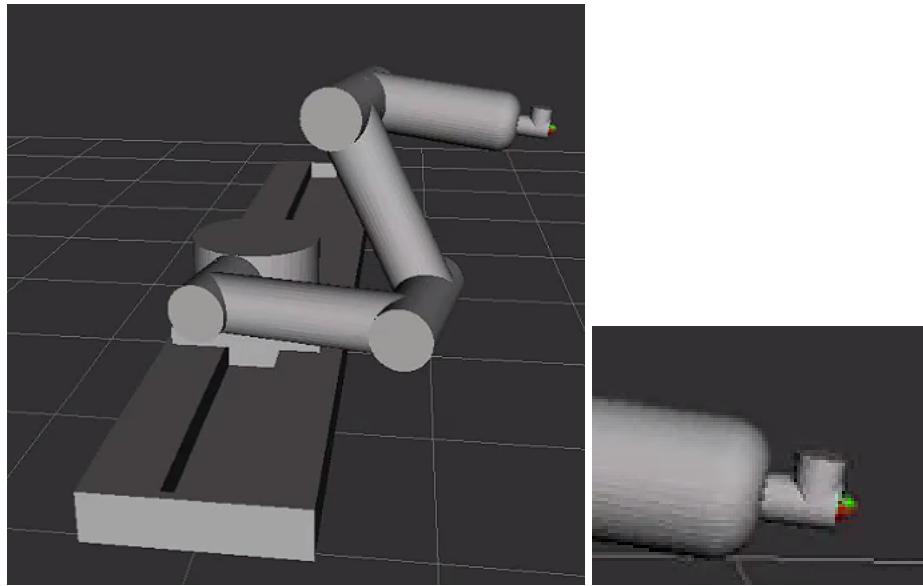


Imagen 4.8. Robot posicionado en las coordenadas deseadas.

Ejemplo 2:

Coordenadas establecidas para la segunda prueba de cinemática inversa.

```
# Desired position
xd = np.array([0.7, 1.2, 1.1])
# Initial configuration
q0 = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
```

Matriz de Transformación obtenida a partir de los q :

```
('Obtained value:\n', array([[ 0.946,  0.306, -0.111,  0.7  ],
   [ 0.325, -0.889,  0.322,  1.2  ],
   [-0.    , -0.34 , -0.94 ,  1.101],
   [ 0.    ,  0.    ,  0.    ,  1.    ]]))
('q:\n', array([ 1.584,  0.331, -0.072,  1.198,  1.525,  0.    ]))
```

Imagen 4.9. Matriz de Transformada Homogénea resultante de la segunda cinemática inversa.

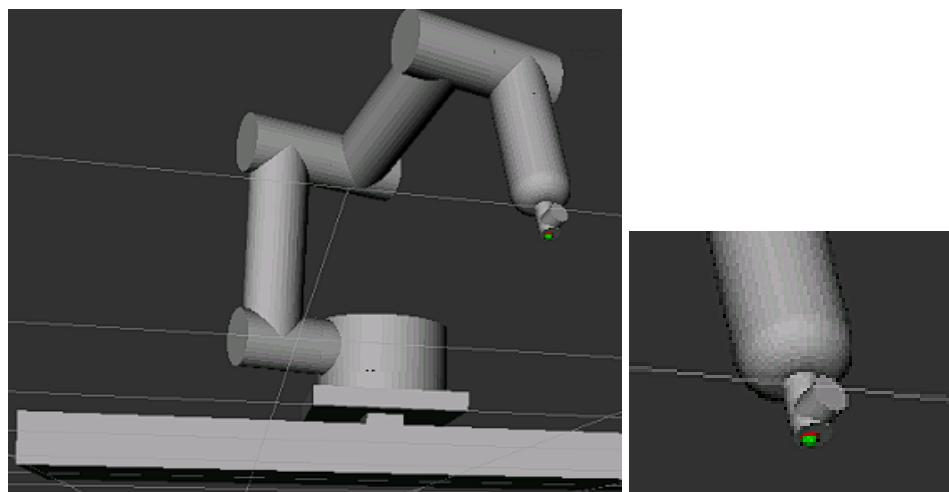


Imagen 4.10. Robot posicionado en la segunda coordenada deseada.

Ejemplo 3:

Coordenadas establecidas para la tercera prueba de cinemática inversa

```
# Desired position
xd = np.array([0.3, 1.7, 0.5])
# Initial configuration
q0 = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0]).
```

Matriz de Transformación obtenida a partir de los q:

```
('Obtained value:\n', array([[ 0.991, -0.104,  0.08 ,  0.294],
   [-0.131, -0.786,  0.605,  1.699],
   [-0.    , -0.61 , -0.792,  0.495],
   [ 0.    ,  0.    ,  0.    ,  1.    ]]))
('q:\n', array([ 2.637, -0.131, -2.017,  4.468,  2.283,  0.    ]))
```

Imagen 4.11. Matriz de Transformada Homogénea resultante de la tercera cinemática inversa.

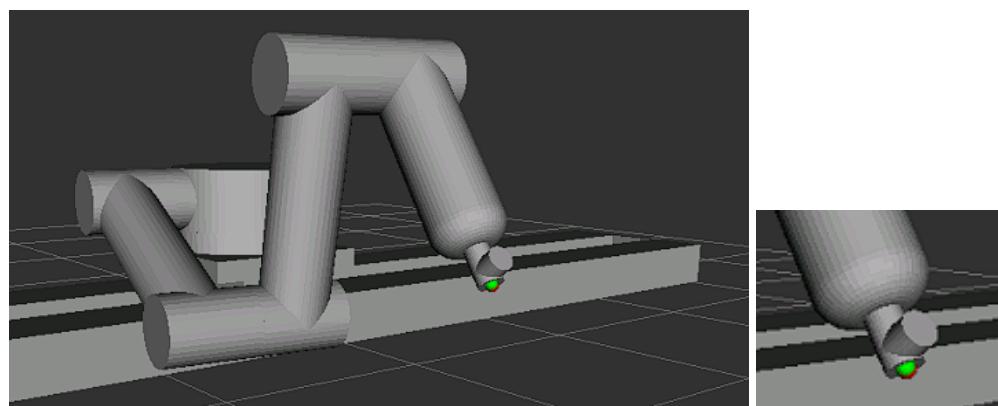


Imagen 4.12. Robot posicionado en la segunda coordenada deseada.

Comentario:

En este caso, utilizando las coordenadas como valor deseado a llegar por el robot, se puede apreciar que el efecto final (marcador rojo) llega a la posición deseada por el código (marcador verde). Las tres pruebas han sido finalizadas con un resultado positivo, como se pueden ver en las imágenes superiores. Al igual que el caso anterior, se procuró brindar valores deseados aleatorios para ver que funcione para todos los casos

5. Control cinemático

- Cálculo de los Jacobianos

Para el presente proyecto, se realizó el cálculo del Jacobiano Analítico dado que este solo se utilizará para calcular posición, no orientación. Como sabemos, las columnas de esta matriz están conformadas por las derivadas parciales de las coordenadas de cada articulación (x, y, z) a razón de cada q , como se puede ver en la siguiente figura.

$$J = \begin{bmatrix} \frac{\partial x}{\partial q_1} & \frac{\partial x}{\partial q_2} & \dots & \frac{\partial x}{\partial q_n} \\ \frac{\partial y}{\partial q_1} & \frac{\partial y}{\partial q_2} & \dots & \frac{\partial y}{\partial q_n} \\ \frac{\partial z}{\partial q_1} & \frac{\partial z}{\partial q_2} & \dots & \frac{\partial z}{\partial q_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial q_1} & \frac{\partial x}{\partial q_2} & \dots & \frac{\partial x}{\partial q_n} \end{bmatrix}$$

Sin necesidad de utilizar las derivadas directamente, estas operaciones se pueden realizar mediante el método de diferencias finitas

$$\frac{\partial x}{\partial q_1} \approx \frac{x(q + \delta q_1) - x(q)}{\delta q_1}$$

Para automatizar el proceso del cálculo de cada valor en el Jacobiano, se realizó la siguiente función en Python.

```
Obtención del Jacobiano (se recibe el q actual y la variación que se le hará)
def jacobian_ur5(q, delta=0.0001):
    """
    Jacobiano analítico para la posición. Retorna una matriz de 3x6 y toma como
    entrada el vector de configuración articular q=[q1, q2, q3, q4, q5, q6]
    """
    # Crear una matriz 3x6
    J = np.zeros((3,6))
    # Transformación homogénea (0-T-6) usando el q dado
    T = fkine_ur5(q)
    # Iteración para la derivada de cada columna
    for i in xrange(6):
        # Copiar la configuración articular(q) y almacenarla en dq
        dq = copy(q)
        # Incrementar los valores de cada q sumándoles un delta a cada uno
        dq[i] = dq[i] + delta
        # Obtención de la nueva Matriz Homogénea con los nuevos valores articulares,
        # luego del incremento (q+delta)
        Td = fkine_ur5(dq)
        # Aproximación del Jacobiano de posición usando diferencias finitas
        for j in xrange(3):
            J[j][i] = (Td[j][3]-T[j][3])/delta
    return J
```

Código 4.2 Cálculo del Jacobiano

- Ley de control

Una vez calculado el Jacobiano, se define la siguiente ley de control para el hallar el error, el cual se da por la diferencia entre la posición actual y la deseada:

$$\mathbf{e} = \mathbf{x} - \mathbf{x}_d.$$

Si la ecuación de arriba se derivase, tomando en cuenta que \mathbf{x}_d es constante, obtendremos: $d\mathbf{e} = d\mathbf{x}$ y sabiendo $d\mathbf{x} = J * dq$, obtenemos:

$$\dot{\mathbf{e}} = J \dot{\mathbf{q}}$$

Despejando dq se obtiene lo siguiente. Notar que $J^\#$ hace referencia a la pseudoinversa de Moore Penrose del Jacobiano (dados que no es una matriz cuadrada)

$$\dot{\mathbf{q}} = J^\# \dot{\mathbf{e}}$$

A continuación se define la ley de control cinemática para que el error tienda a 0, donde la ganancia cinemática, para nuestro caso k se definirá como 0.1:

$$\dot{\mathbf{e}}^* = -k\mathbf{e}$$

Finalmente reemplazando la cuarta ecuación en la tercera y, aplicando la integración por Euler, se obtiene la siguiente ecuación:

$$\mathbf{q}_k = \mathbf{q}_{k-1} + \Delta t \dot{\mathbf{q}}_k$$

Cabe resaltar que utilizamos este método de integración dado que el error por la aproximación es pequeño cuando se trata de periodos de control cortos como es el de nuestro caso. El código que implementa este procedimiento se muestra a continuación:

```
# Main loop
while not rospy.is_shutdown():

    k = 0.1
    # Current time (needed for ROS)
    jstate.header.stamp = rospy.Time.now()
    # Kinematic control law for position (complete here)
    #
    # -----
    #n=np.linalg.matrix_rank(J); m=J.ndim
    J = jacobian_ur5(q0, delta=0.01)
    e = x0 - xd
    e_der = -k*e

    n=np.linalg.matrix_rank(J); m=J.ndim          #np.linalg.det(J)==0
```

```
#####
if (n<m):
    q_der = np.dot(np.dot(np.linalg.inv(J.T*J), J.T), e_der) #pseudo
    inversa amortiguada
else:
    q_der = np.dot(np.linalg.pinv(J), e_der)

q = q + dt*q_der
# -----


T=fkine_ur5(q)
x=T[0:3,3]
x0=x
```

