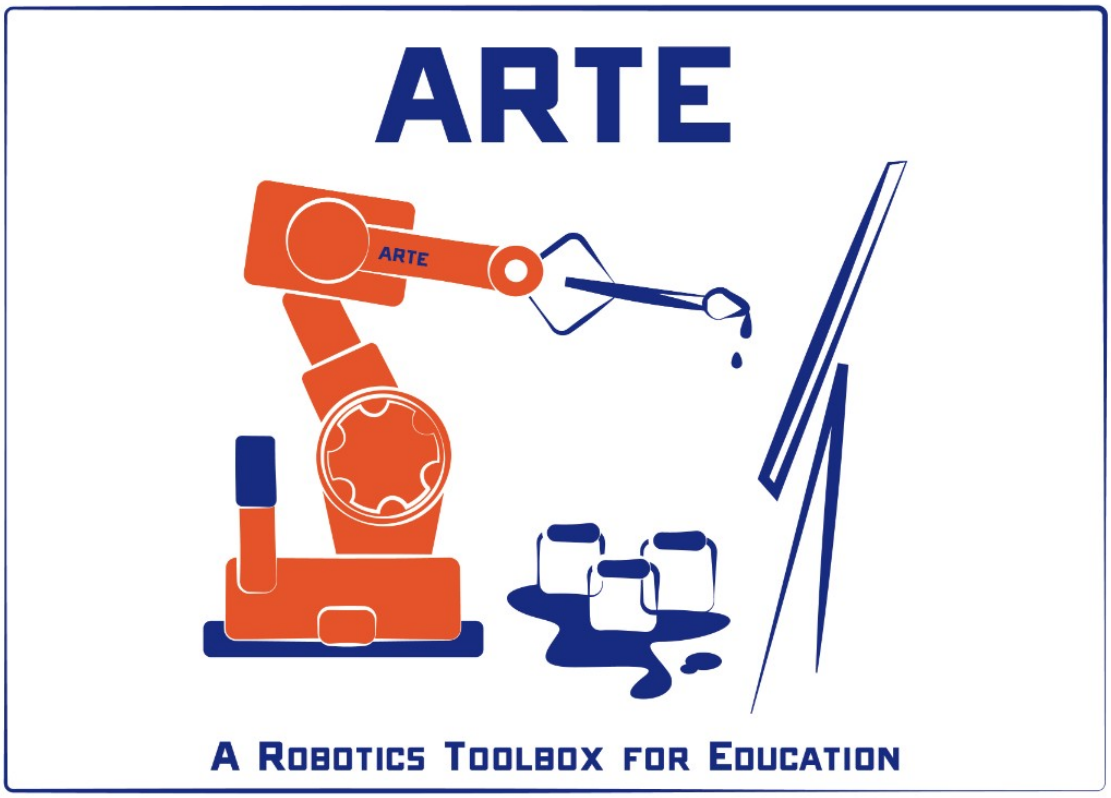
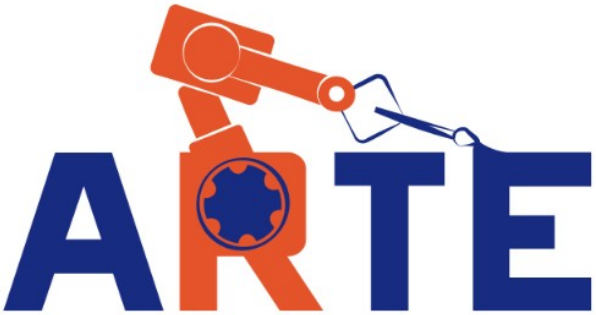


Sesión 2 B: La Jacobiana del manipulador. Conceptos avanzados y aplicación a la cinemática inversa.

Arturo Gil Aparicio

arturo.gil@umh.es



OBJETIVOS

Objetivos generales: Después de esta sesión práctica, el estudiante debería ser capaz de:

- Resolver las ecuaciones de la cinemática inversa de cualquier manipulador de tipo serie incluyendo mecanismos redundantes y muñecas no esféricas.
- Ser capaz de escribir estas ecuaciones en el entorno Matlab para un robot de su elección.

Índice:

1 Primeros pasos

Añada los parámetros de DH del robot en `arte/practicals/UR10/parameters.m`:

```
robot.name= 'UR10';
robot.DH.theta= '[q(1) q(2)+pi/2 q(3) q(4)-pi/2 q(5) q(6)]';
robot.DH.d='[0.128 0.176 -0.128 0.116 0.116 0.092]';
robot.DH.a='[0 0.612 0.572 0 0 0]';
robot.DH.alpha= '[pi/2 0 0 -pi/2 pi/2 0]';
robot.J=[];
```

A continuación cargue el robot y dibújelo:

```
>> pwd
ans =
    '/Users/arturogilaparicio/Desktop/arte'
>> init_lib
>> robot=load_robot('practicals','URtest')
>> T=directkinematic(robot, [0 0 0 0 0 0])
>> drawrobot3d(robot, [0 0 0 0 0 0])
```

El resultado se observa en las figuras siguientes:

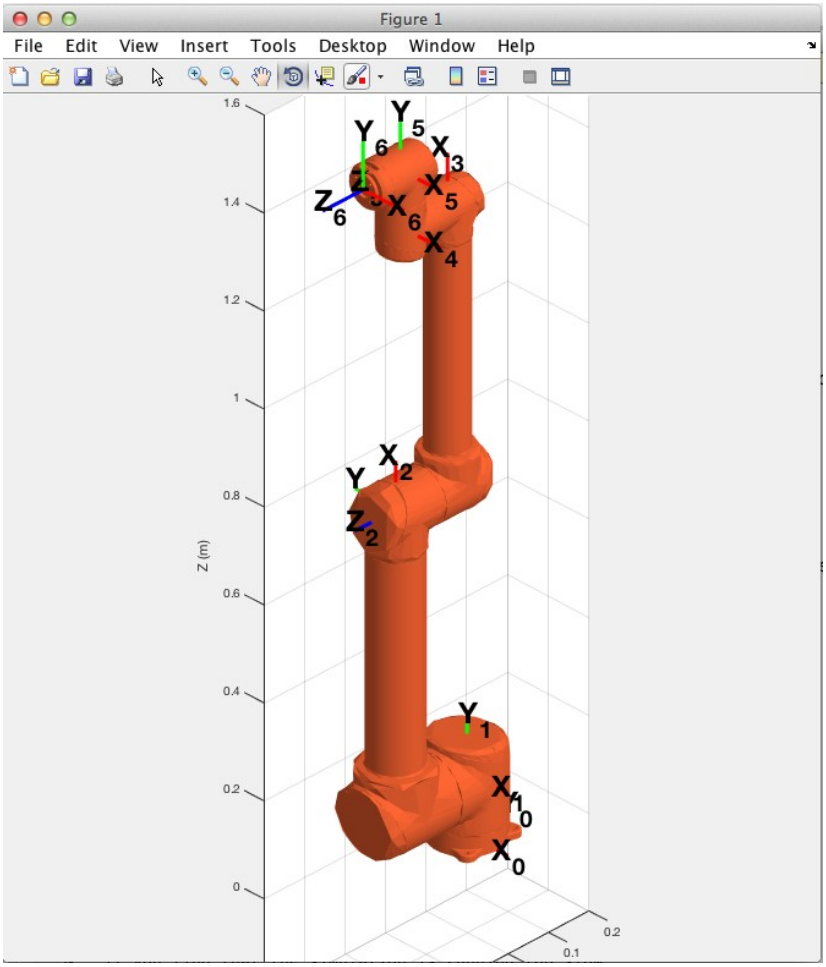
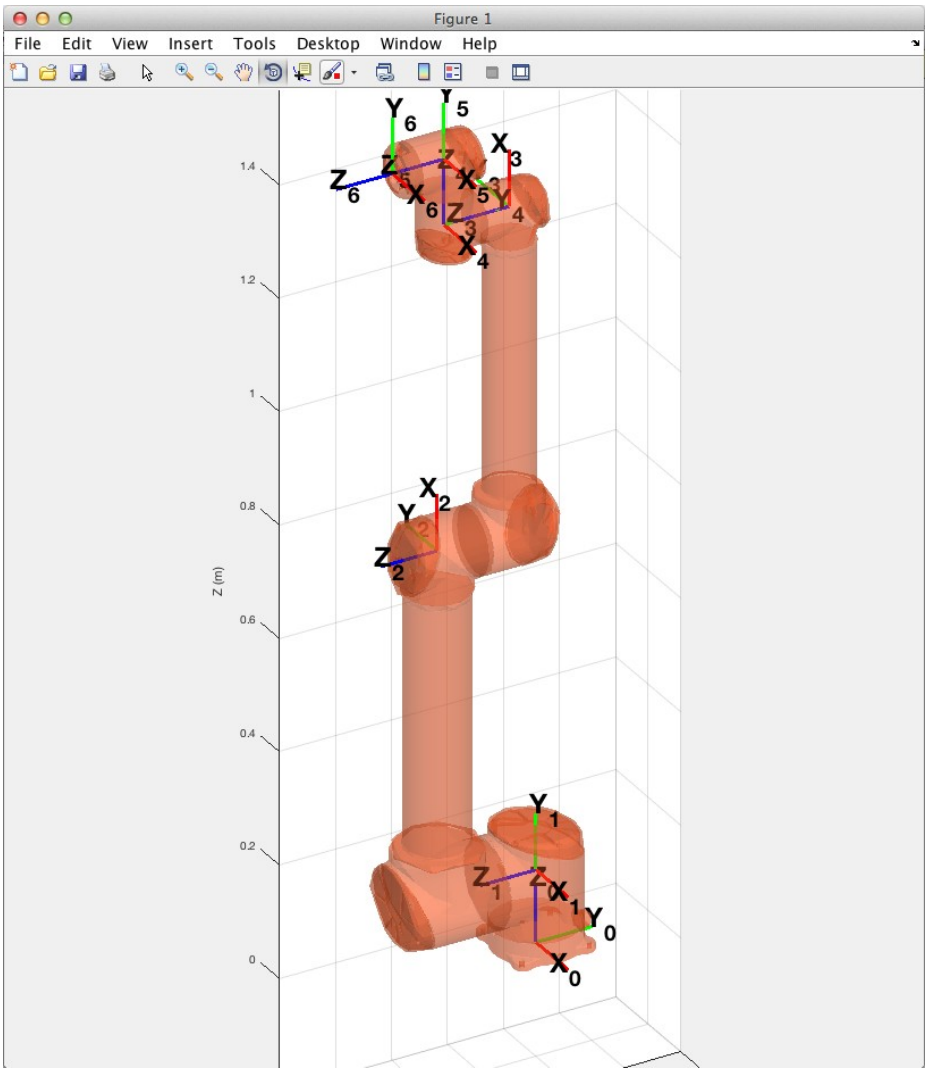
