



Prácticas de Dinámica y simulación de robots manipuladores

Arturo Gil Aparicio

Índice

I Prácticas	11
1. Iniciación a Coppelia Sim	13
1.1. Objetivos	13
1.2. Coppelia Sim	13
1.3. Código proporcionado	14
1.4. Las librerías de interfaz	15
1.5. La librería de prácticas	15
1.6. Paquetes de python	16
1.7. Editores para python	17
1.8. Primeros pasos con Coppelia	17
1.8.1. Cambios en el punto de vista de la escena	18
1.8.2. Mover objetos	19
1.9. Mueva el robot	20
1.10. Un primer script de python	23
1.10.1. Añada objetos al entorno	25
1.10.2. Child scripts	26
1.11. Manejadores y descripción de pyARTE	28
2. Representación de transformaciones en Python	33
2.1. Introducción	33
2.2. Objetivos	33
2.3. ¿Para qué?	34
2.4. Matrices de rotación	35
2.5. Propiedades de una matriz de rotación	35
2.6. Matrices de rotación elementales	37
2.7. Matrices de transformación homogénea	37
2.8. Rotación de un vector	39
2.9. Transformación de las coordenadas de un punto	40
2.10. Ángulos de Euler	40
2.11. Por qué utilizar ángulos de Euler	41
2.12. Conversión de ángulos de Euler a matriz de rotación	41
2.13. Conversión de matriz de rotación a ángulos de Euler	42
2.13.1. Caso normal	43
2.13.2. Caso degenerado	43
2.14. Transformaciones en pyARTE	45
2.14.1. Vectores en pyARTE	45
2.14.2. El modo “debug”	46
2.14.3. Matrices de rotación en pyARTE	47

2.14.4. Matrices de transformación homogénea en pyARTE	48
2.15. La clase <code>Euler</code>	49
2.15.1. Visualizar transformaciones en Coppelia Sim	51
2.16. Ejercicios avanzados	55
2.17. Introducción a <code>numpy</code>	58
2.18. Matrices en <code>numpy</code>	58
3. Una aplicación de paletizado con Coppelia	61
3.1. Introducción	61
3.2. Objetivos	61
3.3. Material proporcionado	61
3.4. Descripción de la escena	62
3.5. Transformaciones	63
3.6. Cálculo de las posiciones de paletizado	64
3.7. Descripción del código	65
3.7.1. función <code>pick_and_place</code>	65
3.7.2. función <code>pick</code>	65
3.7.3. función <code>place</code>	66
3.7.4. Definiendo los <i>target points</i>	67
3.8. Ventosas de vacío	68
3.9. Tiempo real	69
4. Cinemática inversa usando la Jacobiana del manipulador	73
4.1. Objetivos	73
4.2. El robot UR5	73
4.3. Cinemática	74
4.4. Cinemática inversa	77
4.5. Cinemática inversa y planificación	82
4.6. Actividades adicionales	85
4.6.1. Realice una aplicación de paletizado	85
4.7. Resumen	85
4.8. Funciones útiles de <code>numpy</code>	87
5. Aplicaciones industriales	89
5.1. Introducción	89
5.2. Clasificación de objetos en base a su color	89
5.3. Pintura	90
5.4. Soldadura	91
6. Cinemática inversa de un robot redundante	93
6.1. Objetivos	93
6.2. El robot KUKA IIWA	93
6.3. Mueva el robot	93
6.4. Cinemática inversa	94
6.5. Una aplicación de paletizado	95
6.6. Espacio nulo	97
6.7. Objetivos secundarios	98
6.8. Otras funciones secundarias	101
6.9. Ejercicios adicionales	104
6.10. Resumen	106

7. Planificación y obstáculos	109
7.1. Objetivos	109
7.2. Introducción	109
7.3. Planificación en presencia de obstáculos	110
7.3.1. Interpolación de posiciones y orientaciones	110
7.3.2. Funciones de potencial	111
7.3.3. Funciones de potencial y planificación	114
7.4. Distancia del robot a los obstáculos	116
7.5. Distancia a los obstáculos en el espacio nulo	117
7.6. Función de potencial aplicada a la distancia a los obstáculos	120
7.7. Evasión de colisiones del resto del robot	120
7.8. Resumen	122
8. Instalación de Coppelia Sim	125
8.1. Introducción	125
8.2. Plataformas soportadas	125
8.3. Instalación de un entorno virtual de python	125
8.4. Instalación de Coppelia	126
8.5. Instalación de un entorno virtual de Python	127
8.6. Instalación de pyARTE y de las librerías de interfaz	127
8.7. Configuración de pycharm	128
8.8. Pruebas	128
II Proyecto transversal	129

Listado de Figuras

1.1.	Esquema de las comunicaciones entre pyARTE y Coppelia. Ejemplos de funciones.	15
1.2.	Menú git integrado en Pycharm	17
1.3.	Logo de Pycharm.	17
1.4.	Menú git de Pycharm.	18
1.5.	Una vista de la escena <code>scenes/irb140.ttt</code>	18
1.6.	a) Model browser y tree hierarchy en Coppelia. b) Controles para iniciar, suspender y detener la simulación. c) Controles para modificar el punto de vista de la escena. Fuente: Coppelia Sim. . .	19
1.7.	Cambiar la posición/orientación de los objetos en la escena. Fuente: Coppelia Sim.	20
1.8.	a) Menú para el cambio de posición. b) Menú para el cambio de orientación. Fuente: Coppelia Sim.	20
1.9.	Detalle del árbol de jerarquía de la escena <code>irb140.ttt</code> . Fuente: Coppelia Sim.	21
1.10.	Menú integrado en Pycharm para ejecutar scripts de python y depurar.	22
1.11.	a) Seleccione alguno de los robots presentes en la escena de Coppelia. b) Salida por pantalla de la aplicación <code>move_robot.py</code>	23
1.12.	Dos robots añadidos al entorno de Coppelia.	26
1.13.	El <i>child script</i> que crea cubos.	27
1.14.	Código del <i>child script</i> que crea cubos.	30
1.15.	a) Propiedades de un objeto de tipo <i>joint</i> en Coppelia. b) Propiedades dinámicas de una articulación (<i>joint</i>).	31
2.1.	Dos sistemas de referencia con diferente orientación relativa.	35
2.2.	a) Translación pura entre dos sistemas <i>A</i> y <i>B</i> . b) Rotación seguida de translación entre dos sistemas <i>A</i> y <i>B</i>	39
2.3.	Transformación en las coordenadas de un mismo punto en dos sistemas de referencia <i>A</i> y <i>B</i>	41
2.4.	Modo “debug” en el fichero <code>vectors.py</code>	46
2.5.	Modo “debug” en el fichero <code>vectors.py</code>	47
2.6.	Sistemas de referencia en una escena de Coppelia. Fuente: Captura de pantalla sobre Coppelia Sim.	53
2.7.	Sistemas de referencia en una escena de Coppelia. Fuente: Captura de pantalla de Coppelia Sim.	55
3.1.	Una aplicación de paletizado con Coppelia.	62

3.2.	Dos vistas de las transformaciones de interés en la aplicación de paletizado.	70
3.3.	Posiciones (x, y, z) para el paletizado de 27 piezas en un arreglo de 3x3.	71
3.4.	Pinza del robot colisionando con piezas.	71
3.5.	Aumenta o decelara la velocidad de simulación de Coppelia.	71
4.1.	El robot UR5. Fuente: www.universal-robots.com	74
4.2.	El robot UR5 en una aplicación de paletizado.	86
5.1.	Una aplicación de clasificación en base al color.	90
5.2.	Un IRB140 en una aplicación de pintura.	91
5.3.	Un IRB140 en una aplicación de soldadura TIG/MIG.	92
6.1.	El robot KUKA LBR IIWA. Fuente: www.kuka.com . KUKA Roboter GmbH.	94
6.2.	El robot KUKA LBR IIWA con una auto-colisión (pinza-robot). Fuente: www.kuka.com	97
6.3.	Una función secundaria para centrar las articulaciones	103
6.4.	Derivada de la función secundaria para centrar las articulaciones	104
7.1.	Una recta en el espacio de la tarea y un obstáculo en forma de esfera.	112
7.2.	Una recta en el espacio de la tarea y un obstáculo en forma de esfera (se muestra el obstáculo recrescido).	112
7.3.	Una función de potencial inicial.	113
7.4.	Una función de potencial mejorada (Ecuación 7.4).	114
7.5.	Los puntos de la recta movidos según una función de potencial. .	115
7.6.	Mínima distancia del brazo a los obstáculos.	118
7.7.	Abra el “child script” de Lua asociado al robot.	118
7.8.	Código del “child script” de Lua asociado al robot.	124

Lista de Tablas

3.1. Índices (i, j, k) para el paletizado del primer piso ($k = 0$). El índice i se considera alineado con X y j está alineado con Y	65
4.1. Parámetros DH del robot UR5.	73
6.1. Parámetros DH del robot KUKA LBR IIWA 14 R820.	95

Preámbulo

A modo de introducción, se indica aquí un listado de las prácticas de la asignatura. Se utilizarán, como herramientas básicas, las siguientes:

- El simulador Coppelia Sim¹.
- Python.
- La librería PyARTE² basada en Python y que utiliza Coppelia Sim para realizar las simulaciones.
- En la primera parte de documento se recogen una serie de prácticas:
 - **Práctica 1:** Iniciación a Coppelia Sim. En esta práctica se describe de forma sencilla el manejo de Coppelia Sim. Se incluye la simulación de un robot UR5 manejado desde la librería pyARTE. Se ensayan algunas actividades utilizando los scripts de Lua asociados a cada simulación. **Herramientas:** **Coppelia Sim y pyARTE.**
 - **Práctica 2:** Una aplicación de paletizado con Coppelia Sim. La práctica desarrolla una aplicación de paletizado en Coppelia Sim usando un robot IRB140. Se proponen pequeños proyectos evaluables para las prácticas. **Herramientas:** **Coppelia Sim y pyARTE.**
 - **Práctica 3:** Jacobiana del manipulador. Plantea el uso de la Jacobiana del manipulador para resolver la cinemática inversa de manipuladores sin solución cerrada. **Herramientas:** **Python y pyARTE.**
 - **Práctica 4:** Cinemática inversa de un robot redundante. Plantea el uso de la Jacobiana del manipulador y del espacio nulo para resolver la cinemática inversa de manipuladores sin solución cerrada. **Herramientas:** **Python y pyARTE.**
 - **Práctica 5:** Planificación y obstáculos. Se aborda la cinemática inversa de un robot redundante y se utiliza el espacio nulo para evitar obstáculos en el espacio de trabajo del robot. **Herramientas:** **Python y pyARTE.**
 - **Práctica 6:** Instalación de Coppelia Sim y pyARTE. Se detallan los pasos para instalar Coppelia Sim y la librería pyARTE. **Herramientas:** **Python y pyARTE.**
- En la **parte II:** se detallan algunas ideas sobre proyectos a realizar utilizando las herramientas de simulación presentadas.

¹www.coppeliasim.com

²<https://github.com/4rtur1t0/pyARTE>

Parte I

Prácticas

Capítulo 1

Iniciación a Coppelia Sim

1.1. Objetivos

En esta práctica se persiguen los siguientes **objetivos de aprendizaje**:

- Iniciar al estudiante en el uso del simulador Coppelia Sim¹.
- Conocer las principales características de un simulador de robots.
- Conocer la interfaz de programación en python del simulador e introducir la librería pyARTE y su capacidad para manejar el simulador Coppelia Sim.

Durante la práctica se realizarán, entre otras, las siguientes **actividades principales**:

- Cambiar el punto de vista con que se observa la escena en el simulador.
- Modificar la posición y orientación de objetos en el simulador.
- Mover un robot de tipo serie utilizando la librería pyARTE.
- Modificar los scripts de Lua que manejan la simulación en Coppelia.

1.2. Coppelia Sim

En este apartado se presenta el simulador *Coppelia Sim*. El simulador está basado en el conocido (y ampliamente usado) simulador *V-REP*. Las características básicas de *Coppelia Sim* son:

- Software multi-plataforma, con distribuciones para Windows, Linux y Mac.
- Programación: Cuenta con interfaces para Python, C/C++ y Matlab. Adicionalmente, es posible realizar el control desde nodos de ROS (Robot Operating System)². Durante las prácticas de Robótica utilizaremos Python, pues es un lenguaje muy utilizado en la industria.

¹www.coppeliarobotics.com

²www.ros.org

Además, el simulador cuenta con la siguientes capacidades:

- Un motor físico que permite cálculos rápidos para la simulación de un entorno realista con colisiones, fuerzas... etc.
- Simulación de sensores de distancia y cámaras.

Con el manejo de Coppelia Sim, se habilita al alumno a:

- Utilizar un simulador completo de Dinámica multicuerpo.
- Control del robot por pares/fuerzas y posición articular.
- Realizar simulaciones realistas de procesos industriales considerando colisiones y autocolisiones.
- Realizar interacciones entre un robot y otros elementos del entorno.

1.3. Código proporcionado

Durante las prácticas se utilizará la librería pyARTE³. En esencia, esta librería nos permitirá realizar simulaciones que se programarán en python y se materializarán en el simulador Coppelia Sim. Así pues, durante las prácticas, se usarán las interfaces (librería) para Python que proporciona Coppelia. Alternativamente, es posible utilizar Matlab y la interfaz existente en ARTE⁴ para manejar Coppelia. Sin embargo, utilizar Python y la API de Python presenta ventajas y es la opción más recomendable:

- No precisa la adquisición de una licencia de Matlab.
- Permite realizar simulaciones utilizando menos recursos del ordenador.
- Permite la fácil integración del código con otros paquetes de Python como: numpy, scikit-learn, tensorflow, keras, OpenCV y muchos otros. Estas librerías son frecuentes en aplicaciones de Inteligencia Artificial y Visión por Computador.

La Figura 1.1 muestra un esquema organizativo entre la librería pyARTE y el simulador Coppelia. En esencia, el simulador Coppelia mantiene un servicio activo en todo momento. Dicho servicio, escucha en el puerto 19997 del PC en el que se ejecute Coppelia. Las librerías que proporciona Coppelia (API COPPELIA), mandan comandos a ese puerto para modificar las características de la simulación, por ejemplo, si deseamos que una articulación genere un par determinado, o bien se desplace a una posición articular en concreto. Las funciones de la librería de Coppelia se han encapsulado en una serie de clases y de métodos para facilitar la simulación y el trabajo durante las prácticas (pyARTE).

³<https://github.com/4rtur1t0/pyARTE>

⁴www.arvc.umh.es/arte

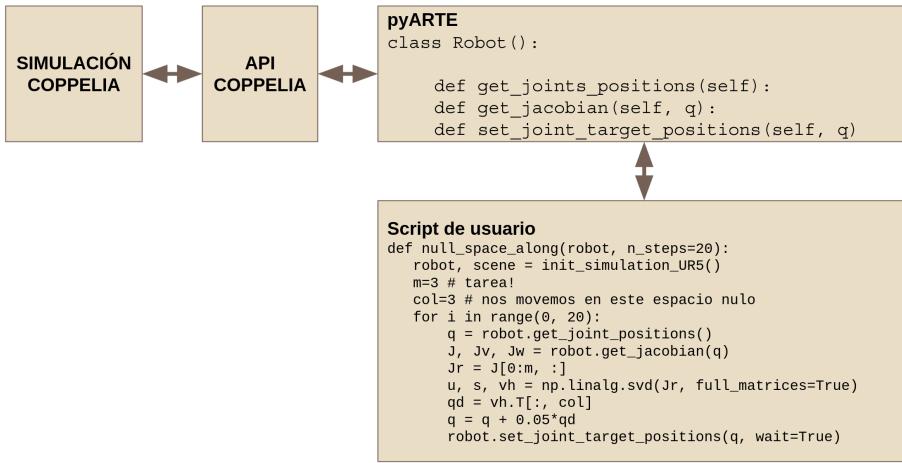


Figura 1.1: Esquema de las comunicaciones entre pyARTE y Coppelia. Ejemplos de funciones.

1.4. Las librerías de interfaz

Según se ha comentado, Coppelia proporciona librerías para poder manejar el simulador utilizando otros lenguajes de programación, como, por ejemplo, C/C++ o python. En la última versión de la librería pyARTE se utiliza la librería Zmq para comunicar a Python con Coppelia. En el Anexo se encuentran detalles para instalar esta librería.

1.5. La librería de prácticas

Los equipos de prácticas están configurados para usar pyARTE y Coppelia Sim sin dificultades. Si se desea instalar el simulador y la librería pyARTE en un PC de uso personal puede seguir las instrucciones del Capítulo 8. Para utilizar la librería pyARTE, siga los siguientes pasos:

- Inicie el editor Pycharm. Ud. es libre de usar cualquier otro editor. Sin embargo, en los ordenadores de prácticas, el editor ya se encuentra configurado con el proyecto de Python/ARTE.
- Abra el proyecto en `/home/isa/Escritorio/pyARTE`.
- Actualice a la última versión del código de las prácticas usando el menú git integrado en Pycharm (Figura 1.2). Tras pulsar sobre la flecha azul, seleccione la opción de “rebase” para configurar el repositorio con la última versión publicada. Con esto, se descargan todos los ficheros con la última versión de la librería. Es necesario tener en cuenta que esto podría sobreescribir cualquier cambio que hayamos realizado sobre los ficheros de la librería en nuestro PC de prácticas.

En estos momentos, debemos tener (de una forma u otra) el repositorio de código de pyARTE perfectamente configurado. A continuación, describiremos

brevemente el código. Se describen, a continuación, los principales directorios del proyecto:

- **artelib**: contiene los robots de la librería y las principales herramientas matemáticas: transformaciones homogéneas, cuaternios, etc, rotaciones, ángulos de Euler.
- **kinematics**: un paquete para añadir fácilmente la cinemática de los robots.
- **scenes**: un conjunto de escenas de coppelia que están integradas dentro de la librería de prácticas.
- **practicals**: en este directorio se encuentran los scripts de python para realizar las prácticas. En algunas ocasiones, se pedirá que el/la estudiante sea capaz de completar el código para realizar las tareas indicadas como ejercicios. El estudiante también podrá desarrollar sus propios scripts para realizar otras simulaciones.
- **demos**: un directorio con aplicaciones extra para demostrar las capacidades de pyARTE y Coppelia.
- **robots**: En este directorio se encuentran algunas clases que representan los robots de interés en la librería. Los siguientes robots están modelados y se encuentran perfectamente definidos en la librería:
 - Robot ABB IRB140.
 - Robot UR5.
 - Robot Kuka LBR.

Todos los anteriores son nombres comerciales de ABB Ltd., Universal Robots A/S y KUKA Roboter GmbH, respectivamente. Igualmente, en este directorio se definen pinzas y sensores utilizados en las simulaciones de Coppelia. La librería pyARTE maneja los modelos proporcionados junto con el simulador Coppelia Sim a los que se les ha realizado ligeras modificaciones.

1.6. Paquetes de python

En los equipos de prácticas se cuenta con una versión de python que cuenta con todos los paquetes ya instalados. Puede instalar los requisitos necesarios para el repositorio de python con las instrucciones que se detallan en el Capítulo 8. Son pocos los paquetes necesarios para ejecutar las simulaciones, en concreto, la librería depende únicamente de:

- **numpy**: computación científica con python.
- **matplotlib**: fundamentalmente pyARTE utiliza el simulador Coppelia para representar los cálculos realizados. No obstante, la librería matplotlib se emplea para representar algunas trayectorias articulares.
- **pynput**: captura de caracteres del teclado.

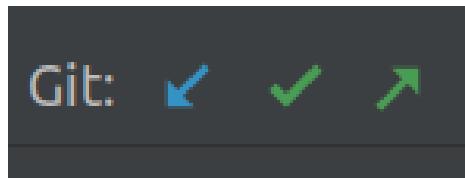


Figura 1.2: Menú git integrado en Pycharm



Figura 1.3: Logo de Pycharm.

1.7. Editores para python

Existen multitud de programas que permiten editar ficheros escritos en lenguaje python. Entre ellos, podemos mencionar PyCharm, pyDev, IDLE y Visual Studio. La elección de un editor depende, en gran medida, de los gustos de cada programador. Durante las prácticas se recomienda el uso del editor pyCharm, el cual ofrece una versión gratuita con un gran número de ayudas a la programación. Realice los siguientes pasos:

- Abra el editor PyCharm (Figura 1.3). Se trata de un editor para python con un gran número de funcionalidades que facilitan la creación de programas.
- El editor PyCharm de las prácticas está configurado para ejecutar una instancia de python. También está configurado para incluir una serie de librerías. En próximas prácticas se indicarán más detalles sobre la configuración de PyCharm.
- El editor PyCharm está configurado para incluir en sus directorios a la librería pyARTE.
- El editor está conectado con el repositorio de la librería. Esto es, se conecta con la url: <https://github.com/4rtur1t0/pyARTE>. En este momento, debe actualizar y obtener la última versión de la librería. Para ello, haga click sobre el ícono azul de la Figura 1.4 y seleccione la opción “merge”.

1.8. Primeros pasos con Coppelia

En este apartado se indican algunas acciones básicas a realizar sobre el simulador. Estas operaciones incluyen el cambio en el punto de vista de la escena, mover y rotar objetos en la escena.

Comience, ejecutando Coppelia Sim y abra la escena `irb140.ttt`: “File” - “Open Scene” - `scenes/irb140.ttt`. Al abrir esta escena veremos una simulación como la mostrada en la Figura 1.5. En esta vista, podemos distinguir:

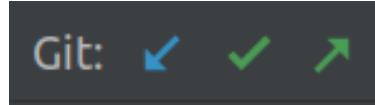


Figura 1.4: Menú git de Pycharm.

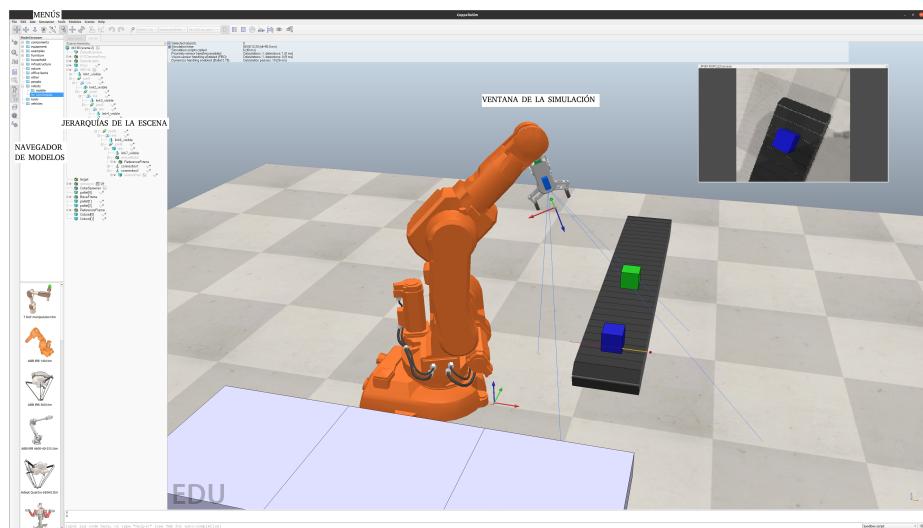


Figura 1.5: Una vista de la escena `scenes/irb140.ttt`.

- Una ventana de simulación: En esta ventana principal se presentará una vista de la simulación de la aplicación robótica.
- Un “navegador de modelos” (izquierda). Denominado *Model Browser*, en este navegador encontraremos un conjunto de robots de tipo serie y paralelo. También se encuentran robots móviles. En el navegador de modelos también encontraremos accesorios útiles para simular aplicaciones robóticas, como, por ejemplo, pinzas o manos robóticas. Finalmente, también podemos encontrar cintas transportadoras y sensores.
- Una jerarquía de la escena (izquierda). Denominado *Scene hierarchy*, esta vista presenta las relaciones entre los objetos de la escena de una forma sencilla.

El árbol de jerarquía de la simulación representa las dependencias en posición y orientación de los objetos de la escena. Se observa, por ejemplo, que en un robot de tipo serie, el movimiento del eslabón 1 es relativo a la base. A su vez, el eslabón 2 depende del eslabón 1... etc. En ocasiones resulta útil ocultar las ventanas de Jerarquía y el navegador de modelos. Para ello, simplemente presiones sobre los botones de la Figura 1.6a.

1.8.1. Cambios en el punto de vista de la escena

Ahora, nos proponemos cambiar el punto de vista con que se observa la escena simulada. En la Figura 1.6c se presentan los controles que permiten variar la visualización de la escena. De izquierda a derecha:



Figura 1.6: a) Model browser y tree hierarchy en Coppelia. b) Controles para iniciar, suspender y detener la simulación. c) Controles para modificar el punto de vista de la escena. Fuente: Coppelia Sim.

- *Camera pan*: translada la escena.
- *Camera rotate*: rota la vista manteniendo un punto fijo.
- *Camera shift*: aleja o acerca la cámara a la escena.
- *Camera opening angle*: abre o cierra el ángulo de la lente de cámara.
- *Fit to view*: incluye todos los elementos de la escena en la vista actual.

Puede utilizar estos controles para observar algunos detalles de la escena.

Ejercicio 1.8.1: Cambios en el punto de vista

Utilice los controles de Coppelia (1.6c) para cambiar el punto de vista de la escena y hacer zoom sobre algún elemento.

1.8.2. Mover objetos

Mostramos, a continuación, los menús de Coppelia que permiten cambiar la posición y orientación de los objetos de la escena. Para ello se utilizan los controles mostrados en la Figura 1.7. De izquierda a derecha:

- *Click selection*: permite utilizar el ratón para seleccionar objetos en la escena.
- *Move*: menú para cambiar la posición de los objetos en la escena.
- *Rotate*: menú para cambiar la orientación de los objetos en la escena.

Por ejemplo, abra la escena `scenes/irb140.ttt` y seleccione cualquier elemento con el ratón (*click selection*). A continuación, presione el botón *Move* (Figura 1.7, izquierda). Aparecerá una ventana como la mostrada en la 1.8a. Este menú permite cambiar la posición del centro de masas del objeto en el sistema de referencia global o bien en el sistema de referencia del objeto “parent”.

A continuación, presione el botón *Rotate* (Figura 1.7, derecha). Aparecerá una ventana como la mostrada en la 1.8b. Igualmente, se pueden realizar rotaciones en el sistema de referencia de la escena (*world*) o en el sistema de referencia del objeto “parent”. Nótese que la orientación se expresa en grados

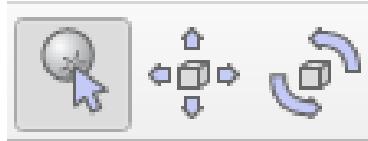


Figura 1.7: Cambiar la posición/orientación de los objetos en la escena. Fuente: Coppelia Sim.

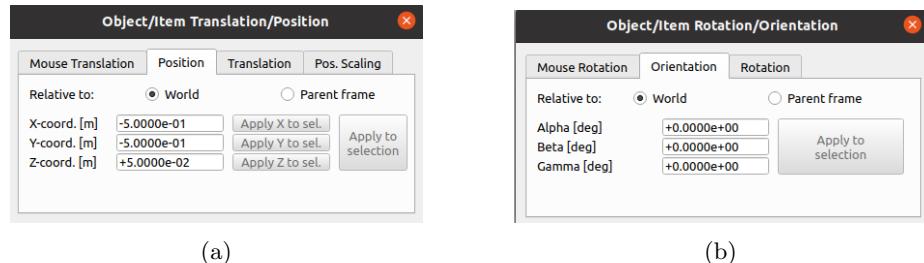


Figura 1.8: a) Menú para el cambio de posición. b) Menú para el cambio de orientación. Fuente: Coppelia Sim.

sexagesimales, utilizando tres ángulos de Euler XYZ en ejes móviles que parten del sistema de coordenadas especificado. De esta manera que la matriz de orientación resultante es:

$$R = R_x(\alpha)R_y(\beta)R_z(\gamma)$$

Importante: La parte gráfica de Coppelia utiliza grados sexagesimales para representar las rotaciones, mientras que la librería de Python (pyARTE) utiliza radianes.

Ejercicio 1.8.2: Mover y rotar objetos en Coppelia

Utilice los menús de Coppelia para:

- Cambiar la posición de un objeto de la escena.
- Cambiar la orientación de un objeto de la escena.

1.9. Mueva el robot

Nos disponemos, a continuación a mover el robot IRB140 usando Python y la librería pyARTE. Utilizaremos el script `practicals/move_robot.py`. Este script permite manejar varios robots de los que están incluidos en la librería. El script captura las pulsaciones del teclado y envía comandos a Coppelia para que modifique las posiciones articulares. Para ejecutar el script de python puede utilizar el menú integrado de Pycharm (Figura 1.10). Simplemente, presione con el botón derecho sobre el script `move_robot.py` y elija la opción Run (ejecutar) o Debug (depurar). Al ejecutar el script de python, se inicializa la librería y python se conecta con Coppelia a través de un socket en el sistema. Automáticamente,

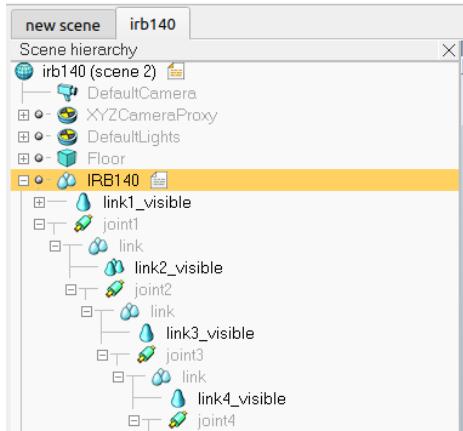


Figura 1.9: Detalle del árbol de jerarquía de la escena `irb140.ttt`. Fuente: Coppelia Sim.

se lanza, en este momento, la simulación en Coppelia. Cuando el script de python finaliza, automáticamente se cierra la simulación. Si, por alguna razón, el script de python no finaliza correctamente, es posible que la simulación en Coppelia no finalice y se deba parar manualmente (botón “stop”).

Ejercicio 1.9.1: Mueva el robot

Abra la escena con un robot ABB IRB140 (`scenes/irb140.ttt`) y ejecute el script `practicals/move_robot.py`. Para ejecutar el script en PyCharm use el botón “play” de la Figura 1.10. El script es, en extremo, sencillo de utilizar, simplemente:

- Asegúrese de hacer click sobre la ventana inferior del terminal (de manera que no edite el código python).
- Elija el primer robot (1). Podrá mover otro tipo de robots con las otras opciones disponibles. Véase la Figura 1.11a.
- Utilice las teclas 1, 2, 3, ..., 6 para incrementar cada articulación q_i .
- Utilice las teclas q, w, e, ..., y para decrementar cada articulación q_i .
- Utilice 'o'/'c' para abrir/cerrar la pinza.
- Utilice 'z' para llevar a cero las articulaciones.
- Utilice 'ESC' para salir del programa.

En todo momento, el script le mostrará por pantalla los valores articulares q actuales y la posición del extremo como (Figura 1.11b):

- Una matriz homogénea T que incluye R y \vec{p} (orientación y posición del extremo del robot).

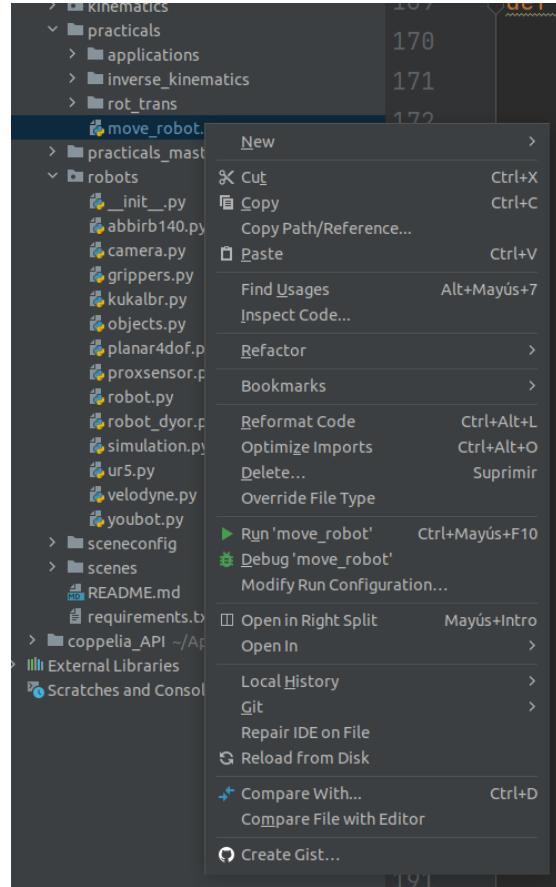


Figura 1.10: Menú integrado en Pycharm para ejecutar scripts de python y depurar.