4.3 Cálculo de la ley de control de q

- Resultados obtenidos

Se verificó que el resultado obtenido sea el adecuado utilizando RViz para obtener las siguientes gráficas:

- Colocando una posición deseada de: $xd=[0.3, 1.7, 0.5]$

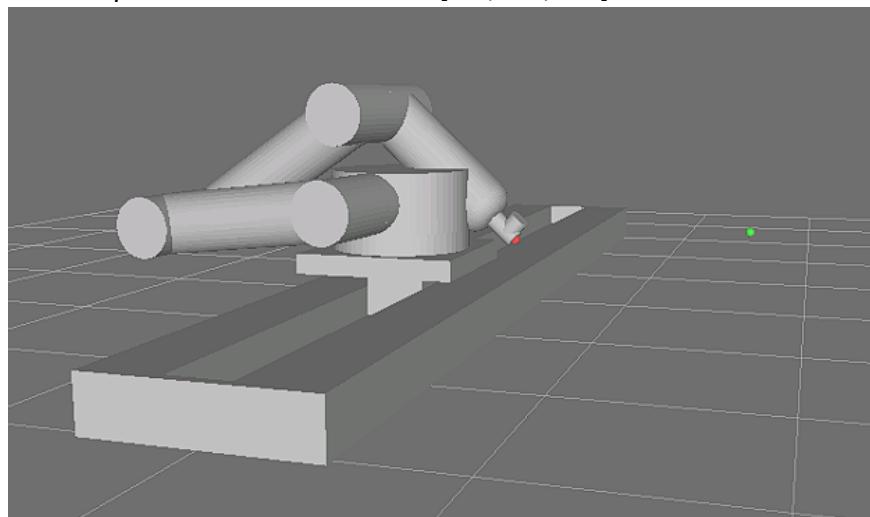


Imagen 5.1. Robot en el instante inicial

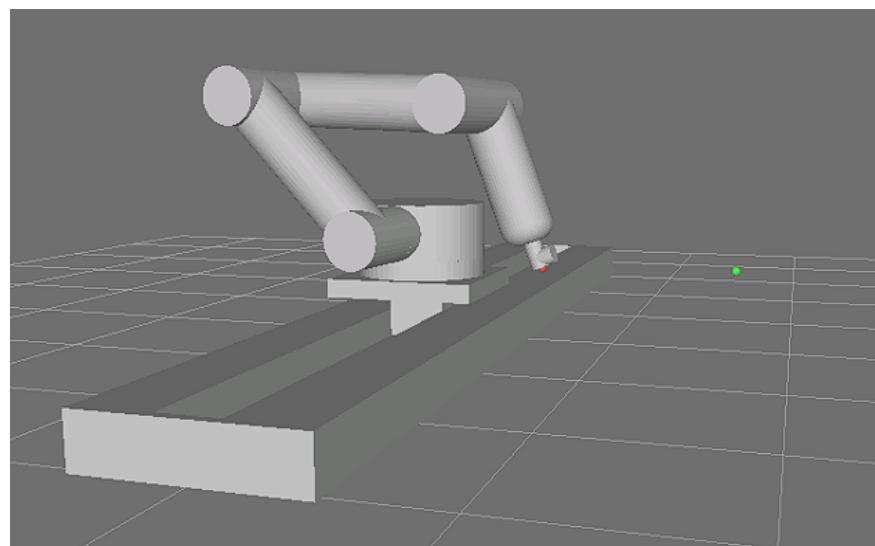


Imagen 5.2. Robot en el instante 2

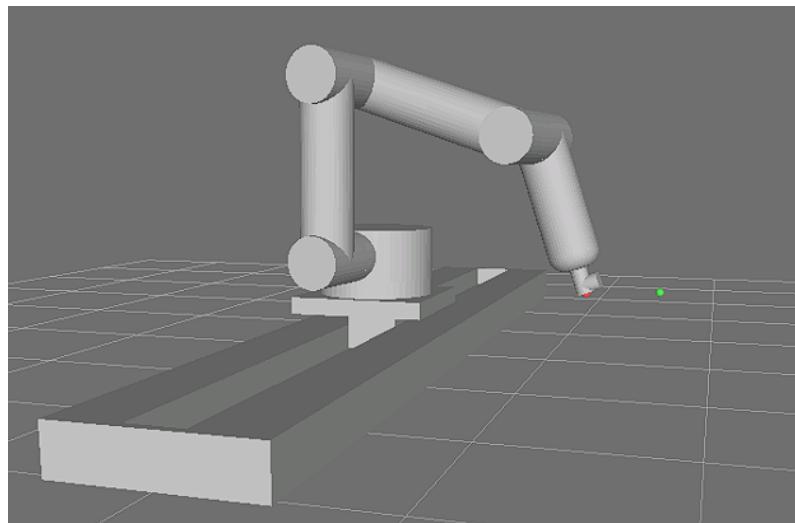


Imagen 5.3. Robot en el instante 3

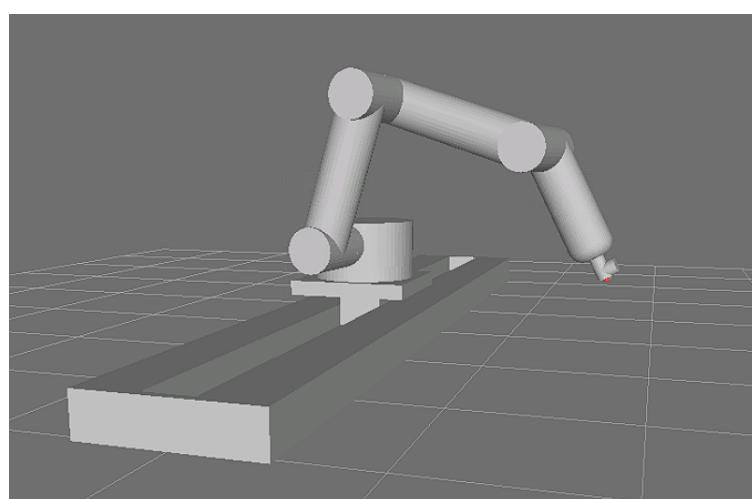


Imagen 5.4. Robot en el instante final

- Gráficas obtenidas:

Se puede apreciar en las imágenes, como para cada instante de tiempo el manipulador se va acercando a la posición deseada para llegar finalmente con éxito a la misma. A continuación, se muestran las gráficas de los valores de q y de posición (x , y , z) en todo el intervalo de tiempo:

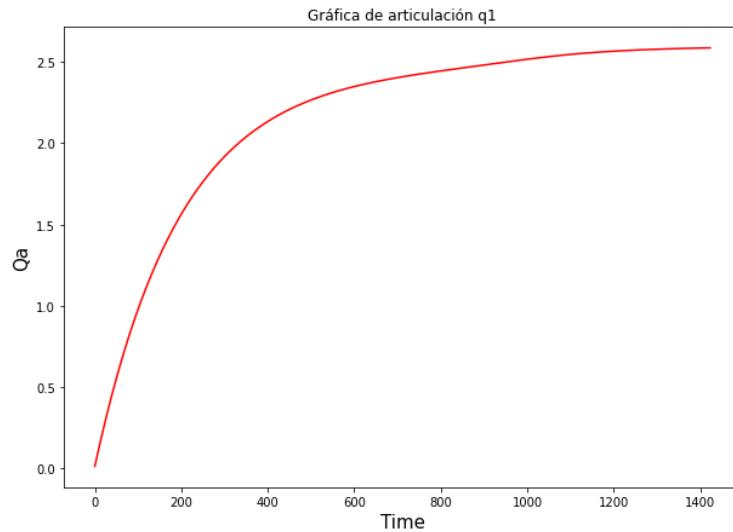


Imagen 5.5. Gráfico de varianza de la articulación q_1

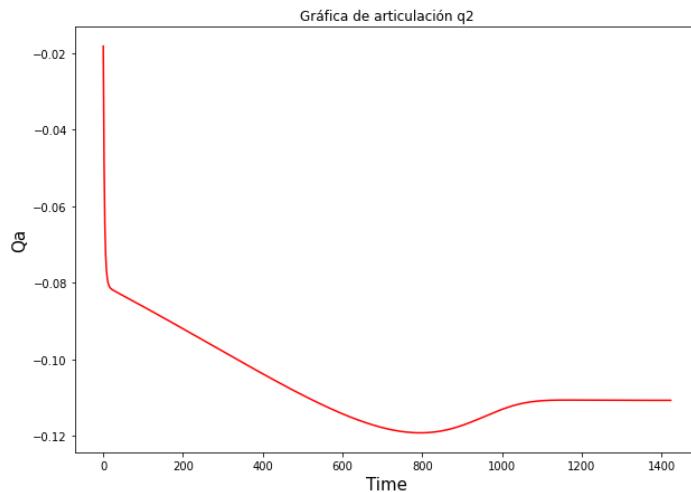


Imagen 5.6. Gráfico de varianza de la articulación q_2

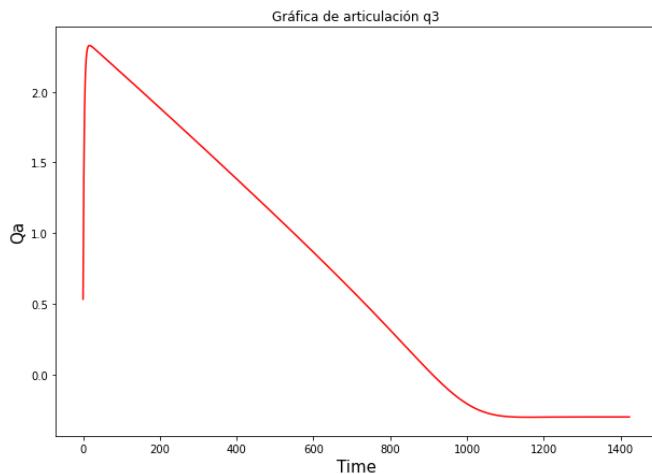


Imagen 5.7. Gráfico de varianza de la articulación q3

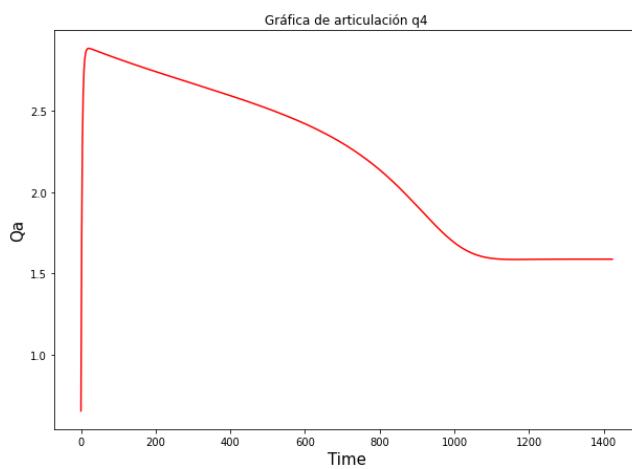


Imagen 5.8. Gráfico de varianza de la articulación q4

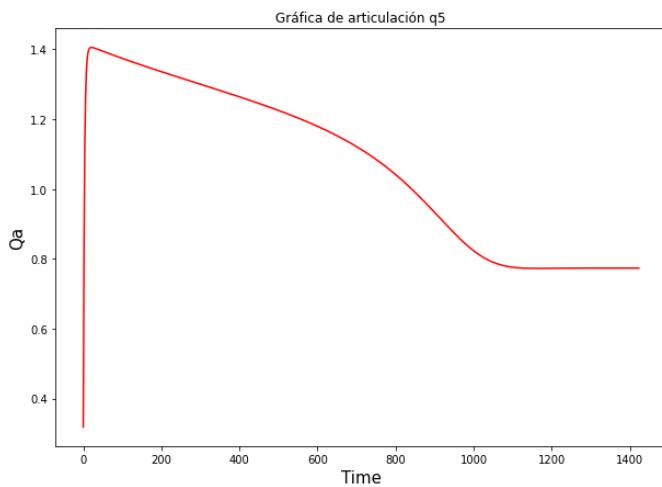


Imagen 5.9. Gráfico de varianza de la articulación q5

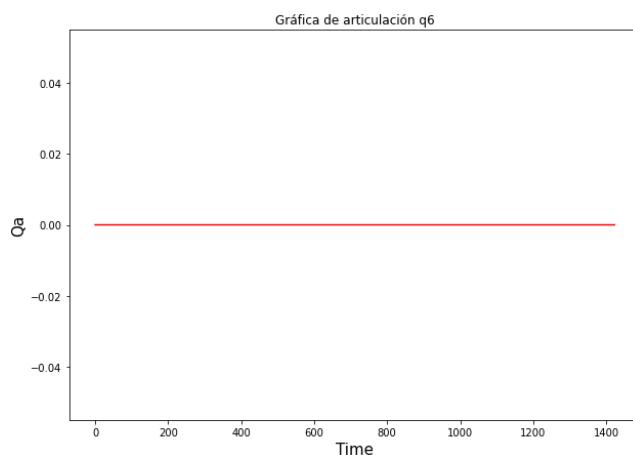


Imagen 5.10. Gráfico de varianza de la articulación q6

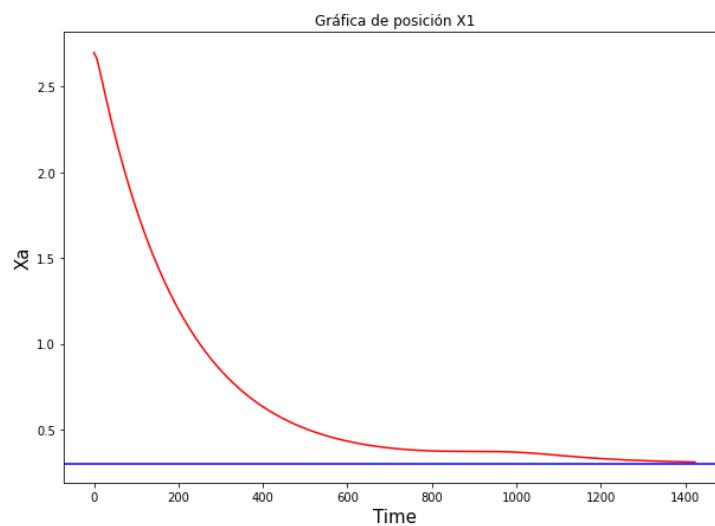


Imagen 5.11. Gráfico de varianza de la coordenada X de posición

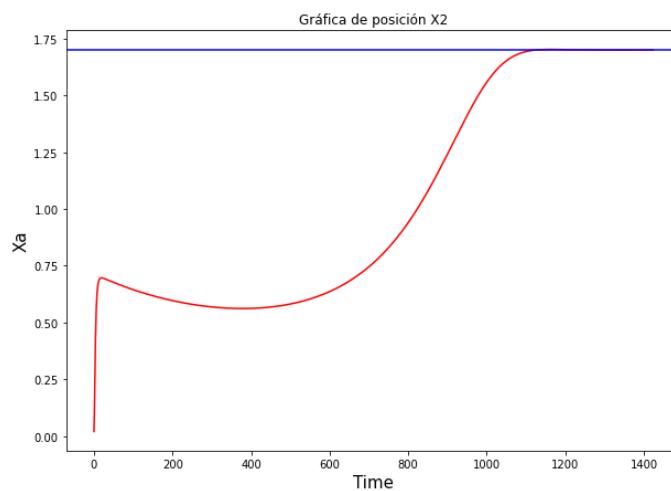


Imagen 5.12. Gráfico de varianza de la coordenada Y de posición

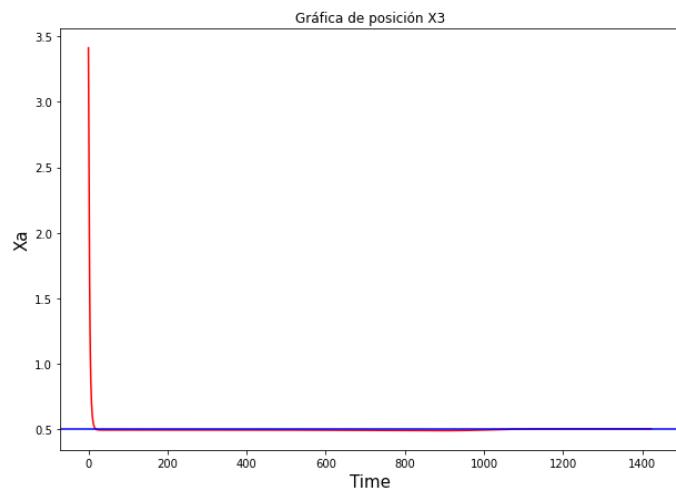


Imagen 5.13. Gráfico de varianza de la coordenada Z de posición.

Comentario:

Se pudo observar cómo es que visualmente (en Rviz y en las gráficas) el robot llega a converger llegando a la posición deseada. Las coordenadas cartesianas llegan correctamente al valor deseado (marcado por una línea horizontal de color azul), mientras que el comportamiento de las articulaciones no es de gran relevancia para este tipo de control.

- Configuraciones singulares

El robot manipulador presentado en este proyecto presenta algunas posibles configuraciones singulares debido a su diseño. Esto significa que cuando se den estos casos, el robot perderá un grado de libertad, no se podrá mover libremente hacia una determinada dirección. Por ejemplo, en la configuración inicial:

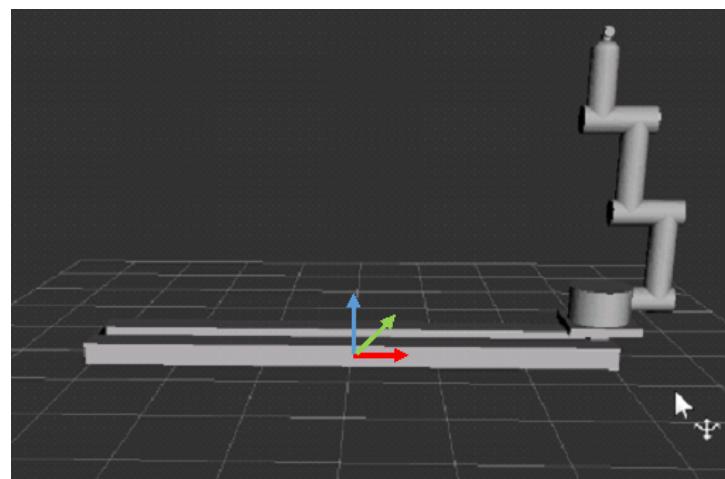


Imagen 5.14. Configuración inicial del robot en su máximo eje Z.

El efecto final pierde capacidad de movimiento con respecto al eje Z inercial (de color azul) debido a que la articulación prismática se encuentra en su límite, por lo tanto, no es posible un movimiento positivo en este eje.

Por otro lado, otra configuración singular se consigue si se realiza un giro de 90 grados en la articulación de revolución q3:

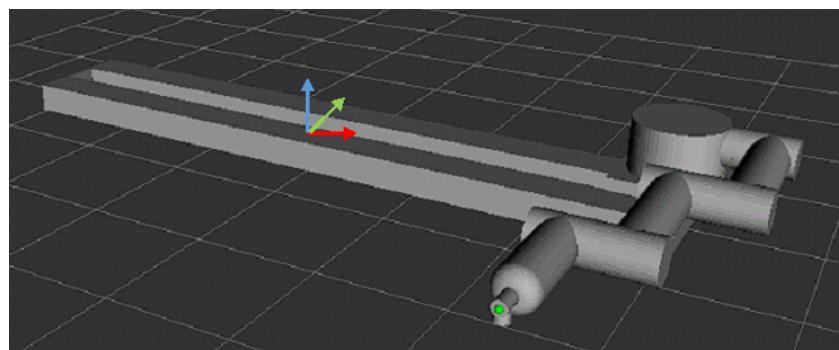


Imagen 5.15. Configuración inicial del robot en su máximo eje Y.

El brazo se encuentra totalmente estirado hacia la pantalla, por lo cual el efecto final se encuentra en el límite con respecto al eje Y inercial (de color verde), ocasionando que no sea posible un movimiento en dirección negativa a este eje.