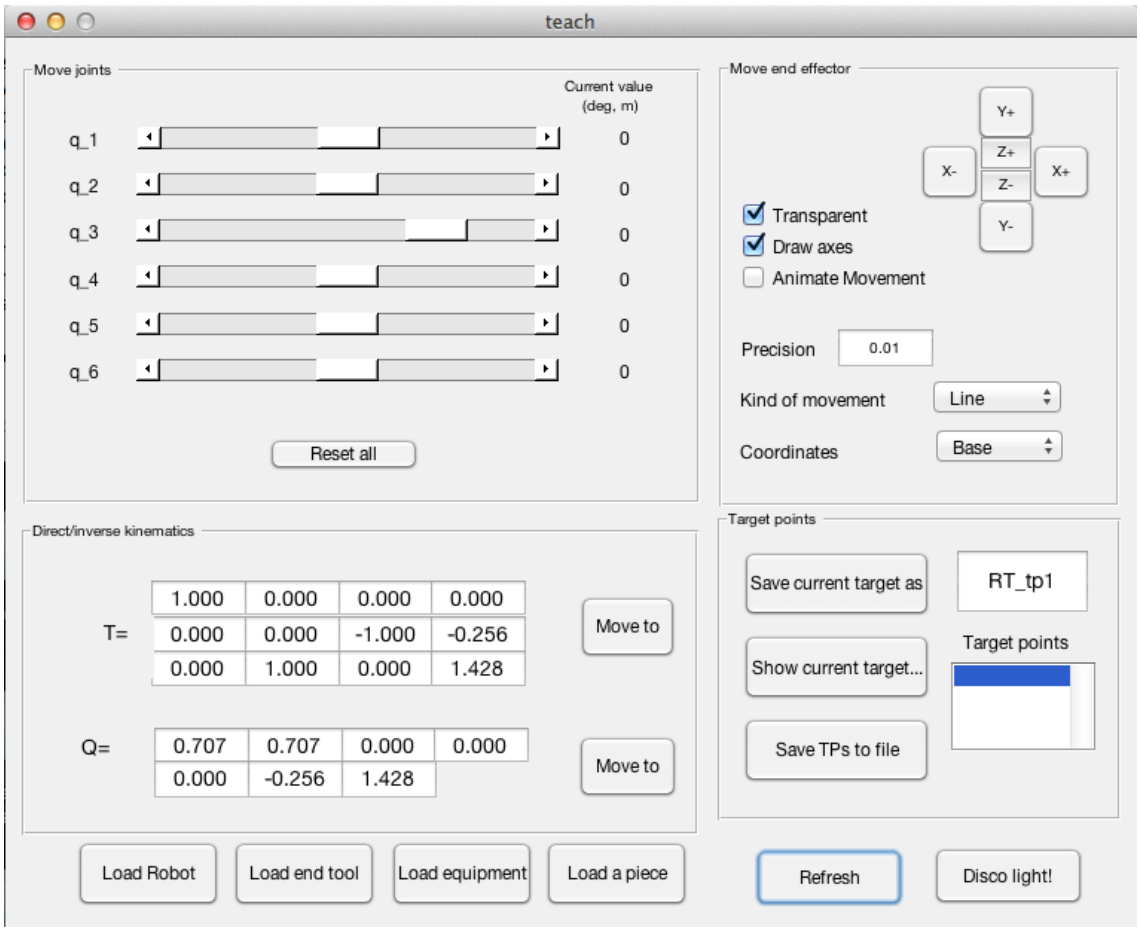


