

Grado en Robótica EPSE-USC Lugo Física II

Práctica 2

CONFIGURACIONES SINGULARES
ELIPSOIDES DE MANIPULABILIDAD/FUERZA
CINEMÁTICA INVERSA



ÁNGEL PIÑEIRO, ABRIL DE 2021

Índice

1. Objetivo de la práctica	2
2. Material necesario	2
3. Matriz Jacobiana	2
4. Configuraciones Singulares	4
5. Elipsoides de Manipulabilidad y Fuerza	4
5.1. Evaluación del volumen del elipsoide de manipulabilidad, utilizando el código de la sección 3	5
6. Cinemática Inversa	7
7. Ejercicios Prácticos	8
7.1. Matriz Jacobiana	8
7.2. Configuraciones singulares	8
7.3. Elipsoides de Manipulabilidad y Fuerza	8
7.4. Cinemática Inversa	8
8. Apéndice	9
8.1. Código para resolución del problema cinemático inverso con Niryo One	9
8.2. Funciones utilizadas en el código que resuelve el problema cinemático inverso del Niryo One	10

1. Objetivo de la práctica

En esta práctica calcularemos la matriz Jacobiana para el robot Niryo One, que utilizaremos para encontrar y caracterizar algunas configuraciones singulares, calcularemos los elipsoides de manipulabilidad y fuerza para algunas configuraciones, y resolveremos el problema cinemático inverso.

2. Material necesario

- Robot Niryo One
- Ordenador con tarjeta Wifi

3. Matriz Jacobiana

La metodología para calcular la matriz Jacobiana de cualquier robot de lazo abierto consiste simplemente en determinar los ejes helicoidales para cada una de las articulaciones del robot en la configuración considerada. Estos ejes son directamente las columnas de la matriz Jacobiana. El protocolo detallado podéis verlo en [este vídeo](#) y el cálculo de esta matriz para el Robot Niryo One está descrito [aquí](#).

Los ejes de referencia y de rotación utilizados en este último vídeo no se corresponden con los utilizados en el software Niryo One Studio ni tampoco con el sentido de rotación positivo de algunas de las articulaciones. En el código que copiamos a continuación se usan exactamente los mismos sistemas de referencia y ejes de rotación utilizados en el software del Robot. El código está desarrollado con cálculo simbólico (utilizando la librería sympy) para tener un resultado genérico, parametrizado en función de las coordenadas de las articulaciones.

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

import numpy as np
import sympy as sp

# Función que convierte un eje de rotación en matriz antisimétrica 3x3 (so3)
def VecToso3(w): return sp.Matrix([[0,-w[2],w[1]], [w[2],0,-w[0]], [-w[1],w[0],0]])

# Definimos ejes de rotación de las articulaciones en la posición cero del robot
w=[]
w.append(np.array([0,0,1]))
w.append(np.array([0,-1,0]))
w.append(np.array([0,-1,0]))
w.append(np.array([1,0,0]))
w.append(np.array([0,-1,0]))
w.append(np.array([1,0,0]))

# Definimos los eslabones
scalefactor=0.001 # para cambiar las unidades a metros
L=np.array([103.0, 80.0, 210.0, 30.0, 41.5, 180.0, 23.7, -5.5])*scalefactor

# Definimos los vectores que van del centro de cada eje al centro del siguiente
q=[]
q.append(np.array([0,0,L[0]]))
q.append(np.array([0,0,L[1]]))
q.append(np.array([0,0,L[2]]))
q.append(np.array([L[4],0,L[3]]))
q.append(np.array([L[5],0,0]))
q.append(np.array([L[6],0,L[7]]))

# Coordenadas de las articulaciones
t=sp.symbols('t0, t1, t2, t3, t4, t5')
```

```

# Calculamos las matrices de rotación a partir de los ejes w, utilizando la fórmula de Rodrigues
R=[]
for i in range(0,6,1):
    wmat=Vectoso3(w[i])
    R.append(sp.eye(3)+sp.sin(t[i])*wmat+(1-sp.cos(t[i]))*(wmat*wmat))

# Aplicamos rotaciones a los vectores q y w para llevarlos a la configuración deseada
qs=[]; ws=[]; Ri=R[0]
qs.append(sp.Matrix(q[0]))
ws.append(sp.Matrix(w[0]))
for i in range(1,6,1):
    ws.append(Ri*sp.Matrix(w[i]))
    qs.append(Ri*sp.Matrix(q[i])+qs[i-1])
    Ri=Ri*R[i]