6. Dinámica y control dinámico

Para probar el correcto funcionamiento de modelo obtenido a partir de RBDL se le darán valores de configuración articular inicial, velocidad articular inicial y posición articular deseada

```
# Configuración inicial
q = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
# Velocidad inicial
dq = np.array([0., 0., 0., 0., 0., 0.])
# Configuración articular deseada
qdes = np.array([0.5, 2.35619, 0., 0., 0., 0.])
```

g:

Dado que M y C dependen de ddq y dq respectivamente, si los igualamos a 0, entonces se podrá obtener la matriz g(q)

$$g(q) = \text{InverseDynamics}(q, 0, 0)$$

$$\begin{bmatrix} 0. & 0. & -3.31 & 2.46 & 2.46 & 0. \end{bmatrix}$$

C:

Dado que M de ddq lo igualo a 0 para que desaparezca, con ello me queda C+g, pero como ya se conoce el valor de g(q) solo bastaría con restarlo

$$c(q, dq) = \text{InverseDynamics}(q, dq, 0) - g(q)$$

$$\begin{bmatrix} 0. & 0. & 0. & 0. & 0. & 0. \end{bmatrix}$$

M:

Para eliminar a la matriz C, se iguala a 0 al dq , y dado que M depende de q y de ddq , entonces es necesario restarle $g(q)$ a este resultado. Cabe resaltar que este valor se coloca en un bucle que recorra toda la matriz de identidad e, la cual es cuadrada y tiene la dimensión de la cantidad de grados de libertad.

$mi = \text{InverseDynamics}(q, 0, e[i]) - g(q)$

$$\begin{bmatrix} 4176.99 & -0.21 & 0.0 & 0.0 & 0.0 & -0.25 \\ -0.21 & 489.86 & -282.59 & -54.58 & -33.25 & 3.23 \\ 0.0 & -282.59 & -5589.85 & -2591.1 & -447.92 & 0.0 \\ 0.0 & -54.58 & -2591.1 & 1394.75 & 274.33 & 0.0 \\ 0.0 & -33.25 & -447.92 & 274.33 & 100.67 & 0.0 \\ 0.25 & 3.23 & 0.0 & 0.0 & 0.0 & 3.23 \end{bmatrix}$$

- Resultados obtenidos

Se verificó que el resultado obtenido sea correcto utilizando RViz para obtener las siguientes gráficas:

Control PD + compensación de la gravedad

dsfsdfsdf

$$\mathbf{u} = \mathbf{g}(\mathbf{q}) + K_p(\mathbf{q}_d - \mathbf{q}) - K_d \dot{\mathbf{q}}$$

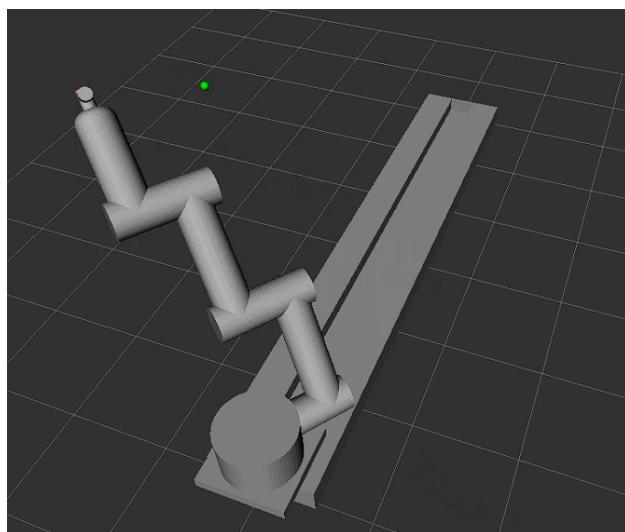


Imagen 6.1. Efecto final en movimiento hacia las coordenadas deseadas (Parte 1).

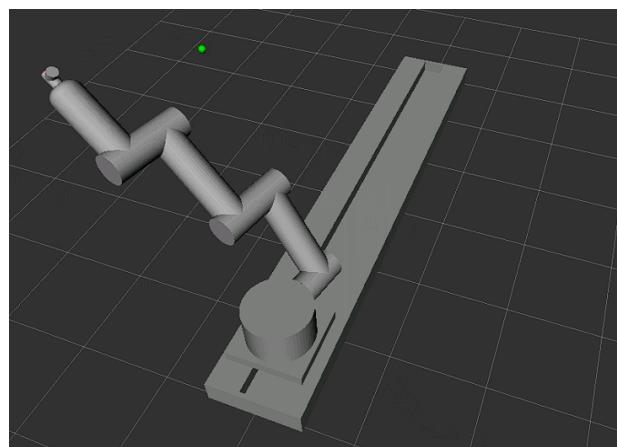


Imagen 6.2. Efecto final en movimiento hacia las coordenadas deseadas (Parte 2).

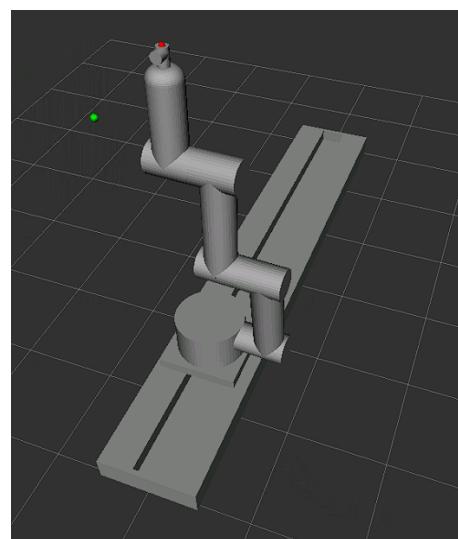


Imagen 6.3. Efecto final en movimiento hacia las coordenadas deseadas (Parte 3).

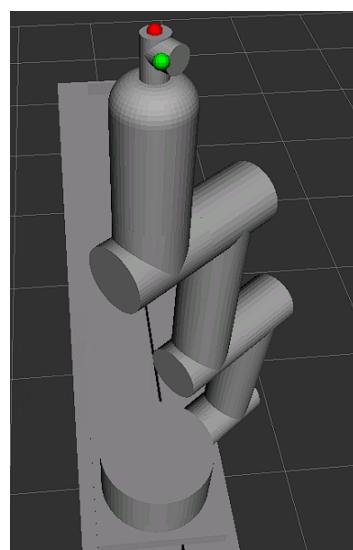


Imagen 6.4. Efecto final en movimiento hacia las coordenadas deseadas (Parte 4).

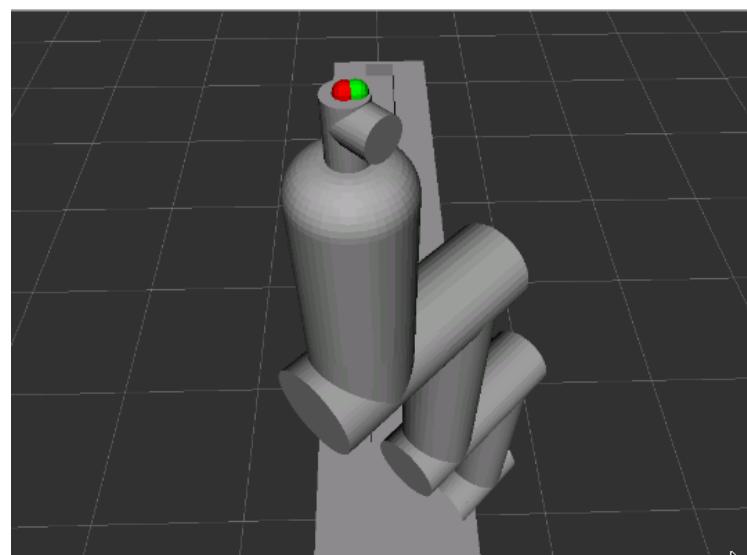


Imagen 6.5. Efecto final llegando a la posición deseada.

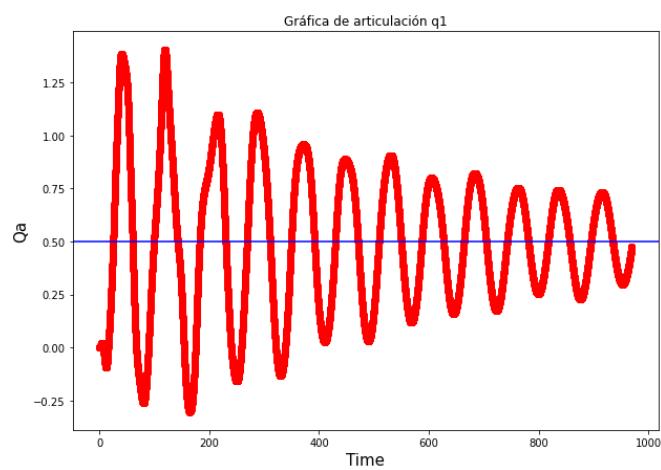


Imagen 6.6. Gráfica de movimiento de la articulación q1.

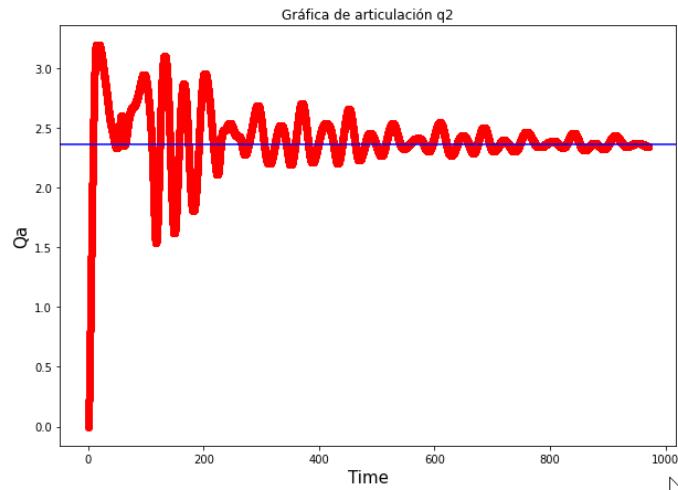


Imagen 6.7. Gráfica de movimiento de la articulación q2.

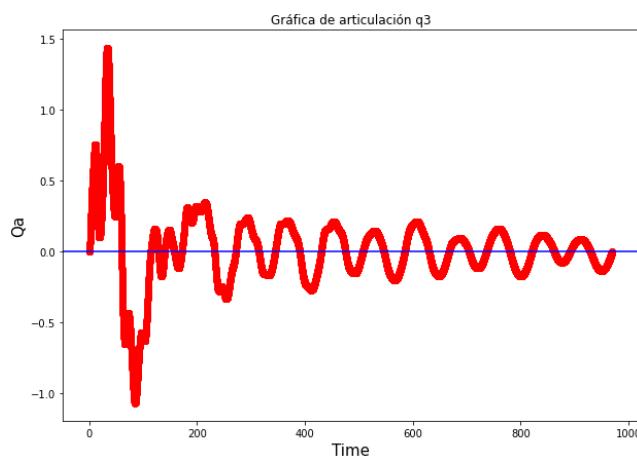


Imagen 6.8. Gráfica de movimiento de la articulación q3.