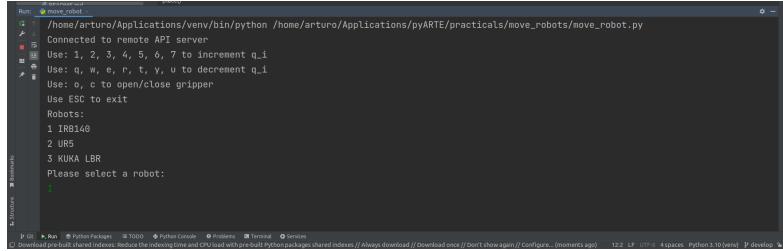
- Un cuaternion Q .
- Tres ángulos de Euler con la orientación de la pinza (en coordenadas de la base).

Considere utilizar esta información durante los siguientes apartados.

Ejercicio 1.9.2: Uso de `move_robot.py` I

Utilice el script `move_robot.py` y realice las actividades siguientes:

- Mueva el robot y encuentre algunas posiciones/orientaciones de interés (p.e. posición de recogida de las piezas, posición para dejar las piezas).
- Mueva el robot a una configuración en la que $R = I$.
- **Intente asir una pieza y moverla con el robot para depositarla en cualquier pallet la izquierda del robot.**

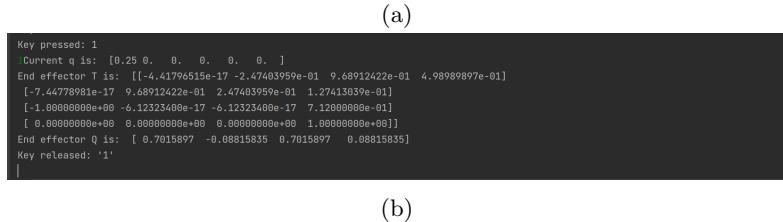


(a)

```

Run: move_robot
/home/arturo/Applications/venv/bin/python /home/arturo/Applications/pyARTE/practicals/move_robots/move_robot.py
Connected to remote API server
Use: 1, 2, 3, 4, 5, 6, 7 to increment q_i
Use: q, w, e, r, t, y, u to decrement q_i
Use: o, c to open/close gripper
Use ESC to exit
Robots:
1 IRB140
2 UR5
3 KUKA LBR
Please select a robot:

```

(b)

```

Key pressed: 1
Current q is: [ 0.25 0. 0. 0. 0. 0. ]
End effector T is: [[-4.41796515e-17 -7.47403959e-01 9.68912422e-01 4.98989897e-01]
[-7.44778981e-17 9.68912422e-01 2.47483959e-01 1.27413039e-01]
[-1.00000000e+00 -6.12323400e-17 -6.12323400e-17 7.12088000e-01]
[ 0.00000000e+00 0.00000000e+00 0.00000000e+00 1.00000000e+00]]
End effector Q is: [ 0.7015897 -0.08815835 0.7015897 0.08815835]
Key released: '1'

```

Figura 1.11: a) Seleccione alguno de los robots presentes en la escena de Coppelia. b) Salida por pantalla de la aplicación `move_robot.py`.

Ejercicio 1.9.3: Uso de `move_robot.py` II

Utilice el script `move_robot.py` y realice las actividades siguientes:

- Experimente con la capacidad de Coppelia para simular colisiones y autocolisiones: intente que el robot se choque consigo mismo o con las piezas que encuentre en el entorno.
- Observe la posición/orientación del extremo en relación con el sistema de referencia de Coppelia. Compare con la salida de texto que produce el programa.
- Anote en un fichero de texto, al menos, 4 posiciones características del robot (posición inicial, posición de aproximación, posición para coger piezas, posición para dejar piezas).

1.10. Un primer script de python

En este apartado presentaremos una simulación básica utilizando Coppelia Sim y la librería pyARTE. Para ello, debemos seguir los siguientes pasos:

- Inicie Coppelia Sim y, a continuación, abra una escena. Deberá ir al menú “File” y después a “Open scene...”. En concreto, abra el fichero `irb140.ttt` que encontrará en `pyARTE/scenes`. En este directorio encontrará todas las escenas que habitualmente se manejan con la librería pyARTE. Tendremos una vista similar a la mostrada en la Figura 1.5.
- Seguidamente abra el script de python que maneja esta escena. El script se llama `pyARTE/practicals/irb140_first_script.py`. Puede utilizar cualquier editor de python para ello.

Si nos fijamos en la Figura 1.5, podemos ver una escena donde se muestra un robot IRB140 en simulación. El robot está equipado con una pinza y, además, en la escena, se simula una cinta transportadora sobre la que se mueven piezas cúbicas. A continuación, se muestra el texto del script `irb140_first_script.py`. Este script es muy básico y se presenta aquí para describir el funcionamiento de la librería pyARTE.

```
import numpy as np
from robots.abbirb140 import RobotABBIRB140
from robots.grippers import GripperRG2
from robots.simulation import Simulation

if __name__ == "__main__":
    simulation = Simulation()
    simulation.start()
    robot = RobotABBIRB140(simulation=simulation)
    robot.start()
    gripper = GripperRG2(simulation=simulation)
    gripper.start(name='/IRB140/RG2/RG2_OpenCloseJoint')

    q1 = np.array([-np.pi/4, np.pi/8, np.pi/8, np.pi/4, -np.pi/4, np.pi/4])
    q2 = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
    q3 = np.array([np.pi/8, 0, -np.pi/4, 0, -np.pi/4, 0])
    q4 = np.array([0, 0, 0, 0, np.pi / 2, 0])

    gripper.open(precision=True)
    gripper.close(precision=True)
    gripper.open(precision=True)
    robot.moveAbsJ(q1, endpoint=True)
    robot.moveAbsJ(q2, endpoint=True)
    robot.moveAbsJ(q3, endpoint=True, precision=True)
    robot.moveAbsJ(q4, endpoint=True, precision=True)

simulation.stop()
```

En concreto, las líneas siguientes crean un objeto de la clase `Simulation` y realizan la conexión con Coppelia. En concreto, el método `start` realiza la conexión entre el script de python y el programa Coppelia.

```
simulation = Simulation()
simulation.start()
```

Seguidamente, las dos líneas siguientes crean un objeto de la clase `RobotABBIRB140` y lo inicializa. También crea un objeto de tipo pinza (pinza RG2) y lo inicializa.

```
robot = RobotABBIRB140(simulation=simulation)
robot.start()
gripper = GripperRG2(simulation=simulation)
gripper.start(name='/IRB140/RG2/RG2_OpenCloseJoint')
```

Importante: la conexión entre Coppelia y los scripts de pyARTE se basan en los nombres de cada objeto dados en Coppelia. Estos nombres se observan en el menú de “Jerarquía” de la Figura 1.9. En concreto, observamos que la base del robot se denomina “/IRB140” y que las articulaciones se denominan ‘joint1’, ‘joint2’, ‘joint3’... etc. Podemos conectar un segundo robot que se denomine ‘/IRB140_2’ si hacemos:

```
robot2 = RobotABBIRB140(simulation=simulation)
robot2.start(base_name='/IRB140_2')
```

Esto permite tener varios robots conectados en una misma simulación de Coppelia. Seguidamente, las líneas siguientes definen 4 vectores de valores articulares. Para ello se utiliza la clase `array` de `numpy`.

```
q1 = np.array([-np.pi/4, np.pi/8, np.pi/8, np.pi/4, -np.pi/4, np.pi/4])
q2 = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
q3 = np.array([np.pi/8, 0, -np.pi/4, 0, -np.pi/4, 0])
q4 = np.array([0, 0, 0, 0, np.pi / 2, 0])
```

Con esto, si queremos que el robot se mueva en Coppelia, podemos utilizar el método `moveAbsJ`. Este método le indica al robot en Coppelia que debe realizar un control articular para llevar al robot a las posiciones articulares especificadas en cada caso (`q1`, `q2`, `q3` y `q4`). Es importante tener en cuenta que el tiempo de simulación está controlado también desde el script de python. Así, si queremos que la simulación avance un paso de simulación (por defecto el paso de simulación es: `dt=50 ms`), debemos utilizar el método `wait` de la clase `Simulation`. En el caso mostrado a continuación esperamos 100 instantes de simulación para que el robot pueda terminar su movimiento.

Ejercicio 1.10.1: Recogida de piezas

Modifique el script `irb140_first_script.py` para que el robot sea capaz de **recoger una pieza de la cinta transportadora** y la deposite en uno de los pallets. Hecha la actividad, el script será el primer programa que hace Ud. en el que se mueve el robot en un modo de cinemática directa para realizar una tarea. Para realizar la actividad, el/la estudiante deberá:

- Modificar los valores de `q1`, `q2`, `q3` para que el robot alcance diferentes posiciones y orientaciones de la pinza.
- Abrir o cerrar la pinza del robot en consecuencia, según las necesidades.

En las próximas prácticas se tratarán formas más convenientes de programar un robot.

1.10.1. Añada objetos al entorno

Coppelia cuenta con un menú denominado *Model browser* (navegador de modelos, véase la Figura 1.6a). Dentro de este menú, Coppelia ofrece un conjunto de modelos predefinidos de robots. Encontramos, en este menú, robots móviles, serie, paralelos, así como otros elementos accesorios como: manos robóticas, cintas transportadoras, muebles, personas, etc. Presione el botón del *Model browser* y añada un robot IRB140 al entorno de Coppelia y un hexápedo móvil, por ejemplo (Figura 1.12). Si inicia la simulación, observará que los robots realizan por sí mismos una serie movimientos.

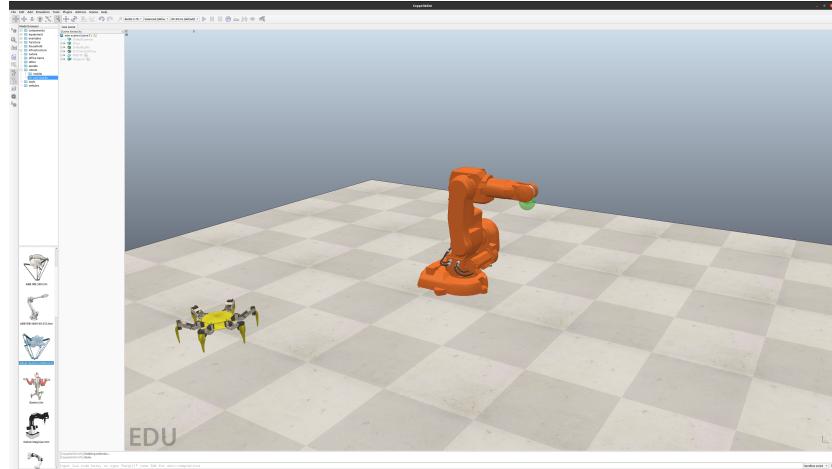


Figura 1.12: Dos robots añadidos al entorno de Coppelia.

Ejercicio 1.10.2: Model Browser

Utilice el “navegador de modelos” y recorra todos los diferentes directorios y tipos de objetos que ofrece Coppelia. En concreto:

- Busque el robot móvil `hexapod.ttm` y arrástrelo a la escena.
- Busque el robot serie (*non-mobile*) denominado UR5 y arrástrelo a la escena.

Hecho esto, puede utilizar el botón con forma triangular (Figura 1.6b) para iniciar la simulación. Observará que los robots comienzan a moverse.

1.10.2. Child scripts

Coppelia cuenta con un intérprete de lenguaje Lua (<https://www.lua.org/>). De esta manera, se pueden asignar scripts en lenguaje Lua a todos los objetos dentro de Coppelia, de manera que se pueden programar comportamientos y movimientos de una forma sencilla. Cuando se inicia la simulación, Coppelia inicia todos los scripts de Lua asociados a la escena.

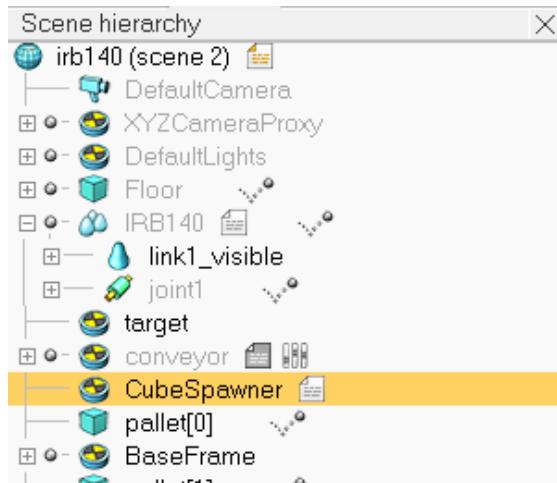


Figura 1.13: El *child script* que crea cubos.

Ejercicio 1.10.3: Child scripts

El ejercicio ejemplifica el uso de los *child scripts* en Coppelia:

- Abra la escena `scenes/irb140.ttt`.
- Inicie la simulación. Observará que la cinta transportadora se mueve y que aparecen cubos sobre ella.
- Edite el *child script* denominado “CubeSpawner”. Para ello realice dos clicks sobre el ícono de tipo documento que aparece al lado del nombre (Figura 1.13). El código Lua se muestra en la Figura 1.14.
- Modifique el código. Por ejemplo: cambie la variable `PIECE_TYPE` y observe el resultado.
- Cambie el tamaño de los cubos y la posición en la que se generan en la escena.

Por tanto, el estudiante debe entender que los *scripts* de Lua permiten programar comportamientos automáticos del simulador que se realizarán siempre que el simulador esté en marcha. Es importante tener esto en cuenta, pues, si al mismo tiempo, utilizamos la librería pyARTE para comandar al robot podríamos tener comportamientos inesperados. En estas prácticas se intenta evitar el uso de scripts de Lua, pues resulta más sencilla la programación de todos los comportamientos desde la librería pyARTE. Por otra parte, los comportamientos que se programen en Lua liberan a python de la realización de muchos cálculos y pueden ser beneficiosos para realizar ciertas simulaciones.

1.11. Manejadores y descripción de pyARTE

Se describe, a continuación, el funcionamiento en conjunto de pyARTE y de Coppelia. Cada simulación realizada con el entorno Coppelia/pyARTE necesitará:

- Una escena .ttt de Coppelia.
- Un script de python en pyARTE que realiza la tarea sobre la escena.

Vamos a ver, a continuación, un ejemplo:

- Abra la escena: `irb140.ttt`.
- Observe el nombre de la base del robot: 'IRB140'.
- Observe el nombre en Coppelia de cada una de las articulaciones del robot. En el árbol de jerarquía de la escena debe encontrar: 'joint1', 'joint2', 'joint3'... etc.

Cada elemento de la escena tiene un nombre dentro de la jerarquía de la escena:

- La articulación 1 se llama: '/IRB140/joint1'.
- La articulación 2 se llama: '/IRB140/joint2'... etc.

A cada elemento de la escena de Coppelia se le asigna un manejador (handle). Este manejador permite enviarle comandos al elemento para su simulación. Las articulaciones (joints) en Coppelia son elementos especiales que pueden recibir diferentes tipos de comandos, en concreto:

- Comandos de posición: En este caso Coppelia realiza un control de posición en la articulación comandada. Se pueden ajustar los parámetros de un controlador PID para realizar esta tarea de control.
- Comandos de velocidad: Coppelia realiza un control de velocidad de la articulación indicada.
- Comandos de fuerza/par. En este caso, el simulador envía la acción de par. Dicha fuerza o par se introduce en el simulador Coppelia para realizar la simulación de dinámica multi-cuerpo.

Si hacemos doble click sobre cualquier articulación, veremos el diálogo de la Figura 1.15a. Si entramos en el menú “Show dynamic properties dialog”, aparecerá el diálogo de la Figura 1.15b. Este diálogo, permite:

- Habilitar o deshabilitar el motor de la articulación (“motor enabled”).
- Fijar el par máximo del motor (*max. torque*).
- Habilitar o deshabilitar el bucle de control en posición: “control loop enabled”. En pyARTE es preciso que el control de cada articulación sea en modo “**velocidad**”.

Nota: Toda la librería pyARTE envía, por defecto, comandos de velocidad a Coppelia. Así pues, es importante que todas las articulaciones de los robots tengan habilitado el bucle de control en **velocidad**.

Los manejadores de Coppelia también pueden ser obtenidos desde python. Esta tarea se realiza desde las funciones que se encuentran en el método **start()** de cada robot que se encuentra en el directorio **robots**.

Ejercicio 1.11.1: Manejadores en python

Por ejemplo:

- Abra el fichero **pyARTE/robots/abbirb140.py**.
- Busque el método **def start()**.
- En esta función, que se muestra bajo este cuadro, encontrará que en las funciones se utiliza el mismo nombre para acceder a los elementos de la escena de Coppelia.
- Para comandar el robot, se utilizarán los métodos de la clase **Robot()** que incluyen funciones que comandan los manejadores de la escena.

Si abre el fichero **robots/irb140.py** observará el siguiente código de python. Es sencillo modificar este código si se desean manejar otras escenas diferentes. Para ello, será necesario indicar los nuevos nombres de las articulaciones. Otros robots en la librería tienen métodos similares al siguiente para adecuarse a los nombres de la escena.

```
def start(self, base_name='/IRB140', joint_name='joint'):  
    armjoints = []  
    # Get the handles of the relevant objects  
    robotbase = self.simulation.sim.getObject(base_name)  
    q1 = self.simulation.sim.getObject(base_name + '/' + joint_name + '1')  
    q2 = self.simulation.sim.getObject(base_name + '/' + joint_name + '2')  
    q3 = self.simulation.sim.getObject(base_name + '/' + joint_name + '3')  
    q4 = self.simulation.sim.getObject(base_name + '/' + joint_name + '4')  
    q5 = self.simulation.sim.getObject(base_name + '/' + joint_name + '5')  
    q6 = self.simulation.sim.getObject(base_name + '/' + joint_name + '6')  
  
    armjoints.append(q1)  
    armjoints.append(q2)  
    armjoints.append(q3)  
    armjoints.append(q4)  
    armjoints.append(q5)  
    armjoints.append(q6)  
    # store the joints  
    self.joints = armjoints
```

En caso de que algún script falle, es importante comprobar que las variables **q1, q2...** son todas mayores que 0.

Child script "/CubeSpawner"

```

1 function sysCall_init()
2     coroutine=coroutine.create(coroutineMain)
3 end
4
5 function sysCall_actuation()
6     if coroutine.status(corout)~='dead' then
7         local ok,errorMsg=coroutine.resume(corout)
8         if errorMsg then
9             error(debug.traceback(corout,errorMsg),2)
10        end
11    end
12 end
13
14 function coroutineMain()
15
16     local CUBE_COUNT = 100
17     local SIZE = {0.08, 0.08, 0.08}
18     local START_POS = {0.6, 0.6, 0.2}
19     local WAIT_TIME = 10
20     local PIECE_TYPE = 0 -- 0: cuboid, 1: sphere, 2: cylinder, 3: cone
21
22
23     -- Define 'constants' to improve code readability
24     local CULLED_BACKFACES=1
25     local VISIBLE_EDGES=2
26     local APPEAR_SMOOTH=4
27     local RESPONDABLE_SHAPE=8
28     local STATIC_SHAPE=16
29     local CYL_OPEN_ENDS=32
30     local PIECE_MASS = 0.1
31     local i = 0
32
33     red = {0.8, 0.1, 0.1}
34     green = {0.1, 0.8, 0.1}
35     blue = {0.1, 0.1, 0.8}
36
37     while i < CUBE_COUNT do
38
39         -- create the shape
40         spawn=sim.createPureShape(PIECE_TYPE, VISIBLE_EDGES + APPEAR_SMOOTH + RESPONDABLE_SHAPE, SIZE, PIECE_MASS)
41
42         -- use this to set the color pure red/green/blue pieces randomly
43         r = math.random()
44         if r < 0.33 then
45             sim.setShapeColor(spawn,nil, sim.colorcomponent_ambient_diffuse, red)
46         elseif r >= 0.33 and r < 0.66 then
47             sim.setShapeColor(spawn,nil, sim.colorcomponent_ambient_diffuse, green)
48         else
49             sim.setShapeColor(spawn,nil, sim.colorcomponent_ambient_diffuse, blue)
50         end
51
52         -- set the position of the object
53         sim.setObjectPosition(spawn, -1, START_POS)
54         -- the following makes the piece detectable by proximity sensors and for vision sensors
55         sim.setObjectProperty(spawn, 32)
56         sim.setObjectSpecialProperty(spawn, 1000)
57
58         -- use this to create pieces with random colours
59         --local colour_components =
60         --{sim.colorcomponent_ambient_diffuse,
61         -- sim.colorcomponent_specular,
62         -- sim.colorcomponent_emission,
63         -- sim.colorcomponent_auxillary}
64         --local rand_rgb = {math.random(), math.random(), math.random()}
65
66         --for i = 1, #colour_components, 1 do
67             -- sim.setShapeColor(spawn, nil, colour_components[i], rand_rgb)
68         --end
69
70         sim.wait(WAIT_TIME)
71         i = i + 1
72     end
73 end
74
75
76 function sysCall_cleanup()
77     -- Put some clean-up code here
78 end
79

```

Figura 1.14: Código del *child script* que crea cubos.

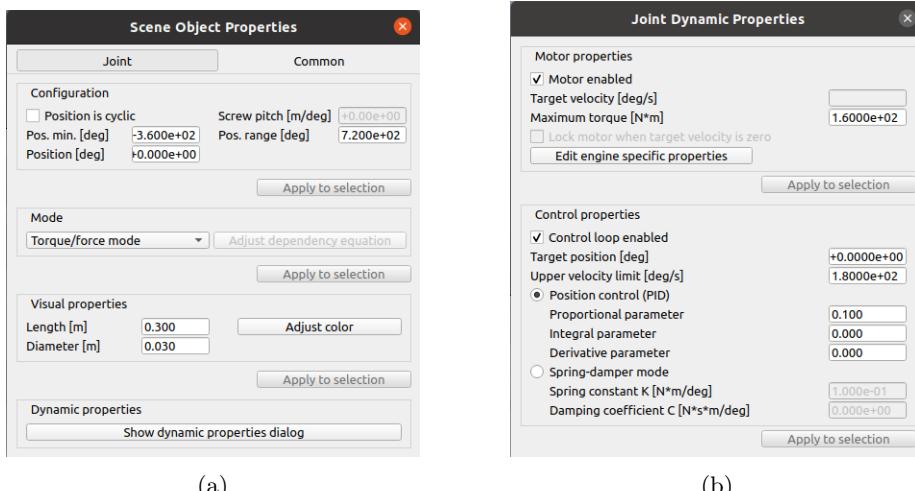


Figura 1.15: a) Propiedades de un objeto de tipo *joint* en Coppelia. b) Propiedades dinámicas de una articulación (*joint*).

Capítulo 2

Representación de transformaciones en Python

2.1. Introducción

El objetivo principal de esta práctica consiste en que el estudiante se familiarice con las diferentes herramientas para la representación de la rotación y la translación entre sistemas de referencia. Esta práctica permite al alumno afianzar los conceptos sobre matrices homogéneas, matrices de rotación y ángulos de Euler, así como la conversión entre estas formas de representar la posición y orientación. La representación combinada de la posición y la traslación se aborda mediante el uso de matrices de transformación homogénea, o bien mediante el uso combinado de un vector de posición y una matriz de rotación o ángulos de Euler.

En esta práctica tenéis una primera parte donde se recuerdan todos los fundamentos teóricos en relación con:

- Matrices de rotación.
- Matrices de transformación homogénea.
- Ángulos de Euler.

Esta parte de la práctica debe haber sido leída y comprendida con anterioridad, pues se tratan conceptos vistos durante las sesiones de teoría. Las actividades a realizar durante la práctica comienzan en el Apartado [2.14](#).

Seguidamente, se describe la implementación de todos estos conceptos en la librería pyARTE y cómo se pueden representar en Coppelia. Finalmente, se os proponen unos ejercicios a realizar con pyARTE y Coppelia.

2.2. Objetivos

En esta práctica se persiguen los siguientes **objetivos de aprendizaje**:

- El estudiante debe comprender la necesidad de utilizar sistemas de referencia para comandar la posición y orientación del extremo del robot. Por

ejemplo, definir el extremo del robot mediante la posición y orientación de un sistema de referencia respecto de la base del robot.

- El estudiante debe ser capaz de definir la orientación relativa entre diferentes sistemas de referencia utilizando:
 - Matrices de rotación.
 - Ángulos de Euler (pronúnciese, por favor: 'ɔɪlər).
- El estudiante debe ser capaz de definir la traslación y rotación entre sistemas de referencia utilizando:
 - Ángulos de Euler y un vector de posición.
 - Un vector de posición y una matriz de rotación.
 - Una matriz de transformación homogénea.
- Aplicar los conocimientos sobre matrices de transformación homogénea y matrices de rotación en un entorno de simulación.

Durante la práctica se realizarán las siguientes **actividades**:

- Definir en python matrices de transformación homogénea y matrices de rotación.
- Realizar conversiones entre matrices de rotación y ángulos de Euler usando pyARTE.
- Representar en el simulador Coppelia Sim sistemas de referencia usando matrices de transformación homogénea.
- Representar en el simulador Coppelia Sim sistemas de referencia definidos por una posición cartesiana y tres ángulos de Euler.
- Practicar con matrices de transformación homogéneas para representar la posición y orientación de sistemas de referencia. En concreto:
 - Dada una matriz de rotación, representar en Coppelia el sistema.
 - Dados tres ángulos de Euler, representar en Coppelia el sistema.
 - Dada una matriz de rotación, convertir a tres ángulos de Euler y representar la orientación en Coppelia.

2.3. ¿Para qué?

En un robot manipulador es necesario especificar la posición y orientación del extremo del robot (p. e., su pinza) respecto de algún sistema de referencia. Generalmente, se debe especificar la posición y orientación respecto del sistema de referencia situado en la base del robot. En ocasiones, también puede ser interesante especificar una posición y orientación respecto de un sistema de referencia ubicado en una mesa de trabajo del robot o cualquier otro elemento de la celda del robot. Especificar correctamente la posición y orientación es especialmente interesante en las siguientes aplicaciones, por ejemplo:

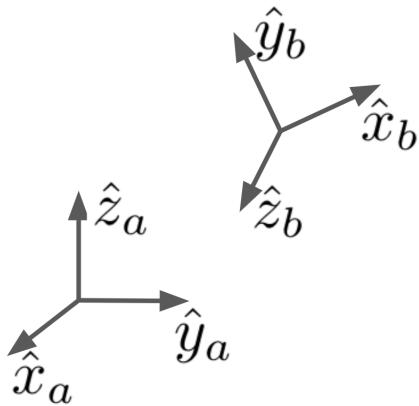


Figura 2.1: Dos sistemas de referencia con diferente orientación relativa.

- Cuando se desea especificar la posición y orientación de una pieza en el espacio de trabajo del robot. El robot necesita conocer esta posición y orientación para poder asir la pieza con precisión.
- Si es necesario definir un punto de soldadura en la carrocería de un vehículo.
- En el caso en el que se indica un conjunto de puntos y orientaciones para realizar una soldadura continua de dos chapas de acero.

2.4. Matrices de rotación

La orientación de un sistema de coordenadas b en relación con un sistema de coordenadas a se puede realizar al expresar los vectores del sistema b ($\hat{x}_b, \hat{y}_b, \hat{z}_b$) en términos de los vectores del sistema a ($\hat{x}_a, \hat{y}_a, \hat{z}_a$). Esta orientación se puede definir mediante la matriz de rotación ${}^a R_b$ (léase: matriz de rotación de a a b), cuyos componentes son el producto escalar de los vectores de ambos sistemas de coordenadas (podemos pensar que los vectores de b se proyectan sobre los vectores de a):

$${}^a R_b = \begin{pmatrix} \hat{x}_b \hat{x}_a & \hat{y}_b \hat{x}_a & \hat{z}_b \hat{x}_a \\ \hat{x}_b \hat{y}_a & \hat{y}_b \hat{y}_a & \hat{z}_b \hat{y}_a \\ \hat{x}_b \hat{z}_a & \hat{y}_b \hat{z}_a & \hat{z}_b \hat{z}_a \end{pmatrix} \quad (2.1)$$

La Figura 2.1 muestra un ejemplo de sistemas de referencia definidos por vectores ($\hat{x}_a, \hat{y}_a, \hat{z}_a$) y ($\hat{x}_b, \hat{y}_b, \hat{z}_b$).

2.5. Propiedades de una matriz de rotación

Suponga que una matriz de rotación se expresa mediante tres vectores columna:

$$R = (\hat{x} \quad \hat{y} \quad \hat{z}) = \begin{pmatrix} x_i & y_i & z_i \\ x_j & y_j & z_j \\ x_k & y_k & z_k \end{pmatrix} \quad (2.2)$$

Si R es una matriz de rotación, entonces sus vectores deben ser mutuamente ortogonales, es decir:

$$\hat{x}^T \hat{y} = 0 \quad \hat{x}^T \hat{z} = 0 \quad \hat{y}^T \hat{z} = 0 \quad (2.3)$$

Y, también, cada vector debe tener norma unitaria:

$$\hat{x}^T \hat{x} = 1 \quad \hat{y}^T \hat{y} = 1 \quad \hat{z}^T \hat{z} = 1 \quad (2.4)$$

Todas estas propiedades se cumplen, también, si definimos R en base a unos vectores fila. En base a estas propiedades (ortonormalidad y unidad), se deriva, claramente que, en una matriz de rotación:

$$R^T R = R R^T = I \quad (2.5)$$

Es decir:

$$R^T R = \begin{pmatrix} \hat{x}^T \\ \hat{y}^T \\ \hat{z}^T \end{pmatrix} (\hat{x} \quad \hat{y} \quad \hat{z}) \quad (2.6)$$

O bien:

$$R^T R = \begin{pmatrix} x_i & x_j & x_k \\ y_i & y_j & y_k \\ z_i & z_j & z_k \end{pmatrix} \begin{pmatrix} x_i & y_i & z_i \\ x_j & y_j & z_j \\ x_k & y_k & z_k \end{pmatrix} = \begin{pmatrix} \hat{x}^T \hat{x} & \hat{x}^T \hat{y} & \hat{x}^T \hat{z} \\ \hat{y}^T \hat{x} & \hat{y}^T \hat{y} & \hat{y}^T \hat{z} \\ \hat{z}^T \hat{x} & \hat{z}^T \hat{y} & \hat{z}^T \hat{z} \end{pmatrix} \quad (2.7)$$

Con lo que, aplicando la ortonormalidad de filas y columnas y considerando su norma unitaria, tenemos:

$$R^T R = R R^T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (2.8)$$

Dado que se cumple que $R^T R = R R^T = I$, podemos multiplicar por R^{-1} por la izquierda y ver que:

$$R^T R R^{-1} = I R^{-1} \quad (2.9)$$

Con lo que:

$$R^T = R^{-1} \quad (2.10)$$

Es decir, la transpuesta de una matriz de rotación equivale a su inversa. Finalmente, en un sistema de referencia dextrógiro, debe ocurrir:

$$\det(R) = 1 \quad (2.11)$$

Mientras que, si el sistema de referencia es levógiro, entonces:

$$\det(R) = -1 \quad (2.12)$$

Normalmente, en el ámbito de la Ingeniería, se utilizan sistemas de referencia dextrógiros. En los que ocurre que:

$$\hat{x} \times \hat{y} = \hat{z} \quad (2.13)$$

En concreto, se considera que estas matrices de rotación pertenecen al grupo ortonormal especial $SO(m)$ (*Special Orthonormal group*) cuando ocurre que $\det(R) = 1$. En el caso de las rotaciones en el espacio tendremos $m = 3$ y hablaremos de $SO(3)$. Cuando trabajamos con rotaciones en el plano, tendremos $m = 2$ y $SO(2)$.

Finalmente, para cualquier vector \vec{u} , si R es una matriz de rotación:

$$\vec{v} = R\vec{u} \quad (2.14)$$

El vector \vec{v} es una versión rotada de \vec{u} y, además, $\|\vec{u}\| = \|\vec{v}\|$, ya que el módulo del vector \vec{u} no cambia debido a la rotación.