# Calculamos las velocidades lineales, los vectores giro correspondientes y la matriz Jacobiana
vs=[]; Ji=[]
i=0
vs.append(qs[i].cross(ws[i]))
Ji.append(ws[i].row_insert(3,vs[i]))
J=Ji[0]
for i in range(1,6,1):
    vs.append(qs[i].cross(ws[i]))
    Ji.append(ws[i].row_insert(3,vs[i]))
    J=J.col_insert(i,Ji[i])

```

El resultado de este código es una matriz Jacobiana parametrizada en función de los ángulos de las articulaciones. A continuación se puede ver cómo queda la Matriz Jacobiana de manera general, en función de todas las coordenadas de las articulaciones, con restricciones en algunas de ellas, y finalmente con valores constantes para las coordenadas de todas las articulaciones:

Resultado general parametrizado:

```
print(J)
```

Resultados con restricciones en algunos ángulos:

```
print(J.subs({t[1]:0, t[2]:0, t[3]:0, t[4]:0}))
print(J.subs({t[0]:0, t[2]:0, t[3]:0, t[4]:0}))
```

Resultados para varias configuraciones específicas:

```
Ja=J.subs({t[0]:0, t[1]:0, t[2]:0, t[3]:0, t[4]:0})
print(np.array(Ja).astype(np.float64).round(decimals=3))

Ja=J.subs({t[0]:np.pi, t[1]:np.pi, t[2]:np.pi, t[3]:np.pi, t[4]:np.pi})
print(np.array(Ja).astype(np.float64).round(decimals=3))

Ja=J.subs({t[0]:np.pi/2., t[1]:np.pi/2., t[2]:np.pi/2., t[3]:np.pi/2., t[4]:np.pi/2.})
print(np.array(Ja).astype(np.float64).round(decimals=3))
```

4. Configuraciones Singulares

En una configuración singular hay direcciones del movimiento que están restringidas. Cada columna de la matriz Jacobiana puede interpretarse como la velocidad del elemento terminal en una situación en la que la velocidad de la coordenada de la articulación correspondiente a esa columna, rota con una velocidad angular de 1 rad/s y el resto de las articulaciones están fijas. Desde esta definición puede deducirse que en una configuración singular 2 o más columnas de la matriz Jacobiana son linealmente dependientes, por lo que su determinante es nulo. Por lo tanto, para encontrar las configuraciones singulares del robot basta con encontrar los valores de θ_i que hacen nula la matriz Jacobiana. De nuevo podemos hacer uso del cálculo simbólico para encontrar estas configuraciones singulares. La resolución del determinante completo, sin ninguna restricción, lleva demasiado tiempo de cómputo. Podemos, sin embargo, imponer restricciones en varias configuraciones para hacer el cálculo más sencillo. Veamos algunos ejemplos:

```
sp.solve(J.subs({t[2]:0, t[3]:0, t[4]:0}).det())
Resultado: [{t1: -1.57079632679490}, {t1: 1.57079632679490}]

sp.solve(J.subs({t[1]:0, t[3]:0, t[4]:0}).det())
Resultado: [{t2: -1.70541733137745},
            {t2: -1.57079632679490},
            {t2: 1.43617532221234},
            {t2: 1.57079632679490}]
```

OJO!!! Estas configuraciones singulares, obtenidas matemáticamente, pueden no ser accesibles por el robot. Antes de mover el robot a alguna de estas configuraciones, verifica que esté dentro del rango alcanzable para las articulaciones correspondientes.

5. Elipsoides de Manipulabilidad y Fuerza

La matriz Jacobiana mapea las velocidades de las articulaciones en una configuración dada, a velocidades del elemento terminal. También sirve para mapear los torques o pares de fuerza en las articulaciones, a la fuerza neta que puede ejercer el elemento terminal en esa configuración.

$$\nu = J(\theta)\dot{\theta} \quad \mathcal{F} = J^{-T}(\theta)\tau$$

En clase vimos como calcular elipses (2D) de manipulabilidad y fuerza para un robot plano 2R, partiendo de una circunferencia de radio unidad en el espacio generado por las velocidades de las articulaciones ($\dot{\theta}$) o en el espacio de los torques (τ). En el caso del robot Niryo One tendríamos un elipsoide en 6 dimensiones en el espacio de velocidades o fuerzas a partir de hiperesferas en $\dot{\theta}$ o τ . No es posible representar gráficamente estas figuras 6D de manera sencilla. Podemos sin embargo escoger sólo 2 ó 3 articulaciones del robot, manteniendo las demás fijas, y calcular el elipsoide de manipulabilidad en esas condiciones. Para utilizar 3 dimensiones, debes partir de los puntos de la superficie de una esfera en el espacio de las velocidades de las articulaciones. recuerda que la ecuación de una esfera de radio unidad en este escenario vendría dada por:

$$\sum_i \dot{\theta}_i^2 = 1$$

por lo que puedes generar puntos en su superficie en coordenadas esféricas como se muestra a continuación:

```
r = 1
u = np.linspace(0, 2 * np.pi, 100)
v = np.linspace(0, np.pi, 100)
for i in range (0,100,1):
    for j in range (0,100,1):
        x = r * np.cos(u[i]) * np.sin(v[j])
        y = r * np.sin(u[i]) * np.sin(v[j])
        z = r * np.cos(v[j])
```

A partir de estos puntos podemos construir los vectores que contienen las 6 velocidades de las articulaciones ($\dot{\theta}$) para cada punto de la esfera:

```
vjoints=np.array([x,y,z,0,0,0])
```

A continuación, multiplicamos estos vectores por la matriz Jacobiana de la conformación que queremos estudiar. Por ejemplo, la matriz Jacobiana de la posición cero del robot se puede calcular introduciendo en el código de la sección 3 la siguiente línea:

```
Jp0=J.subs({t[0]:0, t[1]:0, t[2]:0, t[3]:0, t[4]:0})
```

que nos da como resultado:

```
Matrix([
[0, 0, 1, 0, 1],
[0, -1, -1, 0, -1, 0],
[1, 0, 0, 0, 0, 0],
[0, 0.183, 0.393, 0, 0.423, 0],
[0, 0, 0.423, 0, 0.4175],
[0, 0, 0, -0.2215, 0]])
```

Multiplicando esta matriz por los vectores $\dot{\theta}$ (los puntos de la esfera), obtenemos el vector giro, que podemos almacenar en un array:

```
giro=[]
giro.append(np.dot(Jp0,v))
```

Para cada punto de la esfera en el espacio de velocidades de las articulaciones, se obtiene un punto del elipsoide de manipulabilidad en el espacio de velocidades del elemento terminal. Tanto la esfera en el espacio de $\dot{\theta}$ como los vectores giro se pueden representar en 3D. Esto nos permite visualizar directamente el elipsoide.

Para el elipsoide de fuerza se sigue un proceso totalmente análogo, aunque en este caso se utiliza la inversa de la traspuesta de la matriz Jacobiana para transformar los torques de las articulaciones en la llave que ejerce el elemento terminal.

El caso bidimensional, en el que sólo consideramos 2 articulaciones, es mucho más sencillo y rápido, ya que sólo necesitamos los puntos de una circunferencia, que se transforman en una elipse a través de la matriz Jacobiana.

Para un Robot genérico de lazo abierto y un espacio de tareas con coordenadas $q \in \mathbb{R}^m$ donde $m \leq n$, el elipsoide de manipulabilidad representa las velocidades del elemento terminal para velocidades de articulaciones dadas por $\dot{\theta}$ donde $\|\dot{\theta}\| = 1 \Rightarrow$ una esfera unitaria en el espacio de las velocidades de las articulaciones $\Rightarrow \dot{\theta}^T \dot{\theta} = 1$. En este escenario, podemos definir una matriz $A = J J^T$ a partir de la cual es posible estimar el volumen del elipsoide de manipulabilidad (ver demostración en los apuntes de clase):

$$V \propto \sqrt{\det(J J^T)} \quad (1)$$

Análogamente podemos estimar el volumen del elipsoide de fuerza a partir de la inversa de la matriz A . Los ejes principales del elipsoide de fuerza están alineados con los del elipsoide de manipulabilidad (los autovectores de la matriz A coinciden con los de la matriz A^{-1}) pero los autovalores de A coinciden con los inversos de los autovalores de A^{-1} . Esto quiere decir que conforme el volumen del elipsoide de manipulabilidad aumenta, disminuye el volumen del elipsoide de fuerza y viceversa, de manera que el producto de ambos volúmenes es constante.

5.1. Evaluación del volumen del elipsoide de manipulabilidad, utilizando el código de la sección 3

Primero obtenemos la matriz Jacobiana. Tomando la configuración cero del robot:

```
Jp0=np.array(J.subs({t[0]:0, t[1]:0, t[2]:0, t[3]:0, t[4]:0}), dtype=np.float)
```

A partir de la Jacobiana podemos calcular los elipsoides de manipulabilidad y fuerza en 2 dimensiones, si restringimos las velocidades de las articulaciones a sólo 2 grados de libertad:

```
import numpy as np
import matplotlib.pyplot as plt

Jp0=np.array(J.subs({t[0]:0, t[1]:0, t[2]:0, t[3]:0, t[4]:0}), dtype=np.float)

# Generamos las coordenadas de una circunferencia
u = np.linspace(0, np.pi/2, 100)
```

```

x=[];y=[]
for i in range(0,100,1):
    x.append(np.cos(u[i]))
    y.append(np.sin(u[i]))
x=np.array(x)
xx=np.r_[np.r_[x,-x], np.r_[-x,x]]
y=np.array(y)
yy=np.r_[np.r_[y,y], np.r_[-y,-y]]