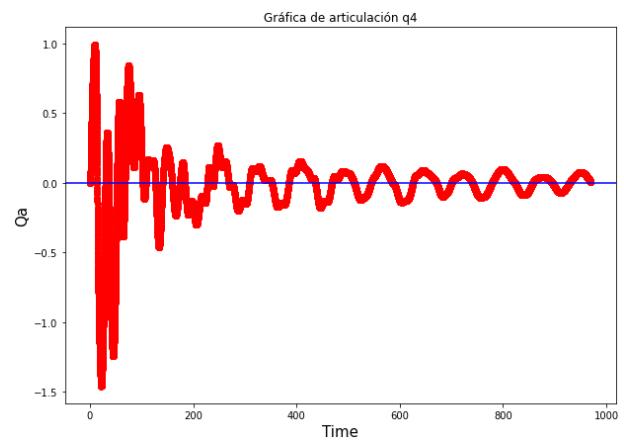


Imagen 6.9. Gráfica de movimiento de la articulación q4.

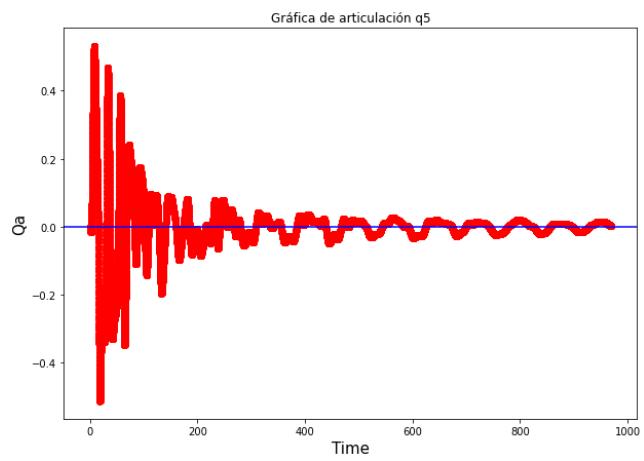


Imagen 6.10. Gráfica de movimiento de la articulación q5.

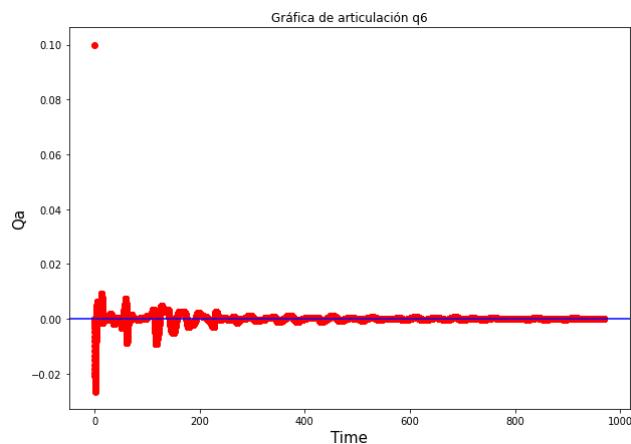


Imagen 6.11. Gráfica de movimiento de la articulación q_6 .

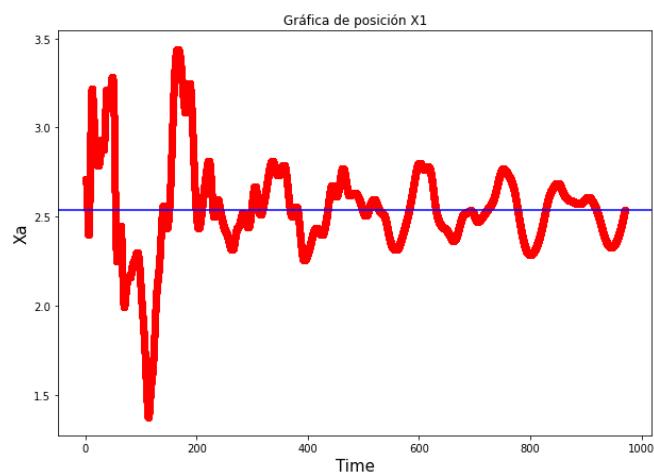


Imagen 6.12. Varianza de la coordenada X de posición.

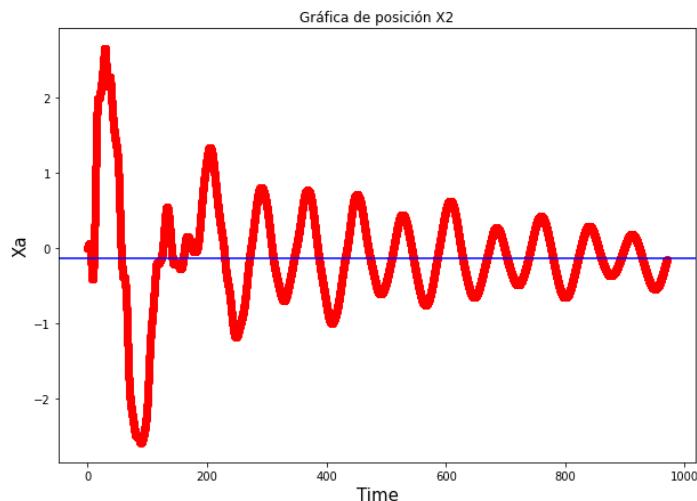


Imagen 6.13. Varianza de la coordenada Y de posición.

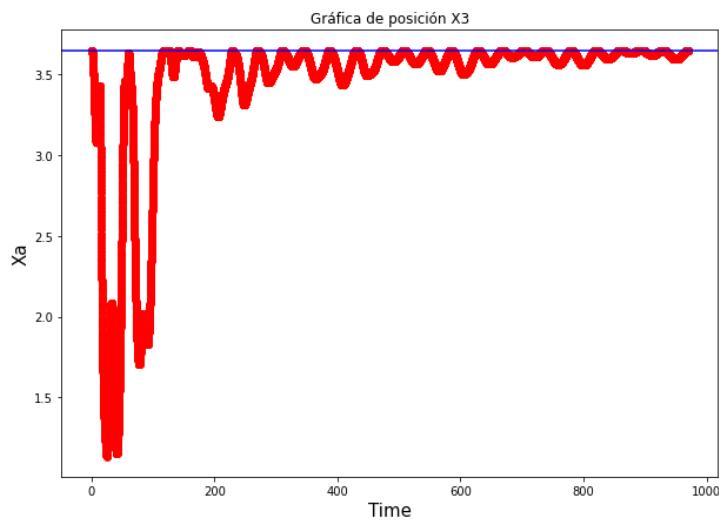


Imagen 6.14. Varianza de la coordenada Z de posición.

Las coordenadas cartesianas llegan al valor deseado (marcada como una línea horizontal de color azul) con un sobreimpulso considerable, al igual que las articulaciones, las cuales presentan oscilación en la respuesta final. Además, cabe resaltar que se demoró un tiempo considerable en llegar a la convergencia, lo cual no es lo más deseado

Control por Dinámica Inversa

$$\mathbf{u} = M(\mathbf{q}) \left(\ddot{\mathbf{q}}_d + K_d (\dot{\mathbf{q}}_d - \dot{\mathbf{q}}) + K_p (\mathbf{q}_d - \mathbf{q}) \right) + C(\mathbf{q}, \dot{\mathbf{q}}) \dot{\mathbf{q}} + \mathbf{g}(\mathbf{q})$$

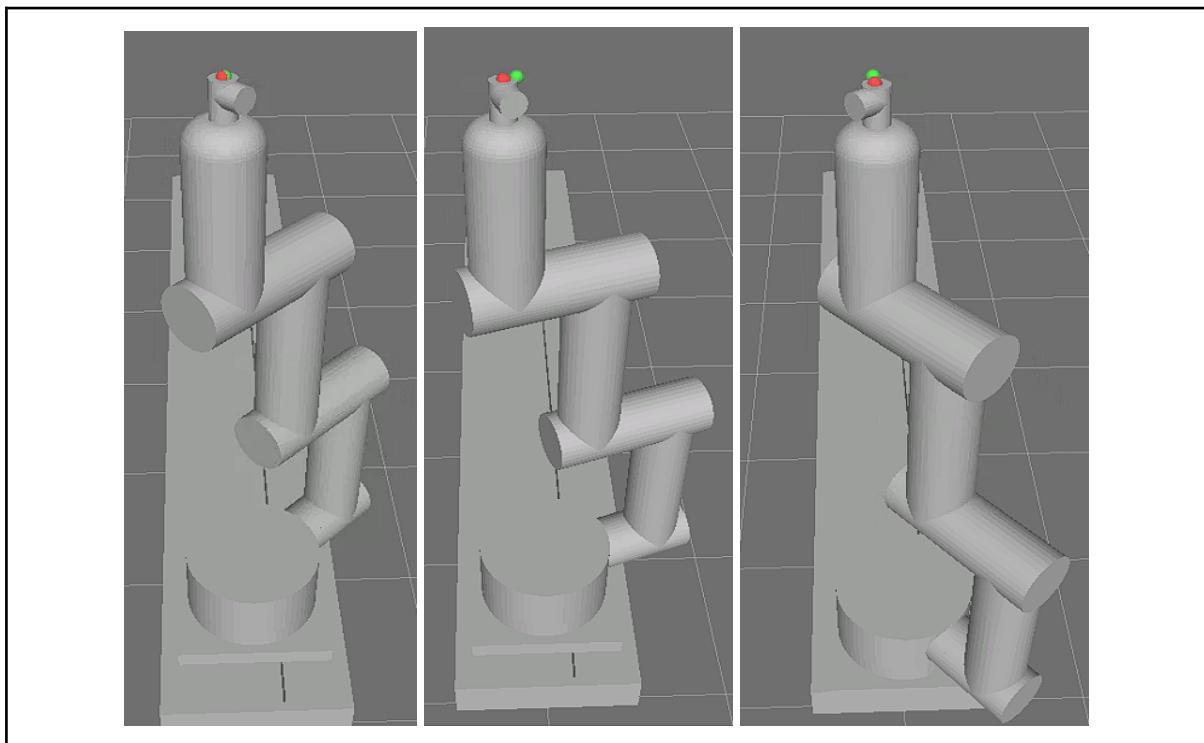


Imagen 6.15. Efecto final ubicado en las coordenadas establecidas por dinámica inversa.