Figure 1
File Edit View Insert Tools Desktop Window Help
Z (m)
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0.2
0.1
Y 5
Y 6
X 3
X 5
Z 6
X 6
X 4
X 2
Y
Z 2
Y 1
X 1
0
X 0



Puede utilizar el comando teach para mover el robot utilizando los sliders:

```
>> teach
```



2 Cinemática inversa de un robot con muñeca no esférica

Para el robot mencionado no existe una solución cerrada para la cinemática inversa. En estos casos podemos considerar lo siguiente:

p_i: posición actual (p_x, p_y, p_z) del extremo del robot.

p_f: posición final (p_x, p_y, p_z) del extremo del robot. Valor deseado de la solución.

Q_i: orientación actual (Q_0, Q_1, Q_2, Q_3) dada por un cuaternio.

Q_f: orientación final (Q_0, Q_1, Q_2, Q_3) del extremo del robot. Valor deseado final de la orientación.

Considere que, en cualquier instante de tiempo existe una función que permite calcular la velocidad lineal y angular del extremo que permite llegar a la solución deseada en posición y orientación.

$$^0\vec{v} = f(p_f, p_i)$$
$$^0\vec{\omega} = f(Q_f, Q_i)$$
$$\hat{V} = [^0\vec{v}\ ^0\vec{\omega}]^T$$

En base a esta velocidad del extremo nos planteamos diferentes formas para resolver la cinemática inversa de forma numérica. Plantearemos dos algoritmos diferentes para resolver este problema:

a) Basado en la Jacobiana inversa.

$$\dot{q} = J^{-1} \cdot \hat{V}$$
$$q = q + \Delta t \cdot \dot{q}$$

b) Basado en la Jacobiana transpuesta.

$$\dot{q} = J^T \cdot \hat{V}$$
$$q = q + \Delta t \cdot \dot{q}$$

TAREAS

1 SOLUCIÓN UTILIZANDO LA JACOBIANA INVERSA

1.1) Modifique el fichero

`/arte/robots/practicals/UR10/`
`inverse_kinematics_ur10_practical.m`

1.2) Añada las líneas necesarias para programar un algoritmo de cinemática inversa basado en la Jacobiana INVERSA. Para probar el algoritmo, ejecute:

`/arte/robots/practicals/UR10/test_kinematics_ur10.m`

1.3) Comience pensando una velocidad lineal que lleva al robot hacia la posición deseada.

J = manipulator_jacobian(robot, J);

V = [v0 0 0 0]';

qd = inv(J)*V;

1.4) Añada, ahora, una velocidad w0 para acomodar también la rotación del extremo.

1.5) Enumere, a continuación los problemas que se encuentra sobre el algoritmo anterior. ¿Qué ocurre si la semilla inicial `q0 = [0 0 0 0 0 0]?. Plantee una solución al problema presentado.`

`Problema 1:`

`Problema 2:`

1.6) Normalización de las velocidades.

Debe Ud. pensar en una forma de normalizar las velocidades articulares obtenidas. Una forma estándar es la siguiente.

A = norm(qd);

if A > 0

    qd = qd/A;

else

    display('CAUTION: we are at a singular point')

end

Fíjese y tenga en cuenta esta situación en el algoritmo. Piense en **cómo huir de esta situación** en la que el robot entra en un punto singular.

1.7) Pruebe a cambiar la ley de actualización para que el algoritmo sea más rápido.

```
q = q + k*qd*step_time;
```

Importante: La constante (k=100) debería ajustarse progresivamente según avanza el algoritmo. Observe el resultado obtenido: en ocasiones el robot oscila alrededor de la solución final. ¿Es este el caso que observa?. Finalmente, combiene decir que step_time tiene el sentido físico del tiempo entre dos muestras. Realmente, se puede considerar esta cantidad dentro de la constante k y hacer.

```
q = q + k*qd;
```

Plantee una forma de ajustar k según el robot avanza hacia la solución final.

1.8) Podemos seguir trabajando y probar una solución algo más refinada.

```
q = q + k*K*qd*step_time;
```

En esta fórmula, estamos ensayando:

```
qd unitario tal que norm(qd)=1
```

```
k: una constante que regula la velocidad de avance hacia la solución final.
```

```
K: es una matriz diagonal que pondera la importancia de alguna de las articulaciones sobre el resto.
```

```
Por ejemplo, podemos tener
```

K = [2 0 0 0 0 0;

0 2 0 0 0 0;

0 0 2 0 0 0;

0 0 0 1 0 0;

0 0 0 0 1 0;

0 0 0 0 0 1];

De manera que se ponderan más las primeras articulaciones. Esto deja entrever que el algoritmo de esta cinemática inversa podría asemejarse a un algoritmo de tipo Levenberg-Marquardt. Comprobad, si, con la matriz K, la convergencia del algoritmo mejora o empeora.

1.9) ¡No vayamos a ciegas!: calcule el error en cada iteración. Al finalizar el algoritmo, compruebe la gráfica de error y observer cómo converge (o diverge) el algoritmo.

1.10) Varíe la línea 44 del algoritmo de prueba para ir a T1, T2 o T3. Proponga otros puntos y orientaciones a los que llegar con el algoritmo.

```
test_kinematics_ur10.m
```

1.11) Enumere los puntos a mejorar de este algoritmo. El algoritmo propuesto permite obtener una única solución. **Proponga cómo obtener más soluciones a la cinemática inversa.**

Pruebe con diferentes semillas iniciales.

1.12) Haga un algoritmo que pruebe el algoritmo que acabamos de realizar.

- Genere una posición q aleatoria.
- Calcule la cinemática directa T.
- Intente llegar a T desde otra posición semilla aleatoria.

- Guarde el error obtenido en la cinemática inversa y el número de iteraciones utilizado.
- ¿Qué conclusiones puede obtener?