# Calculamos los vectores giro, multiplicando por la Matriz Jacobiana
giro=[]
for i in range(0,400,1):
    vjoints=np.array([0,xx[i],yy[i],0,0,0])
    giro.append(np.dot(Jp0,vjoints))
giro=np.array(giro)

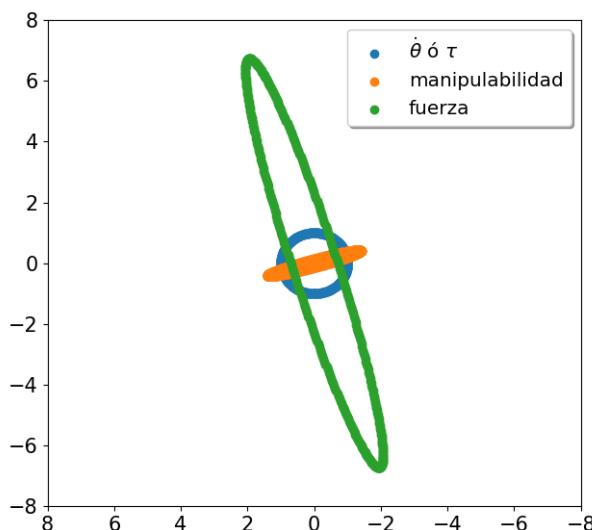
# Calculamos los vectores llave, multiplicando por la inversa de la traspuesta de J
llave=[]
Jp0_T=np.linalg.inv(np.transpose(Jp0))
for i in range(0,400,1):
    taujoints=np.array([0,xx[i],yy[i],0,0,0])
    llave.append(np.dot(Jp0_T,taujoints))
llave=np.array(llave)

# Extraemos las componentes no nulas del vector giro para representarlas
vx=giro[:,1]; vy=giro[:,3]
fx=llave[:,1]; fy=llave[:,3]

# Generamos gráfica con círculo y elipses
plt.scatter(xx,yy, label=r'$\dot{\theta}$ ó $\tau$')
plt.scatter(vx,vy, label="manipulabilidad")
plt.scatter(fx,fy, label="fuerza")
limitplot=8
plt.ylim(top = limitplot, bottom = -limitplot)
plt.xlim(left = limitplot, right = -limitplot)
plt.xticks(fontsize= 15)
plt.yticks(fontsize= 15)
plt.tight_layout()
plt.legend(loc='upper right', shadow=True, fontsize='x-large')
plt.show()

```

Como resultado de este código debes obtener la siguiente gráfica:



Puedes calcular el volumen de los elipsoides añadiendo las siguientes líneas al código que calcula las Matrices Jacobianas (sección 3):

```
# Elipsoide de manipulabilidad
(Jpi*sp.Transpose(Jpi)).det()