- Gráficas

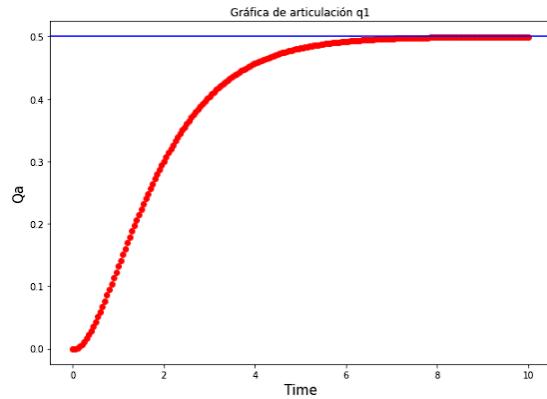


Imagen 6.16. Gráfico de varianza de la articulación q1.

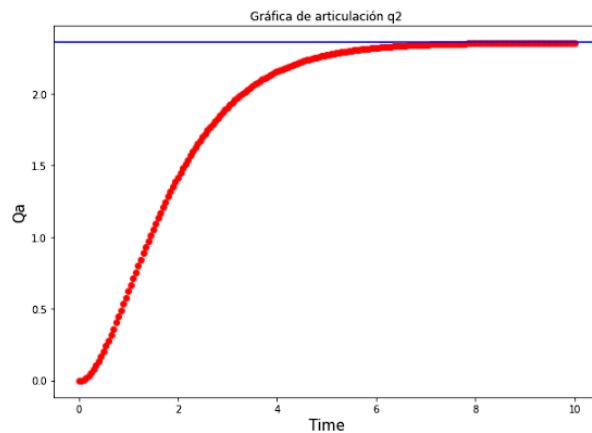


Imagen 6.17. Articulación q2

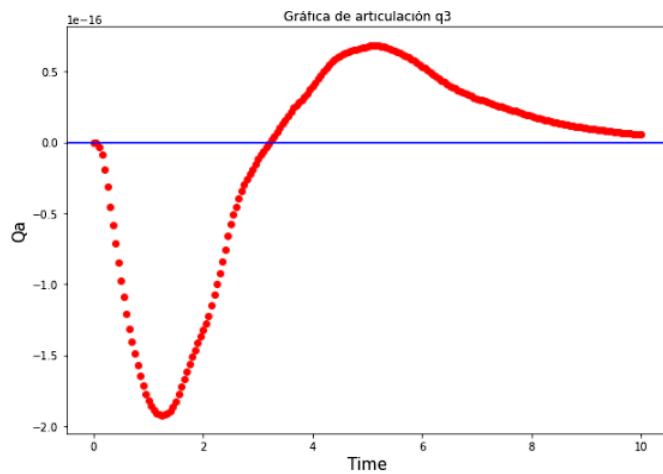


Imagen 6.18. Gráfico de varianza de la articulación q3.

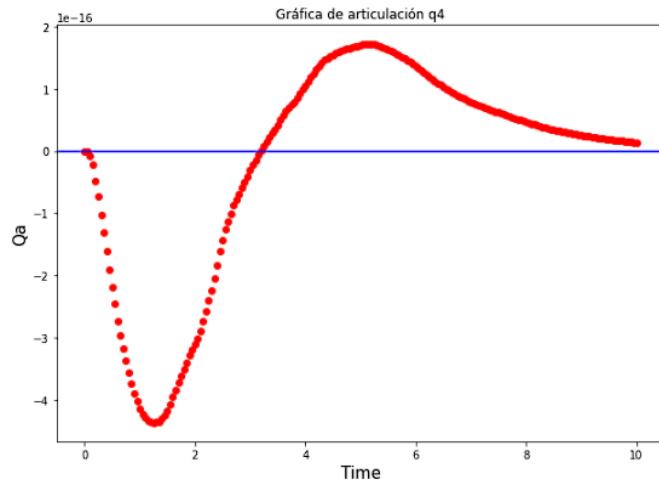


Imagen 6.19. Gráfico de varianza de la articulación q4.

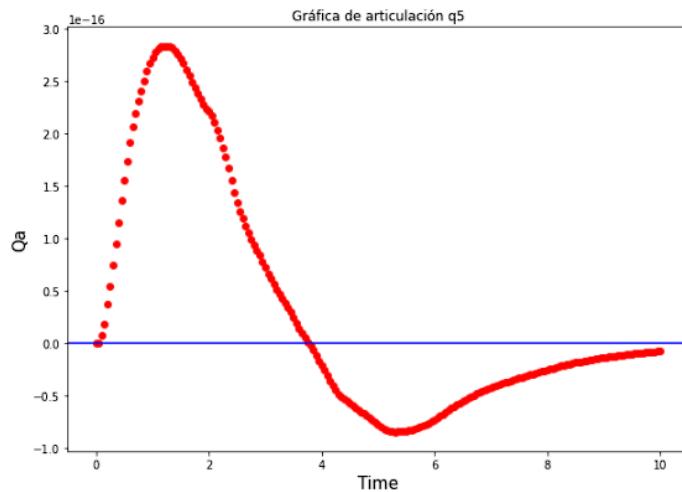


Imagen 6.20. Gráfico de varianza de la articulación q5.

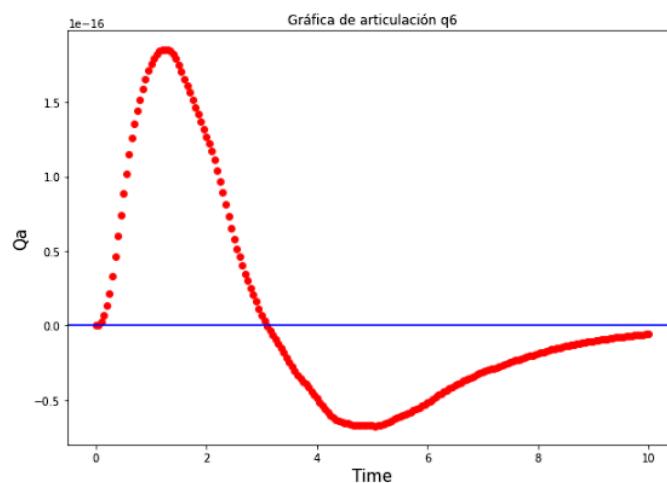


Imagen 6.21. Gráfico de varianza de la articulación q6.

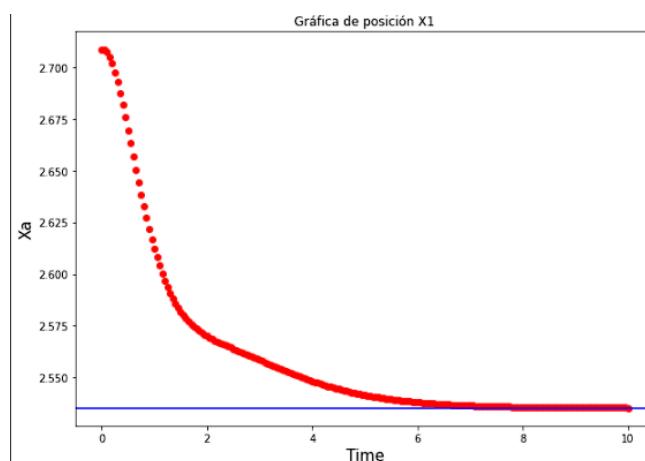


Imagen 6.22. Gráfico de varianza de la coordenada X de posición.

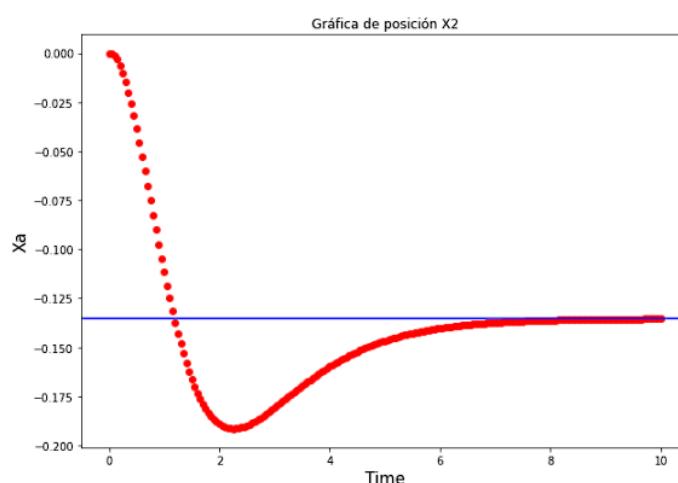


Imagen 6.23. Gráfico de varianza de la coordenada Y de posición.

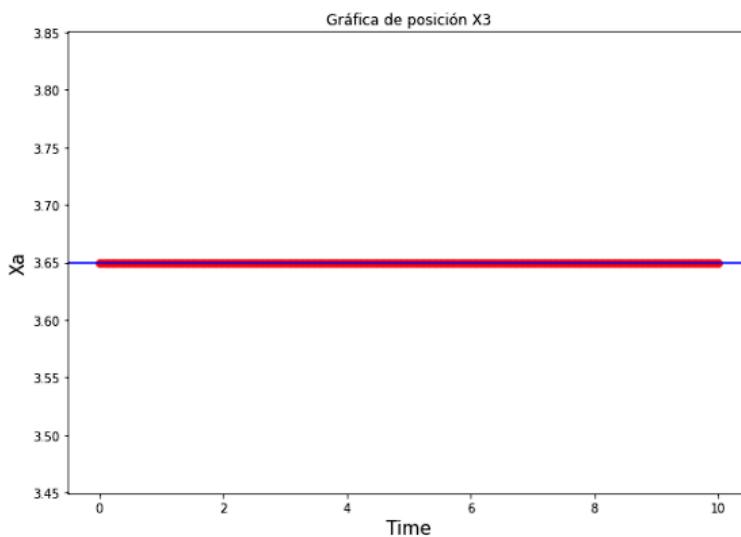


Imagen 6.24. Gráfico de varianza de la coordenada Z de posición.

Comentarios:

Para este caso, como se puede observar en las gráficas, la convergencia fue rápida y precisa. Aquí se puede observar claramente la ventaja en cuanto a precisión que ofrece la dinámica inversa sobre los demás tipos de control, no obstante, cabe resaltar que el costo de aquella precisión es su costo computacional el cual es alto debido a que con cada iteración debe calcular de nuevo las matrices del modelo dinámico.

Notar además que las gráficas de q3, q4, q5 y q6 están en una escala de e-16, por lo que desde el comienzo hasta su estabilización tienen valores iguales a 0

7. Conclusiones

- Se consiguió diseñar un robot original desde cero el cual fue realizado en el software CAD Fusion 360.
- Se consiguió implementar el robot G-O-LIAT de forma exitosa en ROS, de forma que cumplió con todas las instrucciones establecidas en los códigos para llegar a las posiciones indicadas
- Usando un mismo punto de destino, nuestro robot fue capaz de llegar a este mediante la Ley de control PD+g y mediante Dinámica Inversa, no obstante, usando esta última el robot llegó en pocos segundos, mientras que en la primera demoró en estabilizarse.