2.6. Matrices de rotación elementales

Estas matrices de rotación se generan al realizar giros fundamentales sobre uno de los ejes del sistema de referencia. Se definen las siguientes matrices de rotación elementales:

- Giro de α (rad) sobre el eje X:

$$R(\alpha, \hat{x}) = R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha \\ 0 & \sin \alpha & \cos \alpha \end{pmatrix} \quad (2.15)$$

- Giro de β (rad) sobre el eje Y:

$$R(\beta, \hat{y}) = R_y = \begin{pmatrix} \cos \beta & 0 & \sin \beta \\ 0 & 1 & 0 \\ -\sin \beta & 0 & \cos \beta \end{pmatrix} \quad (2.16)$$

- Giro de γ (rad) sobre el eje Z:

$$R(\gamma, \hat{z}) = R_z = \begin{pmatrix} \cos \gamma & -\sin \gamma & 0 \\ \sin \gamma & \cos \gamma & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (2.17)$$

2.7. Matrices de transformación homogénea

Dado un vector $\vec{p} = (p_x, p_y, p_z)^T$, se define el mismo vector \vec{p} en coordenadas homogéneas como $\vec{p} = (\omega p_x, \omega p_y, \omega p_z, \omega)^T$. En Robótica de tipo serie, generalmente, no se producen cambios de escala, con lo que las coordenadas homogéneas se definen con $\omega = 1$ y, por tanto: $\vec{p} = (p_x, p_y, p_z, 1)^T$. El uso de estas coordenadas homogéneas nos permite utilizar transformaciones homogéneas que encapsulan la translación y la rotación. Se define, por tanto, una matriz de transformación homogénea como:

$$T = \begin{pmatrix} R & \vec{t} \\ \vec{0} & 1 \end{pmatrix} \quad (2.18)$$

Donde R es una matriz de rotación y $\vec{t} = (t_x, t_y, t_z)^T$ un vector de traslación. En base a esta definición, podemos indicar matrices homogéneas de rotación pura:

- Giro de α (rad) sobre el eje X:

$$T(\alpha, \hat{x}) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha & 0 \\ 0 & \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (2.19)$$

- Giro de β (rad) sobre el eje Y:

$$T(\beta, \hat{y}) = \begin{pmatrix} \cos \beta & 0 & \sin \beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \beta & 0 & \cos \beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (2.20)$$

- Giro de γ (rad) sobre el eje Z:

$$T(\gamma, \hat{z}) = \begin{pmatrix} \cos \gamma & -\sin \gamma & 0 & 0 \\ \sin \gamma & \cos \gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (2.21)$$

Y también matrices de translación puras, si tienen esta forma:

$$T = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (2.22)$$

También podemos considerar otros ejemplos de matrices homogéneas que combinan giros sobre alguno de los ejes y un vector de translación \vec{t} :

$$T(\alpha, \hat{x}, \vec{t}) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & \cos \alpha & -\sin \alpha & t_y \\ 0 & \sin \alpha & \cos \alpha & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (2.23)$$

$$T(\beta, \hat{y}, \vec{t}) = \begin{pmatrix} \cos \beta & 0 & \sin \beta & t_x \\ 0 & 1 & 0 & t_y \\ -\sin \beta & 0 & \cos \beta & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (2.24)$$

$$T(\gamma, \hat{z}, \vec{t}) = \begin{pmatrix} \cos \gamma & -\sin \gamma & 0 & t_x \\ \sin \gamma & \cos \gamma & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (2.25)$$

Finalmente, veremos un ejemplo práctico. En la Figura 2.2a se representa un sistema A (base) y un sistema móvil B . La transformación (traslación pura) entre ambos sistemas de referencia se representa mediante la matriz:

$${}^A T_B = \begin{pmatrix} 1 & 0 & 0 & 6 \\ 0 & 1 & 0 & -5 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

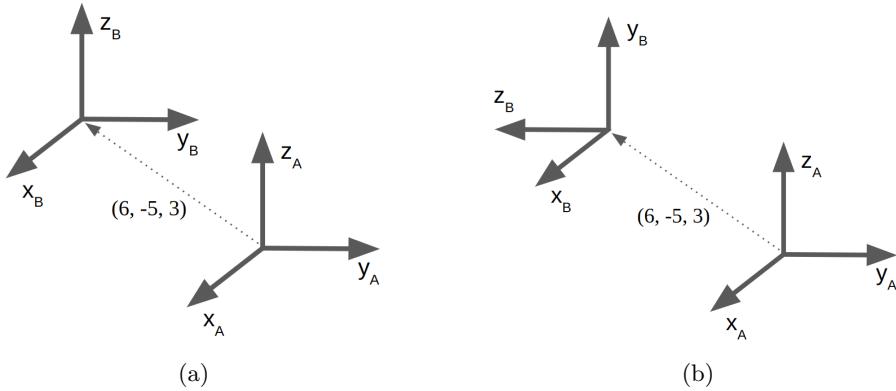


Figura 2.2: a) Translación pura entre dos sistemas A y B . b) Rotación seguida de translación entre dos sistemas A y B .

En la Figura 2.2b se presenta un segundo ejemplo que combina rotación y translación entre un sistema A (base) y un sistema móvil B . La transformación entre ambos sistemas de referencia se representa mediante la matriz:

$${}^A T_B = \begin{pmatrix} 1 & 0 & 0 & 6 \\ 0 & 0 & -1 & -5 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2.8. Rotación de un vector

Considere que tenemos un vector ${}^b \vec{p} = {}^b(p_x, p_y, p_z)$ expresado en el sistema de coordenadas b . Esta notación es común en Robótica: el superíndice a la izquierda indica el sistema de referencia en el que se expresan las coordenadas del vector. Suponga que el sistema b se ha rotado con respecto al a según la matriz ${}^a R_b$. El vector ${}^a \vec{p} = {}^a(p_x, p_y, p_z)$ expresado en coordenadas del sistema a se calcula como:

$${}^a \vec{p} = {}^a R_b {}^b \vec{p} \quad (2.26)$$

Recuerde que los superíndices (a la izquierda), indican el sistema de coordenadas en el que se expresan los vectores. Recuerde, también, la “regla de cancelación de superíndices”, que nos permite deducir que la cantidad a la derecha está expresada en el sistema de coordenadas a . Por ejemplo, para el caso anterior:

$${}^a \vec{p} = {}^a R_b {}^b \vec{p} = {}^a R \vec{p} \quad (2.27)$$

Con lo que debemos deducir que el vector rotado ${}^a R \vec{p}$ se encuentra expresado en coordenadas de del sistema a .

2.9. Transformación de las coordenadas de un punto

Suponga que tiene un sistema fijo que denominaremos sistema de referencia base A . Suponga que tenemos un sistema de referencia B que ha sido rotado y trasladado mediante la matriz ${}^A T_B$. Si conocemos las coordenadas de un punto expresadas en el sistema B : ${}^B \vec{p}$, podemos calcular las coordenadas de ese mismo punto en el sistema A como:

$${}^A \vec{p} = {}^A T_B {}^B \vec{p} \quad (2.28)$$

Supongamos que dos sistemas A y B están relacionados por la transformación ${}^A T_B$:

$${}^A T_B = \begin{pmatrix} 1 & 0 & 0 & 6 \\ 0 & 0 & -1 & -5 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Supongamos que conocemos las coordenadas de un punto en el sistema de referencia B : ${}^B \vec{p} = (-2, 3, -1, 1)^T$ (en coordenadas homogéneas). Ese mismo punto tendrá coordenadas en el sistema A :

$${}^A p = \begin{pmatrix} 1 & 0 & 0 & 6 \\ 0 & 0 & -1 & -5 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} -2 \\ 3 \\ -1 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ -4 \\ 6 \\ 1 \end{pmatrix}$$

Esta transformación entre las coordenadas de un punto en dos sistemas de referencia se aclara en la Figura 2.3.

También, si ahora suponemos conocidas las coordenadas de un punto en el sistema A ${}^A \vec{p} = (4, -4, 6, 1)^T$ y calculamos la matriz $({}^A T_B)^{-1} = {}^B T_A$

$${}^B \vec{p} = ({}^A T_B)^{-1} {}^A \vec{p} = \begin{pmatrix} -2 \\ 3 \\ -1 \\ 1 \end{pmatrix} \quad (2.29)$$

como era de esperar.

2.10. Ángulos de Euler

Los ángulos de Euler constituyen una forma alternativa para representar la orientación relativa entre dos sistemas de referencia. Ideados inicialmente por Herr Leonhard Paul Euler, constituyen un conjunto de tres coordenadas angulares (α, β, γ) que sirven para especificar la orientación de un sistema de referencia móvil de ejes ortogonales, respecto a otro sistema de referencia de ejes ortogonales fijo.

Cuando se usan los ángulos de Euler en cualquier aplicación, es muy importante establecer la convención de ejes que se está usando, en concreto, es importante definir:

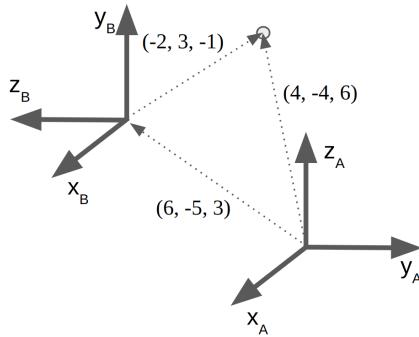


Figura 2.3: Transformación en las coordenadas de un mismo punto en dos sistemas de referencia A y B .

- Si los giros se realizan sobre los ejes móviles o sobre los ejes fijos del sistema.
- El orden y los giros sobre cada eje, por ejemplo, XYZ, ZYZ, ZYX, etc.

En concreto, existen los estándares siguientes:

- Proper Euler angles (ZXZ, XYX, YZY, ZYZ, XZX, YXY).
- Tait–Bryan angles (XYZ, YZX, ZXY, XZY, ZYX, YXZ).

Note que, en el primer caso se repite alguno de los ejes en las rotaciones, mientras que, en el segundo caso (ángulos Tait-Bryan) no se repiten. Lógicamente, cualquier combinación en la que se repita el mismo eje en rotaciones consecutivas no es válido (pues se pierde un grado de libertad).

2.11. Por qué utilizar ángulos de Euler

Es importante entender que especificar la orientación de la pinza del robot usando una matriz de rotación no siempre resultará conveniente, pues es necesario escribir 9 valores. Esto puede resultar tedioso si se está realizando un programa para un robot industrial. Además, la matriz de rotación R debe cumplir con las propiedades enunciadas en el Apartado 2.5, en consecuencia, con frecuencia se deben escribir varios decimales para especificar R correctamente. En cambio, resulta sencillo escribir 3 ángulos de Euler para especificar una orientación. No obstante, en ocasiones no es sencillo establecer “de cabeza” los ángulos de Euler necesarios que indican la orientación dada por una matriz R , de ahí que sea conveniente deducir unas ecuaciones que realicen esta conversión.

2.12. Conversión de ángulos de Euler a matriz de rotación

Si no se dice lo contrario, durante las prácticas se empleará la convención de ángulos de Euler XYZ sobre ejes móviles, pues es el convenio utilizado en el

simulador Coppelia. No obstante, es importante que el lector recuerde que las convenciones de ángulos ZYZ y ZYX son también frecuentes en la Ciencia y la Ingeniería.

Definimos, a continuación, la convención de ángulos de Euler XYZ en ejes móviles. Para ello, considere que primero realizamos una rotación de α (rad) sobre el eje X. A continuación, sobre el sistema de referencia resultante se realiza una rotación de β (rad) sobre el eje Y. Finalmente, sobre el sistema resultante realizamos una rotación de γ (rad) sobre el eje Z. La matriz de rotación resultante, como composición de rotaciones, será:

$$R = R(\alpha, \vec{x})R(\beta, \vec{y})R(\gamma, \vec{z}) \quad (2.30)$$

Por tanto, es sencillo convertir desde cualquier representación de ángulos de Euler a una matriz de rotación. Por ejemplo, si tenemos una convención de ángulos XZY sobre ejes móviles, tendremos que R se calcula como:

$$R = R(\alpha, \vec{x})R(\beta, \vec{z})R(\gamma, \vec{y}) \quad (2.31)$$

Si, por ejemplo, hablamos de ángulos de Euler XZY expresados sobre ejes fijos, es fácil deducir que la matriz resultante R se puede calcular como:

$$R = R(\alpha, \vec{y})R(\beta, \vec{z})R(\gamma, \vec{x}) \quad (2.32)$$

2.13. Conversión de matriz de rotación a ángulos de Euler

Igualmente, nos podría resultar interesante realizar el procedimiento contrario, si conocemos una matriz de rotación R , encontrar los ángulos de Euler que permiten obtener esa orientación. En este apartado se deducen las ecuaciones que permiten realizar esta conversión y más adelante se presentarán las clases de pyARTE que permiten realizarla.

Con todo esto, considere que conoce la matriz de rotación R . Imagine, ahora, que está interesado en calcular tres giros sobre tres ejes no consecutivos, que permiten llegar a la misma matriz R . Considere:

- Que denotamos $e = (\alpha, \beta, \gamma)$ a los ángulos de Euler.
- Que los ejes móviles sobre los que se realiza la rotación son XYZ.

Volviendo al punto anterior, la matriz de rotación R , según esta convención, será:

$$R = R(\alpha, \vec{x})R(\beta, \vec{y})R(\gamma, \vec{z}) \quad (2.33)$$

Si usamos las definiciones de estas rotaciones que se introdujeron en el Apartado 2.6, queda, tras multiplicar las matrices:

$$R = \begin{pmatrix} c_\beta c_\gamma & -c_\beta s_\gamma & s_\beta \\ c_\alpha s_\gamma + s_\alpha s_\beta c_\gamma & c_\alpha c_\gamma - s_\alpha s_\beta s_\gamma & -s_\alpha c_\beta \\ s_\alpha s_\gamma - c_\alpha s_\beta c_\gamma & s_\alpha c_\gamma + c_\alpha s_\beta s_\gamma & c_\alpha c_\beta \end{pmatrix} \quad (2.34)$$

donde $\cos(\theta) = c_\theta$ y $\sin(\theta) = s_\theta$. La matriz R da, como resultado, un conjunto de ecuaciones no lineales donde todos los elementos r_{ij} de la matriz son conocidos. En concreto, tenemos 9 ecuaciones no lineales sobre las 3 variables (α, β, γ) . Resolver estas ecuaciones no es difícil y se indicará en los apartados siguientes.

2.13.1. Caso normal

Podemos comenzar hallando un primer valor de β :

$$\beta = \arcsin(r_{13}) \quad (2.35)$$

Conocido β hallamos un valor alternativo $\beta' = \pi - \beta$, puesto que sabemos que $\sin(\beta) = \sin(\pi - \beta)$. A continuación, continuamos resolviendo el resto de ángulos. Si $\cos \beta \neq 0$ (que ocurre cuando $\beta \neq \pm\pi/2$), entonces:

$$\frac{-r_{12}}{r_{11}} = \frac{\cos \beta \sin \gamma}{\cos \beta \cos \gamma} = \tan \gamma \quad (2.36)$$

$$\gamma = \arctan\left(\frac{-r_{12}}{r_{11}}\right) \quad (2.37)$$

En este momento es importante hacer una puntualización: la función arctan (arcotangente) devuelve un ángulo en el intervalo $[-\pi/2, \pi/2]$. Es decir, esta función supone que el punto definido por el $\cos \gamma$ y el $\sin \gamma$ debe estar en cuadrante I, o bien en el IV. Si deseamos obtener un valor de γ que esté definido en el intervalo $[-\pi, \pi]$ (es decir, en los cuatro cuadrantes: I, II, III y IV), entonces debemos utilizar la función atan2. La función $\text{atan2}(\sin \theta, \cos \theta)$, o arco-tangente de dos argumentos (de ahí el “2”), es una función matemática que se utiliza para obtener, de forma inequívoca, el valor de un ángulo θ a partir de su seno y su coseno. Dicha función devolverá el único ángulo θ en el rango $[-\pi, \pi]$ que tenga el seno y el coseno deseados.

La función $\text{atan2}(\sin \theta, \cos \theta)$ también puede definirse como aquélla que devuelve la fase θ del número complejo $\cos \theta + i \sin \theta$.

Con todo esto, podemos volver a escribir la solución de γ de la Ecuación (2.36) como:

$$\gamma = \text{atan2}\left(\frac{-r_{12}}{\cos \beta}, \frac{r_{11}}{\cos \beta}\right) \quad (2.38)$$

Nótese que existe una solución alternativa γ' si utilizamos el otro valor hallado para β : β' .

$$\gamma' = \text{atan2}\left(\frac{-r_{12}}{\cos \beta'}, \frac{r_{11}}{\cos \beta'}\right) \quad (2.39)$$

Finalmente, α se calcula de forma similar:

$$\alpha = \arctan\left(-\frac{r_{23}}{\cos \beta}, \frac{r_{33}}{\cos \beta}\right) \quad (2.40)$$

Finalmente, otra solución α' se calcula:

$$\alpha' = \arctan\left(-\frac{r_{23}}{\cos \beta'}, \frac{r_{33}}{\cos \beta'}\right) \quad (2.41)$$

2.13.2. Caso degenerado

Analizamos ahora el caso especial en el que $r_{13} = \pm 1$. En esta situación, por tanto, $\sin \beta = \pm 1$ y esto implica que $\cos \beta = 0$ (que ocurre cuando $\beta = \pm\pi/2$). Nos referiremos a esta situación como el caso singular, o bien el caso degenerado.

Veremos que, en esta situación, no es posible determinar todos los ángulos de Euler, sino que existen infinitas soluciones. Por tanto, partimos, igualmente, de la siguiente matriz de rotación R :

$$R = \begin{pmatrix} c_\beta c_\gamma & -c_\beta s_\gamma & s_\beta \\ c_\alpha s_\gamma + s_\alpha s_\beta c_\gamma & c_\alpha c_\gamma - s_\alpha s_\beta s_\gamma & -s_\alpha c_\beta \\ s_\alpha s_\gamma - c_\alpha s_\beta c_\gamma & s_\alpha c_\gamma + c_\alpha s_\beta s_\gamma & c_\alpha c_\beta \end{pmatrix} \quad (2.42)$$

En este caso, cuando $\sin \beta = \pm 1$, entonces $\cos \beta = 0$, con lo que la matriz R tiene la forma:

$$R = \begin{pmatrix} 0 & 0 & \pm 1 \\ r_{21} & r_{22} & 0 \\ r_{31} & r_{32} & 0 \end{pmatrix}$$

Vemos que esta situación es sencilla de detectar, pues el elemento $|r_{13}| = 1$. En la práctica, se utilizará un umbral $\epsilon > 0$ y se detectará este caso siempre que $1 - |r_{13}| < \epsilon$. En este caso, nos damos cuenta de que los cocientes $r_{12}/\cos(\beta)$, $r_{11}/\cos(\beta)$ y, $r_{23}/\cos(\beta)$ y $r_{33}/\cos(\beta)$ que intervienen en las ecuaciones derivadas en el Apartado 2.13.1 no están definidos. En este caso, hacemos:

- Si $r_{13} = \sin \beta = 1$, entonces $\beta = \pi/2$.
- En caso contrario: $r_{13} = \sin \beta = -1$ y $\beta = -\pi/2$.

Para el primer caso, sustituimos $\sin \beta = 1$ en la matriz R , obteniendo:

$$R = \begin{pmatrix} 0 & 0 & 1 \\ c_\alpha s_\gamma + s_\alpha c_\gamma & c_\alpha c_\gamma - s_\alpha s_\gamma & 0 \\ s_\alpha s_\gamma - c_\alpha c_\gamma & c_\alpha s_\gamma + s_\alpha c_\gamma & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ \sin(\alpha + \gamma) & \cos(\alpha + \gamma) & 0 \\ -\cos(\alpha + \gamma) & \sin(\alpha + \gamma) & 0 \end{pmatrix}$$

De esta matriz vemos, que en este caso degenerado, solamente podemos resolver una combinación de $\alpha + \gamma$ que dé como resultado la matriz deseada R :

$$r_{21} = \sin(\alpha + \gamma) \quad (2.43)$$

$$r_{22} = \cos(\alpha + \gamma) \quad (2.44)$$

$$\alpha + \gamma = \text{atan2}(r_{21}, r_{22}) \quad (2.45)$$

Así, pues, observamos que existe un número infinito de combinaciones de $\alpha + \gamma$ que cumplen la Ecuación (2.45). Para obtener una solución, por ejemplo, podemos forzar arbitrariamente $\alpha = 0$, con lo que la solución queda: $e = (\alpha = 0, \beta = \pi/2, \gamma = \text{atan2}(r_{21}, r_{22}))$.

Para el segundo caso, sustituimos $\sin \beta = -1$ en la matriz R , quedando:

$$R = \begin{pmatrix} 0 & 0 & -1 \\ c_\alpha s_\gamma - s_\alpha c_\gamma & c_\alpha c_\gamma + s_\alpha s_\gamma & 0 \\ c_\alpha c_\gamma + s_\alpha s_\gamma & s_\alpha c_\gamma - c_\alpha s_\gamma & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & -1 \\ -\sin(\alpha - \gamma) & \cos(\alpha - \gamma) & 0 \\ \cos(\alpha - \gamma) & \sin(\alpha - \gamma) & 0 \end{pmatrix}$$

De forma similar, resolvemos:

$$r_{32} = \sin(\alpha - \gamma) \quad (2.46)$$

$$r_{22} = \cos(\alpha - \gamma) \quad (2.47)$$

$$\alpha - \gamma = \text{atan2}(r_{32}, r_{22}) \quad (2.48)$$

De nuevo obtenemos un número infinito de soluciones que cumplen con la Ecuación (2.48). Para obtener una solución, por ejemplo, podemos forzar arbitrariamente $\alpha = 0$, con lo que la solución queda: $e = (\alpha = 0, \beta = -\pi/2, \gamma = -\text{atan2}(r_{32}, r_{22}))$.

2.14. Transformaciones en pyARTE

En este apartado se describen el código y clases que tenéis disponibles en pyARTE para manejar todos los conceptos mencionados anteriormente:

- Vectores: clase `Vector()`.
- Matrices de rotación: clase `RotationMatrix()`.
- Matrices de transformación homogénea: clase `HomogeneousMatrix()`.
- Ángulos de Euler: clase `Euler()`.

Además, la librería permite realizar operaciones con matrices de rotación y matrices de transformación homogénea con una sintaxis muy sencilla (similar a la de Matlab, pero usando python). En concreto, estas clases simplifican la sintaxis utilizada en la librería `numpy`. Aunque simples, las clases de pyARTE no disponen de toda la versatilidad y funciones de la librería `numpy`. Por esta razón, al final de esta práctica se dan algunas nociones sobre el uso de la librería `numpy`.

Finalmente, las clases de pyARTE permiten la conversión entre diferentes representaciones de la orientación. Por ejemplo, conocida la matriz de rotación podemos hallar los ángulos de Euler y, al revés, conocidos los ángulos de Euler, podemos hallar una matriz de rotación.

2.14.1. Vectores en pyARTE

El siguiente código presenta un ejemplo de utilización de la clase `Vector` en pyARTE (`practicals/transformations/vectors.py`).

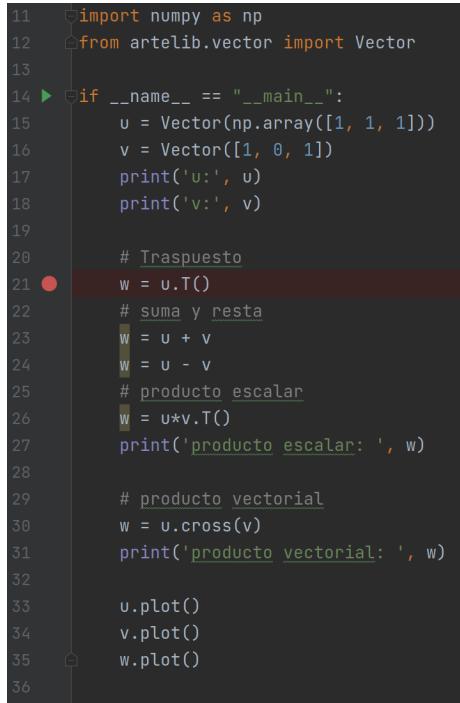
```
import numpy as np
from artelib.vector import Vector

if __name__ == "__main__":
    u = Vector(np.array([1, 1, 1]))
    v = Vector([1, 0, 1])
    print('u:', u)
    print('v:', v)

    # Traspuesto
    w = u.T()
    # suma y resta
    w = u + v
    w = u - v
    # producto escalar
    w = u*v.T()
    print('producto escalar: ', w)

    # producto vectorial
    w = u.cross(v)
    print('producto vectorial: ', w)

    u.plot()
    v.plot()
    w.plot()
```



```

11  import numpy as np
12  from arteleib.vector import Vector
13
14  if __name__ == "__main__":
15      u = Vector(np.array([1, 1, 1]))
16      v = Vector([1, 0, 1])
17      print('u:', u)
18      print('v:', v)
19
20      # Traspuesto
21  ●   w = u.T()
22      # suma y resta
23      w = u + v
24      w = u - v
25      # producto escalar
26      w = u*v.T()
27      print('producto escalar: ', w)
28
29      # producto vectorial
30      w = u.cross(v)
31      print('producto vectorial: ', w)
32
33      u.plot()
34      v.plot()
35      w.plot()
36

```

Figura 2.4: Modo “debug” en el fichero `vectors.py`.

En concreto un objeto de tipo `Vector()` se puede crear con una lista o un array de `numpy`. El método `.T()` transpone el vector. Finalmente, el método `plot()` lo representa en una ventana gráfica. La clase soporta el producto escalar y la suma y resta de vectores.

Ejercicio 2.14.1: Vectores en pyARTE

Ejecute el script `practicals/transformations/vectors.py`. Modifique las coordenadas de los vectores y observe el resultado en las figuras gráficas.

2.14.2. El modo “debug”

Es, en extremo, conveniente, ser capaz de depurar cualquier código en el que se esté trabajando. Si se usa Pycharm, es, en extremo, sencillo, depurar el código de python. Haga click izquierdo sobre la barra junto al código y sitúe un punto rojo (Figura 2.4). Seguidamente, haga click con el botón derecho sobre el fichero `vectors.py` y seleccione la opción ‘‘Debug’’ `vectors`. Pycharm arrancará el script de python y lo detendrá en el primer punto rojo que se haya indicado. En el menú que aparece abajo (Figura 2.5), aparecen las opciones de *step over*, *step into* que permiten ejecutar la siguiente línea o entrar en alguna función, respectivamente.



Figura 2.5: Modo “debug” en el fichero `vectors.py`.

2.14.3. Matrices de rotación en pyARTE

La librería pyARTE representa las matrices usando la clase `RotationMatrix()`. Esta clase simplifica el cálculo con matrices (es parecido a la sintaxis en Matlab) y permite plotear los sistemas de referencia sin necesidad de contar con Coppelia. Deberá cerrar cada ventana gráfica para que el programa continúe.

Se representa, a continuación, un código que ejemplifica el uso de la clase `RotationMatrix` en pyARTE.

```
import numpy as np
from arteleib.rotationmatrix import RotationMatrix, Rx, Ry, Rz
from arteleib.vector import Vector

if __name__ == "__main__":
    R = RotationMatrix(3)
    R.plot('Identity 3x3')

    R = RotationMatrix([[0, 0, -1], [0, 1, 0], [1, 0, 0]])
    R.plot()

    Ra = Rx(np.pi/4)
    Rb = Ry(np.pi/4)
    Rc = Rz(np.pi/4)

    Ra.plot(title='Rx pi/4')
    Rb.plot(title='Ry pi/4')
    Rc.plot(title='Rz pi/4')

    R = Ra*Rb*Rc
    R.plot(title='Three consecutive rotations of pi/4 along XYZ')

    R = R.inv()
    print('Inverse matrix (transpose): ')
    print(R)
    print('Determinant: ', R.det())

    # Rotación de un vector
    u = Vector(np.array([1, 1, 1]))
    u = R*u.T()
    print(u)
```

Como se puede observar, es posible crear matrices a partir de los elementos de la matriz. También se pueden crear matrices Rx, Ry y Rz como rotaciones básicas sobre los ejes X , Y y Z , respectivamente, usando las Ecuaciones (2.15), (2.16) y (2.17).

Vemos, también, que el método `inv()` calcula la inversa de la matriz (su traspuesta) y el método `det()` calcula su determinante. Las matrices se pueden multiplicar usando el operador `*`. Finalmente, para rotar un vector, podemos multiplicar una matriz por un `Vector`. El método `plot()` representa los ejes

de la rotación en un gráfico. Deberá cerrar cada ventana gráfica para que el programa continúe.

Ejercicio 2.14.2: Matrices de rotación en pyARTE

Ejecute el script `practicals/transformations/rotation_matrices.py`. Modifique el giro `np.pi/4` y observe el resultado en la figura gráfica. Utilice el modo “debug” si te surge cualquier duda.

2.14.4. Matrices de transformación homogénea en pyARTE

La librería pyARTE representa las matrices de transformación homogénea usando la clase `HomogeneousMatrix()`. Esta clase simplifica el cálculo con matrices (es parecido a la sintaxis en Matlab) y permite representar sistemas de referencia trasladados y girados sin necesidad de contar con Coppelia.

Se representa, a continuación, un código que ejemplifica el uso de la clase `HomogeneousMatrix` en pyARTE.

```
import numpy as np
from artelib.rotationmatrix import Rx
from artelib.vector import Vector
from artelib.homogeneousmatrix import HomogeneousMatrix
from artelib.euler import Euler

if __name__ == "__main__":
    # Diferentes formas de crear una matriz homogénea
    T1 = HomogeneousMatrix()
    T1.plot('Identity HomogeneousMatrix', block=True)

    T2 = HomogeneousMatrix([[1, 0, 0, 0.5],
                           [0, 1, 0, 0.7],
                           [0, 0, 1, 0.8],
                           [0, 0, 0, 1]])
    T2.plot('Transformation T2')

    print('Position: ', T2.pos())
    print('Rotation: ', T2.R())

    position = Vector([1, 2, 3])
    rotation_matrix = Rx(np.pi/4)
    T3 = HomogeneousMatrix(position, rotation_matrix)
    T3.plot('Transformation T3')

    position = Vector([1, 2, 3])
    orientation = Euler([np.pi/4, np.pi/4, np.pi/4])
    T4 = HomogeneousMatrix(position, orientation)
    T4.plot('Transformation T4')

    T5 = T2*T3
    T5 = T5.inv()
    T5.plot('Transformation T4')

    # Transformar un vector
    u = Vector(np.array([1, 2, 3, 1]))
    u.plot('A 3D vector')
    u = T5*u.T()
    print(u)
```

```
u.plot('A vector transformed by T5')
```

En el código anterior es importante que se observe que un objeto de tipo `HomogeneousMatrix` se puede crear a partir de:

- Un elemento de tipo `Vector`. En este caso, se considera que $R = I$ (transformación T1).
- Los elementos del array que define la matriz (T2).
- Un `Vector` y una matriz de rotación (T3).
- Un `Vector` y un objeto de tipo `Euler`, según se define en el Apartado 2.15 (T4).

En la clase `HomogeneousMatrix` el método `pos()` permite obtener el vector de traslación de la matriz, mientras que el método `R()` permite obtener la matriz de rotación. Se pueden hacer operaciones de multiplicación, inversa y determinante. Observe, finalmente, cómo se pueden multiplicar matrices de transformación homogénea y, también, la transformación de un vector por una matriz.

Ejercicio 2.14.3: Matrices de transformación homogénea en pyARTE

Ejecute: `practicals/transformations/homogeneous_matrices.py` y observe el resultado en la figura gráfica.

2.15. La clase Euler

En pyARTE se añade una clase importante: `Euler`. Esta clase permite la representación de la orientación mediante tres ángulos de Euler. ¿Por qué? o ¿para qué? nos resulta esto interesante. Por las siguientes razones:

- Porque es más fácil de escribir tres valores (α, β, γ) que una matriz de rotación completa (9 valores).
- Además, una matriz de rotación debe ser ortonormal, con lo, en ocasiones, deben escribirse un buen número de decimales de los elementos de la matriz.
- Los ángulos de Euler permiten realizar interpolaciones sencillas entre diferentes orientaciones que pueden ser de utilidad en algunas situaciones.