# Elipsoide de fuerza
((Jpi*sp.Transpose(Jpi)).inv()).det()
```

6. Cinemática Inversa

La resolución del problema cinemático inverso consiste en encontrar las coordenadas de las articulaciones que dan lugar a una posición y orientación específicas del elemento terminal. Al contrario que el caso del problema cinemático directo, la solución no es necesariamente única, es decir, que frecuentemente existen varias configuraciones del robot que dan lugar a la misma posición y orientación del elemento terminal.

El problema cinemático directo de un robot de n grados de libertad se expresa mediante el siguiente producto de matrices exponenciales:

$$T(\theta) = e^{[\mathcal{S}_1]\theta_1} e^{[\mathcal{S}_2]\theta_2} \dots e^{[\mathcal{S}_n]\theta_n} M$$

Si me dan la posición y orientación del elemento terminal como una matriz de transformación homogénea $X \in SE(3)$, el problema cinemático inverso consiste en encontrar las coordenadas de las articulaciones $\theta \in \mathbb{R}^n$ tal que $T(\theta) = X$.

De manera general, el problema cinemático inverso no tiene solución analítica, por lo que se resuelve numéricamente. La solución numérica también sirve para optimizar resultados analíticos, por ejemplo en los casos en los que el alineamiento entre ejes del brazo no son perfectos (la solución analítica asume idealidad). En estos casos la solución analítica puede usarse para inicializar el cálculo numérico.

También se utilizan métodos de optimización numéricos en los casos en los que no existe una solución exacta y buscamos una aproximación, o en los casos en los que hay infinitas soluciones (robots redundantes) y queremos una solución óptima con respecto a algún criterio como por ejemplo minimizar la potencia consumida o el torque máximo en las articulaciones.

Hay muchos algoritmos numéricos que se pueden implementar para resolver el problema cinemático inverso. Nosotros utilizaremos un algoritmo muy simple (método de Newton-Raphson) que en general converge muy rápidamente, aunque conviene sugerir una solución inicial próxima a la real. A continuación describimos el protocolo que implementamos en nuestro código:

- La configuración deseada para el elemento terminal de un Robot viene dada por una matriz de transformación homogénea T_{sd} .
- Planteamos una hipótesis inicial θ^0 para las coordenadas de las articulaciones, que esté razonablemente cerca de la solución deseada.
- Calculamos el vector *Giro* en el SR del elemento terminal ν_b , que indica la dirección de cambio para ir del conjunto de coordenadas θ^i a la solución deseada. En el SR situado en el elemento terminal: $[\nu_b] = \log(T_{sb}^{-1}(\theta^i)T_{sd})$. Si utilizamos el SR fijo: $[\nu_s] = [Ad_{T_{sb}}]\nu_b$
- Mientras los módulos de este vector no superen un umbral predeterminado $\Rightarrow \|\omega\| \geq \epsilon_\omega$ ó $\|v\| \geq \epsilon_v$, hacemos $\theta^{i+1} = \theta^i + J^\dagger(\theta^i)\nu$ e incrementamos el valor de i .

El cálculo también se puede realizar tanto en el SR fijo como en el SR situado en el elemento terminal. En el primer caso utilizamos $J_s(\theta)$ mientras que en el segundo caso utilizamos $J_b(\theta)$. Recuerda que los vectores giro se relacionan a través de la matriz adjunta: $\nu_s = [Ad_{T_{sb}}]\nu_b$.

La Matriz pseudoinversa J^\dagger es análoga a la matriz inversa pero es posible calcularla incluso para matrices no cuadradas y no invertibles. Se puede calcular en Python utilizando la función `numpy.linalg.pinv`.

- Si $n < m$, el algoritmo devuelve la solución más aproximada al objetivo.
- Si $n > m$ hay infinitas soluciones y el algoritmo devuelve la que da el cambio más pequeño en las coordenadas de las articulaciones.

7. Ejercicios Prácticos

7.1. Matriz Jacobiana

Asegúrate de entender el código que calcula la matriz Jacobiana del robot Niryo One. Introdúcelo en un archivo y prueba a imprimir su resultado en función de los valores de θ y también para varias configuraciones específicas. Mueve el robot a las mismas configuraciones para las cuales calculas la matriz Jacobiana y razona los resultados obtenidos.

7.2. Configuraciones singulares

Tomando como referencia los ejemplos de la sección 4, encuentra varias configuraciones singulares del robot Niryo One, mueve el robot a esas configuraciones y trata de entender porqué esas configuraciones son singulares. Utiliza para ello la resolución de la ecuación $\det(\text{Jacobiana}) = 0$ con algunas de las articulaciones fijas para encontrar la relación entre las coordenadas del resto de las articulaciones que nos llevan a configuraciones singulares.

7.3. Elipsoides de Manipulabilidad y Fuerza

Tomando como referencia el código de la subsección 5.1, desarrolla un código que calcule el elipsoide de manipulabilidad y el elipsoide de fuerza para una configuración aleatoria del robot y para una configuración singular. Limita el cálculo a 2 articulaciones para que puedas representar el elipsoide en un plano fácilmente. Mueve el robot entre las dos configuraciones y observa cómo varía el volumen del elipsoide de manipulabilidad al acercarnos a la configuración singular. Repite los cálculos para el elipsoide de fuerza y comprueba que el producto del volumen de ambos elipsoides es constante. Interpreta los resultados.

7.4. Cinemática Inversa

- En el apéndice tienes el código que resuelve el problema cinemático inverso para el Robot Niryo One. Comprueba que funciona bien, comparando diferentes configuraciones del robot con los resultados que obtienes del código. Corrige los ángulos que resultan para que siempre estén dentro del intervalo $[-\pi, \pi]$. Haz una tabla con posiciones y orientaciones teóricas y reales del robot y compara los resultados. Para los valores teóricos utiliza al menos 5 "semillas" diferentes y comprueba la reproducibilidad del resultado del código para diferentes valores iniciales de las coordenadas de las articulaciones.
- Los resultados del cálculo deben ser más precisos cuanto más cerca está la "semilla" de la solución final. Aprovecha esto para hacer que el robot dibuje una trayectoria punto-a-punto, por ejemplo, puedes hacer que el robot describa un círculo sobre la superficie en la que está apoyado, manteniendo la orientación vertical de la herramienta.
- Tomando como referencia los códigos disponibles de esta práctica y de la anterior, desarrolla un código que barra un rango amplio de posiciones y orientaciones predeterminadas teóricas (utilizando el método `move.joints`) lee el valor real correspondiente de las coordenadas y orientaciones (utilizando el método `get_arm_pose`) y salva los resultados en un archivo. Dentro de tu código debes validar que las coordenadas de las articulaciones a las que mueves el robot están dentro del rango aceptable.

8. Apéndice

8.1. Código para resolución del problema cinemático inverso con Niryo One