8. Recomendaciones

- Si se tiene como primer sistema un desplazamiento prismático, recomendamos multiplicar a la matriz de Transformación Homogénea por matrices T de rotación de forma que este primer sistema coincida con el eje de coordenadas del visualizador, donde normalmente 'z' es el eje de altura.
- No escatimar en decimales y tampoco en aproximaciones visuales al momento de medir las distancias, dado que con ello se obtiene la matriz de DH, la cual es la base

de todo el movimiento del robot, si esta está mal, todo fallará, no debe haber fallas en las mediciones

9. Anexos

Funciones utilizadas:

```
import numpy as np
from copy import copy
import rbdl

pi = np.pi
cos = np.cos; sin=np.sin; pi=np.pi

# Matrices np de rotacion homogeneisadas
def sTrotx(ang):
    Tx = np.array([[1, 0, 0, 0],
                  [0, np.cos(ang), -np.sin(ang), 0],
                  [0, np.sin(ang), np.cos(ang), 0],
                  [0, 0, 0, 1]])
    return Tx

def sTroty(ang):
    Ty = np.array([[np.cos(ang), 0, np.sin(ang), 0],
                  [0, 1, 0, 0],
                  [-np.sin(ang), 0, np.cos(ang), 0],
                  [0, 0, 0, 1]])
    return Ty

def sTrotz(ang):
    Tz = np.array([[np.cos(ang), -np.sin(ang), 0, 0],
                  [np.sin(ang), np.cos(ang), 0, 0],
                  [0, 0, 1, 0],
                  [0, 0, 0, 1]])
    return Tz

#Pure traslation
def tf_tr(d,c):

    if c == 'x':
```

```

Ttrx = np.array([[1, 0, 0, d],
                [0, 1, 0, 0],
                [0, 0, 1, 0],
                [0, 0, 0, 1]])
return Ttrx

elif c == 'y':
    Ttry = np.array([[1, 0, 0, 0],
                    [0, 1, 0, d],
                    [0, 0, 1, 0],
                    [0, 0, 0, 1]])
    return Ttry

elif c == 'z':
    Ttrz = np.array([[1, 0, 0, 0],
                    [0, 1, 0, 0],
                    [0, 0, 1, d],
                    [0, 0, 0, 1]])
    return Ttrz

def dh(d, theta, a, alpha):
    cth=cos(theta)
    sth=sin(theta)
    ca=cos(alpha)
    sa=sin(alpha)
    T = np.array([[cth, -ca*sth, sa*sth, a*cth],
                  [sth, ca*cth, -sa*cth, a*sth],
                  [0, sa, ca, d],
                  [0,0,0,1]])
    return T

def fkine_ur5(q):

    # Matrices DH (completar), emplear la función dh con los parámetros DH
    # para cada articulación

```

```

T1 = dh( 3-0.1-q[0], pi, 0, pi/2)
T2 = dh( 0.700, q[1]+pi, 0, pi/2)
T3 = dh( 0.6343965, q[2]-pi/2, -1, pi)
T4 = dh( 0.424906, q[3], -1, 0)
T5 = dh( 0.400922, q[4]-pi/2, 0, pi/2)
T6 = dh( 0.950, q[5]-pi/2, 0, 0)

# Efector final con respecto a la base
T =
sTroty(pi/2).dot(sTrotz(pi/2)).dot(T1).dot(T2).dot(T3).dot(T4).dot(T5).do
t(T6)
return T

# Obtencion del Jacobiano (se recibe el q actual y la variacion que se le
hara)
def jacobian_ur5(q, delta=0.0001):
    """
    Jacobiano analitico para la posicion. Retorna una matriz de 3x6 y toma
    como
    entrada el vector de configuracion articular q=[q1, q2, q3, q4, q5, q6]
    """
    # Crear una matriz 3x6
    J = np.zeros((3,6))
    # Transformacion homogenea (0-T-6) usando el q dado
    T = fkine_ur5(q)
    # Iteracion para la derivada de cada columna
    for i in xrange(6):
        # Copiar la configuracion articular(q) y almacenarla en dq
        dq = copy(q)
        # Incrementar los valores de cada q sumandole un delta a cada uno
        dq[i] = dq[i] + delta
        # Obtencion de la nueva Matriz Homogenea con los nuevos valores
        # articulares, luego del incremento (q+delta)
        Td = fkine_ur5(dq)
        # Aproximacion del Jacobiano de posicion usando diferencias finitas
        for j in xrange(3):
            J[j][i] = (Td[j][3]-T[j][3])/delta
    return J

```

```

# Metodo de Newton, se recibe la posicion a la que se desea ir(xdes) y el
valor actual del q
def ikine_ur5(xdes, q0):
    """
    Calcular la cinematica inversa de UR5 numericamente a partir de la
    configuracion articular inicial de q0.
    Emplear el metodo de newton
    """

    epsilon = 0.001      #error(distancia) minima requerida
    max_iter = 1000       #max cantidad de iteraciones
    delta     = 0.00001    #variacion(movimiento) en el espacio

    q = copy(q0)  #se copia el valor de la articulacion inicial(q0) y se
    almacena en q
    for i in range(max_iter):
        J=jacobian_ur5(q,delta)  # Matriz Jacobiana
        Td=fkine_ur5(q)          # Matriz Actual
        #Posicion Actual: extrayendo la parte translacional de la Matriz
        Homogenea(d1,d2,d3) y almacenandolos en el vector xact
        xact=Td[0:3,3]

        # Error entre pos deseada y pos actual (cuanta distancia separa ambos
        puntos)
        e=xdes-xact  #diferencia de ambos, dado que son 2 vectores con mismo
        origen, la resta significa un vector que va desde xact a xdes
                    #e=var(x,y,z) -> variacion espacial para llegar a pos
        deseada

        # Metodo de Newton (se actualiza el valor de q y vuelve al loop si
        tdv no llega a la max iteracion)
        q=q+np.dot(np.linalg.pinv(J),e)  #q=q+var(q) ->
        q=q+(inv(J)*var(x,y,z)) -> q=q+(inv(J)*e)
        #Condicion de termino
        if(np.linalg.norm(e)<epsilon):  #norma=modulo, el modulo es la
        distancia en magnitud faltante para llegar a la posicion deseada
            break
    pass

```

```
return q

# Lo mismo que se realizo anteriormente, solo cambiando la linea de
# metodo de Newton por el del Metodo de la Gradiente
#Este metodo necesita ademas del parametro delta
def ik_gradient_ur5(xdes, q0):
    """
    Calcular la cinematica inversa de UR5 numericamente a partir de la
    configuracion articular inicial de q0.

    Emplear el metodo gradiente
    """
    epsilon = 0.001
    max_iter = 1000
    delta = 0.00001
    alpha = 0.5

    q = copy(q0)
    for i in range(max_iter):
        # Main loop
        #Matriz Jacobiana
        J=jacobian_ur5(q,delta)
        #Matriz Actual
        Td=fkine_ur5(q)
        #Posicion Actual
        xact=Td[0:3,3]
        # Error entre pos deseada y pos actual
        e=xdes-xact

        # Metodo de la gradiente
        q=q+alpha*np.dot(J.T,e)
        #Condicion de termino
        if(np.linalg.norm(e)<epsilon):
            break
        pass
    return q

class Robot(object):
    def __init__(self, q0, dq0, ndof, dt):
```

```

        self.q = q0      # numpy array (ndof x 1)
        self.dq = dq0    # numpy array (ndof x 1)
        self.M = np.zeros([ndof, ndof])
        self.b = np.zeros(ndof)
        self.dt = dt
        self.robot =
rbdl.loadModel('../urdf/FR_Proyecto_Manipulador.urdf')

def send_command(self, tau):
    rbdl.CompositeRigidAlgorithm(self.robot, self.q, self.M)
    rbdl.NonlinearEffects(self.robot, self.q, self.dq, self.b)
    ddq = np.linalg.inv(self.M).dot(tau-self.b)
    self.q = self.q + self.dt*self.dq
    self.dq = self.dq + self.dt*ddq

def read_joint_positions(self):
    return self.q

def read_joint_velocities(self):
    return self.dq

```

Cinemática directa:

```

#!/usr/bin/env python

import rospy
from sensor_msgs.msg import JointState

from markers import *
from funciones import *

rospy.init_node("testForwardKinematics")
pub = rospy.Publisher('joint_states', JointState, queue_size=1000)
bmarker = BallMarker(color['GREEN'])

# Joint names
#jnames = ['Revolucion1_position_controller',
'Revolucion2_position_controller',
'Revolucion3_position_controller','Revolution4_position_controller',

```

```
'Revolucion5_position_controller', 'Corredera6_position_controller']
#jnames = ['Revolucion1', 'Revolucion2', 'Revolucion3','Revolucion4',
'Revolucion5', 'Corredera6']
jnames = ['Corredera6', 'Revolucion5', 'Revolucion4','Revolucion3',
'Revolucion2', 'Revolucion1']

# Joint Configuration
q = [0, 0, 0, 0, 0, 0] # 2.35619, 1.5708, 0.785398

# End effector with respect to the base
T = fkine_ur5(q)
print( np.round(T, 3) )
bmarker.position(T) #Set position (4x4 NumPy homogeneous matrix) for
the ball and publish it

# Object (message) whose type is JointState
jstate = JointState()
# Set values to the message
jstate.header.stamp = rospy.Time.now()
jstate.name = jnames
# Add the head joint value (with value 0) to the joints
jstate.position = q

# Loop rate (in Hz)
rate = rospy.Rate(100)
# Continuous execution loop
while not rospy.is_shutdown():
    # Current time (needed for ROS)
    jstate.header.stamp = rospy.Time.now()
    # Publish the message
    pub.publish(jstate)
    bmarker.publish()
    # Wait for the next iteration
    rate.sleep()
```

Cinemática inversa:

```
#!/usr/bin/env python
```

```

#IN:
#posicion deseada x_des
#q0 inicial

#OUT:
#robot con efecto final en posicion del x_des
#1 Marker en la posicion espacial del efecto final
#1 Marker en la posicion espacial del x deseado

import rospy
from sensor_msgs.msg import JointState
from markers import *
from funciones import *

rospy.init_node("testInvKine")
#Publica los JointState en el topico joint_states
pub = rospy.Publisher('joint_states', JointState, queue_size=1000)

bmarker      = BallMarker(color['RED'])
bmarker_des  = BallMarker(color['GREEN'])

# Joint names
jnames = ['Corredora6', 'Revolucion5', 'Revolucion4', 'Revolucion3',
'Revolucion2', 'Revolucion1']

# Desired position
xd = np.array([0.3, 1.7, 0.5])
# Initial configuration
q0 = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# Inverse kinematics -> obtiene el valor de q necesario para llegar a la
# posicion deseada
q = ikine_ur5(xd, q0)                      #usando metodo de Newton
#q = ik_gradient_ur5(xd, q0)                  #usando metodo de la gradiente

# Se obtiene la nueva Matriz Homogenea en base al nuevo q -> (0-T-xdes)