Además, la clase `Euler` permite la conversión sencilla entre algunas representaciones de la orientación que hemos visto, en concreto:

- Dados tres ángulos de Euler, permite calcular la matriz de orientación R .
- Dada una matriz de orientación R , es posible obtener dos conjuntos de ángulos de Euler (y ambos dan, como resultado R).

Importante: La clase Euler utiliza la convención XYZ de ángulos en ejes móviles para estas transformaciones. En el siguiente script de python se ejemplifica el uso de la clase `Euler()`: (`transformations/euler_conversions.py`).

```

import numpy as np
from artelib.euler import Euler
from artelib.rotationmatrix import RotationMatrix

if __name__ == '__main__':
    e = Euler([-np.pi/2, 0, 0])
    print('Euler angles XYZ:')
    print(e.abg)

    print('Conversion from Euler to a rotation matrix:')
    R = e.R()
    print('R\n', R)
    R.plot()

    # convert R to Euler angles
    print('Euler angles (XYZ) that yield R (e1, e2):')
    [e1, e2] = R.euler()
    print(e1.abg, e2.abg)
    print('Please check that R1, R2 and R are equal')
    R1 = e1.R()
    R2 = e2.R()
    print('R1:\n', R1)
    print('R2:\n', R2)

    # Convert any R to Euler angles
    R = RotationMatrix(np.array([[0, 0, -1], [0, 1, 0], [1, 0, 0]]))
    print('R\n', R)
    R.plot()
    print('Euler angles (XYZ) that yield R:')
    [e3, e4] = R.euler()
    print(e3.abg, e4.abg)
    print('Please check that R3, R4 and R are equal')
    R3 = e3.R()
    R4 = e4.R()
    print('R3:\n', R3)
    print('R4:\n', R4)

```

Nótese que `Euler([alpha, beta, gamma])` crea el objeto de la clase `Euler()`. Para convertir a una matriz de rotación, usando la Ecuación (2.33), se empleará el método `R()` de la clase. Al revés, si tenemos una matriz de rotación, podemos llamar al método `euler()` para obtener los ángulos de Euler. Note que `euler()` devuelve dos objetos de la clase `Euler()`, con lo que podemos convertir ambos, de nuevo, a un objeto de la clase `RotationMatrix()`.

Ejercicio 2.15.1: Conversión de angulos de Euler a matriz de rotación

En el código anterior, compare las definiciones de `R` con `R1`, `R2`, `R3` y `R4`.

Ejercicio 2.15.2: Conversión de angulos de Euler a matriz de rotación

Calcule las matrices de rotación definidas por los siguientes ángulos de Euler en convención intrínseca XYZ. Utilice el script `euler_conversions.py`.

- $e = (\pi/2, 0, \pi/2)$.
- $e = (\pi/4, \pi/4, \pi/4)$.
- $e = (\pi/2, \pi/2, -\pi/2)$.

Solución: Usando el código anterior, vemos que:

$$\begin{aligned} e = (\pi/2, 0, \pi/2) &\longrightarrow R = \begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \end{pmatrix} \\ e = (\pi/4, \pi/4, \pi/4) &\longrightarrow R = \begin{pmatrix} 0.5 & -0.5 & 0.707 \\ 0.853 & 0.146 & -0.5 \\ 0.146 & 0.853 & 0.5 \end{pmatrix} \\ e = (\pi/2, \pi/2, -\pi/2) &\longrightarrow R = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{pmatrix} \end{aligned}$$

2.15.1. Visualizar transformaciones en Coppelia Sim

En las clases `Vector`, `RotationMatrix`, `HomogeneousMatrix` y `Euler`, ya hemos utilizado el método `plot()` que permite visualizarlos en una ventana gráfica.

Ahora, presentamos al alumno la forma de dibujar estos elementos en Coppelia. Visualizar un elemento junto con un robot en una escena de Coppelia nos va a resultar muy útil y clarificador. En concreto, vamos a utilizar la clase `ReferenceFrame()` para visualizar las transformaciones en Coppelia. En concreto, esta clase nos permite representar un sistema de referencia en Coppelia. La apariencia de este sistema de referencia la podemos ver en la Figura 2.6, donde se representan tres sistemas de referencia. Observe el eje X (rojo), el Y (verde) y el Z (azul). Uno de los sistemas se coloca frente a la base del robot, el otro es relativo al extremo del robot (efector final) y el otro sistema de referencia móvil lo moveremos a placer en este apartado. Sobre este sistema de referencia vamos a poder:

- Cambiar la posición del origen del sistema de referencia.
- Especificar su orientación con:
 - Una matriz de rotación.
 - Tres ángulos de Euler (XYZ sobre ejes móviles).
- Especificar directamente posición y orientación con:
 - Una matriz homogénea.

- Un Vector y una matriz de rotación.
- Un Vector y un objeto Euler.

En la escena `irb140.ttt` y el script `transformations/reference_frames.py` se utiliza la clase `ReferenceFrame()` para representar la posición y orientación de un sistema de referencia. El uso básico de la clase se representa a continuación:

```
import numpy as np
from artelib.rotationmatrix import RotationMatrix
from artelib.vector import Vector
from artelib.euler import Euler
from artelib.homogeneousmatrix import HomogeneousMatrix
from robots.objects import ReferenceFrame
from robots.simulation import Simulation

if __name__ == "__main__":
    simulation = Simulation()
    simulation.start()
    frame = ReferenceFrame(simulation=simulation)
    frame.start()

    # Change position and orientation of the frame
    # using Vector and RotationMatrix
    position = Vector([0.5, 0, 0.3])
    orientation = RotationMatrix(np.array([[0, 1, 0],
                                           [-1, 0, 0],
                                           [0, 0, 1]]))

    frame.set_position(position)
    simulation.wait(50)
    frame.set_orientation(orientation)
    simulation.wait(50)
    # same as before
    frame.set_position_and_orientation(position, orientation)
    simulation.wait(50)

    # using Vector and Euler
    position = Vector([.6, 0, .6])
    orientation = Euler([np.pi/4, np.pi/4, np.pi/4])
    frame.set_position(position)
    simulation.wait(50)
    frame.set_orientation(orientation)
    simulation.wait(50)
    # same as before
    frame.set_position_and_orientation(position, orientation)
    simulation.wait(50)

    # using a HomogeneousMatrix
    T = HomogeneousMatrix([[0, 0, -1, 0.6],
                           [0, 1, 0, -0.3],
                           [1, 0, 0, 0.8],
                           [0, 0, 0, 1]])
    frame.set_position(position=T.pos())
    simulation.wait(50)
    frame.set_orientation(orientation=T.R())
    simulation.wait(50)
    frame.set_position_and_orientation(T)

simulation.stop()
```

En las funciones anteriores se le indica a Coppelia que modifique la posición y orientación del sistema de referencia. Observe que es posible indicar la posición (`frame.set_position`) mediante un `Vector`, una lista o un array de numpy. La

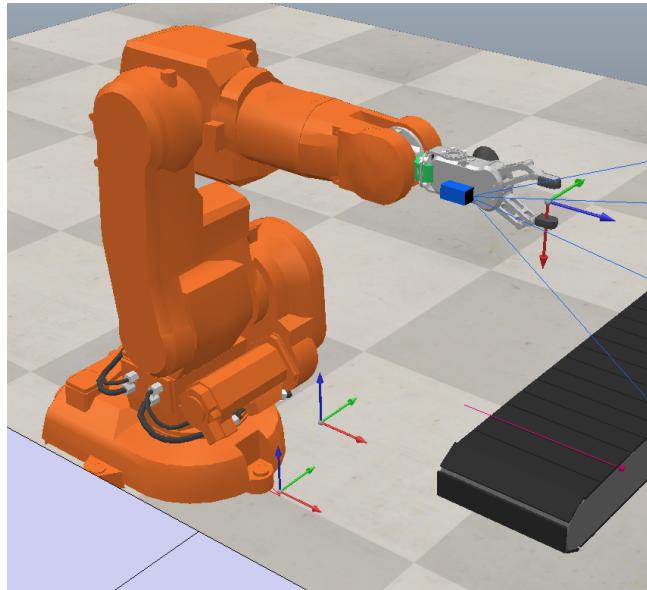


Figura 2.6: Sistemas de referencia en una escena de Coppelia. Fuente: Captura de pantalla sobre Coppelia Sim.

orientación se cambia con `frame.set_orientation`, especificando una matriz de rotación o bien un objeto de la clase `Euler`. Finalmente, se puede especificar posición y orientación simultáneamente con cualquiera de las modalidades vistas. Es importante utilizar el método (`wait()`) para darle tiempo a Coppelia a cambiar la posición y orientación del sistema de referencia.

Ejercicio 2.15.3: Mover un sistema de referencia en Coppelia

Realice los siguientes pasos:

- a) Abra la escena `irb140.ttt` y el script `transformations/reference_frames.py`.
- b) Podrá observar que se puede cambiar la posición y orientación del sistema de referencia usando un script de python.
- c) Dibuje los sistemas de referencia que corresponden con las siguientes matrices de transformación homogénea y observe el resultado en Coppelia.

$$T_a = \begin{pmatrix} 0 & 0 & -1 & 0.5 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0.3 \\ 0 & 0 & 0 & 1 \end{pmatrix} T_b = \begin{pmatrix} 0 & 0 & -1 & 1 \\ 0 & 1 & 0 & 2 \\ 1 & 0 & 0 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$T_c = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & -1 \\ 0 & -1 & 0 & 0.1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Dibuje el sistema de referencia que corresponde con tres ángulos de Euler XYZ $e = (\pi/2, -\pi/2, \pi/2)$ y un vector de posición $\vec{t} = (0.3, 0.5, 0.3)^T$ (m).

Note que las orientaciones de los vectores del sistema de referencia móvil, leído por columnas, deben ser los coeficientes de la matriz de rotación en el sistema $S_0 = (X_0, Y_0, Z_0)$ (la base).

Ejercicio 2.15.4: Especificando puntos de destino para el robot

La programación de un robot requiere ser capaces de especificar puntos de destino para su extremo. Es decir, el sistema de referencia situado sobre la punta del robot deberá coincidir con el sistema de referencia que comandemos como destino (*target*).

- a) Abra la escena `irb140.ttt` y el script `transformations/reference_frames.py`.
- Observe la Figura 2.7. Secuencialmente, sitúe el sistema de referencia móvil sobre todos los puntos que se indican en la figura.

En las prácticas siguientes comandaremos al robot a estos puntos de destino para asir una pieza, paletizarla o realizar una operación de soldadura, entre otras aplicaciones.

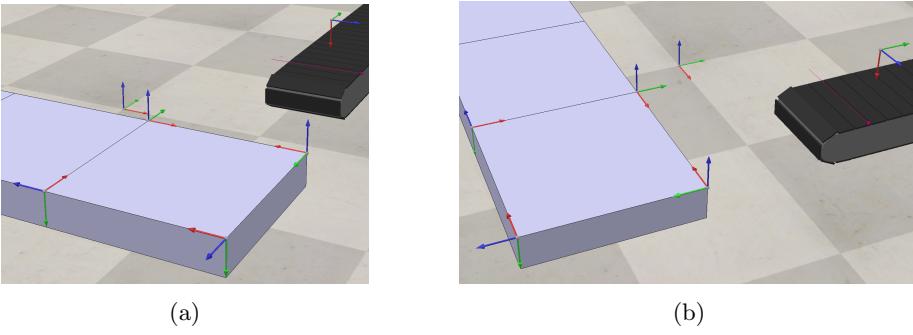


Figura 2.7: Sistemas de referencia en una escena de Coppelia. Fuente: Captura de pantalla de Coppelia Sim.

2.16. Ejercicios avanzados

Ejercicio 2.16.1: Conversión de matriz de rotación a ángulos de Euler

Usando el script `transformations/euler_conversions.py` anterior, calcule los ángulos de Euler que dan, como resultado, las siguientes matrices de rotación.

$$R_a = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix} R_b = \begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \end{pmatrix}$$

$$R_c = \begin{pmatrix} 0 & 0 & 1 \\ 0 & -1 & 0 \\ 1 & 0 & 0 \end{pmatrix} R_d = \begin{pmatrix} 0 & 0 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{pmatrix}$$

¿Cuáles de las anteriores matrices corresponden con un caso degenerado?

Solución: Usando el script `transformations/euler_conversions.py`, tenemos:

$$R_a = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix} \quad e_a = (-\pi/2, 0, 0), (\pi/2, \pi, -\pi)$$

$$R_b = \begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \end{pmatrix} \quad e_b = (\pi/2, 0, \pi/2), (-\pi/2, \pi, -\pi/2)$$

$$R_c = \begin{pmatrix} 0 & 0 & 1 \\ 0 & -1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad e_c = (0, \pi/2, \pi), (\pi, \pi/2, 0) \text{ (degenerado)}$$

$$R_d = \begin{pmatrix} 0 & 0 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{pmatrix} \quad e_d = (0, -\pi/2, \pi), (\pi, -\pi/2, 0) \text{ (degenerado)}$$

Ejercicio 2.16.2: Conversión de ángulos de Euler a matriz de rotación

Escriba una función que calcule la matriz de rotación R para diferentes convenciones de los ángulos de Euler. En concreto:

- XYX
- ZXZ
- XZX

Solución: Se proporciona, a continuación, el código de esta tarea. Encontrarás el código en `practicals/euler2rot.py`

```
import numpy as np
from artelib.rotationmatrix import Rx, Ry, Rz

def euler2rot(abg, convention):
    """
    Compute the rotation matrix for a given convention
    (e. g. XYZ) always working on mobile axes.
    """
    if convention == 'xyz':
        Ra = Rx(abg[0])
        Rb = Ry(abg[1])
        Rc = Rz(abg[2])
    elif convention == 'zxz':
        Ra = Rz(abg[0])
        Rb = Rx(abg[1])
        Rc = Rz(abg[2])
    elif convention == 'xzx':
        Ra = Rx(abg[0])
        Rb = Rz(abg[1])
        Rc = Rx(abg[2])
    else:
        print('UNDEFINED CONVENTION')
        raise Exception
    R = Ra*Rb*Rc
    return R

if __name__ == '__main__':
    Rxyz = euler2rot([np.pi/2, 0, np.pi/2], 'xyz')
    Rzxz = euler2rot([np.pi/2, 0, np.pi/2], 'zxz')
    Rxzx = euler2rot([np.pi/2, 0, np.pi/2], 'xzx')

    print(Rxyz)
    print(Rzxz)
    print(Rxzx)
    Rxyz.plot('Rxyz')
    Rzxz.plot('Rzxz')
    Rxzx.plot('Rxzx')
```

Ejercicio 2.16.3: Conversión entre diferentes convenciones de ángulos de Euler

Escriba una función que convierta entre diferentes convenciones de ángulos de Euler (en ejes móviles). En concreto:

- De XYX a XYZ .
- De ZXZ a XYZ .
- De XZX a XYZ .

Solución: La conversión entre diferentes convenios de ángulos de Euler se muestra en las líneas siguientes ([transformations/euler2rot.py](#)). Nótese que se utiliza el mismo script indicado antes. El proceso plantea convertir cualquier convención de Euler a una matriz de rotación R . Conocida R , se puede calcular los ángulos de Euler XYZ con las ecuaciones expuestas en los Apartados [2.13.1](#) y [2.13.2](#). Nótese que estas ecuaciones están implementadas en la clase `Euler`.

```
import numpy as np
from artelib.rotationmatrix import Rx, Ry, Rz

def euler2rot(abg, convention):
    """
    Compute the rotation matrix for a given convention.
    """
    if convention == 'xyz':
        Ra = Rx(abg[0])
        Rb = Ry(abg[1])
        Rc = Rz(abg[2])
    elif convention == 'zxz':
        Ra = Rz(abg[0])
        Rb = Rx(abg[1])
        Rc = Rz(abg[2])
    elif convention == 'xzx':
        Ra = Rx(abg[0])
        Rb = Rz(abg[1])
        Rc = Rx(abg[2])
    else:
        print('UNDEFINED CONVENTION')
        raise Exception
    R = Ra*Rb*Rc
    return R

if __name__ == '__main__':
    Rxyz = euler2rot([np.pi/2, 0, np.pi/2], 'xyz')
    Rzxz = euler2rot([np.pi/2, 0, np.pi/2], 'zxz')
    Rxzx = euler2rot([np.pi/2, 0, np.pi/2], 'xzx')

    print('Resulting matrices: ')
    print('Rxyz:\n', Rxyz)
    print('Rzxz:\n', Rzxz)
    print('Rxzx:\n', Rxzx)
    Rxyz.plot('Rxyz')
    Rzxz.plot('Rzxz')
    Rxzx.plot('Rxzx')

    print('Convert every R to XYZ Euler angles')
```

```

print('Rxyz to XYZ (obvious):')
print(Rxyz.euler()[0], Rxyz.euler()[1])
print('Rzxz to XYZ:')
print(Rxyz.euler()[0], Rxyz.euler()[1])
print('Rzxz to XYZ:')
print(Rxyz.euler()[0], Rxyz.euler()[1])

```

Nótese que en la clase `RotationMatrix` se implementa siempre una conversión hacia ángulos de Euler en versión XYZ. Así pues, el código anterior solamente funcionará si se desea convertir cualquier convención de ángulos de Euler al estándar XYZ (en ejes móviles).

2.17. Introducción a numpy

En este último apartado se dan algunas nociones sobre la librería `numpy`. Este apartado no es esencial para el desarrollo de las prácticas con pyARTE, pero es de utilidad si se desea desarrollar nuevo código con python.

2.18. Matrices en numpy

En este apartado aprenderemos a representar y manejar matrices usando la librería `numpy`. NumPy (Numerical Python) es una librería de código abierto que se utiliza en múltiples áreas de ingeniería y ciencia. La librería NumPy contiene estructuras de tipo matriz n-dimensionales. En concreto, proporciona el tipo `ndarray` (un array n-dimensional) y un conjunto de métodos para operar sobre él. Este tipo de dato nos permitirá definir vectores y matrices de forma sencilla.

El código siguiente muestra cómo crear dos matrices (una de 3x3 y otra matriz 3x4), para, después, multiplicarlas. Además, se añade un poco de orden al script de python, añadiendo una función `main`, que, generalmente, facilita la comprensión del código y su organización

```

import numpy as np

def multiplica_matrices():
    a = np.array([1, 2, 3], [4, 5, 6], [7, 8, 9])
    b = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
    print('MATRICES:')
    print(a)
    print(b)
    print('PRODUCTO:')
    c = np.dot(a, b)
    print(c)

if __name__ == "__main__":
    multiplica_matrices()

```

En este script de python, primero se ejecutará el código en la función `main`, la cual invoca a la función `multiplica_matrices()`. Esta función, finalmente, realiza las operaciones deseadas.

Nota: Se proporciona, a continuación, información para aclarar las operaciones con matrices más usuales usando la librería `numpy` de python:

- Importar la librería `numpy`:

```
import numpy as np
```

A partir de este momento se importa la librería numpy y nos podemos referir a ella como `np`.

- Creación de un vector:

```
q0 = np.array([-np.pi, 0, np.pi/2, 0, 0, 0])
```

Atención: se puede crear una lista en python con:

```
q0 = [1, 2, 3, 4, 5]
```

Sin embargo, un array de numpy cuenta con propiedades avanzadas que lo distinguen de una lista estándar de python.

- Creación de una matriz:

```
A = np.array([[1, 2], [3, 4]])
```

- Producto de Matrices:

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[1, 2], [3, 4]])
C = np.dot(A, B)
```

- Determinante de una matriz:

```
A = np.array([[1, 2], [3, 4]])
detA = np.linalg.det(A)
```

- Producto $A \cdot A^T$:

```
A = np.array([[1, 2], [3, 4]])
prodAAT = np.dot(A, A.T)
```

- Inversa:

```
A = np.array([[1, 2], [3, 4]])
invA = np.linalg.inv(A)
```


Capítulo 3

Una aplicación de paletizado con Coppelia

3.1. Introducción

En esta práctica se le propone al estudiante que realice una aplicación completa de paletizado en el simulador Coppelia Sim en combinación con la librería pyARTE. La aplicación de paletizado es común en la industria, pues una gran cantidad de bienes de consumo se transportan sobre pallets. El paletizado consiste, generalmente, en la colocación de cajas (u otros objetos) sobre un pallet con medidas estándar, buscando el mayor aprovechamiento posible del volumen, para reducir los costes relativos al transporte de las mercancías. Un ejemplo de este paletizado se muestra en la Figura 3.1.

3.2. Objetivos

En esta práctica se persiguen los siguientes **objetivos de aprendizaje**:

- Capacitar al estudiantado para crear una simulación real de un proceso industrial robotizado usando Coppelia Sim y la librería pyARTE.
- Permitir el desarrollo de una aplicación de paletizado en un entorno de simulación.

En la práctica se proponen, entre otras, las siguientes **actividades**:

- Cálculo y representación de la soluciones de la cinemática inversa de un robot industrial usando Coppelia Sim y la librería pyARTE.
- Interacción del robot con objetos del entorno: p. e. asir piezas con una pinza o bien utilizando una ventosa de vacío.

3.3. Material proporcionado

Para la práctica se proporciona al estudiante el siguiente material:

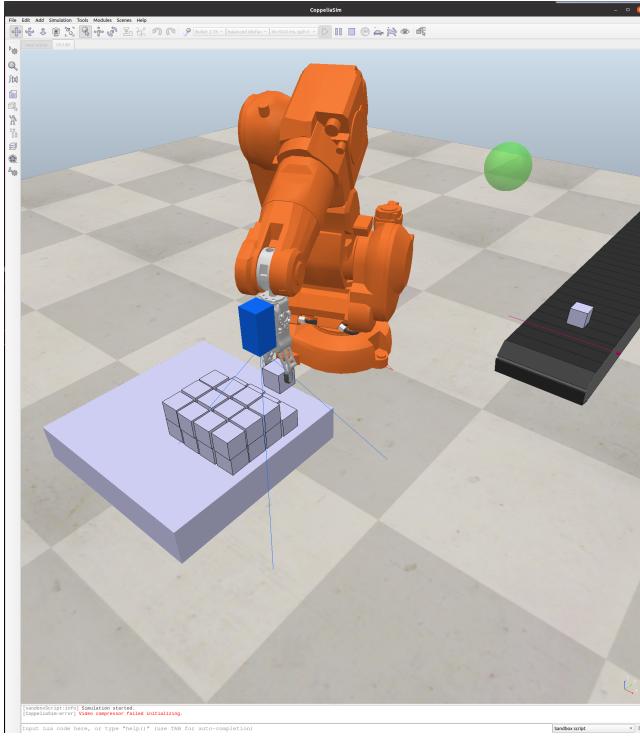


Figura 3.1: Una aplicación de paletizado con Coppelia.

- Una escena de Coppelia: `scenes/irb140.ttt`.
- Manejo básico del robot: `practicals/move_robot.py`.
- Un script de python que se encarga de realizar la aplicación de paletizado: `practicals/applications/irb140_palletizing.py`.

3.4. Descripción de la escena

Abra la escena (`scenes/irb140.ttt`) en Coppelia. Puede iniciar la simulación de la escena (botón “play” en el menú superior). Observará lo siguiente:

- Que existe unas superficies (pallets) en las que se pueden depositar las piezas. Estos pallets están ubicados en diferentes posiciones y orientaciones
- Que hay una cinta transportadora a la izquierda del robot. En la cinta transportadora existe un sensor que la detiene cuando algún objeto corta su haz.
- En el explorador de la escena de Coppelia observamos, además, un elemento denominado CubeSpawner. Este elemento es el encargado de crear cubos y piezas que deberá paletizar el robot. El/la estudiante puede modificar el script de Lua para crear otras tareas similares a la descrita aquí, en concreto, las siguientes variables determinan el número total de piezas

a producir, sus dimensiones, la posición inicial de las piezas y el tiempo (s) a esperar para crear una nueva pieza.

```
local CUBE_COUNT = 100
local SIZE = {0.08,0.08,0.08}
local START_POS = {0.6,0.5,0.2}
local WAIT_TIME = 7
```

También es posible modificar la siguiente línea para crear otro tipo de piezas:

```
local spawn = sim.createPureShape(0, 10, SIZE, 0.1, NULL)
```

Por ejemplo, para crear esferas:

```
local spawn = sim.createPureShape(1, 10, SIZE, 0.1, NULL)
```

Con:

- 0: cuboides.
- 1: esferas.
- 2: cilindros.
- 3: conos.

Si se inicia la simulación en Coppelia (sin iniciar ningún script de python), veremos que el script `CubeSpawner` comenzará a crear piezas. También veremos que las piezas se desplazan sobre la cinta transportadora hasta llegar al sensor de luz. Cuando el sensor de luz es interrumpido por una pieza, la cinta transportadora se detiene automáticamente. De esta manera, la cinta transportadora avanzará solamente cuando se retire la pieza que corta el haz de luz. En consecuencia, si no se retira la pieza, se producirá un amontonamiento de las piezas creadas por `CubeSpawner`.

3.5. Transformaciones

Si observamos la Figura 3.2, podemos observar las relaciones entre los sistemas de referencia de interés durante el paletizado. La transformación 0T_m hace referencia a la transformación entre la base del robot y un sistema móvil (solidario al pallet). La transformación mT_p indica la transformación entre el sistema de referencia móvil y la posición y orientación de la pieza.

En la Figura 3.2 debemos fijarnos, además, en las siguientes transformaciones:

- 0A_6 : la transformación entre la base del robot y el extremo del robot (cinemática directa del robot).
- T_{tcp} : la transformación entre el extremo del robot y la pinza (TCP).

- 0T_m : la posición/orientación del pallet.
- mT_p : la posición y orientación de la pieza en el sistema de coordenadas del pallet.

Importante: La transformación 0T_m cambiará cuando se necesite paletizar en diferentes posiciones y orientaciones. La transformación mT_p cambiará para cada pieza a paletizar. Finalmente, T_{tcp} cambiará en función de la pinza o ventosa que estemos utilizando.

Definimos la posición y orientación de paletizado como:

$$T_{target} = {}^0T_m {}^mT_p \quad (3.1)$$

Con todo esto, para que el robot deposite una pieza en el pallet, tendrá que ocurrir que:

$${}^0A_6T_{tcp} = T_{target} = {}^0T_m {}^mT_p \quad (3.2)$$

Importante: En la ecuación anterior, la parte de la derecha permite definir una transformación global en coordenadas de la base del robot. En concreto, tendremos una transformación mT_p para cada una de las piezas del pallet y habrá una transformación 0T_m global para el pallet. Para poder llegar a la transformación global T_{target} el algoritmo de cinemática inversa habrá hallado los valores articulares para que ${}^0A_6T_{tcp} = T_{target}$. Es decir:

$${}^0A_6 = T_{target}T_{tcp}^{-1} \quad (3.3)$$

Por tanto, podemos paletizar en cualquier posición y orientación del entorno si:

- Modificamos 0T_m para adaptarnos al pallet que deseamos llenar y,
- para cada pieza, calculamos cada nuevo *target point* como $T_{target} = {}^0T_m {}^mT_p$.

3.6. Cálculo de las posiciones de paletizado

En el código proporcionado, vamos a prestar especial atención a la función `compute_3D_coordinates` que se utiliza en la función `place`. Suponga que desea realizar un paletizado de $n_x \times n_y \times n_z$ elementos de un cubo de lado d (m). Suponga, además, que desea dejar un espacio ϵ (m) entre los cubos. Es fácil obtener los índices de un array multidimensional como el definido. Así, por ejemplo, si elegimos $n_x = 3$, $n_y = 4$, $n_z = 2$ estaremos definiendo un arreglo de piezas de 3x4 con dos alturas. Sea $\vec{v}_i = (k_x, k_y, k_z)_i$ un vector con los índices de la pieza i . Entonces, calculamos la posición 3D de la pieza i como

$$\vec{p}_i = k_x(d + \epsilon, 0, 0) + k_y(0, d + \epsilon, 0) + k_z(0, 0, d)$$

En la Figura 3.3 se presentan, como ejemplo, las posiciones 3D de las piezas para un paletizado de 3x3x3 elementos y un cubo de lado 0.08 m. **Importante:** Las posiciones \vec{p}_i así definidas se refieren a un sistema de coordenadas ubicado en la pieza 0.

fila/col.	0	1	2	3
0	(0, 0, 0)	(0, 1, 0)	(0, 2, 0)	(0, 3, 0)
1	(1, 0, 0)	(1, 1, 0)	(1, 2, 0)	(1, 3, 0)
2	(2, 0, 0)	(2, 1, 0)	(2, 2, 0)	(2, 3, 0)

Tabla 3.1: Índices (i, j, k) para el paletizado del primer piso ($k = 0$). El índice i se considera alineado con X y j está alineado con Y .

3.7. Descripción del código

Examine el código del script `applications/irb140_palletizing.py`. Encontrará que la función `main` llama a la función `pick_and_place`. En los apartados siguientes se describen cada una de las funciones. A su vez, la función `pick_and_place` llama a `pick` y a `place`.

3.7.1. función `pick_and_place`

Esta función llama, alternativamente, a la función `pick` (recoger pieza) y `place` (dejar pieza). Las líneas siguientes realizan una espera. El robot únicamente iniciará el proceso de recogida de la pieza cuando una pieza active el sensor de la cinta transportadora. Recuerda que, típicamente, se emplean sensores infrarrojos de presencia equipados con una salida por relé.