2 SOLUCIÓN UTILIZANDO LA JACOBIANA TRASPUESTA

2.1) Modifique el fichero

`/arte/robots/practicals/UR10/inverse_kinematics_ur10_practical.m`

2.2) Añada las líneas necesarias para programar un algoritmo de cinemática inversa basado en la Jacobiana TRANSPUESTA. Para probar el algoritmo, ejecute:

`/arte/robots/practicals/UR10/test_kinematics_ur10.m`

2.3) Enumere, a continuación los problemas que se encuentra sobre el algoritmo anterior. ¿Qué ocurre si la semilla inicial `q0 = [0 0 0 0 0 0]`?

2.4) Enumere, a continuación los problemas que se encuentra sobre el algoritmo anterior. ¿Qué ocurre si la semilla inicial `q0 = [0 0 0 0 0 0]`?. Plantee una solución al problema presentado.

`Problema 1:`

`Problema 2:`

2.5) Normalización de las velocidades.

Debe Ud. pensar en una forma de normalizar las velocidades articulares obtenidas. Una forma estándar es la siguiente.

```
A = norm(qd);
if A > 0
    qd = qd/A;
else
    display('CAUTION: we are at a singular point')
end
```

Fíjese y tenga en cuenta esta situación en el algoritmo. Piense en **cómo huir de esta situación** en la que el robot entra en un punto singular.

2.6) Pruebe a cambiar la ley de actualización para que el algoritmo sea más rápido.

```
q = q + k*qd*step_time;
```

Importante: La constante (k=100) debería ajustarse progresivamente según avanza el algoritmo. Observe el resultado obtenido: en ocasiones el robot oscila alrededor de la solución final. ¿Es este el caso que observa?. Finalmente, combiene decir que step_time tiene el sentido físico del tiempo entre dos muestras. Realmente, se puede considerar esta cantidad dentro de la constante k y hacer.

```
q = q + k*qd;
```

Plantee una forma de ajustar k según el robot avanza hacia la solución final.

2.7) Podemos seguir trabajando y probar una solución algo más refinada.

```
q = q + k*K*qd*step_time;
```

En esta fórmula, estamos ensayando:

```
qd unitario tal que norm(qd)=1

k: una constante que regula la velocidad de avance
hacia la solución final.

K: es una matriz diagonal que pondera la importancia
de alguna de las articulaciones sobre el resto.

Por ejemplo, podemos tener
```

K = [2 0 0 0 0 0;

0 2 0 0 0 0;

0 0 2 0 0 0;

0 0 0 1 0 0;

0 0 0 0 1 0;

0 0 0 0 0 1];

De manera que se ponderan más las primeras articulaciones. Esto deja entrever que el algoritmo de esta cinemática inversa podría asemejarse a un algoritmo de tipo Levenberg-Marquardt. Comprobad, si, con la matriz K, la convergencia del algoritmo mejora o empeora.

2.8) ¡No vayamos a ciegas!: calcule el error en cada iteración. Al finalizar el algoritmo, compruebe la gráfica de error y observer cómo converge (o diverge) el algoritmo.

2.9) Varíe la línea 44 del algoritmo de prueba para ir a T1, T2 o T3. Proponga otros puntos y orientaciones a los que llegar con el algoritmo.

`test_kinematics_ur10.m`

2.10) Enumere los puntos a mejorar de este algoritmo. El algoritmo propuesto permite obtener una única solución. **Proponga cómo obtener más soluciones a la cinemática inversa.**

Pruebe con diferentes semillas iniciales.

2.11) Haga un algoritmo que pruebe el algoritmo que acabamos de realizar.

- Genere una posición q aleatoria.
- Calcule la cinemática directa T.
- Intente llegar a T desde otra posición semilla aleatoria.
- Guarde el error obtenido en la cinemática inversa y el número de iteraciones utilizado.
- ¿Qué conclusiones puede obtener?

3 COMPARE AMBAS SOLUCIONES

Compare la solución de cinemática inversa de los dos algoritmos para la misma matriz T (T1, T2 o T3). ¿Es la misma solución? Para verlo, rellene la siguiente tabla:

Algoritmo	**T1**	**T2**	**T4**	**T3**
Jacobiana inversa	**qinv=** **n_iterations =**	**qinv=** **n_iterations =**	**qinv =** **n_iterations =**	**qinv=** **n_iterations =**
Jacobiana transpuesta	**qinv=** **n_iterations =**	**qinv=** **n_iterations =**	**qinv =** **n_iterations =**	**qinv=** **n_iterations =**

Observe que es muy complejo asegurar la convergencia del algoritmo y, al mismo tiempo, tener un número de iteraciones bajo.

Plantee una forma de combinar ambos algoritmos para obtener una solución con independencia de puntos singulares.

AYUDA:

La Jacobiana de cualquier manipulador se puede obtener usando la función manipulator_jacobian sobre la posición articular actual.