```

#!/usr/bin/env python
#
# By Angel.Pineiro at usc.es
# Version April, 2021
#
# EJEMPLO DE USO:
# python CinematicaInversaNiryo_03.py -a '0 0 90' -r '0.2 0 0.1' -j '0 -0.8 -0.8 0 -1.5 0'
#
#
import numpy as np
import sympy as sp
import optparse

desc="Resolución del problema cinemático inverso para el Robot Niryo One."

def main():
    parser = optparse.OptionParser(description=desc, version='%prog version 1.0')
    parser.add_option('-j', '--seed_joints', help='Coordenadas iniciales de las articulaciones en rad', action='store')
    parser.add_option('-r', '--xyz', help='Coordenadas xyz finales para el elemento terminal', action='store')
    parser.add_option('-a', '--ang', help='Ángulos de Euler finales para el elemento terminal en grados', action='store')
    parser.add_option('-o', '--eomg', help='Error en la orientación del elemento terminal (0.01 por defecto)', action='store')
    parser.add_option('-e', '--er', help='Error en la posición del elemento terminal (0.001 por defecto)', action='store')
    parser.set_defaults(seed_joints='0 0 0 0 0', xyz='0.1 0.1 0.1', ang='0 0 0', eomg='0.01', ev=0.001)
    options, arguments = parser.parse_args()
    ****
    seed_joints=str(options.seed_joints).split()
    t=[]
    for i in range (0,6,1): t.append(np.float(seed_joints[i]))

    xyz=str(options.xyz).split()
    r=[]
    for i in range (0,3,1): r.append(np.float(xyz[i]))

    ang=str(options.ang).split()
    orientation=[]
    for i in range (0,3,1): orientation.append(np.deg2rad(np.float(ang[i])))

    eomg=float(options.eomg)
    ev=float(options.ev)

    scalefactor=0.001
    L=np.array([103.0, 80.0, 210.0, 30.0, 41.5, 180.0, 23.7, -5.5])*scalefactor

    # Calculamos la Matriz de Transformación Homogénea a partir de posiciones y ángulos
    T=getT(orientation, r)
    #print(np.round(T,2))
    # Calculamos los ejes helicoidales en la posición cero del robot
    S=getS()
    #print(S)
    # Matriz de transformación homogénea en la posición cero del robot
    M=np.array([[1,0,0,L[6]+L[5]+L[4]],[0,1,0,0],[0,0,1,L[0]+L[1]+L[2]+L[3]+L[7]],[0,0,0,1]])

    thetalist = np.array(t).copy()
    i = 0
    maxiterations = 20
    Tsb = CinematicaDirecta(M,S, thetalist) # Resuelve la Cinemática Directa para thetalist
    Vb = MatrixLog6(np.dot(np.linalg.inv(Tsb), T)) # vector Giro para ir a la posición deseada en {b}
    Vs = np.dot(Adjunta(Tsb), se3ToVec(Vb)) # vector Giro en el SR de la base {s}

    # condición de convergencia: módulo de velocidad angular < eomg y velocidad lineal < ev
    err = np.linalg.norm([Vs[0], Vs[1], Vs[2]]) > eomg or np.linalg.norm([Vs[3], Vs[4], Vs[5]]) > ev

    while err and i < maxiterations:
        J=Jacobiana(thetalist);
        J=np.array(J.tolist()).astype(np.float64)
        thetalist = thetalist + np.dot(np.linalg.pinv(J), Vs)
        i = i + 1
        Tsb = CinematicaDirecta(M, S, thetalist)
        Vb = MatrixLog6(np.dot(np.linalg.inv(Tsb), T))

```

```

Vs = np.dot(Adjunta(Tsb), se3ToVec(Vb)); print (Vs)
err = np.linalg.norm([Vs[0], Vs[1], Vs[2]]) > eomg or np.linalg.norm([Vs[3], Vs[4], Vs[5]]) > ev

print ("\n\nCoordenadas de las articulaciones:\n", thetalist)
print ("Error en w:", np.round(np.linalg.norm([Vs[0], Vs[1], Vs[2]]),8))
print ("Error en v:", np.round(np.linalg.norm([Vs[3], Vs[4], Vs[5]]),8))
print ("Número de iteraciones:", i)

if __name__=="__main__":
    main()

```

8.2. Funciones utilizadas en el código que resuelve el problema cinemático inverso del Niryo One