```

```

# Resulting position (end effector with respect to the base link)
T = fkine_ur5(q)
print('Obtained value:\n', np.round(T,3))

# Red marker shows the achieved position
bmarker.xyz(T[0:3,3])
# Green marker shows the desired position
bmarker_des.xyz(xd)

# Objeto (mensaje) de tipo JointState
jstate = JointState()
# Asignar valores al mensaje
jstate.header.stamp = rospy.Time.now()
jstate.name = jnames
# Add the head joint value (with value 0) to the joints
jstate.position = q
print('q:\n', np.round(jstate.position,3))

# Loop rate (in Hz)
rate = rospy.Rate(100)
# Continuous execution loop
while not rospy.is_shutdown():                                #mientras que no se presiona
    Ctrl+C
        # Current time (needed for ROS)
        jstate.header.stamp = rospy.Time.now()
        # Publicando el estado de las articulaciones del robot y las
        # posiciones de las bolas roja y verde
        pub.publish(jstate)
        bmarker.publish()
        bmarker_des.publish()
        # Wait for the next iteration
        rate.sleep()

```

Control cinemático:

```

#!/usr/bin/env python
#

```

```
#IN:  
#posicion cartesiana deseada (xd)  
#pos articular inicial (q0)  
  
#OUT:  
#Movimiento hacia la posicion deseada (se guardaran datos de xactual xd,  
error)  
  
from __future__ import print_function  
import rospy  
from sensor_msgs.msg import JointState  
  
from markers import *  
from funciones import *  
  
from numpy import matrix, rank  
  
# Initialize the node  
rospy.init_node("testKineControlPosition")  
print('starting motion ... ')  
# Publisher: publish to the joint_states topic  
pub = rospy.Publisher('joint_states', JointState, queue_size=10)  
# Files for the logs  
fxcurrent = open("/tmp/xcurrent.txt", "w")  
fxdesired = open("/tmp/xdesired.txt", "w")  
fq = open("/tmp/q.txt", "w")  
  
# Markers for the current and desired positions  
bmarker_current = BallMarker(color['RED'])  
bmarker_desired = BallMarker(color['GREEN'])  
  
# Joint names  
jnames = ['Corredera6', 'Revolucion5', 'Revolucion4','Revolucion3',  
'Revolucion2', 'Revolucion1']  
  
# Desired position  
xd = np.array([0.3, 1.7, 0.5])  
# Initial configuration  
q0 = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
```

```

# Resulting initial position (end effector with respect to the base link)
T = fkine_ur5(q0)
x0 = T[0:3,3]      #initial position(x,y,z)

# Red marker shows the achieved position
bmarker_current.xyz(x0)
# Green marker shows the desired position
bmarker_desired.xyz(xd)

# Instance of the JointState message
jstate = JointState()
# Values of the message
jstate.header.stamp = rospy.Time.now()
jstate.name = jnames
# Add the head joint value (with value 0) to the joints
jstate.position = q0

# Frequency (in Hz) and control period
freq = 20
dt = 1.0/freq
rate = rospy.Rate(freq)

# Initial joint configuration
q = copy(q0)
min_err=0.01

# Main loop
while not rospy.is_shutdown():

    k = 0.1
    # Current time (needed for ROS)
    jstate.header.stamp = rospy.Time.now()
    # Kinematic control law for position (complete here)
    # -----
    #n=np.linalg.matrix_rank(J); m=J.ndim
    J = jacobian_ur5(q0, delta=0.01)
    e = x0 - xd
    e_der = -k*e

```

```

n=np.linalg.matrix_rank(J); m=J.ndim           ##### #np.linalg.det(J)==0

#####
if (n<m):
    q_der = np.dot(np.dot(np.linalg.inv(J.T*J),J.T), e_der) #pseudo
inversa amortiguada
else:
    q_der = np.dot(np.linalg.pinv(J), e_der)

#q_der = np.dot(np.linalg.pinv(J), e_der)
q = q + dt*q_der
# ----

T=fkine_ur5(q)
x=T[0:3,3]
x0=x
# Log values
fxcurrent.write(str(x[0])+' '+str(x[1])+' '+str(x[2])+'\n')
fxdesired.write(str(xd[0])+' '+str(xd[1])+' '+str(xd[2])+'\n')
fq.write(str(q[0])+" "+str(q[1])+" "+str(q[2])+" "+str(q[3])+" "+
         str(q[4])+" "+str(q[5])+"\n")

if(np.linalg.norm(e)<min_err):
    break

# Publish the message
jstate.position = q
pub.publish(jstate)
bmarker_desired.xyz(xd)
bmarker_current.xyz(x)

# Wait for the next iteration
rate.sleep()

print('ending motion ...')
fxcurrent.close()

```

```
fxdesired.close()
fq.close()
```

Control dinámico:

```
#!/usr/bin/env python

#IN:
#pos articular inicial (q0)
#velocidad articular inicial (dq)
#pos articular deseada (qdes)

#OUT:
#desplazamiento del robot hacia el punto indicado, así como la aparición
de los markers, uno en el lugar deseado y otro en el efecto final del
robot

import rospy
from sensor_msgs.msg import JointState
from markers import *
from funciones import *
from roslib import packages

import rbdl

rospy.init_node("control_pdg")
pub = rospy.Publisher('joint_states', JointState, queue_size=1000)
bmarker_actual = BallMarker(color['RED'])
bmarker_deseado = BallMarker(color['GREEN'])

# Archivos donde se almacenara los datos
fqact = open("/tmp/qactual.dat", "w")
fqdes = open("/tmp/qdeseado.dat", "w")
fxact = open("/tmp/xactual.dat", "w")
fxdes = open("/tmp/xdeseado.dat", "w")

# Nombres de las articulaciones
# Joint names
jnames = [ 'Corredera6', 'Revolucion5', 'Revolucion4','Revolucion3',
```

```
'Revolucion2', 'Revolucion1']
# Objeto (mensaje) de tipo JointState
jstate = JointState()
# Valores del mensaje
jstate.header.stamp = rospy.Time.now()
jstate.name = jnames

# =====
# Configuracion articular inicial (en radianes)
q = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
# Velocidad inicial
dq = np.array([0., 0., 0., 0., 0., 0.])
# Configuracion articular deseada
qdes = np.array([0.5, 2.35619, 0., 0., 0., 0.])
# =====

# Posicion resultante de la configuracion articular deseada
xdes = fkine_ur5(qdes)[0:3,3]

# Copiar la configuracion articular en el mensaje a ser publicado
jstate.position = q
pub.publish(jstate)

# Modelo del robot
modelo = rbdl.loadModel('../urdf/FR_Proyecto_Manipulador.urdf')
#print(dir(modelo))
ndof = modelo.q_size # Grados de libertad

# Frecuencia del envío (en Hz)
freq = 20
dt = 1.0/freq
rate = rospy.Rate(freq)

# Simulador dinamico del robot
robot = Robot(q, dq, ndof, dt)

# Se definen las ganancias del controlador
valores = 1*np.array([1.0, 1.0, 1.0, 1.0, 1.0, 1.0])
```

```

Kp = np.diag(valores)
Kd = 2*np.sqrt(Kp)

# Bucle de ejecucion continua
t = 0.0

# Arrays numpy
zeros = np.zeros(ndof)           # Vector de ceros
tau   = np.zeros(ndof)           # Para torque
g     = np.zeros(ndof)           # Para la gravedad
c     = np.zeros(ndof)           # Para el vector de Coriolis+centrifuga
M    = np.zeros([ndof, ndof])    # Para la matriz de inercia
e    = np.eye(6)                 # Vector identidad

min_err=0.001

rbdl.InverseDynamics(modelo,q,zeros,zeros,g)
rbdl.InverseDynamics(modelo,q,dq,zeros,tau)
C = tau - g
for i in range(ndof):
    rbdl.InverseDynamics(modelo,q,zeros,e[i],tau)
    M[i] = tau - g
print("g:")
print(g.round(3))
print("C:")
print(C.round(3))
print("M:")
print(np.round(M,2))

while not rospy.is_shutdown():
    # Leer valores del simulador
    q  = robot.read_joint_positions()
    dq = robot.read_joint_velocities()
    # Posicion actual del efecto final
    x = fkine_ur5(q)[0:3,3]
    # Tiempo actual (necesario como indicador para ROS)
    jstate.header.stamp = rospy.Time.now()

```

```

# Almacenamiento de datos
fxact.write(str(t)+' '+str(x[0])+''+str(x[1])+''+str(x[2])+'\n')
fxdes.write(str(t)+' '+str(xdes[0])+''+str(xdes[1])+''+
'+str(xdes[2])+'\n')
fqact.write(str(t)+' '+str(q[0])+''+str(q[1])+''+str(q[2])+' '+
str(q[3])+''+str(q[4])+''+str(q[5])+'\n ')
fqdes.write(str(t)+' '+str(qdes[0])+''+str(qdes[1])+''+
str(qdes[2])+''+str(qdes[3])+''+str(qdes[4])+''+str(qdes[5])+'\n ')

# -----
# Control dinamico (COMPLETAR)
# -----
u = np.zeros(ndof)    # Reemplazar por la ley de control

rbdl.InverseDynamics(modelo,q,zeros,zeros,g)
rbdl.InverseDynamics(modelo,q,dq,zeros,tau)
C = tau - g
for i in range(ndof):
    rbdl.InverseDynamics(modelo,q,zeros,e[i],tau)
    M[i] = tau - g
    error_q = qdes - q
    error_dq = theta - dq

ddqd = 0
if(np.linalg.norm(error_q)<min_err):
    break

#torque = Kp.dot(error_q) + Kd.dot(error_dq) # Control PD
#torque = Kp.dot(error_q) + Kd.dot(error_dq) + g # Control PD + G
torque = M.dot(ddqd+Kp.dot(error_q) + Kd.dot(error_dq)) + C + g
#control por dim inv
u = torque
#print(u)

# Simulacion del robot
robot.send_command(u)

# Publicacion del mensaje
jstate.position = q

```

```
pub.publish(jstate)
bmarker_deseado.xyz(xdes)
bmarker_actual.xyz(x)
t = t+dt
# Esperar hasta la siguiente iteracion
rate.sleep()

fqact.close()
fqdes.close()
fxact.close()
fxdes.close()
```