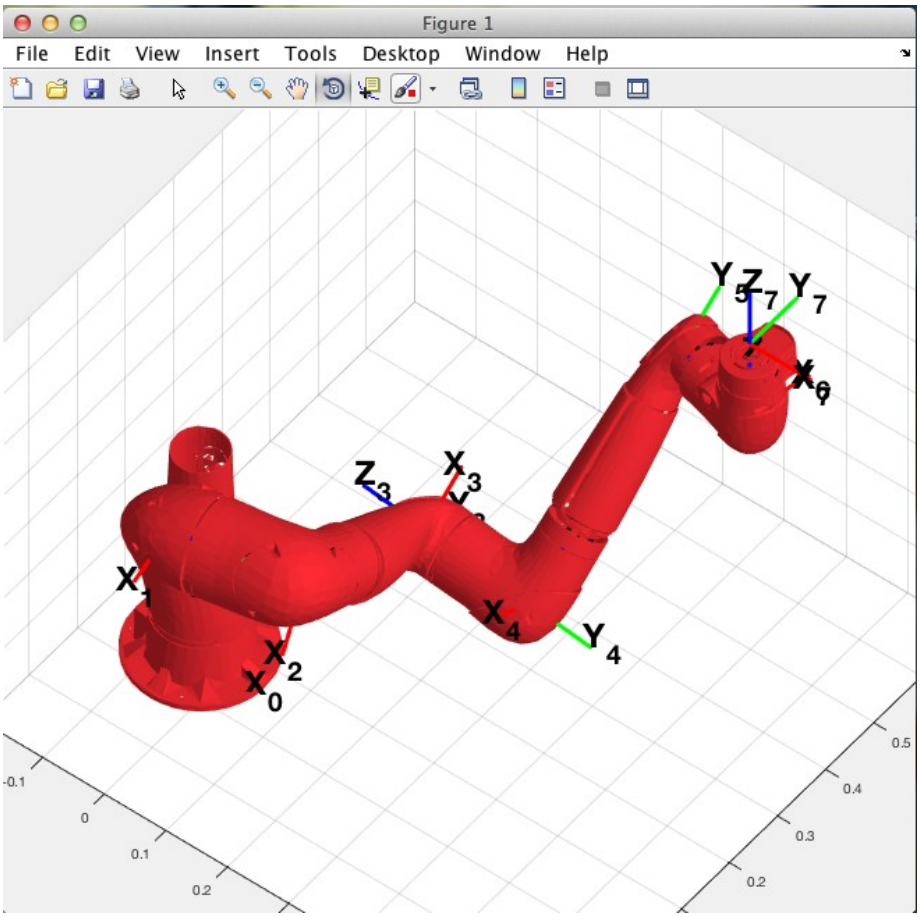
```
J = manipulator_jacobian(robot, q);
```

Es necesario prestar atención a la escala de cada iteración. Así pues, dada la naturaleza no lineal de la cinemática de un robot, actualizaciones muy grandes de qd darán como resultado fallos en la convergencia. Una propuesta para normalizar cada iteración es la siguiente

```
qd = qd/norm(qd);
qd = qd*norm(Vref);
```

3 Cinemática inversa de un robot redundante

Se propone ahora extender el algoritmo para un robot redundante de 7GDL. El robot Sawyer de Rethink Robotics. Se proponen tres algoritmos.



a) Basado en la Jacobiana pseudo Inversa Moore-Penrose.

$$\dot{q} = J^{\dagger} \cdot \hat{V}$$
$$q = q + \Delta t \cdot \dot{q}$$

b) Basado en la Jacobiana pseudo Inversa Moore-Penrose atenuada (damped).

$$\dot{q} = (J + \lambda I)^{\dagger} \cdot \hat{V}$$
$$q = q + \Delta t \cdot \dot{q}$$

b) Basado en la Jacobiana transpuesta.

$$\dot{q} = J^{T} \cdot \hat{V}$$
$$q = q + \Delta t \cdot \dot{q}$$

Tenga en cuenta todas las consideraciones que se apuntaron en el apartado anterior de cinemática inversa de un robot de 6 GDL y muñeca no esférica. Ahora, extienda la solución (y hágala más general) para atacar el problema de un robot redundante de 7GDL.

TAREAS

1 SOLUCIÓN UTILIZANDO LA JACOBIANA PSEUDO-INVERSA Y DAMPED

a) Modifique el fichero

```
/arte/robots/practicals/SAWYER/
inverse_kinematics_sawyer_practical.m
```

b) Añada las líneas necesarias para programar un algoritmo de cinemática inversa basado en la Jacobiana PSEUDO-INVERSA. Para probar el algoritmo, ejecute:

```
/arte/robots/practicals/SAWYER/
test_kinematics_sawyer.m
```

c) Enumere, a continuación los problemas que se encuentra sobre el algoritmo anterior. ¿Qué ocurre si la semilla inicial `q0 = [0 0 0 0 0 0]`?. Plantee una solución al problema presentado. Pruebe, en este caso a utilizar la solución atenuada.

$$\dot{q} = (J + \lambda I)^{\dagger} \cdot \hat{V}$$
$$q = q + \Delta t \cdot \dot{q}$$

d) Pruebe a cambiar la ley de actualización para que el algoritmo sea más rápido.

```
q = q + 100*qd*step_time;
```

e) Varíe la línea 44 del algoritmo de prueba para ir a T1, T2 o T3. Proponga otros puntos y orientaciones a los que llegar con el algoritmo.

```
test_kinematics_ur10.m
```

f) Enumere los puntos a mejorar de este algoritmo. El algoritmo propuesto permite obtener una única solución. Proponga cómo obtener más soluciones a la cinemática

inversa. Considere la utilización del null space de la Jacobiana. En Matlab, se puede obtener fácilmente este espacio nulo.

```
>> null(J)
ans =
   0.0987
  -0.0384
   0.3028
  -0.1191
  -0.7913
   0.1769
   0.4749
```