```

def VecToso3(w): # convierte un eje de rotación en una matriz antisimétrica 3x3
    return np.array([[0,-w[2],w[1]], [w[2],0,-w[0]], [-w[1],w[0],0]])

def Vectose3(V): # convierte un vector giro o eje helicoidal en matriz 4x4 se3
    return np.r_[np.c_[VecToso3([V[0], V[1], V[2]]), [V[3], V[4], V[5]]], np.zeros((1, 4))]

def so3ToVec(so3mat): # extrae un vector de 3 componentes de una matriz antisimétrica so3
    return np.array([so3mat[2][1], so3mat[0][2], so3mat[1][0]])

def se3ToVec(se3mat): # Convierte una matriz se3 en un vector giro 1x6
    return np.r_[[se3mat[2][1], se3mat[0][2], se3mat[1][0]],
                [se3mat[0][3], se3mat[1][3], se3mat[2][3]]]

def getR(w,ang): # Formula de Rodrigues para obtener matriz de Rotación
    wmat=VecToso3(w)
    return np.eye(3)+np.sin(ang)*wmat+(1.-np.cos(ang))*np.dot(wmat,wmat)

def getejes(): # Definimos ejes de rotación del Robot Niryo One
    w=[]
    w.append(np.array([0,0,1]))
    w.append(np.array([0,-1,0]))
    w.append(np.array([0,-1,0]))
    w.append(np.array([1,0,0]))
    w.append(np.array([0,-1,0]))
    w.append(np.array([1,0,0]))
    return w

def getqs(): # Definimos vectores q para la posición cero del Robot Niryo One
    # vectores que van de cada eje al siguiente
    q=[]; scalefactor=0.001
    L=np.array([103.0, 80.0, 210.0, 30.0, 41.5, 180.0, 23.7, -5.5])*scalefactor
    q.append(np.array([0,0,L[0]]))
    q.append(np.array([0,0,L[1]]))
    q.append(np.array([0,0,L[2]]))
    q.append(np.array([L[4],0,L[3]]))
    q.append(np.array([L[5],0,0]))
    q.append(np.array([L[6],0,L[7]]))
    return q

def getT(orientation,r): # Devuelve la matriz de transformación homogénea
    i=np.array([1,0,0]); j=np.array([0,1,0]); k=np.array([0,0,1]);
    Ri=getR(i,orientation[0]); Rj=getR(j,orientation[1]); Rk=getR(k,orientation[2]);
    R=np.matmul(Rk,np.matmul(Rj,Ri))
    aux=np.array([[0,0,0,1]])
    return np.r_[np.c_[R,r],aux]

def getS(): # Devuelve los ejes helicoidales del robot Niryo One en la posición cero
    w=getejes() # Definimos ejes de rotación
    q=getqs() # Definimos vectores q

    # Definimos los eslabones
    scalefactor=0.001 # por si quiero los resultados en otras unidades
    L=np.array([103.0, 80.0, 210.0, 30.0, 41.5, 180.0, 23.7, -5.5])*scalefactor

    # Calculamos los ejes de giro y vectores posición en la configuración final del robot
    qs=[]; ws=[]
    qs.append(np.array(q[0]))
    ws.append(np.array(w[0]))
    for i in range(1,6,1):
        ws.append(np.array(w[i]))
        qs.append(np.array(q[i])+qs[i-1])

```

```

# Calculamos las velocidades lineales para construir los ejes helicoidales
vs=[]; Si=[]
for i in range(0,6,1):
    vs.append(np.cross(qs[i],ws[i]))
    Si.append(np.r_[ws[i],vs[i]])
return Si

def MatrixExp6(se3mat): # convierte un vector giro en forma matricial 4x4 se3 en una MTH a través de la exponencial
    se3mat = np.array(se3mat) # vector giro en representación matricial se3 (4x4)
    v=se3mat[0: 3, 3] # extraemos el vector v*theta (velocidad lineal)
    omgmattheta=se3mat[0: 3, 0: 3] # extraemos omega*theta en forma matricial 3x3 (so3)
    omgtheta = so3ToVec(omgmattheta) # lo pasamos a forma vectorial

    if (np.linalg.norm(omgtheta))<1.e-6: # en el caso de que no haya giro (omega despreciable)
        return np.r_[np.c_[np.eye(3), v], [[0, 0, 0, 1]]] # concatena columnas y filas. Sólo traslación

    else: # caso general
        theta = np.linalg.norm(omgtheta)
        omgmat = omgmattheta / theta # omega en forma matricial 3x3 (so3) Normalizada
        # a continuación aplicamos la definición de matriz exponencial que vimos en clase (slide 42, tema 2)
        G_theta=np.eye(3)*theta+(1-np.cos(theta))*omgmat+(theta-np.sin(theta))*np.dot(omgmat,omgmat)
        R=np.eye(3)+np.sin(theta)*omgmat+(1.-np.cos(theta))*np.dot(omgmat,omgmat)
        return np.r_[np.c_[R,np.dot(G_theta,v)/theta],[[0, 0, 0, 1]]]