```
while True:
    if conveyor_sensor.is_activated():
        break
    robot.wait()
```

En este caso, el sensor de Coppelia está enlazado con Python y cuando se llama a `conveyor_sensor.is_activated()` se obtiene un valor de `True` cuando se corta el haz de luz.

3.7.2. función `pick`

Esta función realiza la acción de recoger una pieza. Nótese que, para realizar esta acción:

- La pinza debe estar abierta.
- El robot debe desplazarse sobre la pieza (punto de aproximación).
- Se debe colocar la pinza envolviendo la pieza (punto de recogida).
- Seguidamente, se debe cerrar la pinza.
- Debe elevarse el extremo del robot. Coppelia Sim simula que la pinza aplica unas fuerzas sobre la pieza y esta se mueve en consecuencia.

Seguidamente se presenta el código de esta función:

```
def pick(robot, gripper):
    q0 = np.array([0, 0, 0, 0, np.pi/2, 0])
    tp1 = Vector([0.6, 0.267, 0.23])
    tp2 = Vector([0.6, 0.267, 0.19])
```

```

to1 = Euler([0, np.pi, 0])
to2 = Euler([0, np.pi, 0])
robot.moveAbsJ(q0, precision=True)
gripper.open(precision=True)
robot.moveJ(target_position=tp1, target_orientation=to1, precision=True)
robot.moveL(target_position=tp2, target_orientation=to2, precision='last')
gripper.close(precision=True)
robot.moveL(target_position=tp1, target_orientation=to1, precision=False)

```

Las variables `tp1`, `tp2`, `to1` y `to2` almacenan la posición y orientación de dos puntos a los que debe dirigirse el extremo del robot. Se trata de un primer punto de aproximación cerca de la pieza y, seguidamente, un punto de recogida. Nótese que se especifica la orientación del extremo usando ángulos de Euler (XYZ, móviles). El código es sencillo de entender, pues utiliza las instrucciones de movimiento `moveAbsJ`, `moveJ` y `moveL` que ya se estudiaron.

Ejercicio 3.7.1: Solución inicial

Utilice el script `irb140_palletizing.py`. En la función `pick` cambie el valor de `q0` y observe el resultado.

3.7.3. función place

Se presenta, a continuación, el código de la función `place` es similar a la función `pick`. En este caso, la diferencia radica en que se pretende dejar una pieza. En el paletizado, generalmente, la posición en la que se recoge la pieza es siempre la misma, pero la posición en la que se debe dejar la pieza varía. Se presenta, a continuación, el código de la función `place`:

```

def place(robot, gripper, i):
    piece_length = 0.08
    piece_gap = 0.02
    q0 = np.array([0, 0, 0, 0, 0, 0])

    T0m = HomogeneousMatrix(Vector([-0.15, -0.65, 0.1]), Euler([0, 0, 0]))

    pi = compute_3D_coordinates(index=i, n_x=3, n_y=4, n_z=2,
                                  piece_length=piece_length, piece_gap=piece_gap)

    p0 = pi + np.array([0, 0, 2.5 * piece_length])
    Tmp0 = HomogeneousMatrix(p0, Euler([0, np.pi, 0]))

    p1 = pi + np.array([0, 0, 0.5 * piece_length])
    Tmp1 = HomogeneousMatrix(p1, Euler([0, np.pi, 0]))

    # TARGET POINT 0 y 1
    T0 = T0m*Tmp0
    T1 = T0m*Tmp1

    robot.moveAbsJ(q0, precision=True)
    robot.moveJ(target_position=T0.pos(), target_orientation=T0.R(), precision=True)
    robot.moveL(target_position=T1.pos(), target_orientation=T1.R(), precision='last')
    gripper.open(precision=True)
    robot.moveL(target_position=T0.pos(), target_orientation=T0.R(), precision='last')

```

Nota: Observará que en el script `irb140_palletizing.py` se utilizan las funciones `moveL` y `moveJ`. En ambas, se debe especificar un valor `q_0`. La librería

elige comandar al robot a la solución de la cinemática inversa más cercana a q_0 (en distancia Euclídea en el espacio articular).

3.7.4. Definiendo los *target points*

Nos fijamos, ahora, en estas líneas de la función `place`:

```
T0m = HomogeneousMatrix(Vector([-0.15, -0.65, 0.1]), Euler([0, 0, 0]))

pi = compute_3D_coordinates(index=i, n_x=3, n_y=4, n_z=2,
                             piece_length=piece_length, piece_gap=piece_gap)

p0 = pi + np.array([0, 0, 2.5 * piece_length])
Tmp0 = HomogeneousMatrix(p0, Euler([0, np.pi, 0]))

p1 = pi + np.array([0, 0, 0.5 * piece_length])
Tmp1 = HomogeneousMatrix(p1, Euler([0, np.pi, 0]))
```

En el código anterior, tenemos:

- $T0m$: hace referencia a la matriz 0T_m definida anteriormente. Esta matriz almacena la posición y orientación del pallet respecto de la base del robot.
- pi : la posición de la pieza i en el pallet.
- $p0$: posición sobre la pieza p_i a una altura $2.5 * d$.
- $p1$: posición donde dejar la pieza i (se debe sumar la mitad del lado de la pieza).

Usando las posiciones $p0$ y $p1$ se construyen las matrices $Tmp0$ y $Tmp1$. Estas matrices definen mT_p para cada pieza. La orientación en el sistema móvil es fija y se define como `Euler([0, np.pi, 0])`. Finalmente, la posición y orientación de la pieza se calcula con:

```
T0 = T0m*Tmp0
T1 = T0m*Tmp1
```

Estas líneas calculan la posición y orientación de la pieza en el sistema de referencia de la base del robot, de acuerdo con la Ecuación (3.1).

Finalmente, es necesario comandar al robot a las posiciones y orientaciones calculadas. Para ello, podemos usar las instrucciones de movimiento `moveJ` y `moveL`, según se observa a continuación:

```
robot.moveAbsJ(q0, precision=True)
robot.moveJ(target_position=T0.pos(), target_orientation=T0.R(), precision=True)
robot.moveL(target_position=T1.pos(), target_orientation=T1.R(), precision='last')
gripper.open(precision=True)
robot.moveL(target_position=T0.pos(), target_orientation=T0.R(), precision='last')
```

Ejercicio 3.7.2: Paletizado

Utilice el script `irb140_palletizing.py`. Realice las siguientes tareas:

- Modifique el número total de piezas a paletizar (`n_pieces`).
- Modifique el arreglo de piezas a paletizar. Por ejemplo, cambie en la función `place` a un paletizado de 2x3x2, 4x2x2.
- En la función `place`, puede probar a modificar el espaciado entre piezas: `piece_gap`. Entienda que una mayor densidad de piezas/volumen, en general, rebaja los costes de transporte. Sin embargo, con la pinza que monta el robot, podemos tener colisiones.
- Usando un arreglo de 2x2 encuentre la mayor altura que puede paletizar.

Ejercicio 3.7.3: Un paletizado general

Posicione las piezas arregladas sobre el pallet de la izquierda del robot (girado $\pi/6$ (rad)) sobre el eje Z. Modifique la matriz 0T_m para poder paletizar en una posición y orientación diferentes. La posición y orientación de los target points de acuerdo con la transformación $T_{target} = {}^0T_m {}^mT_p$.

3.8. Ventosas de vacío

En esta práctica se simula que el robot está equipado con una pinza de dos dedos. Una pinza robótica proporciona la capacidad de abordar un gran número de tareas en robótica. En la aplicación actual, debemos tener en cuenta que los dedos de la pinza, posiblemente, rozarán con otras piezas cuando se abran (véase la Figura 3.4). Por tanto, es necesario elegir muy bien el espaciado entre piezas y las posiciones finales a las que se llevará el extremo del robot. Siempre existe la opción de dejar caer la pinza desde posiciones más altas, de tal manera que los dedos no interfieran con las piezas que ya se han dejado. En este caso, Coppelia intenta simular esta caída de forma aleatoria, con lo que la posición de la pieza no será exactamente la programada. La pinza que se ha elegido para esta práctica, no obstante, posibilita realizar un gran número de actividades diferentes. Igualmente, la librería incorpora otras pinzas (p.e. la barrett hand), que pueden ser igualmente utilizadas durante el proyecto propuesto. Es habitual realizar aplicaciones de paletizado utilizando útiles con ventosas de vacío.

Se ha incluido, como herramienta alternativa, una ventosa de vacío. Podemos utilizar esta ventosa, en la aplicación para tener una mejor precisión en el agarra. Para usarla, debemos usar la clase `SuctionPad` y modificar los puntos de destino en las funciones de `pick` y `place`.

La ventosa simula un agarre perfecto utilizando una herramienta no prensil. De esta manera, se pueden ubicar las piezas sobre el pallet con mayor precisión. También: note que se puede dejar la pieza sin tener que soltarla a una altura.

Nota: la cinta transportadora introduce pequeñas variaciones de posición y orientación en las piezas. De esta manera, la colocación de las piezas sobre el

pallet no se puede realizar de forma totalmente exacta.

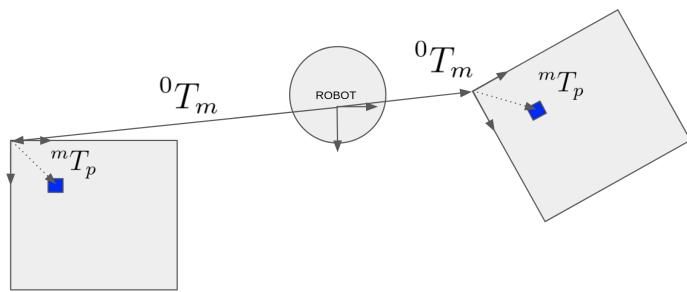
Ejercicio 3.8.1: Ventosas

Cambie el TCP del robot y modifique `irb140_palletizing.py` para utilizar un objeto de tipo `SuctionPad` en vez de gripper. Además:

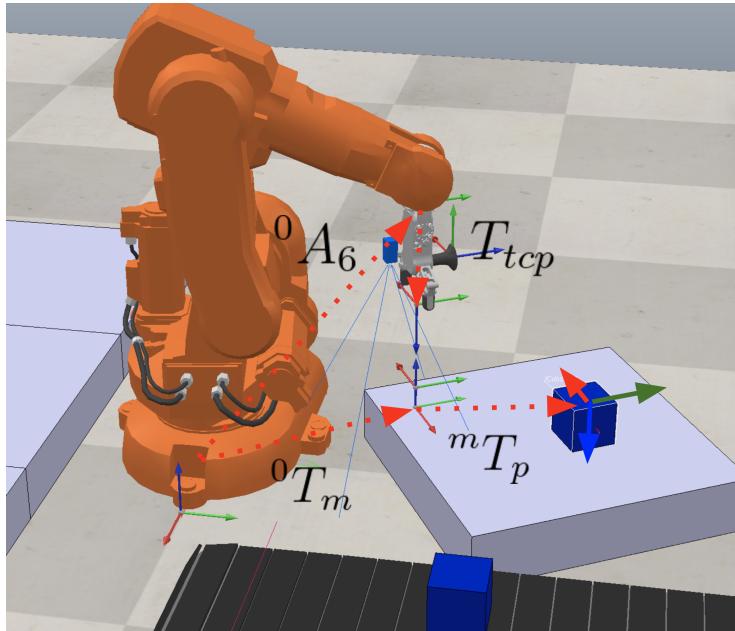
- Será necesario que cambie el TCP de la herramienta. Deberá modificar ligeramente, la posición y orientación de los target points en la función `pick` y `place`.
- Reduzca ahora la variable `piece_gap` para maximizar el rendimiento volumétrico del pallet.

3.9. Tiempo real

Cuando ejecute cualquier script en Coppelia, el simulador, intentará funcionar a *tiempo real*. Es decir, el movimiento del robot dependerá de las velocidades máximas de cada articulación, su capacidad para realizar par en cada articulación... etc. Se pueden utilizar los botones mostrados en la Figura 3.5 para acelerar/retrasar la simulación. Esto, en ocasiones, resulta beneficioso si queremos simular rápidamente una tarea muy larga realizada por el robot.



(a)



(b)

Figura 3.2: Dos vistas de las transformaciones de interés en la aplicación de paletizado.

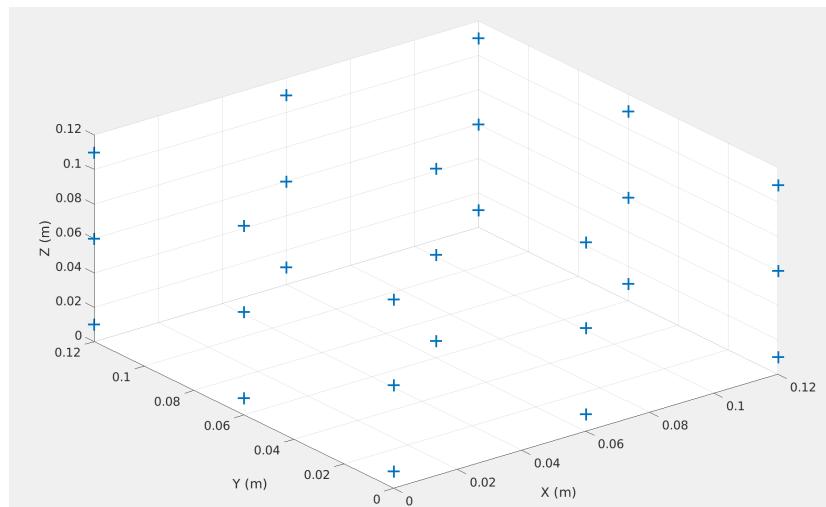


Figura 3.3: Posiciones (x, y, z) para el paletizado de 27 piezas en un arreglo de 3x3.

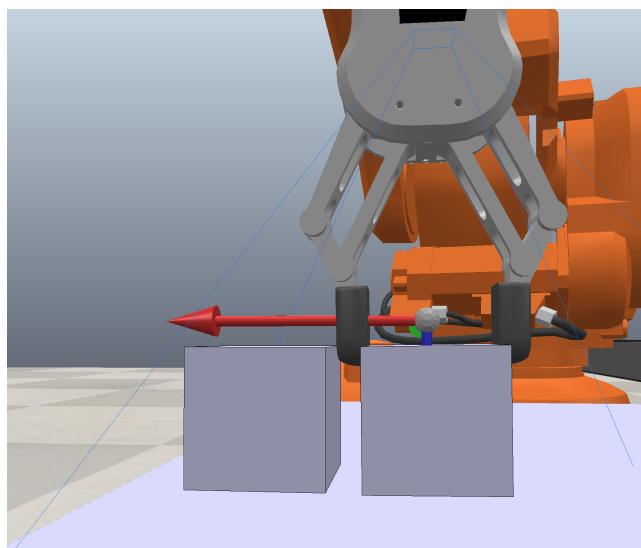


Figura 3.4: Pinza del robot colisionando con piezas.

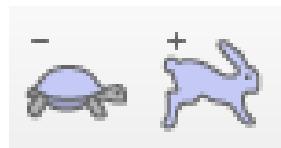


Figura 3.5: Aumenta o decelara la velocidad de simulación de Coppelia.

Capítulo 4

Cinemática inversa usando la Jacobiana del manipulador

4.1. Objetivos

En esta práctica se persiguen los siguientes objetivos:

- Ensayar y practicar conceptos de cinemática inversa en velocidad.
- Plantear un planificador de trayectorias simple.
- Desarrollar un algoritmo de cinemática inversa para un robot con muñeca no esférica.

4.2. El robot UR5

En la Figura 4.1 se presenta el robot UR5. En la Tabla 4.1 se muestran los parámetros DH que definen su cinemática. Un aspecto destacable de este robot es que su muñeca no es esférica. De esta manera, no es posible obtener soluciones cerradas a su cinemática inversa en posición/orientación. En concreto, no es posible aplicar el concepto de “desacople cinemático” para resolver de forma cerrada la posición y orientación del manipulador. En esta práctica utilizaremos un método basado en la Jacobiana para obtener soluciones en posición/orientación.

Transformación	θ (rad)	d (m)	a (m)	α (rad)
$0 \rightarrow 1$	$q_1 - \pi/2$	0.089159	0	$\pi/2$
$1 \rightarrow 2$	$q_2 + \pi/2$	0	0.425	0
$2 \rightarrow 3$	q_3	0	0.39225	0
$3 \rightarrow 4$	$q_4 - \pi/2$	0.10915	0	$-\pi/2$
$4 \rightarrow 5$	q_5	0.09465	0	$\pi/2$
$5 \rightarrow 6$	q_6	0.0823	0	0

Tabla 4.1: Parámetros DH del robot UR5.



Figura 4.1: El robot UR5. Fuente: www.universal-robots.com

4.3. Cinemática

En la librería ARTE¹ y pyARTE se emplean métodos conocidos para el cálculo de las matrices de transformación y matriz Jacobiana de los robots. En concreto se define una tabla de parámetros DH para cada robot y se calculan las matrices de DH para cada transformación entre eslabones consecutivos. Esto nos permite: En estas prácticas, con la idea de tener un código más simple y fácil de mantener, se utilizan expresiones simbólicas para:

- Definir la posición/orientación del robot (matriz T).
- Definir la Jacobiana del manipulador (J).

Puede observar ambas expresiones de T y J en el fichero:
kinematics/kinematics_ur5.py.

En concreto en las funciones `eval_symbolic_jacobian_UR5` y `eval_symbolic_T_UR5`. Note que en estas funciones se define las ecuaciones cerradas de la matriz homogénea de posición/orientación T y de la Jacobiana del manipulador J como función de las articulaciones $\vec{q} = \{q_1, q_2, \dots, q_6\}$.

En ocasiones, puede resultar complejo obtener estas ecuaciones cerradas que expresen la posición del extremo del robot o bien su Jacobiana. En consecuencia,

¹www.arvc.umh.es/arte

se presenta aquí una forma de deducir las ecuaciones utilizando la librería ARTE y Matlab. En la librería ARTE puede obtenerse una expresión de la matriz $T = {}^0A_6$ con el siguiente script: **Este código no es necesario reproducirlo en esta práctica, pero se deja aquí como documentación para el estudiante.**

```
function direct_kinematics_symbolic_UR5

syms q1 q2 q3 q4 q5 q6
robot = load_robot('UR', 'UR5')

% link lengths
d = eval(robot.DH.d);
a = eval(robot.DH.a);
alpha = eval(robot.DH.alpha);

% matrices DH
A01 = dh_sym(q1-pi/2, d(1), a(1), alpha(1));
A12 = dh_sym(q2+pi/2, d(2), a(2), alpha(2));
A23 = dh_sym(q3, d(3), a(3), alpha(3));
A34 = dh_sym(q4-pi/2, d(4), a(4), alpha(4));
A45 = dh_sym(q5, d(5), a(5), alpha(5));
A56 = dh_sym(q6, d(6), a(6), alpha(6));

A02 = A01*A12;
A03 = A02*A23;
A04 = A03*A34;
A05 = A04*A45;
A06 = A05*A56;
A06 = simplify(A06)

function A = dh_sym(theta, d, a, alpha)
syms q1 q2 q3 q4 q5 q6
% avoid almost zero elements in cos(alpha) and sin(alpha)
ca = cos(alpha);
sa = sin(alpha);
if abs(ca) < 1e-6
    ca = 0;
end
if abs(sa) < 1e-6
    sa = 0;
end
A=[cos(theta) -ca*sin(theta) sa*sin(theta) a*cos(theta);
    sin(theta) ca*cos(theta) -sa*cos(theta) a*sin(theta);
    0           sa            ca            d;
    0           0              0              1];
```

También, en la librería ARTE puede obtenerse una expresión de la Jacobiana del manipulador con el siguiente script. **Este código no es necesario reproducirlo en esta práctica, pero se deja aquí como documentación para el estudiante.**

```
function jacobian_symbolic_UR5
% link lengths
```

```

syms q1 q2 q3 q4 q5 q6
robot = load_robot('UR', 'UR5')

d = eval(robot.DH.d);
a = eval(robot.DH.a);
alpha = eval(robot.DH.alpha);

% matrices DH
A01 = dh_sym(q1-pi/2, d(1), a(1), alpha(1));
A12 = dh_sym(q2+pi/2, d(2), a(2), alpha(2));
A23 = dh_sym(q3, d(3), a(3), alpha(3));
A34 = dh_sym(q4-pi/2, d(4), a(4), alpha(4));
A45 = dh_sym(q5, d(5), a(5), alpha(5));
A56 = dh_sym(q6, d(6), a(6), alpha(6));

A02 = A01*A12;
A03 = A02*A23;
A04 = A03*A34;
A05 = A04*A45;
A06 = A05*A56;

z0 = [0 0 1]';
z1 = A01(1:3,3);
z2 = A02(1:3,3);
z3 = A03(1:3,3);
z4 = A04(1:3,3);
z5 = A05(1:3,3);
% simplify expressions
z4 = simplify(z4);
z5 = simplify(z5);

% Jacobian in angular speed
Jw = [z0 z1 z2 z3 z4 z5];

p06=A06(1:3,4);
p16=A06(1:3,4)-A01(1:3,4);
p26=A06(1:3,4)-A02(1:3,4);
p36=A06(1:3,4)-A03(1:3,4);
p46=A06(1:3,4)-A04(1:3,4);
p56=A06(1:3,4)-A05(1:3,4);

% jacobian in linear speed
Jv = [cross(z0, p06) cross(z1, p16) cross(z2, p26)...
       cross(z3, p36) cross(z4, p46) cross(z5, p56) ];

Jw = simplify(Jw)
Jv = simplify(Jv)
Js = [Jv; Jw]

function A = dh_sym(theta, d, a, alpha)
syms q1 q2 q3 q4 q5 q6
% avoid almost zero elements in cos(alpha) and sin(alpha)
ca = cos(alpha);

```

```

sa = sin(alpha);
if abs(ca) < 1e-6
    ca = 0;
end
if abs(sa) < 1e-6
    sa = 0;
end
A=[cos(theta) -ca*sin(theta) sa*sin(theta) a*cos(theta);
    sin(theta) ca*cos(theta) -sa*cos(theta) a*sin(theta);
    0           sa                  ca          d;
    0           0                   0           1];

```

4.4. Cinemática inversa

En los robots de 6 GDL equipados con una muñeca esférica es posible deducir un conjunto de ecuaciones cerradas que permiten calcular (en general) las coordenadas articulares que permiten al robot alcanzar una posición y orientación dadas (por ejemplo) por una matriz homogénea (T) de posición/orientación. Definamos, en este momento, el problema de cinemática inversa como aquel que pretende hallar una posición articular q del robot de manera que el extremo del robot alcance la posición y orientación deseadas expresadas, por ejemplo, por la matriz T_d :

$$T_d = \begin{pmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.1)$$

El robot UR5, por contra, no está equipado con una muñeca esférica. De esta manera, existe una dependencia entre el valor articular de q_4 , q_5 y q_6 y la posición del extremo. Por tanto, no es posible encontrar una solución cerrada para su cinemática inversa. Para hallar una solución de la cinemática inversa nos centraremos en el uso de la Jacobiana del Manipulador. En concreto, tenemos que:

$$\begin{pmatrix} {}^0\vec{v}_e \\ {}^0\vec{\omega}_e \end{pmatrix} = {}^0J\dot{q}$$

donde $\dot{\vec{x}} = ({}^0\vec{v}_e, {}^0\vec{\omega}_e)^T$ define la velocidad lineal y angular del extremo del robot y 0J hace referencia a la Jacobiana del Manipulador. El vector de velocidades articulares es \dot{q} . Ambas cantidades se expresan en el sistema de coordenadas de la base del robot. Para abbreviar y hacer la nomenclatura más clara, escribiremos, normalmente:

$$\begin{pmatrix} \vec{v}_e \\ \vec{\omega}_e \end{pmatrix} = J\dot{q}$$

Esta ecuación, se suele aproximar por la siguiente expresión:

$$\vec{e} = J\Delta q$$

Esta expresión define la relación entre el error \vec{e} en el extremo del robot, J y un pequeño movimiento articular Δq . El error \vec{e} define, por tanto, un error

de posición y orientación en el extremo del robot. Así pues, los algoritmos de cinemática inversa se basan en realizar las siguientes iteraciones hasta que, para una posición articular q_{i+1} tengamos un error $|\vec{e}| = 0$.

$$\Delta q_i = J^{-1} \vec{e} \quad (4.2)$$

$$q_{i+1} = q_i + \Delta q_i \quad (4.3)$$

Si J no es cuadrada, entonces utilizaremos las siguientes expresiones:

$$\Delta q_i = J^\dagger \vec{e} \quad (4.4)$$

$$q_{i+1} = q_i + \Delta q_i \quad (4.5)$$

donde J^\dagger es la pseudo-inversa Moore-Penrose.

Debemos, en este momento, definir el error $\vec{e} = (\vec{e}_p, \vec{e}_o)$ que consideramos compuesto de un error de posición \vec{e}_p y un error de orientación \vec{e}_o . Recordemos que deseamos que el extremo del robot alcance la posición $\vec{p}_d = (p_x, p_y, p_z)$ de la matriz T_d en la Ecuación (4.1). Habitualmente, se define el error de posición como:

$$\vec{e}_p = \vec{p}_d - \vec{p}_i$$

donde \vec{p}_d es la posición deseada del extremo del robot y \vec{p}_i corresponde con la posición del extremo dados los valores articulares actuales q_i (cinemática directa $T_i = f(q_i)$).

Debemos dar también una expresión para el error de orientación. Asumamos que deseamos que el robot alcance una orientación dadas por la matriz $R_d = (\vec{n}_d, \vec{o}_d, \vec{a}_d)$. Supongamos que en una iteración del algoritmo, el extremo del robot tiene una orientación dada por la matriz $R_i = (\vec{n}_i, \vec{o}_i, \vec{a}_i)$. Con esto, una posible expresión del error de orientación \vec{e}_o es [1]:

$$\vec{e}_o = \frac{1}{2}(\vec{n}_d \times \vec{n}_i + \vec{o}_d \times \vec{o}_i + \vec{a}_d \times \vec{a}_i)$$

En el ejercicio siguiente se busca que el alumno programe un algoritmo de cinemática inversa para un robot con muñeca no esférica. Se proporciona al alumno un script de python que deberá completar para realizar las tareas que se proponen a continuación. Note que se plantea un método basado en la Jacobiana del manipulador que parte de una semilla inicial q_0 .

Ejercicio 4.4.1: CINEMÁTICA INVERSA

El ejercicio plantea que el/la estudiante complete el código de un script de Python para programar un método de cinemática inversa basado en la Jacobiana. Instrucciones:

- Abra la escena `scenes/ur5.ttt`.
- Edite el script `practicals/inverse_kinematics/ur5_ikine.py`. Se pide que complete el código proporcionado para resolver la cinemática inversa del robot.
- Complete la función siguiente. Solamente se proporciona un esqueleto de la función y el estudiante deberá completarlo para que la función devuelva una posición articular q tal que $T = T(q)$:

```
def inverse_kinematics(robot,
                        target_position,
                        target_orientation, q0)
```

- Halle el error en cada paso del algoritmo utilizando la función:

```
e, error_dist, error_orient =
    compute_kinematic_errors(Tcurrent=Ti, Ttarget=Ttarget)
```

Note que la función proporciona el error e , a partir de la posición/orientación actuales T_i y la posición/orientación deseadas T_{target} .

- Halle el vector de actualización Δq en cada paso del algoritmo como:

$$\Delta q = J^\dagger \vec{e}$$

donde J^\dagger es la pseudo-inversa Moore Penrose definida como:

$$J^\dagger = J^T (JJ^T)^{-1}$$

- Actualice la posición articular:

$$q = q + \Delta q$$

- Devuelva una solución cuando considere que el error en posición/orientación es bajo. Note que la función `compute_kinematic_errors` ya devuelve la norma del error de posición y orientación.

El estudiante debe programar un método de cinemática inversa basado en la Jacobiana del manipulador. Para facilitar el trabajo, se proporcionan al alumno unas funciones:

```
Ti = robot.direct_kinematics(q)
J, Jv, Jw = robot.get_jacobian(q)
e, error_dist, error_orient = compute_kinematic_errors(Tcurrent=Ti, Ttarget=Ttarget)
```

La primera: `robot.direct_kinematics(q)` consiste en un método de ci-

nemática directa del robot. Devuelve una matriz de posición/orientación del extremo del robot. Seguidamente, la función:

```
J, Jv, Jw = robot.get_jacobian(q)
```

devuelve la Jacobiana del manipulador J completa y, además, ambas matrices separadas en velocidad lineal y angular. Finalmente, la función:

```
e, error_dist, error_orient = compute_kinematic_errors(Tcurrent=Ti,target=Ttarget)
```

permite calcular el error en posición y orientación (la cantidad que debe moverse el extremo del robot hacia la posición y orientación deseadas).

Nota: de forma alternativa, podríamos utilizar J^{-1} en vez de J^\dagger (la pseudo-inversa Moore-Penrose). Se ha elegido J^\dagger por ser un procedimiento más general, lo que nos permitirá utilizar las mismas funciones para robots redundantes con J rectangular (no cuadrada). Se plantea, por tanto, el uso de la pseudoinversa J^\dagger puesto, que, de esta manera, podremos extender fácilmente en el futuro el algoritmo para el control de robots redundantes. La pseudo inversa Moore-Penrose se define, para este caso:

$$J^\dagger = J^T (JJ^T)^{-1}$$

Note que, la siguiente línea, define la semilla inicial del algoritmo:

```
q0 = np.array([-np.pi, -np.pi/8, np.pi/2, 0, 0, 0])
```

Debe haber notado que, aunque elegante, la semilla $q_0 = (0, 0, 0, 0, 0, 0)$ no resulta apropiada en este caso, **por tratarse de un punto singular del robot** para el cual J^{-1} no existe. Igualmente, en una singularidad la matriz (JJ^T) está mal condicionada y al calcular la inversa $(JJ^T)^{-1}$ obtenemos $\Delta q = \text{nan}$ (*not a number*). Si entramos en esta condición, el algoritmo no puede converger.

Para mejorar el algoritmo, se propone al alumno que implemente una versiones mejoradas que evitan esta situación, en concreto, se propone al alumno que realice dos ejercicios que plantean soluciones esta situación crítica. Estas dos soluciones son:

- Método “**Damped least squares**” o Moore-Penrose atenuado. La regla de actualización es:

$$\Delta q = J^T (JJ^T + kI)^{-1} \vec{e}$$

con $k = 0.01$.

- **Método de la transpuesta.** La regla de actualización es:

$$\Delta q = \alpha J^T \vec{e}$$

Se justifica el uso de la transpuesta a partir de la propagación de las fuerzas y pares en el extremo del robot a las articulaciones, pues, sabemos que:

$$\tau = J^T \vec{f}$$

El valor de α resulta crítico para la convergencia del algoritmo, algunos autores proponen:

$$\alpha = \frac{\vec{e} \cdot JJ^T \vec{e}}{JJ^T \vec{e} \cdot JJ^T \vec{e}}$$

con todo esto, la regla de actualización queda:

$$\Delta q = \frac{\vec{e} \cdot JJ^T \vec{e}}{JJ^T \vec{e} \cdot JJ^T \vec{e}} J^T \vec{e}$$

Ejercicio 4.4.2: MOORE PENROSE VERSIÓN ATENUADA (damped)

Programe una versión atenuada de la cinemática inversa:

- Edite la función `moore_penrose`, script `ur5_ikine.py`.
- Compruebe el determinante $\det(JJ^T)$. Si el determinante se encuentra por debajo de un umbral .001, entonces, deberá utilizarse la conocida como *Damped Least Squares*:

$$\Delta q = J^T(JJ^T + kI)^{-1}\vec{e}$$

con $k > 0$. De esta manera, usando valores de $k = 0.01$, p.e., se aumenta la condición de la matriz a invertir y, de esta manera, la matriz es siempre invertible.

- En caso contrario, use la versión normal de la pseudoinversa.
- Para probar su solución, modifique la semilla inicial a $q_0 = (0, 0, 0, 0, 0, 0)^T$. (`q0 = np.array([0, 0, 0, 0, 0, 0])`).

Ejercicio 4.4.3: MÉTODO DE LA TRANSPUESTA

Programe el método de la transpuesta para resolver la cinemática inversa:

- Edite la función `delta_q_transpose`, script `ur5_ikine.py`.
- La regla de actualización será:

$$\Delta q = \frac{\vec{e} \cdot JJ^T \vec{e}}{JJ^T \vec{e} \cdot JJ^T \vec{e}} J^T \vec{e}$$

- Para probar su solución, modifique la semilla inicial a $q_0 = (0, 0, 0, 0, 0, 0)^T$. (`q0 = np.array([0, 0, 0, 0, 0, 0])`).

Ejercicio 4.4.4: COMPARACIÓN ENTRE LOS MÉTODOS

Compare los métodos propuestos:

- Script: `ur5_ikine.py`.
- ¿Qué diferencias encuentra entre el método basado en la transpuesta y el método basado en la pseudo-inversa atenuada?
- Cuente, por ejemplo, el número de iteraciones necesarias para alcanzar la solución final en todos los métodos.

En el script se han proporcionado los siguientes métodos que resultan de ayuda para manejar el robot:

```
robot.set_target_position_orientation(target_positions[0], target_orientations[0])
robot.set_joint_target_positions(q, wait=True)
```

```

robot.open_gripper(wait=True)
robot.close_gripper(wait=True)
robot.plot_trajectories()
robot.stop_arm()
scene.stop_simulation()

```

- La función `robot.set_joint_target_positions(q, precise=True)` comanda al robot para que alcance la posición articular especificada en q . Más en concreto, le indica a Coppelia que debe fijar un conjunto de referencias para que el sistema de control del brazo intente alcanzarlas. En el modo de control utilizado en esta práctica, Coppelia utiliza un conjunto de controladores PID en posición en cada una de las articulaciones. Si se especifica `precise=True`, la función devolverá el control al script principal únicamente cuando el robot en Coppelia haya alcanzado la posición q especificada. Si se especifica `precise=False` entonces el método comanda al robot la nueva posición q y sale inmediatamente sin esperar a que el robot en simulación haya alcanzado q . Si se hace `precise=False`, puede utilizar la función `robot.wait(steps)` para esperar el número de intervalos de simulación (`steps`) que se desee.
- Las funciones `open_gripper` y `close_gripper` sirven para abrir/cerrar la pinza del robot. Finalmente, la función `stop_arm` detiene el brazo y `stop_simulation` detiene la simulación. Se recomienda usar `stop_simulation` antes de que el script de python finalice. Se proporciona también una función denominada `plot_trajectories`. Esta función genera una gráfica con cada una de las posiciones articulares comandadas al robot.
- El método `set_target_position_orientation()` simplemente pinta un objeto de tipo *dummy* en el entorno de Coppelia con la posición y orientación que se especifiquen. Esta función resulta interesante para depurar errores y saber el destino al que el robot debe dirigirse.

Ejercicio 4.4.5: SEMILLAS INICIALES

En cualquiera de los anteriores algoritmos, la solución alcanzada depende de la semilla inicial:

- Proponga una modificación del algoritmo que genere semillas aleatorias y almacene las soluciones del algoritmo. Deberá general, al menos 16 semillas aleatorias.
- Tenga en cuenta que algunas de las semillas darán, como resultado, la misma solución de la cinemática inversa.
- Utilice `robot.set_joint_target_positions` para visualizar las soluciones.

4.5. Cinemática inversa y planificación

En este caso planteamos una tarea similar a la del Apartado 4.4. Buscamos construir una función que realice una planificación de trayectorias que:

- Permite al robot seguir una trayectoria recta en el espacio.
- La velocidad del extremo debe estar planificada y conocida. Consideraremos movimientos a orientación constante y movimientos a posición constante.
- Consiga mantener (en la medida de lo posible) una magnitud de la velocidad lineal y angular constante.
- Consiga que las velocidades articulares no superen ciertos valores máximos.
- Deben monitorizarse las posiciones articulares del robot y comprobar que están dentro de sus límites articulares.

En lo sucesivo, pensaremos que debemos realizar una función que realice una planificación lineal entre una posición \vec{p}_a y una posición \vec{p}_b . Además, el extremo del robot deberá seguir esta trayectoria con una velocidad lineal \vec{v}_e con $|\vec{v}_e| = v_{max}$. De forma simple, consideraremos que el tiempo necesario para que el robot siga la línea en el espacio de la tarea, será:

$$t_{total} = \frac{|\vec{p}_b - \vec{p}_a|}{v_{max}}$$

Por otra parte, cada intervalo de simulación en Coppelia es de δt (s). En particular, $\delta t = 0.05$ s es el intervalo estándar de simulación en Coppelia. En consecuencia, deberemos planificar n puntos sobre la recta:

$$n = ceil\left(\frac{t_{total}}{\delta t}\right)$$

Suponga que la variable $c \in [0, 1]$, entonces, un punto \vec{p}_i perteneciente a la recta entre \vec{p}_a y \vec{p}_b es:

$$\vec{p}_i = (1 - c) \cdot \vec{p}_a + c \cdot \vec{p}_b$$

En consecuencia, generar n puntos sobre la recta implica generar c valores distribuidos uniformemente en $[0, 1]$. La librería numpy nos permite hacer esto fácilmente con:

```
c = np.linspace(0, 1, n)
```

Este planificador de trayectorias es, en extremo sencillo. En concreto, no se comprueba si el robot es capaz de alcanzar esa velocidad lineal en el extremo, pues, recordemos que:

$$\begin{pmatrix} \vec{v} \\ \vec{\omega} \end{pmatrix} = J \dot{q}$$

entonces, suponiendo J invertible:

$$\dot{q} = J^{-1} \begin{pmatrix} \vec{v}_{max} \\ \vec{\omega} \end{pmatrix}$$

Debe ocurrir que $|\dot{q}_i| < \dot{q}_{i,max}$, $\forall i$. Es decir, durante la trayectoria, todas las velocidades articulares deben ser menores a las es especificadas por el fabricante del robot.

En relación con la orientación, suponga que la orientación en el comienzo de la línea está dada por la matriz de orientación R_a . En el final de la recta, el extremo del robot deberá tener la orientación R_b , en este caso, **la siguiente expresión no es correcta**:

$$R_i = (1 - c) \cdot R_a + c \cdot R_b$$

pues la matriz resultado debe ser ortonormal.

Una posible forma de realizar esta interpolación entre dos orientaciones diferentes plantea el uso de cuaterniones y la expresión que definiremos ahora. Así, suponga que la orientación en el inicio de la trayectoria está dada por el cuaternion Q_a y Q_b representa la orientación al final de la trayectoria. Entonces, una orientación intermedia entre las dos puede ser generada a través de la fracción c como:

$$Q = Q_a \frac{\sin((1 - c)\theta)}{\sin(\theta)} + Q_b \frac{\sin(c\theta)}{\sin(\theta)}$$

siendo θ el ángulo sostenido por el arco entre los dos cuaterniones, calculado como el producto escalar entre dos cuaterniones $\cos \theta = Q_a \cdot Q_b$. La fracción $c = 0$ para el inicio de la trayectoria y $c = 1$ para el final [2].

Ejercicio 4.5.1: APLICACIÓN DE PALETIZADO CON PLANIFICACIÓN LINEAL

- Utilice la escena `ur5.ttt`.
- Abra el script `ur5_ikine_path.py`.
- Complete la función `n_movements` que permite calcular el número de puntos n .
- Complete la función `path_planning_p` que permite obtener n puntos 3D entre dos puntos p_a y p_b .
- Haga que se resuelva la cinemática inversa para cada uno de los puntos de la recta. Para esto, complete la función siguiente:

```
q_path = inversekinematics_path(robot, path_p, path_o, q)
```

- Esta función se encarga de llamar internamente a `inverse_kinematics` para cada punto de la recta a partir de una semilla inicial `q`.
- Una vez resuelta esta tarea, el extremo del robot deberá seguir trayectorias rectas. Cuando el extremo se reorienta, se observará también un cambio lineal en la orientación.

Note que el script ahora utiliza la función `set_joint_target_trajectory`, a la que se le pasa una trayectoria articular. Note que esta función llama internamente a `set_joint_target_positions` con lo que el robot intentará realizar el control del brazo para llevarlo a la posición articular que se le indique.

La función `set_joint_target_trajectory` cuenta con diferentes opciones, por ejemplo, si especificamos:

```

robot.set_joint_target_trajectory(q_path, precision='all')
robot.set_joint_target_trajectory(q_path, precision='none')
robot.set_joint_target_trajectory(q_path, precision='last')

```

Cada una de las opciones anteriores confiere un comportamiento diferente al simulador. En la opción primera ('all'), la librería pyARTE esperará a que Coppelia haya alcanzado cada una de las posiciones articulares de la trayectoria con una precisión alta. En la segunda opción, pyARTE enviará los comandos de posición a Coppelia sin esperar a que se alcance cada posición articular en Coppelia. En la tercera opción, pyARTE únicamente esperará a que se alcance la última de las posiciones articulares de la trayectoria `q-path`.

Es muy importante tener en cuenta que si el robot choca con algún elemento del entorno o bien las posiciones articulares están fuera de su rango, no será posible que el sistema de control llegue a las posiciones articulares comandadas.

4.6. Actividades adicionales

4.6.1. Realice una aplicación de paletizado

Llegados a este punto, el alumno cuenta con las funciones necesarias para realizar una programación en el entorno realista de Coppelia. Se plantea que el robot coja todas las piezas de la cinta transportadora y las coloque sobre un pallet.

Ejercicio 4.6.1: APLICACIÓN DE PALETIZADO

Realice una aplicación de paletizado:

- Utilice el script `ur5_ikine.py` o `ur5_ikine_path.py`.
- Añada más `target_positions` y `target_orientations` y utilice el método de cinemática inversa para que el robot se desplace hasta los puntos deseados y recoja las piezas y las deposite en su lugar. Un posible resultado de la simulación se muestra en la Figura 4.2.

4.7. Resumen

Se resumen los conceptos más importantes trabajados en cada uno de los ejercicios.

- Cinemática inversa usando la solución **Moore-Penrose**. Esta solución falla si, en algún paso del algoritmo, la matriz JJ^T es no invertible.
 - Script `ur5_ikine.py`.
 - Escena `scenes/ur5.ttt`.
- Cinemática inversa usando la solución **Damped Moore-Penrose**. Se añade un término si el determinante de JJ^T se encuentra por debajo de un umbral.

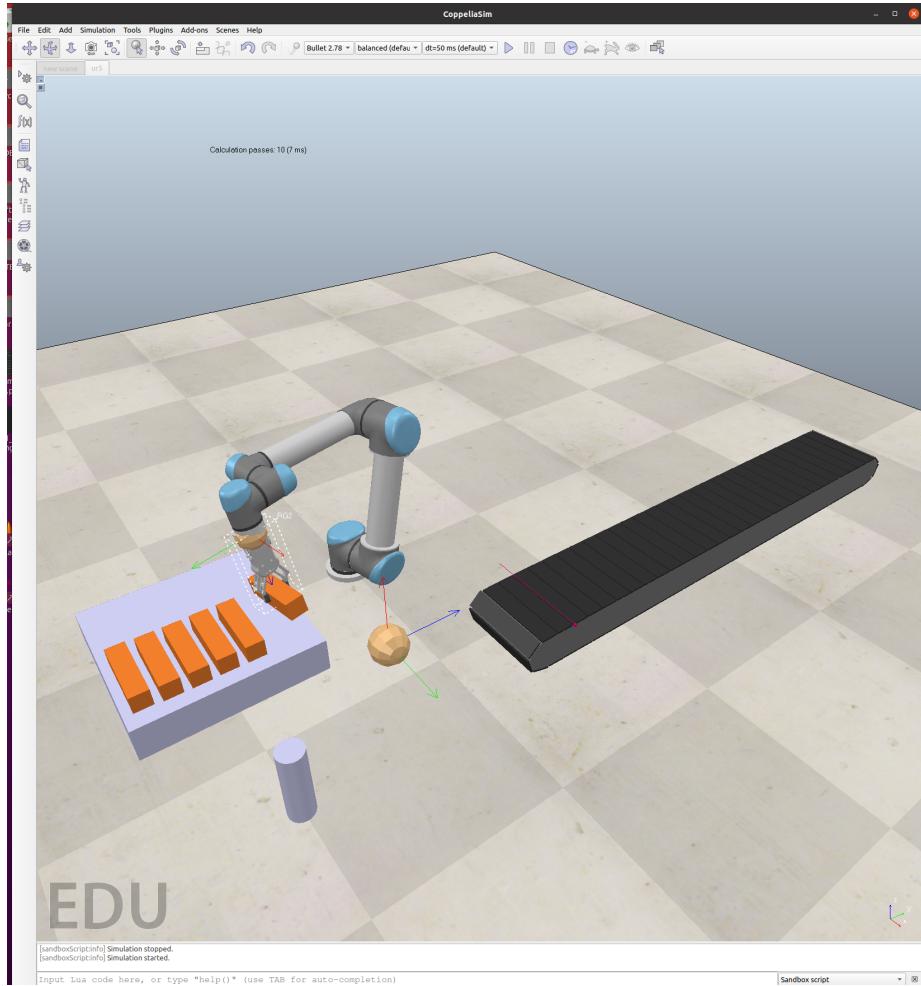


Figura 4.2: El robot UR5 en una aplicación de paletizado.

- Script `ur5_ikine.py`.
- Escena `scenes/ur5.ttt`.
- Cinemática inversa usando la solución **Transpuesta**. Se utiliza una solución alternativa basada en la matriz J^T .
 - Script `ur5_ikine.py`.
 - Escena `scenes/ur5.ttt`.
- **Una aplicación de paletizado con interpolación lineal entre posiciones/orientaciones.** A diferencia de los ejercicios anteriores, en este caso, el extremo del robot se mueve sobre líneas rectas en el espacio de la tarea. Se realiza una interpolación lineal de las trayectorias. Tenga en cuenta los comentarios realizados sobre la interpolación de las posiciones del extremo del robot y la interpolación de las orientaciones.
 - Script `ur5_ikine_path.py`.

- Escena `scenes/ur5.ttt`.

4.8. Funciones útiles de numpy

Nota: Se proporciona, a continuación, información para aclarar las operaciones con matrices más usuales usando la librería `numpy` de python:

- Importar la librería `numpy`:

```
import numpy as np
```

A partir de este momento se importa la librería `numpy` y nos podemos referir a ella como `np`.

- Creación de un vector:

```
q0 = np.array([-np.pi, 0, np.pi/2, 0, 0, 0])
```

- Creación de una matriz:

```
A = np.array([[1, 2], [3, 4]])
```

- Producto de Matrices:

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[1, 2], [3, 4]])
C = np.dot(A, B)
```

- Determinante de una matriz:

```
A = np.array([[1, 2], [3, 4]])
detA = np.linalg.det(A)
```

- Producto $A \cdot A^T$:

```
A = np.array([[1, 2], [3, 4]])
prodAAT = np.dot(A, A.T)
```

- Inversa:

```
A = np.array([[1, 2], [3, 4]])
invA = np.linalg.inv(A)
```

Ejemplos de operaciones en esta práctica:

- Multiplicar $A = J^T J$:

```
A = np.dot(J.T, J)
```

- Cálculo de la Least Squares Pseudoinverse $J^\dagger = J^T (JJ^T)^{-1}$:

```
Jp = np.dot(J.T, np.linalg.inv(np.dot(J, J.T)))
```

- Manipulabilidad:

```
manip = np.linalg.det(np.dot(J, J.T))
```

Capítulo 5

Aplicaciones industriales

5.1. Introducción

En una práctica anterior se exploró en detalle una simulación de una aplicación de paletizado. En esta práctica se presentan otras aplicaciones industriales utilizando el simulador Coppelia Sim y la librería pyARTE. Se da una explicación resumida sobre cada una de las aplicaciones presentadas, dejando al estudiante que explore de forma autónoma las soluciones y buscando que las mejore y amplíe.

Se presentan a continuación algunas aplicaciones industriales:

- Clasificación de objetos en base a su color.
- Pintura de vehículos automóviles.
- Soldadura TIG/MIG.

5.2. Clasificación de objetos en base a su color

Se propone la simulación de una aplicación de recogida y clasificación de objetos. Como punto de partida, se proporciona:

- Escena: `irb140.ttt`. Se proporciona una escena que incluye un robot IRB140 y una cinta transportadora (Figura 5.1). El script `CubeSpawner` crea objetos con un determinado color. El robot cuenta con una cámara que le permite capturar imágenes del entorno simulado.
- Script: `demos/irb140_color_classification.py`. El script utiliza las funciones:
 - `robot.get_image`: Obtiene una imagen de la cámara sobre la pinza del robot. La resolución, FOV y parámetros se pueden editar desde el menú de Coppelia. Se recomienda mantener una resolución baja para no ralentizar la simulación.
 - `robot.get_mean_color`: Devuelve el valor medio RGB de la imagen capturada por el robot (normalizado a un color unidad).

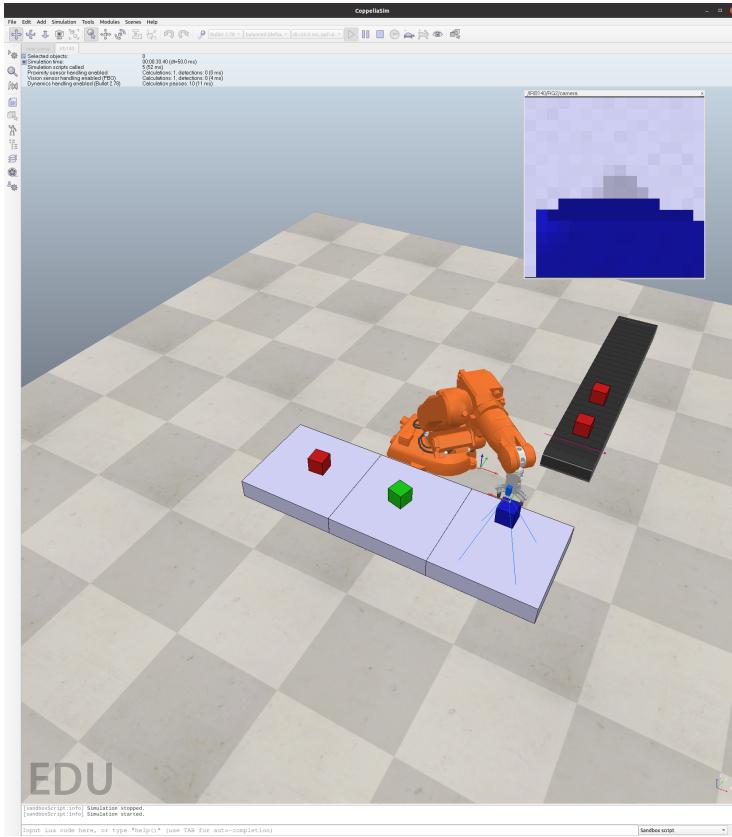


Figura 5.1: Una aplicación de clasificación en base al color.

Para clasificar cada pieza, el robot sitúa una cámara sobre la pieza y captura una imagen. Esta imagen se procesa para obtener un color medio. En base al color detectado (R, G o B) el robot deja la pieza en una zona de la escena.

Objetivos:

- **Objetivo 1:** Paletice las piezas en función de su color en tres montones separados.
- **Objetivo 2:** Considere que la posición de la pieza sobre la cinta transportadora es desconocida. Debe calcular la posición de la pieza utilizando la imagen capturada por el robot. Deberá calcularse el giro necesario para poder asir las piezas correctamente. Igual que antes, paletice las piezas en base a su color.

5.3. Pintura

Se propone la simulación de una aplicación de pintura en la producción de automóviles. Como punto de partida, se proporciona:

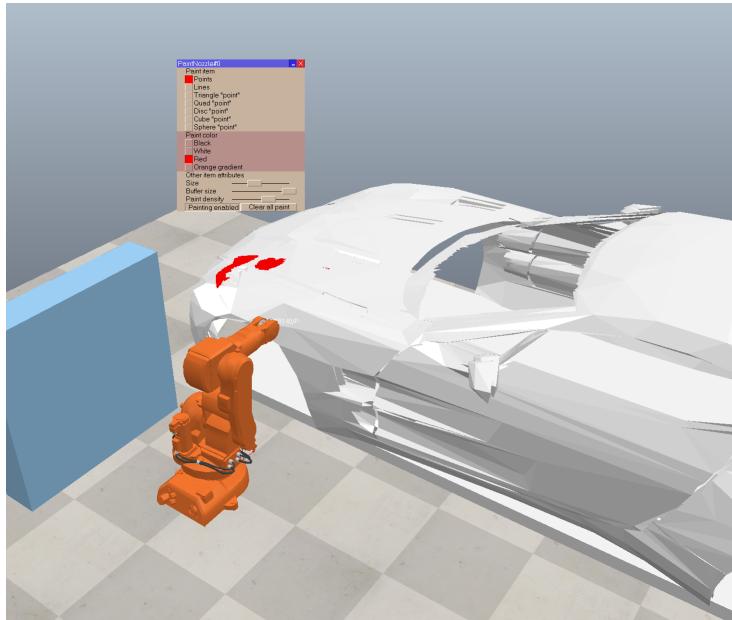


Figura 5.2: Un IRB140 en una aplicación de pintura.

- Escena: `irb140_paint_application.ttt`. Se proporciona una escena que incluye un robot IRB140 y una carrocería de un vehículo (Figura 5.2). Cuando se inicia la simulación, automáticamente, la pistola de pintura proyecta partículas sobre la carrocería, simulando un proceso de pintura industrial. En el modelo del objeto de la carrocería se ha incluido un movimiento que simula la cinta de transporte sobre la que se mueve el chasis a velocidad constante.
- Script: `demos/irb140_paint_application.py`. En el script, simplemente, se mueve el robot a diferentes posiciones articulares para simular el inicio del proceso de pintura.

Objetivos:

- **Objetivo 1:** Pinte uno de los lados de la carrocería.
- **Objetivo 2:** Proponga una aplicación de pintura con otro objeto. Para ello puede importar objetos desde el menú File – import – Mesh. Coppelia acepta objetos en formato obj y stl.

5.4. Soldadura

Se propone la simulación de una aplicación de soldadura (Figura 5.3). Como punto de partida, se proporciona:

- Escena: `irb140_welding.ttt`. Se proporciona una escena que incluye un robot IRB140 y una herramienta de soldadura en el extremo.

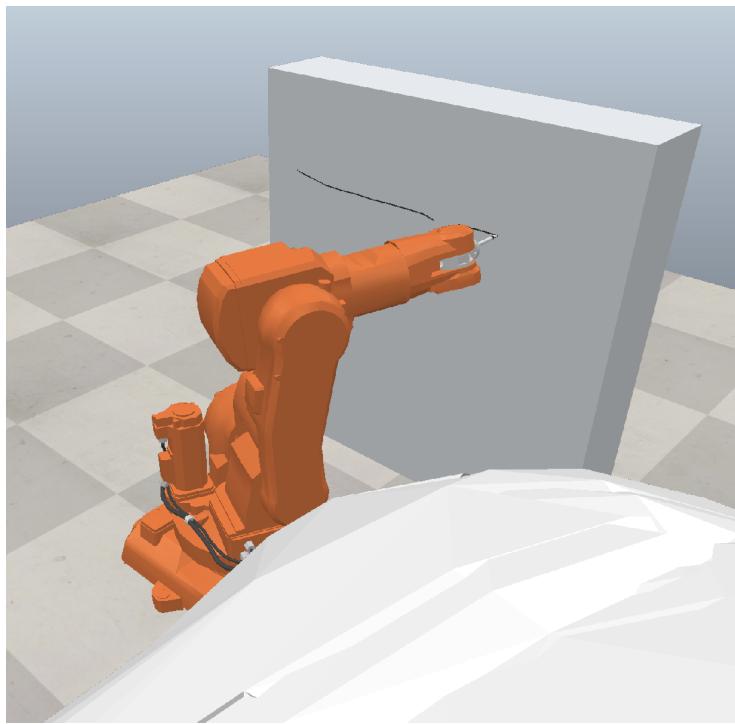


Figura 5.3: Un IRB140 en una aplicación de soldadura TIG/MIG.

- Script: `irb140_welding.py`. La *welding torch* se encuentra siempre activada en Coppelia. Cuando la herramienta se sitúa cerca de algún elemento de la escena, realizará la operación de soldadura pintando sobre el elemento más cercano.
- **Objetivo:** Realice varias trayectorias de soldadura sobre una pieza o bien el chasis de un vehículo. Es importante saber seleccionar las configuraciones iniciales de la solución para que toda la trayectoria se pueda realizar correctamente. Será necesario derivar un algoritmo para mejorar la cinemática inversa (con más puntos intermedios), buscando que la línea de soldadura sea lo más recta posible.

Capítulo 6

Cinemática inversa de un robot redundante

6.1. Objetivos

En esta práctica se persiguen los siguientes objetivos:

- Ensayar un método de cinemática inversa con un robot redundante de 7DOF.
- Realizar una aplicación de paletizado con un robot redundante.
- Calcular los movimientos del robot en el espacio nulo de una tarea con $m = 6$, considerando un espacio articular $n = 7$.
- Utilizar el espacio nulo para conseguir objetivos secundarios, por ejemplo:
 - Alejar las articulaciones de sus límites articulares.
 - Incrementar la manipulabilidad del mecanismo.

6.2. El robot KUKA IIWA

En la Figura 4.1 se presenta el robot KUKA LBR IIWA. En la Tabla 6.1 se muestran los parámetros DH que definen su cinemática. Un aspecto destacable de este robot es que cuenta con 7 GDL y, por tanto, se considera redundante para la mayoría de las tareas. No es posible obtener soluciones cerradas a su cinemática inversa en posición/orientación. En este caso, utilizaremos un método basado en la Jacobiana para obtener soluciones en posición/orientación.

6.3. Mueva el robot

Comenzaremos la práctica de forma similar a la anterior. Se propone al estudiante mover las articulaciones del robot, para familiarizarse con él. Nos damos cuenta de que, según se ha comentado antes, el robot cuenta con 7GDL.

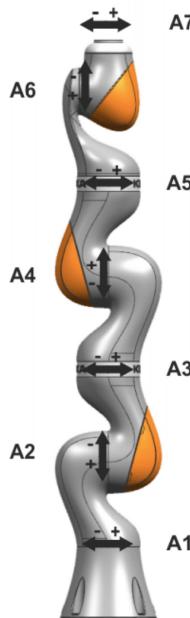


Figura 6.1: El robot KUKA LBR iiwa. Fuente: www.kuka.com. KUKA Roboter GmbH.

Ejercicio 6.3.1: MUEVA EL ROBOT KUKA

El ejercicio busca que el/la estudiante sea capaz de mover el robot en simulación mediante control articular.

- Abra la escena de Coppelia `scenes/kuka_14_R820_1.ttt`.
- **TAREA:** Mueva el robot con el script `kuka_move_robot.py`. Reja una pieza usando 'o' y 'c' para abrir/cerrar la pinza. Deposite una pieza en el pallet.

6.4. Cinemática inversa

Utilizaremos los mismos conceptos de cinemática inversa vistos en prácticas anteriores. Así, pues consideramos que el error en posición y orientación está dado por $\vec{e} = (\vec{e}_p, \vec{e}_o)$ y se cumple:

$$\vec{e} = J\Delta q$$

En el caso que nos ocupa, deseamos hallar Δq de la Ecuación (6.4) para poder hacer:

$$q_{k+1} = q_k + J^* \vec{e} \quad (6.1)$$

Transformación	θ (rad)	d (m)	a (m)	α (rad)
$0 \rightarrow 1$	q_1	0.36	0	$-\pi/2$
$1 \rightarrow 2$	q_2	0	0	$\pi/2$
$2 \rightarrow 3$	q_3	0.420	0	$\pi/2$
$3 \rightarrow 4$	q_4	0	0	$-\pi/2$
$4 \rightarrow 5$	q_5	0.4	0	$-\pi/2$
$5 \rightarrow 6$	q_6	0.0	0	$\pi/2$
$6 \rightarrow 7$	q_7	0.111	0	0

Tabla 6.1: Parámetros DH del robot KUKA LBR IIWA 14 R820.

donde suponemos que J^* es una solución a la Ecuación (6.4). Recordemos que en prácticas anteriores se dieron soluciones diferentes a esta ecuación:

- Una solución basada en la pseudo inversa Moore-Penrose.

$$\Delta q = J^\dagger \vec{e}$$

- Una solución basada en la pseudoinversa atenuada Moore-Penrose.

$$\Delta q = J^T (JJ^T + kI)^{-1} \vec{e}$$

- Una solución basada en la Jacobiana transpuesta.

$$\Delta q = \frac{\vec{e} \cdot JJ^T \vec{e}}{JJ^T \vec{e} \cdot JJ^T \vec{e}} J^T \vec{e}$$

6.5. Una aplicación de paletizado

Nos planteamos ahora probar la misma aplicación de paletizado que resolvimos en la práctica anterior.

Ejercicio 6.5.1: SOLUCIONES NO VÁLIDAS DE LA CINEMÁTICA INVERSA

Se pretende que el alumno observe el resultado de una simulación donde el robot colisiona consigo mismo.

- Abra, para ello, el fichero `practice3.1_kuka.py`.
- Abra la escena de Coppelia `scenes/kuka_14_R820_1.ttt`.
- Ejecute el script de python y observe el resultado.
- **Todo debería funcionar sin ninguna modificación y, en efecto, el robot debe colisionar consigo mismo.** En la Figura 6.2 se presenta esta situación.
- Note que la línea `robot.secondary_objective=False` está inhibiendo específicamente el algoritmo que evita esta colisión.

El resultado de la simulación se observa en la Figura 6.2. Durante la primera trayectoria del robot, la pinza colisiona con una parte del brazo. El sistema de simulación de Coppelia detecta esta situación y detiene el movimiento del brazo. Las trayectorias que realiza el robot, a continuación, no son demasiado satisfactorias, pues el robot colisiona o pasa cerca de los obstáculos con el consiguiente riesgo de colisión. Realizar una simulación ha sido, en esta ocasión, positivo, pues ejecutar esta trayectoria en el brazo real conlleva el riesgo de dañar el brazo.

Pensamos, a continuación, una forma de resolver este problema. Debemos pensar que en cada iteración del algoritmo, hacemos:

$$q_{k+1} = q_k + \Delta q \quad (6.2)$$

$$q_{k+1} = q_k + J^\dagger \vec{e} \quad (6.3)$$

Debemos recordar que, en la Ecuación 6.2 la solución $\Delta q = J^\dagger \vec{e}$ es de mínimos cuadrados, es decir, es aquella solución para la cual $|\Delta q|$ es mínima. Por tanto, podemos asegurar que, en todo instante de simulación, dado q_k el siguiente valor q_{k+1} será el más cercano a q_k y, por tanto, la solución q_{T+1} pasadas T iteraciones, también será la más cercana a la semilla inicial q_0 . En definitiva, no hay nada en este algoritmo (basado en la Ecuación 6.2) que impida al robot chocar o que sus articulaciones giren más allá de algún límite articular, salvo cambiar la semilla inicial q_0 . Esto no es siempre factible en una aplicación real, con lo que no resulta una solución apropiada. Por lo tanto, en este momento, debemos detenernos a:

- Recordar el proceso que se siguió en la anterior práctica para el cálculo de la cinemática inversa usando la pseudo inversa Moore-Penrose.
- Recordar el concepto de espacio nulo y su aplicación a este caso.
- Pensar en una solución basada en el empleo del espacio nulo. ¿De qué manera podemos mantener la posición/orientación del extremo pero que el robot tenga una pose mejor (que no existan colisiones)?

Debemos, por tanto, recordar que el método para la solución de la cinemática inversa está basado en la Jacobiana del manipulador. Recuerde que la solución, en cada instante, depende del valor inicial de semilla que se le proporcione al método. En un principio, deberíamos probar diferentes semillas iniciales q_0 para observar si, de esta manera, se puede obtener una solución válida.

Ejercicio 6.5.2: Una aplicación de paletizado. Semilla inicial

El ejercicio busca que el alumno observe las diferentes soluciones de la cinemática a las que se llega cuando se parte de diferentes semillas iniciales q_0 .

- Abra, para ello, el fichero `practice3.1_kuka.py`. Abra la escena de Coppelia `scenes/kuka_14_R820_1.ttt`.
- Varíe la posición articular inicial q_0 del robot y observe los resultados.

Variar la posición articular inicial q_0 no resulta la mejor solución al problema. En los apartados siguientes se explora una solución que explota el espacio nulo de la aplicación Jacobiana.

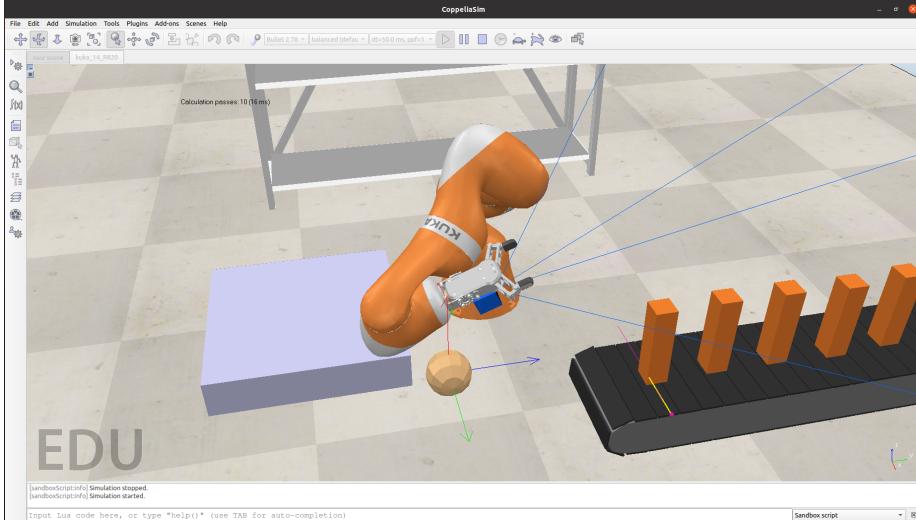


Figura 6.2: El robot KUKA LBR IIWA con una auto-colisión (pinza-robot).
Fuente: www.kuka.com.

6.6. Espacio nulo

Recordemos que la descomposición SVD de la Jacobiana del manipulador nos proporcionaba una forma sencilla de encontrar un vector \dot{q}_0 perteneciente al espacio nulo. Movernos con una velocidad \dot{q}_0 permite mover el brazo sin mover su extremo. De esta manera, se pueden hallar las infinitas soluciones de la cinemática inversa para una posición y orientación determinadas. En `numpy`, la descomposición SVD se puede calcular como:

```
u, s, vh = np.linalg.svd(J, full_matrices=True)
```

Nota: la función `np.linalg.svd` devuelve las matrices U , S y V^T (en efecto, V está ya transpuesta), de manera que se puede reconstruir la matriz original como:

$$J = USV^T$$

o, bien, en `numpy`:

```
J = np.dot(np.dot(u, s), vh)
```

Ejercicio 6.6.1: ESPACIO NULO

El ejercicio pretende que el alumno sea capaz de calcular el espacio nulo del robot para una determinada posición articular q . A continuación, deberá integrar las velocidades en el espacio nulo para mover el robot. Realice las siguientes tareas:

- Abra el fichero `practice_3.2_kuka.py`.
- Modifique la función `null_space(J)` para que devuelva un vector Δq alineado con el espacio nulo de J .
- Integre la velocidad del robot haciendo:

$$\dot{q} = q + qd$$

- Ejecute `practice_3.2_kuka.py` y observe los resultados en Coppelia. ¿Cómo es el movimiento del robot?
- Fíjese en la gráfica de posiciones articulares: ¿qué observa? Deberá observar si los valores de las articulaciones están siempre dentro de los límites del fabricante.
- Utilice el método `q=robot.apply_joint_limits(q)` para limitar los valores articulares del robot y observe el resultado.