Con lo que la regla de actualización para mover el robot en el espacio nulo es:

```
qd = null(J)
q = q + 100*qd*step_time;
```

2 SOLUCIÓN UTILIZANDO LA JACOBIANA TRANSPUESTA

a) Modifique el fichero

`/arte/robots/practicals/SAWYER/test_kinematics_sawyer.m`

b) Añada las líneas necesarias para programar un algoritmo de cinemática inversa basado en la Jacobiana TRANSPUESTA. Para probar el algoritmo, ejecute:

`/arte/robots/practicals/SAWYER/test_kinematics_sawyer.m`

c) Enumere, a continuación los problemas que se encuentra sobre el algoritmo anterior. ¿Qué ocurre si la semilla inicial `q0 = [0 0 0 0 0 0]`?

d) Pruebe a cambiar la ley de actualización para que el algoritmo sea más rápido.

`q = q + 100*qd*step_time;`

e) Varíe la línea 44 del algoritmo de prueba para ir a T1, T2 o T3. Proponga otros puntos y orientaciones a los que llegar con el algoritmo.

`test_kinematics_sawyer.m`

f) Enumere los puntos a mejorar de este algoritmo.

3 COMPARE AMBAS SOLUCIONES

Compare la solución de cinemática inversa de los dos algoritmos para la misma matriz T (T1, T2 , T3 o T4). ¿Es la misma solución? Para verlo, rellene la siguiente tabla:

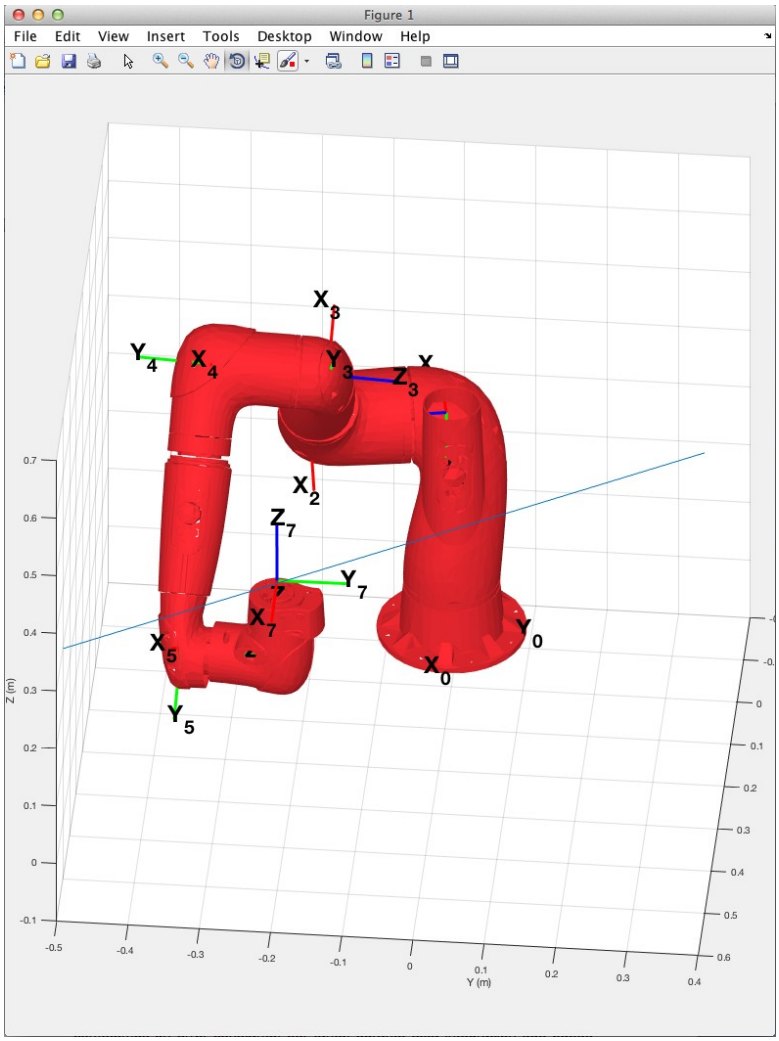
Algoritmo	**T1**	**T2**	**T4**	**T3**
Jacobiana inversa	**qinv=** **n_iterations =**	**qinv=** **n_iterations =**	**qinv =** **n_iterations =**	**qinv=** **n_iterations =**
Jacobiana transpuesta	**qinv=** **n_iterations =**	**qinv=** **n_iterations =**	**qinv =** **n_iterations =**	**qinv=** **n_iterations =**

Observe que es muy complejo asegurar la convergencia del algoritmo y, al mismo tiempo, tener un número de iteraciones bajo.

Plantee una forma de combinar ambos algoritmos para obtener una solución con independencia de puntos singulares.