def MatrixLog3(R): # Calcula la matriz logaritmo de una matriz de rotación
    acosinput = (np.trace(R) - 1) *0.5
    if np.trace(R) >= 3: return np.zeros((3, 3))
    elif np.trace(R) <= -1:
        if abs(1 + R[2][2])>1.e-6: omg = (1.0 / np.sqrt(2 * (1 + R[2][2]))) * np.array([R[0][2], R[1][2], 1 + R[2][2]])
        elif abs(1 + R[1][1])>1.e-6: omg = (1.0 / np.sqrt(2 * (1 + R[1][1]))) * np.array([R[0][1], 1 + R[1][1], R[2][1]])
        else: omg = (1.0 / np.sqrt(2 * (1 + R[0][0]))) * np.array([1 + R[0][0], R[1][0], R[2][0]])
        return VecToso3(np.pi * omg)
    else:
        theta = np.arccos(acosinput)
        return (theta*0.5)/np.sin(theta) * (R-np.array(R).T)

def MatrixLog6(T): # Calcula la matriz logaritmo de una MTH
    R=T[0: 3, 0: 3]; p = T[0: 3, 3] # separa la MTH en matriz de rotación y vector traslación
    omgmat = MatrixLog3(R) # coordenadas exponenciales de la matriz de rotación
    # o sea, un vector de rotación como matriz antisimétrica so3 (3x3)
    if np.array_equal(omgmat, np.zeros((3, 3))): # Si no hay rotación, es una matriz de ceros
        return np.r_[np.c_[np.zeros((3, 3)),p],[[0, 0, 0, 0]]]
    else:
        omgvec= so3ToVec(omgmat) # expresa la rotación como un vector en la dirección del eje por el ángulo
        omgmat=omgmat/np.linalg.norm(omgvec) # el vector en el eje de rotación normalizado y en forma matricial
        theta = np.linalg.norm(omgvec) # también se puede calcular como np.arccos((np.trace(R)-1)/2.0)
        # a continuación aplicamos la definición que vimos en clase (ver diapositivas)
        invG_theta=np.eye(3)/theta-omgmat*0.5+(1.0/theta-0.5/np.tan(theta*0.5))*np.dot(omgmat,omgmat)
        v=np.dot(invG_theta,p)
        return np.r_[np.c_[omgmat,v],[[0, 0, 0, 0]]]*theta # primero concatena columnas y luego filas

def Adjunta(T): # Calcula la matriz adjunta de una MTH
    R=T[0: 3, 0: 3]; p = T[0: 3, 3]
    return np.r_[np.c_[R, np.zeros((3, 3))], np.c_[np.dot(VecToso3(p), R), R]]

def CinematicaDirecta(M,S,t):
    T=np.eye(4);
    for i in range(0,6,1): T=np.dot(T,MatrixExp6(VecToso3(S[i]*t[i])))
    return np.dot(T,M)

def Jacobiana(theta):
    w=getejes() # Definimos ejes de rotación
    q=getqs() # Definimos vectores q
    # Definimos los eslabones
    scalefactor=0.001 # por si quiero los resultados en otras unidades
    L=np.array([103.0, 80.0, 210.0, 30.0, 41.5, 180.0, 23.7, -5.5])*scalefactor

    t=sp.symbols('t0, t1, t2, t3, t4, t5') #Coordenadas de las articulaciones

    # Calculamos las matrices de rotación a partir de los ejes w, utilizando la fórmula de Rodrigues
    R=[]
    for i in range(0,6,1):
        wmat=sp.Matrix(VecToso3(w[i]))
        R.append(sp.eye(3)+sp.sin(t[i])*wmat+(1-sp.cos(t[i]))*(wmat*wmat))

    # Aplicamos rotaciones a los vectores q y w para llevarlos a la configuración del robot que queremos

```

```
qs=[]; ws=[]; Ri=R[0]
qs.append(sp.Matrix(q[0]))
ws.append(sp.Matrix(w[0]))
for i in range(1,6,1):
    ws.append(Ri*sp.Matrix(w[i]))
    qs.append(Ri*sp.Matrix(q[i])+qs[i-1])
    Ri=Ri*R[i]

# Calculamos las velocidades lineales, los vectores giro correspondientes y la matriz Jacobiana
vs=[]; Ji=[]; i=0
vs.append(qs[i].cross(ws[i]))
Ji.append(ws[i].row_insert(3,vs[i]))
J=Ji[0]
J=J.col_insert(i,Ji[i])
return np.array(J.subs({t[0]:theta[0], t[1]:theta[1], t[2]:theta[2], t[3]:theta[3], t[4]:theta[4]}))
```