Nota: asegúrese de que la velocidad de la articulación 3 es siempre positiva (o negativa). Note que el espacio nulo calculado con el método SVD es un vector unitario: es decir, no tenemos por qué esperar que la velocidad articular (Δq) sea continua. Note que deberá escalar esta velocidad articular para acomodarla a las velocidades máximas de cada articulación. En el anterior ejercicio, probablemente, hayamos obtenido una trayectoria que hacía al robot oscilar alrededor de un punto determinado. Puede obtener un movimiento en una dirección determinada asegurándose de que el signo de, al menos, una de las velocidades \dot{q}_i (Δq_i) sea constante (o bien positivo, o bien negativo).

Vemos, por otra parte, que es necesario considerar que, si saturamos el valor de alguna de las articulaciones, entonces no nos moveremos en el espacio nulo (y en consecuencia el extremo del robot se mueve).

6.7. Objetivos secundarios

En estos momentos estamos preparados para utilizar el espacio nulo con la idea de resolver el problema inicial: **la muñeca del robot choca con la pinza**. Así pues, una posible solución consideraría utilizar el espacio nulo para hallar soluciones de la cinemática inversa que hicieran que la muñeca y pinza no chocaran. Una posible forma de resolver este caso es conseguir que el valor del ángulo q_6 sea mayor, de manera que muñeca y pinza no colisionen. Para

conseguir esto, podríamos, por ejemplo, intentar minimizar la siguiente función:

$$\omega(q) = \sum_{i=1}^n \frac{(q_i - \hat{q}_i)^2}{q_{i,max} - q_{i,min}} \quad (6.4)$$

donde: q_i es el valor de la articulación i y \hat{q}_i es el valor deseado para la articulación i . El denominador es un normalizador basado en el rango de movimiento de cada articulación. Minimizar la Ecuación (6.4) busca que cada articulación q_i se encuentre lo más cercana posible a \hat{q}_i .

En la literatura estas funciones se denominan **objetivos secundarios** [3]. El objetivo primario consistiría en alcanzar una posición y orientación del extremo, mientras que podríamos tener otros objetivos secundarios, siempre que los vectores Δq pertenezcan al espacio nulo. También se puede plantear una versión ponderada de esta función:

$$\omega(q) = \sum_{i=1}^n \frac{k_i(q_i - \hat{q}_i)^2}{q_{i,max} - q_{i,min}}$$

donde $\vec{k} = \{k_1, k_2, \dots, k_n\}$ es un vector de pesos positivos que permite darle más o menos importancia a cada una de las articulaciones de interés. Esto nos permite hacer $k_i = 0$ si no nos importa el rango de movimiento de la articulación i . Igualmente, podemos hacer $k_i = 10$ si queremos que esa articulación se encuentre más cerca de su valor \hat{q}_i . En nuestro caso, el rango de movimiento de todas las articulaciones es idéntico, con lo que tenemos:

$$\omega(q) = \sum_{i=1}^n k_i(q_i - \hat{q}_i)^2$$

Al vector de valores deseados lo denominamos posición articular central \hat{q}_c :

$$\hat{q}_c = (\hat{q}_1, \hat{q}_2, \dots, \hat{q}_n)$$

Para minimizar la función $\omega(q)$, se propone calcular un vector articular como el siguiente:

$$\Delta q_0 = -\frac{\partial \omega}{\partial \vec{q}} \quad (6.5)$$

Note el sentido negativo de este gradiente, pues deseamos minimizar el valor de $\omega(q)$. En concreto, el vector Δq_0 se puede obtener mediante la diferenciación de $\omega(k)$ y tiene la siguiente expresión:

$$\Delta q_0 = -2[k_1(q_1 - \hat{q}_1), k_2(q_2 - \hat{q}_2), \dots, k_7(q_7 - \hat{q}_7)]^T$$

Seguidamente, se debe proyectar el vector Δq_0 al espacio nulo, con lo que el movimiento articular Δq_b que permite mantener las articulaciones cerca de una pose determinada es:

$$\Delta q_b = (I - J^\dagger J)\Delta q_0$$

Ejercicio 6.7.1: CINEMÁTICA INVERSA CON UN OBJETIVO SECUNDARIO

El ejercicio busca que el estudiante sea capaz de generar un vector de velocidades articulares Δq_0 que acerque al robot, en cada instante, a la posición articular central \hat{q} . Esta velocidad articular se genera con la Ecuación (6.5). Seguidamente (**importante**) debe proyectarse esa velocidad articular al espacio nulo, para que no produzca un movimiento activo del extremo.

Siga las siguientes instrucciones:

- Abra el script `practice3.3_kuka.py`.
- Observe la función `inversekinematics_line`. Esta función realiza una planificación en el espacio de la tarea. Después, para cada punto en el espacio de la tarea, se llama a la función `inversekinematics_secondary`.
- Esta función `inversekinematics_secondary`, en cada iteración, intenta reducir el error en posición/orientación y además minimizar la función $\omega(q)$
- Complete la función `diff_w_central` para calcular Δq_0 .
- Complete la función:

```
def null_space_projector(J):
```

La función deberá devolver un proyector al espacio nulo:

$$P = I - J^\dagger J$$

- Complete la función:

```
def minimize_w_central(J, q, qc, K):
```

Esta función deberá generar una velocidad Δq_0 de acuerdo con la Ecuación (6.5) que acerca al robot hacia la pose central. Además, deberá proyectar Δq_0 al espacio nulo para que no genere un movimiento activo del extremo.

La función `diff_w_central` debe devolver un vector Δq_0 que acerca las articulaciones a los valores especificados. Para encontrar el vector más cercano a Δq_0 que pertenezca al espacio nulo, será necesario proyectarlo al espacio nulo. De esta manera, tendremos un vector Δq_b que mueve al robot en el espacio nulo para que se minimice la función $\omega(q)$ y, por tanto, las articulaciones estén lo más cerca de sus posiciones centrales especificadas \hat{q}_i . Si P es un proyector al espacio nulo de J , entonces, la proyección al espacio nulo es:

$$\Delta q_b = (I - J^\dagger J)\Delta q_0 = P\Delta q_0$$

En las líneas especificadas anteriormente, `qda` es la parte activa de la velo-

ciudad que mueve el robot hacia la posición y orientación deseadas. La variable \mathbf{qdb} mueve al robot en el espacio nulo para minimizar $\omega(\mathbf{q})$. Siendo \mathbf{qdb} una velocidad, se toma arbitrariamente, como condición que la norma de \mathbf{qdb} sea la mitad de la norma de \mathbf{qda} . Esta última condición la variaremos en el ejercicio siguiente.

Ejercicio 6.7.2: CINEMÁTICA INVERSA CON UN OBJETIVO SECUNDARIO (bis)

Siga las siguientes instrucciones:

- Observe la función de `inversekinematics_secondary` y cambie la constante 0.5 por otros valores. Observe los resultados:

```
qda = robot.moore_penrose_damped(J, vwref)
qdb = minimize_w_central(J, q, qc, K)
qdb = 0.5 * np.linalg.norm(qda) * qdb
```

- Modifique la posición central \hat{q}_i en la función `inversekinematics2`. Observe los resultados:

```
qc = [0, 0, 0, 0, 0, 0, 0]
K = [1, 1, 1, 1, 1, 1, 1]
```

- El vector de pesos K en la función `inversekinematics_secondary`. Observe los resultados:

```
qc = [0, 0, 0, 0, 0, 0, 0]
K = [1, 1, 1, 1, 1, 1, 1]
```

- Compruebe el funcionamiento del script `practice3.3_kuka.py`. ¿Se consiguen los objetivos secundarios buscados en la aplicación de paletizado?
- Compruebe que las articulaciones no se salen de sus rangos de funcionamiento en ningún momento.

6.8. Otras funciones secundarias

La función que hemos utilizado hasta ahora nos obliga a definir una posición de referencia $\hat{\mathbf{q}}$. En ocasiones, esto no resulta conveniente. Nos gustaría definir una función que, simplemente, actuara cerca del límite articular y su influencia fuera mínima cuando la articulación se encontrara en sus valores centrales.

Existe, por tanto, otra forma de definir un objetivo secundario:

$$\omega(q) = \frac{1}{2n} \sum_{i=1}^n \frac{q_{i,max} - q_{i,min}}{(q_{i,max} - q_i)(q_i - q_{i,min})} \quad (6.6)$$

La forma de esta función, para una sola articulación q , se muestra en la Figura 6.3. Note que la función crece rápidamente cuando el valor se acerca al límite articular (lo cual resulta muy interesante) y es aproximadamente constante en el resto de valores. La función no está definida en los límites articulares $q_{i,max}$ y $q_{i,min}$, pero es fácil limitar su valor. En la Figura 6.3 se muestra la forma de esta función para una sola articulación, donde se observa la saturación de la función en los extremos. Si derivamos la Ecuación (6.6) respecto de la articulación q_i , obtenemos:

$$\frac{\partial w}{\partial q_i} = \frac{(q_{i,min} - q_{i,max})(-2q_i + q_{i,max} + q_{i,min})}{((q_i - q_{i,max})(q_i - q_{i,min}))^2} \quad (6.7)$$

El vector $q_b = -\frac{\partial w}{\partial \vec{q}}$ se forma como $\frac{\partial w}{\partial \vec{q}} = (\frac{\partial w}{\partial q_1}, \frac{\partial w}{\partial q_2}, \dots, \frac{\partial w}{\partial q_n})$. En la Figura 6.4 se muestra la derivada $\frac{\partial \omega(q)}{\partial q_i}$. Se muestra el valor de la derivada para una sola articulación. Note que la derivada toma valores cercanos a cero en una gran parte del rango de la articulación. Cerca del límite q_{min} , la derivada toma un valor negativo y positivo cerca del límite q_{max} . Merece la pena destacar que el objetivo es minimizar la Ecuación (6.6), de ahí el signo negativo en $-\frac{\partial w}{\partial \vec{q}}$. El algoritmo completo es:

$$\begin{aligned} \Delta q_a &= J^\dagger \vec{e} \\ \Delta q_b &= -\frac{\partial w}{\partial \vec{q}} \\ P &= (I - J^\dagger J) \\ \Delta q_0 &= P \cdot q_b \\ q_{i+1} &= q_i + \Delta q_a + \Delta q_0 \end{aligned}$$

Nótese que Δq_a es una solución activa de la cinemática inversa. Por otra parte, Δq_b corresponde a un vector diferencial calculado en base a la Ecuación (6.7). Nótese que Δq_b corresponde con un movimiento activo del extremo. Tras proyectarlo al espacio nulo, queda: $q_o = Pq_b$. La última suma puede ser escalada para atender a parámetros de suavidad en la trayectoria, por ejemplo:

$$q_{i+1} = q_i + \Delta q_a + \Delta q_0$$

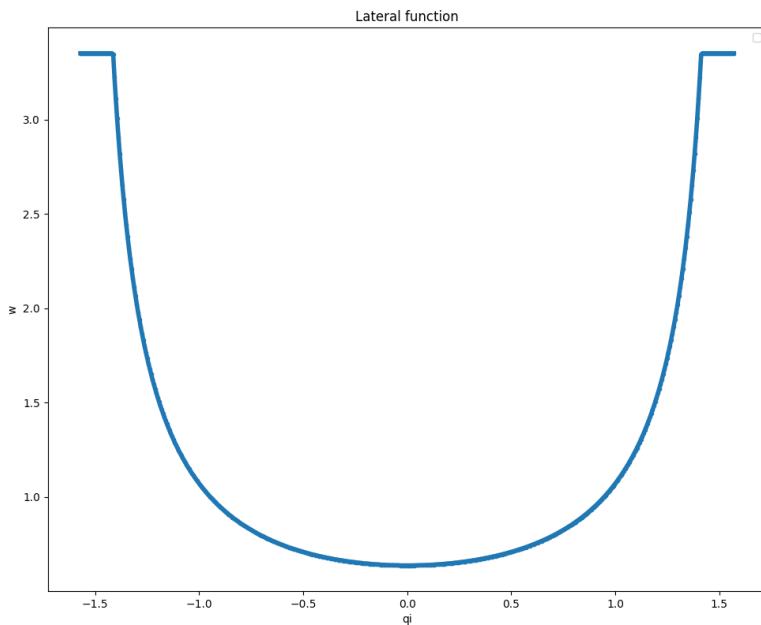


Figura 6.3: Una función secundaria para centrar las articulaciones

Ejercicio 6.8.1: OTRO OBJETIVO SECUNDARIO

Utilice la Ecuación (6.6) para evitar que el robot se acerque a sus límites articulares.

- Modifique el script `practice3.4_kuka.py` para completar el algoritmo usando la función diferencial de la Ecuación (6.7).
- **Importante:** deberá saturar el valor de la función cuando q_i se acerque a $q_{i,min}$ o $q_{i,max}$.
- Observe las diferencias en las trayectorias calculadas por el script `practice3.3_kuka.py` y `practice3.4_kuka.py`.

Concluimos en que las funciones utilizadas como objetivos secundarios pueden ser interesantes en función de la aplicación a la que vaya destinado el robot.

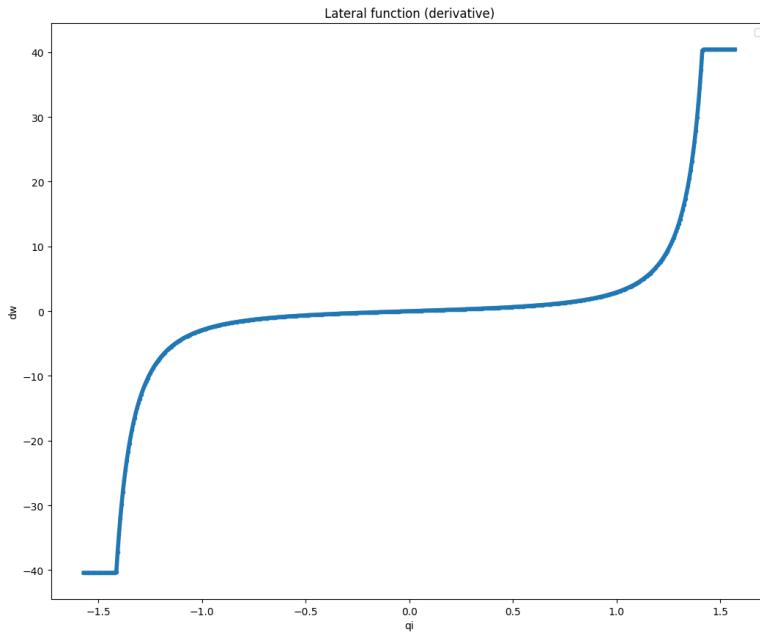


Figura 6.4: Derivada de la función secundaria para centrar las articulaciones

6.9. Ejercicios adicionales

Si volvemos a la definición de la cinemática directa de un robot en velocidad, tenemos:

$$\begin{pmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} = J \begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dots \\ \dot{q}_n \end{pmatrix}$$

Con esto, durante esta práctica, nuestro robot es redundante, puesto que:

- La tarea tiene dimensión $m = 6$.
- El robot tiene $n = 7$ GLD.
- Tenemos un espacio nulo de dimensión 1.

Sin embargo, no todas las aplicaciones en robótica requieren 6 GDL. Por ejemplo, ¿qué ocurre si las piezas en nuestra aplicación fueran cilíndricas, o esféricas?. También, además, piense que es innecesario mantener la orientación constante mientras la pieza se traslada desde la posición inicial hasta la final. Durante todo ese trayecto, las dimensiones adicionales del espacio nulo se podrían utilizar para maximizar otros objetivos secundarios, como, por ejemplo:

- Mantener las articulaciones alejadas de sus límites articulares (hecho en esta práctica).
- Maximizar la manipulabilidad $m = \sqrt{\det(JJ^T)}$ (lo que evitaría caer dentro de las singularidades).
- Alejar al robot de operadores/as humanos y/o obstáculos en el entorno. Este objetivo se aborda en otra de las prácticas de la asignatura.

Ejercicio 6.9.1: AMPLIANDO EL ESPACIO NULO

El ejercicio busca que el/la estudiante amplíe su conocimiento sobre el uso del espacio nulo. Siga las siguientes instrucciones:

- Cree un nuevo script llamado `practice3.5_kuka.py`.
- Redefina J como J_v :

$$\begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = J_v \begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dots \\ \dot{q}_n \end{pmatrix}$$

- ¿Cuál es la dimensión del espacio nulo ahora?
- Mueva el robot en las 4 dimensiones que, ahora, quedan libres en el espacio nulo del robot. Use el método descrito basado en `np.linalg.svd`.
- Anote sus propias conclusiones.
- Redefina ahora J como:

$$\begin{pmatrix} v_x \\ v_y \\ v_z \\ \omega_z \end{pmatrix} = J \begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dots \\ \dot{q}_n \end{pmatrix}$$

- ¿Cuál es la dimensión del espacio nulo ahora?
- Mueva el robot en las 3 dimensiones que, ahora, quedan libres en el espacio nulo del robot. Use el método descrito basado en `np.linalg.svd`.
- Anote sus propias conclusiones.

Ejercicio 6.9.2: AMPLIANDO EL ESPACIO NULO (II)

- Cree un nuevo script llamado `practice3.5_kuka.py`.

- Redefina J :

$$\begin{pmatrix} v_x \\ v_y \\ v_z \\ w_x \\ w_y \end{pmatrix} = J \begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dots \\ \dot{q}_n \end{pmatrix}$$

- Utilice ahora el espacio nulo para:

- Minimizar la distancia de la solución a la posición central \hat{q} .
- Maximizar la manipulabilidad del robot en toda la trayectoria.

En conclusión, el estudiante debe haber notado que la Jacobiana del manipulador “fija” las variables activas. Es decir, movernos en el espacio nulo no cambiará esas variables. Además, si el espacio de las variables activas es, por ejemplo, $m = 3$ y nuestro robot tiene $n = 7$ grados de libertad, tendremos, en total un espacio nulo con dimensión $n - m = 4$. Estas dimensiones se pueden obtener como las columnas de V que corresponden con valores singulares nulos de la descomposición SVD. En concreto, si hacemos:

$$\begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = J_v \begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dots \\ \dot{q}_n \end{pmatrix}$$

Entonces, podemos movernos:

- A posición constante $(v_x, v_y, v_z) = \vec{0}$, variando la orientación en el eje X .
- A posición constante $(v_x, v_y, v_z) = \vec{0}$, variando la orientación en el eje Y .
- A posición constante $(v_x, v_y, v_z) = \vec{0}$, variando la orientación en el eje Z .
- A posición constante $(v_x, v_y, v_z) = \vec{0}$, variando la última dimensión del espacio nulo que queda libre.

6.10. Resumen

Se resumen aquí los objetivos de cada uno de los ejercicios, así como los scripts y escenas implicados.

- **Ejercicio 3.1:** Busca que el estudiante se familiarice con el movimiento del robot.

- Escena: `scenes/kuka_14_R820_1.ttt`
- Script: `kuka_move_robot.py`

- **Ejercicio 3.2:** Busca que el estudiante compruebe que algunas soluciones de la cinemática inversa son no válidas.

- Escena: `scenes/kuka_14_R820_1.ttt`
 - Script: `practice_3.1_kuka.py`
- **Ejercicio 3.3:** El alumno deberá observar qué ocurre cuando se varía la semilla inicial q_0 y compruebe a qué soluciones de la cinemática inversa se llega.
- Escena: `scenes/kuka_14_R820_1.ttt`
 - Script: `practice_3.1_kuka.py`
- **Ejercicio 3.4:** Se propone al estudiante que mueva al robot en el espacio nulo.
- Escena: `scenes/kuka_14_R820_1.ttt`
 - Script: `practice_3.2_kuka.py`
- **Ejercicio 3.5:** Se propone al estudiante que resuelva la cinemática inversa del robot teniendo en cuenta un objetivo secundario. El objetivo secundario busca acercar al robot hacia una solución de ejemplo \hat{q} .
- Escena: `scenes/kuka_14_R820_1.ttt`
 - Script: `practice_3.3_kuka.py`
- **Ejercicio 3.6:** Se propone al estudiante que realice pruebas variando las constantes K y qc :
- Escena: `scenes/kuka_14_R820_1.ttt`
 - Script: `practice_3.3_kuka.py`
- **Ejercicio 3.7:** Se propone al estudiante que compruebe si se consiguen los objetivos de la aplicación de paletizado al usar los objetivos secundarios.
- Escena: `scenes/kuka_14_R820_1.ttt`
 - Script: `practice_3.3_kuka.py`
- **Ejercicio 3.8:** Se propone al estudiante que utilice una función secundaria diferente para apartar las articulaciones de sus valores máximos.
- Escena: `scenes/kuka_14_R820_1.ttt`
 - Script: `practice_3.4_kuka.py`
- **Ejercicio 3.9:** Se propone al estudiante que mueva el robot en un espacio nulo ampliado. En este caso, se considera $m = 3$ y, por tanto, la dimensión del espacio nulo es 4.
- Escena: `scenes/kuka_14_R820_1.ttt`
 - Script: `practice_3.5_kuka.py`

Capítulo 7

Planificación y obstáculos

7.1. Objetivos

En esta práctica se persiguen los siguientes objetivos:

- Implementar un método de cinemática inversa para robots redundantes.
Además,
- Implementar un método de cinemática inversa que utilice el espacio nulo para mover el robot y:
 - Evite colisiones con objetos estáticos en el entorno en su extremo efector.
 - Evite colisiones de todo el robot con el entorno.
 - Realizar una aplicación de seguimiento de líneas evitando colisiones.

7.2. Introducción

El objetivo que perseguimos, en general, consiste en mover al robot sin que ninguna parte de él colisione con el entorno mientras, al mismo tiempo, se realiza una tarea. En la literatura encontraremos diferentes formas para mover al robot en presencia de obstáculos. Estos métodos se pueden clasificar, inicialmente en:

- **Métodos reactivos:** Cuando el robot reacciona en cada instante de simulación. Generalmente, estos métodos consideran algún tipo de función de potencial que aleja al extremo del robot de los obstáculos o bien aleja también al resto de eslabones del robot.
- **Métodos deliberativos:** Cuando existe una planificación del extremo que, de forma planificada, evita los obstáculos.

Plantearemos en esta práctica una combinación de ambos métodos para resolver el objetivo que nos hemos planteado.

7.3. Planificación en presencia de obstáculos

Consideramos el problema de realizar una planificación en el espacio de la tarea. En nuestra planificación, deseamos:

- Comenzar desde una posición y orientación del extremo \vec{p}_a , Q_a .
- Finalizar en una posición y orientación del extremo \vec{p}_b , Q_b .
- Evitar una esfera (obstáculo) en la posición \vec{p}_e .

7.3.1. Interpolación de posiciones y orientaciones

El algoritmo para crear la trayectoria se presenta en las Figuras 7.1 y 7.2. Comienza planificando una serie de posiciones y orientaciones sobre una recta. Comenzamos considerando que la velocidad lineal a la que se mueve el extremo del robot es velocidad lineal \vec{v}_e con $|\vec{v}_e| = v_{max}$. Es frecuente que el extremo siga un perfil trapezoidal de velocidad sobre la recta. En este caso, con el objetivo de simplificar el problema, consideraremos que la velocidad sobre la recta es constante y el tiempo total para realizar el movimiento será:

$$t_{total} = \frac{|\vec{p}_b - \vec{p}_a|}{v_{max}}$$

Por otra parte, cada intervalo de simulación en Coppelia es de δt (s). En particular, $\delta t = 0.05$ s es el intervalo estándar de simulación en Coppelia. En consecuencia, deberemos planificar n puntos sobre la recta:

$$n = \frac{t_{total}}{\delta t}$$

y redondeamos al siguiente entero:

$$n = ceil\left(\frac{t_{total}}{\delta t}\right)$$

Suponga que la variable $c \in [0, 1]$, entonces, un punto \vec{p}_i perteneciente a la recta entre \vec{p}_a y \vec{p}_b es:

$$\vec{p}_i = (1 - c) \cdot \vec{p}_a + c \cdot \vec{p}_b \quad (7.1)$$

En consecuencia, generar n puntos sobre la recta implica generar n valores de c distribuidos uniformemente en $[0, 1]$. La librería `numpy` nos permite hacer esto fácilmente con:

```
c = np.linspace(0, 1, n)
```

Este planificador de trayectorias es, en extremo sencillo. En concreto, no se comprueba si el robot es capaz de alcanzar esa velocidad lineal en el extremo, pues, recordemos que:

$$\begin{pmatrix} \vec{v} \\ \vec{\omega} \end{pmatrix} = J \dot{q}$$

entonces, suponiendo J invertible:

$$\dot{q} = J^{-1} \begin{pmatrix} \vec{v}_{max} \\ \vec{\omega} \end{pmatrix}$$

Debe ocurrir que $|\dot{q}_i|$, $\forall i$ en el camino, debe ser menor que la máxima velocidad articular especificada por el fabricante del robot.

En relación con la orientación, suponga que la orientación en el comienzo de la línea está dada por la matriz de orientación R_a . En el final de la recta, el extremo del robot deberá tener la orientación R_b , en este caso, **la siguiente expresión no es correcta:**

$$R_i = (1 - c) \cdot R_a + c \cdot R_b$$

pues la matriz resultado debe ser ortonormal.

Una posible forma de realizar esta interpolación entre dos orientaciones diferentes plantea el uso de cuaterniones y la siguiente expresión. Así, suponga que la orientación en el inicio de la trayectoria está dada por el cuaternion Q_a y Q_b representa la orientación al final de la trayectoria. Entonces, una orientación intermedia entre las dos puede ser generada a través de la fracción c como:

$$Q = Q_a \frac{\sin((1 - c)\theta)}{\sin(\theta)} + Q_b \frac{\sin(c\theta)}{\sin(\theta)} \quad (7.2)$$

, siendo θ el ángulo sostenido por el arco entre los dos cuaterniones, calculado como el producto escalar entre dos cuaterniones:

$$\cos \theta = Q_a \cdot Q_b \quad (7.3)$$

Ejercicio 7.3.1: Planificar posiciones y orientaciones en el espacio de la tarea

El ejercicio persigue que el estudiante realice una función para planificar una recta en el espacio en posiciones y orientaciones. Instrucciones:

- Abra la escena `kuka_14_R820_2.ttt`. Abra el script `practice4.1_kuka.py`.
- **TAREA 1:** Debe crear la función `generate_target_positions(pa, pb, n)`: y la función `generate_target_orientations(Qa, Qb, n)`: que implemente las Ecuaciones (7.1) y (7.2).
- **TAREA 2:** El script `practice4.1_kuka.py` está preparado para llamar al método que resuelve la cinemática inversa para todos los puntos del camino calculado.

```
robot.inversekinematics_path(target_positions=path[0],
                             target_orientations=path[1], q0=q0)
```

- **TAREA 3:** Represente los puntos de la recta en un gráfico de `matplotlib`. Use la función de pyARTE:

```
plot3d(p_positions[:, 0], p_positions[:, 1], p_positions[:, 2])
```

7.3.2. Funciones de potencial

Vamos a definir una función de potencial centrada en la esfera con posición \vec{p}_s . Asuma que, esta función se evalúa para un punto del espacio \vec{p}_i y que $\rho =$

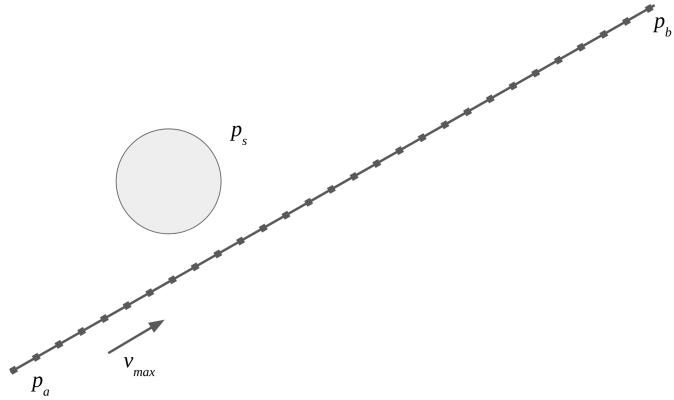


Figura 7.1: Una recta en el espacio de la tarea y un obstáculo en forma de esfera.

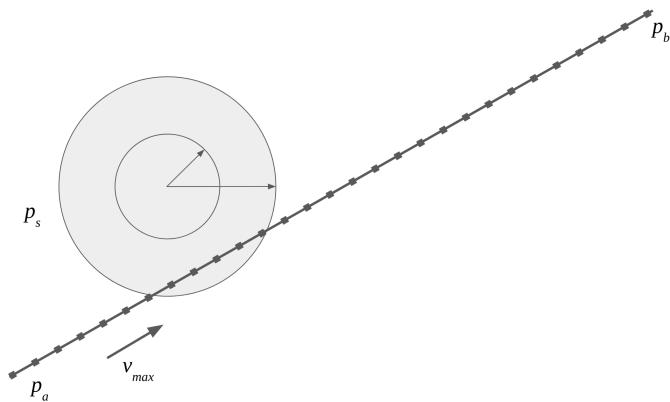


Figura 7.2: Una recta en el espacio de la tarea y un obstáculo en forma de esfera (se muestra el obstáculo recrecido).

$|\vec{p}_s - \vec{p}_i|$ es la distancia hasta el centro de la esfera. Con esto definimos una fuerza de repulsión \vec{f}_r hacia el exterior de la esfera:

$$\vec{f}_r = \begin{cases} K \left(\frac{1}{\rho_{min}} - \frac{1}{\rho_{max}} \right) \vec{u} & \text{si } \rho < \rho_{min} \\ K \left(\frac{1}{\rho} - \frac{1}{\rho_{max}} \right) \vec{u} & \text{si } \rho_{min} < \rho < \rho_{max} \\ 0\vec{u} & \text{si } \rho > \rho_{max} \end{cases} \quad (7.4)$$

donde:

- K : es una constante que permite ajustar la velocidad máxima de repulsión.
- ρ_{min} : es una distancia mínima (coincide con el radio de la esfera). Esta constante $\rho_{min} > 0$ se define, de tal manera, que fija un valor máximo del módulo de \vec{v}_r .
- ρ_{max} : es la distancia máxima de influencia (a esta distancia el módulo de la fuerza es nulo).

La forma de esta función de potencial se observa en la Figura 7.4.

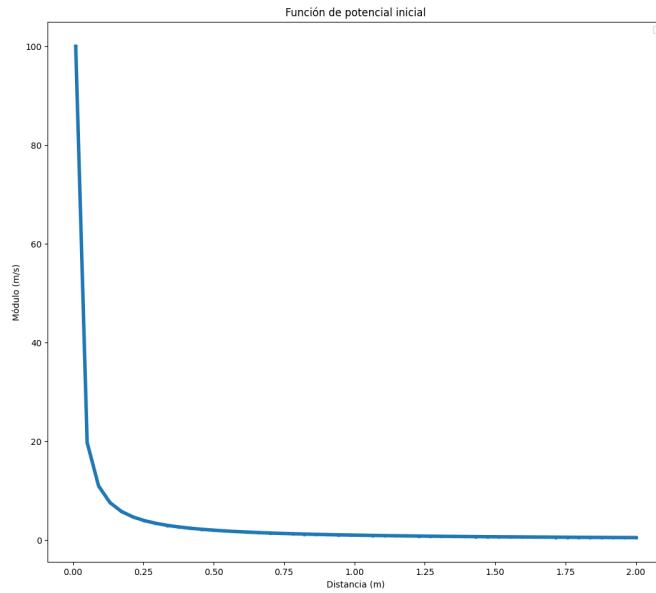


Figura 7.3: Una función de potencial inicial.