3 Seguimiento de trayectorias en el espacio de trabajo

Se propone seguir una trayectoria recta en el espacio de trabajo de la tarea. Ejecute el código en `arte/robots/practicals/SAWYER/path_planning_line.m`



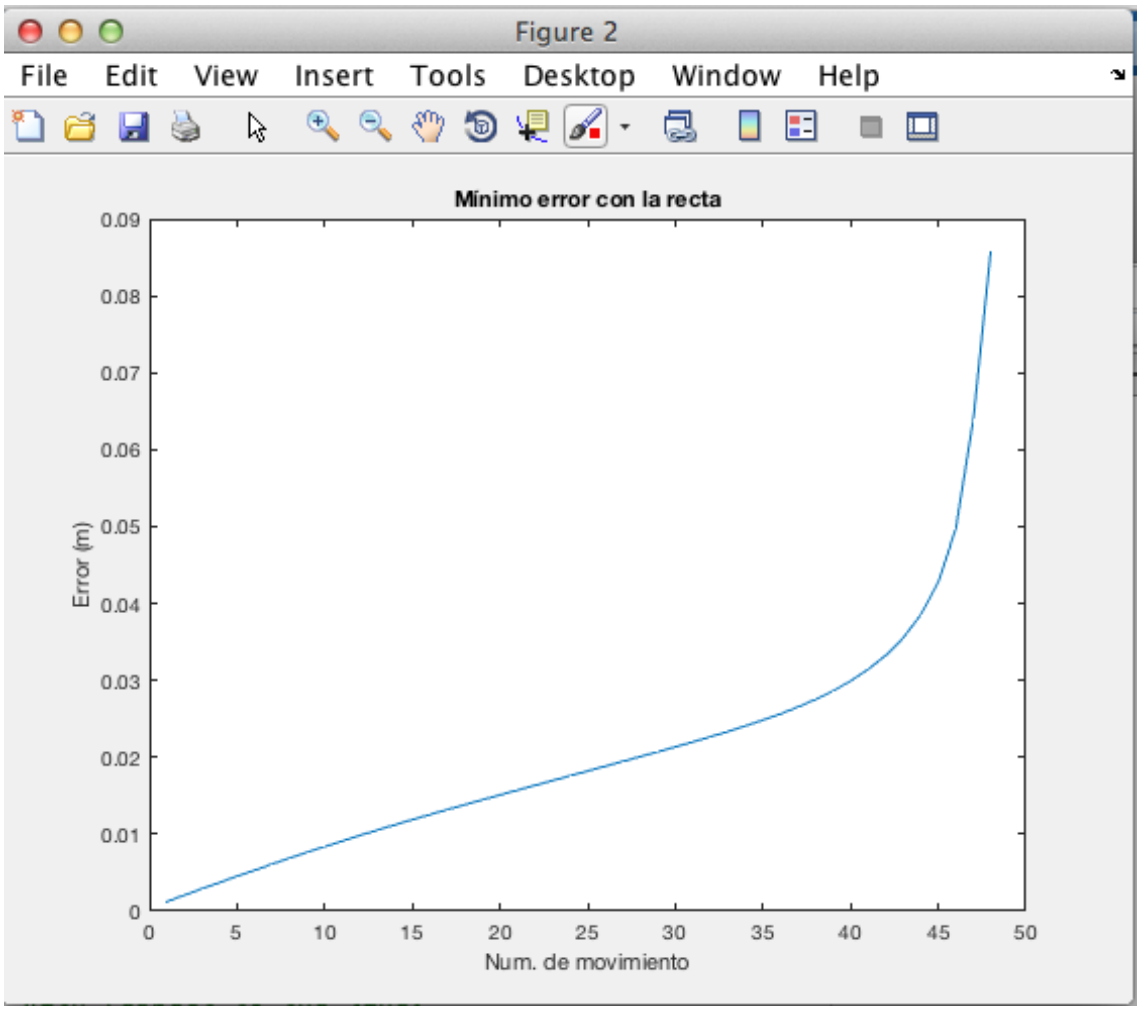
Cualquiera de las acciones de control planteadas hasta ahora sufre de un pequeño problema, por ejemplo:

$$\dot{q} = J^{\dagger} \cdot \hat{V}$$

La velocidad articular así calculada es una aproximación lineal (pues se calcula usando la Jacobiana) válida únicamente en el instante t. Esta actualización de la posición articular q es, en esencia, una aproximación de Taylor de orden 1. Al integrar y hacer

$$q = q + \Delta t \cdot \dot{q}$$

cometemos un error constante, por tanto, durante esta integración que puede llegar a ser problemático en algún caso. Se presenta a continuación el error con respecto a la recta a seguir.



Proponga una solución a este problema. En el script de seguimiento de línea cuenta ya con la función:

```
[delta_end, error_line, error_line_vector] = find_errors(start_point, end_point, p);
```

Dicha función proporciona un vector unitario en dirección con el mínimo error del extremo y la recta. Plantee, también, formas de corregir el error de orientación. Tenga en cuenta que se proporciona la variable `error_line_vector`, que define un vector perpendicular a la recta y que pasa por el extremo del robot. Plantee una solución que intente minimizar este error.