Ejercicio 7.3.2: Programe una función de potencial

El ejercicio persigue que el estudiante cree una función de potencial y la visualice. Instrucciones:

- Abra la escena `kuka_14_R820_2.ttt`. Abra el script `practice4.1_kuka.py`.
- **TAREA 1:** Cree una función de potencial de acuerdo con la Ecación (7.4).
- Indique razonables para los parámetros. P.e. $\rho_{min} = 0.1$ (m) y $\rho_{max} = 0.5$ (m).
- **TAREA 2:** Plotee la magnitud de la función de potencial en función de la distancia a la esfera. Use la función `plot_xy` de pyARTE. Puede comprobar la forma de esta función con el siguiente código:

```
f = []
r = np.linspace(0.01, 1.0, 200)
for ri in r:
    f.append(potential(ri))
plot_xy(r, f, title='Función de potencial inicial')
```

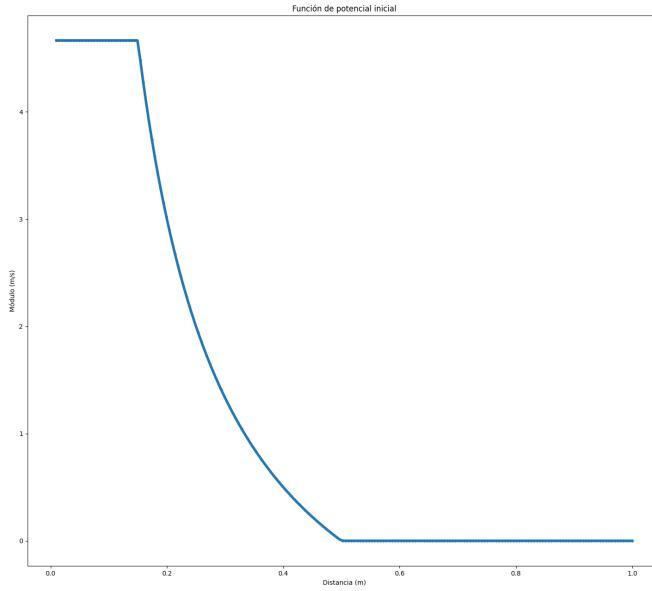


Figura 7.4: Una función de potencial mejorada (Ecuación 7.4).

7.3.3. Funciones de potencial y planificación

En este punto, fijándonos en la Figura 7.2, deseamos que la función de potencial se aplique a los puntos de la recta que se encuentran en el interior de la esfera exterior. Deseamos que esos puntos que interfieren con la esfera exterior se alejen de su zona de influencia. La forma de hacer esto es calcular, para cada punto \vec{p}_i de la trayectoria recta se calcula un nuevo punto como:

$$\vec{p}_i^* = \vec{p}_i + \Delta s \vec{f}_r \quad (7.5)$$

donde Δs es una constante de integración. Este proceso se repite hasta que ningún punto de la recta se mueve. En la Figura 7.5 se presenta un posible resultado del algoritmo.

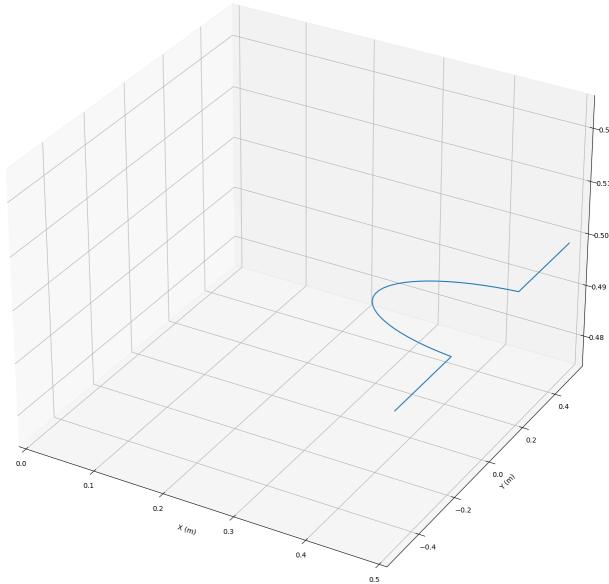


Figura 7.5: Los puntos de la recta movidos según una función de potencial.

Ejercicio 7.3.3: Aleje los puntos de la recta

El ejercicio persigue que el estudiante modifique los puntos de la trayectoria según una función de potencial:

- Abra la escena `kuka_14_R820_2.ttt`. Abra el script `practice4.1_kuka.py`.
- **TAREA 1:** Se proporciona una función que mueve los puntos alejándolos de la esfera: `move_target_positions_obstacles`
- **TAREA 2:** Plotee los nuevos puntos de la trayectoria. Use la función `plot_xy` de pyARTE.
- **TAREA 3:** Haga que el robot siga los puntos de esta nueva trayectoria y compruebe si colisiona con la esfera.
- **TAREA 4:** Pruebe variando la posición de la esfera y las posiciones iniciales y finales de la trayectoria del robot. Observe los resultados. Pruebe con diferentes posiciones de la esfera. Por ejemplo:

```
sphere.set_object_position([0.55, 0.0, 0.45])
sphere.set_object_position([0.5, 0.0, 0.5])
sphere.set_object_position([0.5, 0.0, 0.4])
sphere.set_object_position([0.35, 0.0, 0.4])
```

Como resultado del ejercicio anterior, debe comprobar que, en función de la semilla inicial q_0 el robot podrá seguir la trayectoria planificada o no, ya que únicamente estamos buscando que la trayectoria del extremo no colisione con el obstáculo. Puede comprobar que si la trayectoria del extremo pasa por, exactamente, el centro de la esfera, el método fallará.

7.4. Distancia del robot a los obstáculos

Coppelia proporciona funciones que permiten obtener la mínima distancia que existe entre un objeto y otro durante la simulación. Nótese que hallar la distancia mínima entre un punto y una recta es una tarea sencilla. Igualmente es fácil encontrar la distancia mínima entre un punto y la superficie de una esfera. Sin embargo, resulta muy complejo obtener la distancia mínima entre dos objetos con formas complejas (como ocurre con el KUKA iiwa LBR). Conviene, en este momento, realizar el siguiente ejercicio:

Ejercicio 7.4.1: Objetos más cercanos

El ejercicio plantea que el alumno visualice los objetos más cercanos al robot en la simulación de Coppelia.

- Abra la escena de Coppelia `LBR_iiwa_14_R820_2.ttt`.
- Abra el script `kuka_move_robot.py`.
- Mueva al robot a una posición articular q en Coppelia y observe que esta distancia mínima se marca con una recta azul (Figura 7.6). El valor de la distancia se observa también en un gráfico que se actualiza a tiempo real (Figura 7.6).

Asuma que cuenta con una función que calcula la distancia más corta entre todos los objetos del entorno y el eslabón 1 del robot. Llamaremos a esta distancia: $d_{1,min}$. Considere que la misma función permite calcular la mínima distancia entre todos los objetos y el eslabón 2 ($d_{2,min}$). Por tanto, la distancia entre el robot y todos los obstáculos del entorno es d_o :

$$d_o = \arg \min_i d_{i,min}$$

Por tanto, nos interesa conocer cuál es la mínima de las distancias calculadas para cada uno de los eslabones del robot. El cálculo de esta distancia mínima lo realiza Coppelia, usando un script en lenguaje Lua que está asociado al robot. Estos *scripts* se denominan “child scripts” en el argot de Coppelia. Puede abrir este script haciendo “doble click” sobre el ícono mostrado en la Figura 7.7. Aparecerá el código mostrado en la Figura 7.8. Puede echar un vistazo a este código. Todos estos cálculos costosos de distancia los realiza CoppeliaSim de forma independiente al código de Python. Un punto interesante es que, al usar esta función en Lua:

```
sim.setFloatSignal('min_distance_to_objects', dist_var[7])
```

se genera una variable que podrá ser leída desde nuestra API de python. En el argot de Coppelia, esta variable se denomina “señal” (*signal*). Las señales

son variables globales que pueden ser leídas fácilmente desde la API de python. En concreto, en cualquier instante de simulación, podremos leer esta distancia desde Python con esta función:

```
def get_min_distance_to_objects(self):
    error, distance = sim.simxGetFloatSignal(self.clientID,
                                              sim.simx_opmode_oneshot_wait)
    return distance
```

En este código se le está diciendo a Coppelia que nos devuelva la variable que hemos llamado `min_distance_to_objects` en la función `setFloatSignal`.

Nota: La distancia mínima a los obstáculos puede ser, en ocasiones, poco útil para la tarea que se está resolviendo. En ocasiones se utiliza la distancia media a los obstáculos. Esta distancia la definimos como:

$$\hat{d}_o = \frac{1}{n} \sum_{i=1}^n d_{i,min}$$

Ejercicio 7.4.2: Distancia mínima a los objetos

En este ejercicio se propone al estudiante que modifique el programa de Lua para tener en cuenta otras distancias.

- Abra la escena de Coppelia `LBR_iwa_14_R820_2.ttt`.
- Abra el child script asociado al robot Kuka.
- Observe qué eslabones se están evaluando y qué objetos.
- ¿Por qué no se evalúa la distancia de los objetos al eslabón 0 (base)?
- ¿Por qué no se evalúa la distancia de los objetos al extremo efector?
- Añada un objeto a la escena y modifique el código de Lua para evaluar la distancia del objeto al robot. Para añadir un objeto, simplemente, haga click con el botón derecho y elija: “add” - “primitive shape” - (Sphere, disc, cuboid).
- Note que Lua puede obtener un manejador (handle) de cada objeto a través de su nombre.
- Utilice el script `kuka_move_robot.py` para mover el robot y observar el comportamiento.

7.5. Distancia a los obstáculos en el espacio nulo

Hasta este punto, hemos propuesto una manera de evitar que el extremo del robot colisione con un objeto en el entorno (la esfera). El método presentado genera una trayectoria de forma deliberativa que aparta al robot de la esfera. Nos hemos dado cuenta de que, de esta manera, **podemos evitar las colisiones del extremo del robot, pero el resto del robot podría colisionar con algún objeto del entorno**. Se propone, en lo siguiente, utilizar el espacio nulo

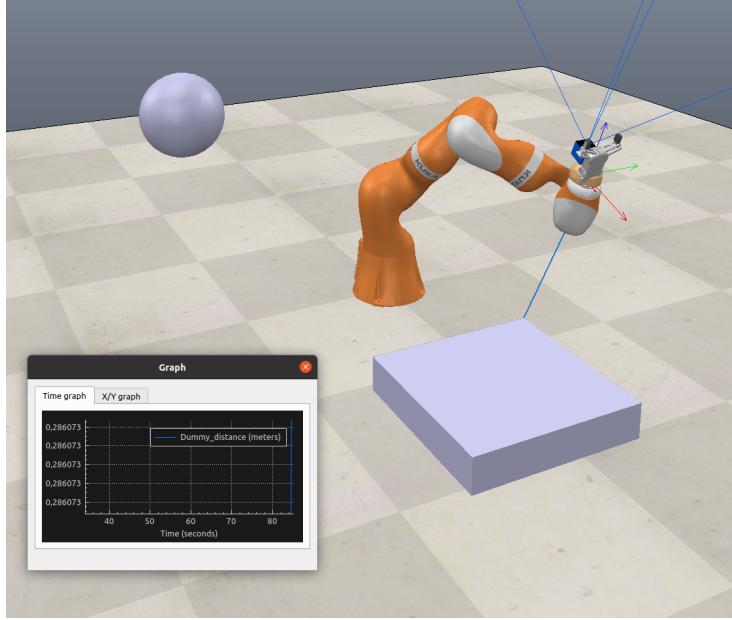


Figura 7.6: Mínima distancia del brazo a los obstáculos.

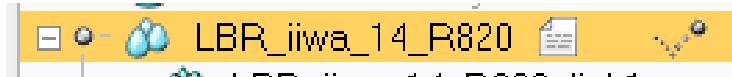


Figura 7.7: Abra el “child script” de Lua asociado al robot.

para conseguir que el resto del robot se aleje lo más posible de los obstáculos. Así pues, planteamos mover el brazo en el espacio nulo para hacer que todo el brazo (salvo el extremo) se aleje de los obstáculos. Dicho de otra manera, volviendo a la solución general de la cinemática inversa:

$$\dot{q} = \dot{q}_a + \dot{q}_b = J^\dagger \vec{v}_{ref} + (I - J^\dagger J)\dot{q}_0$$

La velocidad articular activa \dot{q}_a moverá el extremo del robot siguiendo una velocidad \vec{v}_{ref} sobre la trayectoria planificada. Por otra parte, la velocidad pasiva \dot{q}_b no moverá el extremo del robot, pero sí podemos elegirla para que aleje al brazo de los obstáculos. En concreto utilizaremos para ayudarnos de una función de potencial calculada sobre la distancia del robot al obstáculo más cercano d_o :

$$f_o = \begin{cases} K \left(\frac{1}{d_{min}} - \frac{1}{d_{max}} \right) & \text{si } d_o < d_{min} \\ K \left(\frac{1}{d_o} - \frac{1}{d_{max}} \right) & \text{si } d_{min} < d_o < d_{max} \\ 0 & \text{si } d_o > d_{max} \end{cases} \quad (7.6)$$

Esta función de potencial es un escalar, a diferencia de la función de potencial anteriormente utilizada. Nos interesa mover el robot en la dirección en la que f_o se minimice.

Nótese que no es posible diferenciar la distancia d_o , pues no conocemos una función $d_o = d_o(q)$. Es decir, si contáramos con esta función, podríamos plantear

el cálculo de:

$$\frac{\partial d_o}{\partial \vec{q}}$$

y proyectar esta derivada al espacio nulo. Esta idea no es realizable en la práctica, por no contar con una función conocida para d_o , por tanto, será necesario mover leer el valor de d_o de Coppelia, mover al robot en el espacio nulo, y repetir la acción. La dirección de movimiento en el espacio nulo será aquella que maximice d_o . La idea de usar una función como la mostrada en la Ecuación (7.6) es anular la influencia del obstáculo cuando todo el robot se encuentra alejado de los obstáculos. De esta manera se consiguen trayectorias más simples y suaves.

En esencia, el ejercicio siguiente 4.6 busca mover al robot en el espacio nulo y hallar la posición articular q que maximiza las distancias a los obstáculos, es decir:

$$\arg \max_q (d_o)$$

Ejercicio 7.5.1: Maximice la distancia en el espacio nulo

El ejercicio persigue mover al robot en el espacio nulo y hallar q de manera que se maximice la distancia a todos los obstáculos. Es decir: que el robot tenga una pose en la que se aleje, lo más posible, de los obstáculos. Instrucciones:

- Abra la escena `kuka_14_R820_2.ttt`. Abra el script `practice4.2_kuka.py`. Observe la función `def maximize_distance_to_obstacles(robot, q)`. Esta función deberá encontrar la posición articular q para la cual la distancia del robot a los objetos se maximiza. Esta función, llama, a su vez, a la función `move_null_space` que realiza una serie de movimientos en el espacio nulo del robot.
- **TAREA 1:** Debe modificar ligeramente la función `move_null_space` de manera que se almacene en un array `ds` todas las distancias mínimas en cada paso del algoritmo. En cada instante de simulación, puede obtener la distancia mínima del robot con todos los objetos del entorno con `d = robot.get_min_distance_to_objects()`.
- **TAREA 2:** Modifique ligeramente la función `find_min_distance`, de manera que permita obtener la posición articular del array `qs` que maximiza la distancia.
- **TAREA 3:** Finalmente, observe el resultado de ejecutar el script. Note que, en la posición inicial se ha maximizado la distancia del robot a los obstáculos. No obstante, en cada paso siguiente de la trayectoria, el robot puede colisionar con el suelo u otros objetos, pues no hay nada que, durante el cálculo de la trayectoria, mueva al robot alejándolo de los obstáculos (en cada instante de simulación).

7.6. Función de potencial aplicada a la distancia a los obstáculos

En el ejercicio 4.6, el estudiante deberá haber notado que la combinación de ambos movimientos $\Delta q_a + \Delta q_b$ se hace de forma arbitraria. Con lo que deberíamos preguntarnos:

- ¿Debemos aplicar una velocidad Δq_b con $|\Delta q_b| > 0$ en todos los pasos del algoritmo? Es decir, ¿debemos intentar alejarnos siempre de los obstáculos?.
- ¿Por qué no aplicamos esta estrategia únicamente cuando estemos cerca de los obstáculos?
- ¿Por qué no es la cantidad diferencial $|\Delta q_b|$ inversamente proporcional a la distancia a los obstáculos? ¿Por qué no es proporcional $|\Delta q_b|$ a la velocidad \dot{d}_o (velocidad con la que se acercan/alejan los obstáculos)? Nota: Estas últimas preguntas no se abordan en esta práctica.

Una forma de contestar a estas cuestiones consisten en calcular una función de potencial utilizando la distancia d_o .

$$\Delta q_b^* = \kappa(d_o)\Delta q_b$$

siendo $\kappa(d_o)$ una función de potencial de la distancia d del robot al obstáculo más cercano:

$$\kappa(d_o) = \begin{cases} K_o \left(\frac{1}{d_{min}} - \frac{1}{d_{max}} \right) & \text{si } d_o < d_{min} \\ K_o \left(\frac{1}{d_o} - \frac{1}{d_{max}} \right) & \text{si } d_{min} < d_o < d_{max} \\ 0 & \text{si } d_o > d_{max} \end{cases} \quad (7.7)$$

De forma análoga a la Ecuación (7.4), d_{min} , d_{max} y K_o definen la forma de la función y su influencia en función de la distancia. Note que es posible definir una distancia a los obstáculos d_{max} de tal manera, que anule $\kappa(d_o)$ cuando el robot se encuentre alejado de los obstáculos.

7.7. Evasión de colisiones del resto del robot

Buscamos ahora que el estudiante mueva el robot **en cada instante** del cálculo de la trayectoria con:

- Calcular la velocidad activa (movimiento diferencial) $\Delta q_a = J^\dagger \vec{v}_{ref}$, siendo \vec{v}_{ref} la velocidad activa del robot que permite seguir la trayectoria en el espacio de la tarea. Recordemos que esta trayectoria ya se calculaba para evitar algunos obstáculos como, por ejemplo, esferas.
- Calcular un movimiento en el espacio nulo: Δq_b que aumente la distancia del robot a los obstáculos.

Esto último es sencillo, pues, en cada instante de simulación, se puede calcular un pequeño movimiento Δq_b perteneciente al espacio nulo y decidir si es más conveniente:

- Hacer $q = q + \Delta q_b$

- o hacer $q = q - \Delta q_b$

Se tomará una decisión u otra dependiendo de si aumenta o disminuye la distancia a los obstáculos. Es decir, la librería mueve efectivamente el brazo en cada instante de simulación y lee la distancia a los obstáculos y tomando la decisión más conveniente. Podemos ver esto en la función `increase_distance_to_obstacles`. El ejercicio siguiente busca que, en cada instante, el robot calcule la cinemática inversa para seguir la trayectoria del extremo y, al mismo tiempo, realizará movimientos en el espacio nulo que lo alejen de los obstáculos.

Ejercicio 7.7.1: Maximice la distancia en cada instante de la trayectoria

El ejercicio persigue que el alumno compruebe el algoritmo propuesto, que en cada instante de la solución de la cinemática inversa, intenta alejar al robot de los obstáculos en base a un movimiento en el espacio nulo. Se variarán los parámetros para ver qué influencia tienen sobre el resultado del algoritmo. Instrucciones:

- Abra la escena `kuka_14_R820_2.ttt`. Abra el script `practice4.3_kuka.py`.
- Observe la función `def increase_distance_to_obstacles(robot, q)`. Esta función comprueba si el robot debe moverse con dirección Δq_b o $-\Delta q_b$ dependiendo de si aumenta o disminuye la distancia a los obstáculos. De ahí que, en la primera trayectoria del robot (aprendizaje) este realice una trayectoria muy oscilatoria (pues en todo momento debe valorar entre dos movimientos alternativos).
- **TAREA 1:** Ejecute el script `practice4.3_kuka.py` y varíe el número de iteraciones en la posición inicial del robot (cambie a 0, 1, 5, 100...).

```
for i in range(30):
    qd = increase_distance_to_obstacles(robot, q)
    q = q + 0.05*qd
```

¿Qué observa?

- **TAREA 2:** Ejecute el script `practice4.3_kuka.py` y varíe el módulo de la velocidad \dot{q}_b (cambie a 0.8, 0.5, 1.5...).

```
qdb = increase_distance_to_obstacles(robot, q)
qdb = 0.8*np.linalg.norm(qda)*qdb
qd = qda + qdb
```

Observe las soluciones que se producen cada caso. Debe considerarse importante la suavidad de la trayectoria que seguirá el robot, por ejemplo.

7.8. Resumen

Hasta ahora, las actividades realizadas, han implicado:

- **EJERCICIO 4.1:** Planificar rectas con interpolación entre posiciones cartesianas y orientaciones.
 - Escena: `kuka_14_R820_2.ttt`.

- Script: `practice4.1_kuka.py`.
- **EJERCICIO 4.2:** Programación de una función de potencial para el extremo del robot.
 - Escena: `kuka_14_R820_2.ttt`.
 - Script: `practice4.1_kuka.py`.
- **EJERCICIO 4.3:** Alejando los puntos de la recta de acuerdo con la función de potencial. Planificación sencilla en el espacio de la tarea para evitar obstáculos.
 - Escena: `kuka_14_R820_2.ttt`.
 - Script: `practice4.1_kuka.py`.
- **EJERCICIO 4.4:** Objetos más cercanos. Plantea mover el robot y observar la distancia a los objetos más cercanos.
 - Escena: `kuka_14_R820_2.ttt`.
 - Script: `kuka_move_robot.py`.
- **EJERCICIO 4.5:** Distancia del robot a los objetos más cercanos. Se propone al estudiante que modifique el programa deLua para tener en cuenta otras distancias y otros objetos de colisión.
 - Escena: `kuka_14_R820_2.ttt`.
 - Script: `kuka_move_robot.py`.
- **EJERCICIO 4.6:** El ejercicio persigue mover al robot en el espacio nulo y hallar q de manera que el robot se aleje lo más posible de los obstáculos.
 - Escena: `kuka_14_R820_2.ttt`.
 - Script: `practice4.2_kuka.py`.
- **EJERCICIO 4.7:** El ejercicio persigue mover al robot en el espacio nulo en cada movimiento de la trayectoria. Se desea que el estudiante modifique los principales parámetros del algoritmo y observe los resultados.
 - Escena: `kuka_14_R820_2.ttt`.
 - Script: `practice4.3_kuka.py`.

Child script (LBR_iiwa_14_R820)

```

1 function sysCall_init()
2     -- Handles to the robot links
3     hrob_link4=sim.getObjectHandle('LBR_iiwa_14_R820_link4')
4     hrob_link5=sim.getObjectHandle('LBR_iiwa_14_R820_link5')
5     hrob_link6=sim.getObjectHandle('LBR_iiwa_14_R820_link6')
6     hrob_link7=sim.getObjectHandle('LBR_iiwa_14_R820_link7')
7
8     -- Objects that may produce collisions
9     h_sphere=sim.getObjectHandle('Sphere')
10    h_cuboid=sim.getObjectHandle('Cuboid')
11    h_floor=sim.getObjectHandle('Floor')
12    col_objects = {h_sphere, h_cuboid, h_floor}
13    n_col_objects = 3
14
15    graph=sim.getObjectHandle('Graph')
16    local color={0,0.5,1}
17    streamId=sim.addGraphStream(graph,'dummy_distance','meters',0,color)
18    distanceSegment=sim.addDrawingObject(sim.drawing_lines,2,0,-1,1,color)
19 end
20
21
22 function sysCall_actuation()
23
24 end
25
26 function sysCall_sensing()
27     -- use this to check the distance to all measurable objects in the scene
28     -- local res,dist=sim.checkDistance(h,sim.handle_all)
29     -- use this to check the distance to robot link 5
30
31     -- find min distance of robot link to all objects
32     dist4_m, dist4_var = find_min_distance_to_objects(hrob_link4)
33     dist5_m, dist5_var = find_min_distance_to_objects(hrob_link5)
34     dist6_m, dist6_var = find_min_distance_to_objects(hrob_link6)
35     dist7_m, dist7_var = find_min_distance_to_objects(hrob_link7)
36     n_robot_links = 4
37
38     local all_distances_m = {dist4_m, dist5_m, dist6_m, dist7_m}
39     local all_distances_var = {dist4_var, dist5_var, dist6_var, dist7_var}
40     index = index_of_min(all_distances_m, n_robot_links)
41     dist_var = all_distances_var[index]
42     if dist_var then
43         sim.setGraphStreamValue(graph,streamId,dist_var[7])
44         sim.addDrawingObjectItem(distanceSegment,nil)
45         sim.addDrawingObjectItem(distanceSegment,dist_var)
46         sim.setFloatSignal('min_distance_to_objects', dist_var[7])
47     end
48 end
49
50
51 function find_min_distance_to_objects(hrob_link)
52     -- check the distance to all collision objects from this robot link
53     local dist_variables = {}
54     local dist_meters = {}
55
56     for i=1,n_col_objects,1 do
57         -- checks minimum distance between handle1 and handle2
58         local res,dist=sim.checkDistance(hrob_link, col_objects[i])
59         dist_variables[i] = dist
60         dist_meters[i] = dist[7]
61     end
62     index = index_of_min(dist_meters, n_col_objects)
63     return dist_meters[index], dist_variables[index]
64 end
65
66 function index_of_min(arr, n)
67     -- return the index that corresponds to
68     min_dist = arr[1]
69     min_index = 1

```

Figura 7.8: Código del “child script” de Lua asociado al robot.

Capítulo 8

Instalación de Coppelia Sim

8.1. Introducción

Coppelia y pyARTE están instalados en los equipos de prácticas. Si se desea instalar el simulador y la librería en un PC de uso personal, por favor, realice los siguientes pasos:

- a) Instalación de un entorno virtual de python.
- b) Instalación de Coppelia Sim.
- c) Instalación de pyARTE y de las librerías de interfaz con Coppelia.
- c) Configuración de pycharm.

En lo que sigue, se asume que se cuenta con un sistema operativo Ubuntu 20.04 funcionando correctamente.

8.2. Plataformas soportadas

Coppelia Sim cuenta con versiones instalables para Windows, Mac y Linux. Se indican aquí las plataformas sobre las que se ha probado Coppelia Sim y pyARTE y se puede garantizar que funciona correctamente:

- Ubuntu 20.04 + Coppelia Sim 20.04 + python 3.8.
- Ubuntu 22.04 + Coppelia Sim 22.04 + python 3.10.

8.3. Instalación de un entorno virtual de python

Para simplificar el proceso, nos disponemos a crear un entorno virtual de python. Un entorno virtual de python es, en esencia, una estructura de directorios donde tendremos:

- El intérprete de python (en nuestro caso python3.8) en `venv/bin/python`.
- El gestor de paquetes en `venv/bin/pip`.

- Un conjunto de librerías instaladas.

En Ubuntu, para crear entornos virtuales, necesitaremos instalar el paquete del sistema virtualenv:

```
$ sudo apt install virtualenv
```

A continuación creamos el entorno virtual de python, en este caso, los vamos a crear en el escritorio del usuario actual:

```
$ cd /home/usuario/Escritorio
$ sudo virtualenv venv
```

Modifique, `usuario` por el nombre de su usuario. El comando anterior debe responder con algo parecido a:

```
created virtual environment CPython3.8.10.final.0-64 in 98ms
creator CPython3Posix(dest=/home/arvc/venv, clear=False, global=False)
seeder FromAppData(download=False, pip=latest, setuptools=latest,
wheel=latest, pkg_resources=latest, via=copy, app_data_dir=
/home/arvc/.local/share/virtualenv/seed-app-data/v1.0.1.debian.1)
activators BashActivator,CShellActivator,FishActivator,PowerShellActivator,
PythonActivator,XonshActivator
```

Llegados a este punto, tenemos un entorno virtual en nuestro escritorio del sistema. Podemos ejecutar una instancia del intérprete de python que hemos creado si hacemos:

```
$ cd /home/usuario/Escritorio/venv/bin
$ ./python
Python 3.8.10 (default, Jun 22 2022, 20:18:18)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 4+5
9
>>>
```

Use CTRL+d o `exit()` para salir del intérprete. Si usamos el gestor de paquetes `venv/bin/pip` para instalar paquetes, las librerías se instalarán en el entorno virtual en `venv/lib/python3.8/site-packages`. De esta manera, podemos mantener en la misma computadora diferentes intérpretes de python (p.e. python 2.7, python 3.8 y python 3.9, con diferentes librerías instaladas).

Instalamos, a continuación, las librerías necesarias para la librería pyARTE:

```
$ sudo apt install python3-dev
$ cd /home/usuario/Escritorio/venv/bin
$ sudo ./pip install numpy matplotlib pyinputplus
```

8.4. Instalación de Coppelia

Coppelia permite descargar distribuciones del simulador desde:

www.coppeliarobotics.com/downloads

En nuestro caso, usaremos la versión EDU del simulador. Descargue y descomprima el simulador. Nota: en los equipos de prácticas ya se ha descargado el simulador.

8.5. Instalación de un entorno virtual de Python

Si estás utilizando un sistema Linux, probablemente el sistema operativo venga con una instalación básica del intérprete de python. Durante las prácticas se utiliza un método ligeramente diferente, que consiste en crear un entorno virtual de Python. Un entorno virtual de Python es, simplemente, un directorio donde se instalará el programa ejecutable que interpreta el lenguaje y, además, también se instalarán las librerías con las que trabaja python. De esta manera, es posible mantener diferentes versiones de python con diferentes librerías en un mismo sistema operativo. La ventaja principal de esto es asegurar el buen funcionamiento del sistema aún cuando haya incompatibilidades entre diferentes librerías.

Creamos, a continuación, un entorno virtual de python. Comenzamos instalando el paquete `virtualenv`:

```
$ sudo apt install virtualenv
```

Ejecutamos el comando `virtualenv` y creamos un directorio en:

```
home  
usuario  
Applications:
```

```
$ cd  
$ mkdir Applications  
$ cd Applications  
$ virtualenv venv
```

Para comprobar si todo ha ido bien, haremos:

```
$ cd  
$ Applications/venv/bin/python  
Python 3.8.10 (default, May 26 2023, 14:05:08)  
[GCC 9.4.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> 5+5  
10
```

El código anterior lanza el intérprete de python en modo “línea de comandos” y permite realizar pequeños scripts y operaciones.

8.6. Instalación de pyARTE y de las librerías de interfaz

Clone la librería pyARTE. Con el comando siguiente se utiliza la aplicación git para descargar un repositorio público alojado en www.github.com

```
$ cd /home/usuario/Escritorio  
$ git clone https://github.com/4rtur1t0/pyARTE.git
```

Instale, a continuación, la librería de comunicaciones entre procesos Zmq:

```
$ cd  
$ cd Applications/venv/bin  
$ ./pip3 install coppeliasim-zmqremoteapi-client
```

8.7. Configuración de pycharm

En este apartado se indica cómo configurar el editor de pycharm para ejecutar pyARTE y sus scripts sin problemas. Estos pasos ya se han realizado en los equipos de prácticas. En un PC de uso personal, realice los siguientes pasos:

- Abra pycharm.
- Abra el proyecto pyARTE.
- Configure el intérprete de python:
File - Settings - Project - Python interpreter- “engranaje” - add - existing environment - seleccione
`/home/usuario/Applications/venv/bin/python`. Esto le dice a pycharm dónde se encuentra el intérprete de python.

8.8. Pruebas

En este momento deberías tener todo preparado para poder realizar simulaciones en Coppelia manejadas desde python con la ayuda de la librería pyARTE. Para probar si todo se encuentra instalado correctamente, simplemente:

- Inicie Coppelia.
- Abra la escena de Coppelia `scenes/ur5.ttt`.
- Abra uno de los scripts que maneja esta escena. Por ejemplo, abra:
`practicals/ur5_move_robot.py`.
- Ejecute el script (botón *play* verde sobre el script).
- Pinche con el ratón sobre el terminal de ejecución que aparece abajo.
- El programa permite mover las articulaciones con las teclas 1, 2, 3, 4... del teclado. Utilice 'o' y 'c' para abrir/cerrar la pinza del robot.

Parte II

Proyecto transversal

Bibliografía

- [1] Luigi Villani Bruno Siciliano Lorenzo Sciavicco y Giuseppe Oriolo. *Robotics: Modelling, planning and control*. London: Springer, 2009.
- [2] Peter Corke. *Robotics, Vision and Control - Fundamental Algorithms in MATLAB®*. Springer Tracts in Advanced Robotics. Springer, 2011. ISBN: 978-3-642-20143-1.
- [3] Bruno Siciliano y J.-J.E. Slotine. “A general framework for managing multiple tasks in highly redundant robotic systems. In Fifth international conference on advanced robotics (Vol. 2”. En: jul. de 1991, 1211-1216 vol.2. ISBN: 0-7803-0078-5. DOI: [10.1109/ICAR.1991.240390](https://doi.org/10.1109/ICAR.1991.240390).