



Prácticas de 1770 Robótica

Arturo Gil Aparicio



# Índice

<b>I Prácticas</b>	<b>15</b>
<b>1. Iniciación a Coppelia Sim</b>	<b>17</b>
1.1. Objetivos . . . . .	17
1.2. Coppelia Sim . . . . .	17
1.3. Código proporcionado . . . . .	18
1.4. Las librerías de interfaz . . . . .	19
1.5. La librería de prácticas . . . . .	19
1.6. Paquetes de python . . . . .	21
1.7. Editores para python . . . . .	21
1.8. Primeros pasos con Coppelia . . . . .	22
1.8.1. Cambios en el punto de vista de la escena . . . . .	22
1.8.2. Mover objetos . . . . .	23
1.9. Un primer script de python . . . . .	25
1.10. Mueva el robot . . . . .	26
1.10.1. Añadir objetos al entorno . . . . .	29
1.10.2. Child scripts . . . . .	30
1.11. Manejadores y descripción de pyARTE . . . . .	33
1.12. Comentarios finales . . . . .	36
<b>2. Representación de transformaciones en Python</b>	<b>37</b>
2.1. Introducción . . . . .	37
2.2. Objetivos . . . . .	37
2.3. ¿Para qué? . . . . .	38
2.4. Matrices de rotación . . . . .	39
2.5. Propiedades de una matriz de rotación . . . . .	39
2.6. Matrices de rotación elementales . . . . .	41
2.7. Matrices de transformación homogénea . . . . .	41
2.8. Rotación de un vector . . . . .	43
2.9. Transformación de las coordenadas de un punto . . . . .	44
2.10. Ángulos de Euler . . . . .	44
2.11. Por qué utilizar ángulos de Euler . . . . .	45
2.12. Conversión de ángulos de Euler a matriz de rotación . . . . .	45
2.13. Conversión de matriz de rotación a ángulos de Euler . . . . .	46
2.13.1. Caso normal . . . . .	47
2.13.2. Caso degenerado . . . . .	47
2.14. Transformaciones en pyARTE . . . . .	49
2.14.1. Vectores en pyARTE . . . . .	49
2.14.2. Matrices de rotación en pyARTE . . . . .	50

2.14.3. Matrices de transformación homogénea en pyARTE . . . . .	51
2.15. La clase <b>Euler</b> . . . . .	52
2.15.1. Visualizar transformaciones en Coppelia Sim . . . . .	54
2.16. Ejercicios avanzados . . . . .	58
2.17. Introducción a <code>numpy</code> . . . . .	61
2.18. Matrices en <code>numpy</code> . . . . .	61
<b>3. Cinemática directa</b>	<b>63</b>
3.1. Introducción . . . . .	63
3.2. Objetivos . . . . .	63
3.3. Instalación de la librería . . . . .	64
3.4. Carga de robots en ARTE . . . . .	65
3.5. Parámetros de DH de un robot . . . . .	67
3.6. Cinemática directa del robot IRB140 . . . . .	67
3.7. Transformaciones intermedias . . . . .	70
3.8. Uso de la aplicación <code>teach</code> . . . . .	71
3.9. Ejercicios finales . . . . .	74
<b>4. Cinemática inversa</b>	<b>79</b>
4.1. Objetivos . . . . .	79
4.2. El problema cinemático inverso . . . . .	79
4.3. Soluciones cerradas de la cinemática inversa . . . . .	80
4.4. Cinemática inversa de un robot plano de 3 GDL . . . . .	81
4.5. Cinemática inversa de un robot plano de 3 GDL en ARTE . . . . .	83
4.6. Solución de la cinemática inversa para un robot de 6 GDL . . . . .	84
4.7. Uso de la aplicación <code>teach</code> . . . . .	87
<b>5. Cinemática inversa y planificación de trayectorias en Coppelia</b>	<b>89</b>
5.1. Introducción . . . . .	89
5.2. Cinemática directa del ABB IRB140 . . . . .	89
5.2.1. Manejo básico del robot IRB140 . . . . .	89
5.2.2. El TCP de la herramienta . . . . .	90
5.2.3. Definición del TCP . . . . .	90
5.3. Cinemática inversa del robot IRB140 . . . . .	91
5.4. Generación de trayectorias en el espacio de trabajo del robot . . . . .	95
5.4.1. Interpolación de posiciones y orientaciones en el espacio de la tarea . . . . .	96
5.4.2. Generación de trayectorias rectas a orientación constante . . . . .	97
5.4.3. Seguimiento de trayectorias por el robot . . . . .	98
5.4.4. Instrucciones de movimiento en un robot . . . . .	98
5.4.5. Programas de ejemplo . . . . .	100
<b>6. Una aplicación de paletizado con Coppelia</b>	<b>105</b>
6.1. Introducción . . . . .	105
6.2. Objetivos . . . . .	105
6.3. Material proporcionado . . . . .	105
6.4. Descripción de la escena . . . . .	106
6.5. Transformaciones . . . . .	107
6.6. Cálculo de las posiciones de paletizado . . . . .	108
6.7. Descripción del código . . . . .	109

6.7.1. función <code>pick_and_place</code> . . . . .	109
6.7.2. función <code>pick</code> . . . . .	109
6.7.3. función <code>place</code> . . . . .	110
6.7.4. Definiendo los <i>target points</i> . . . . .	111
6.8. Selección de la herramienta . . . . .	112
6.9. Ventosas de vacío . . . . .	112
6.10. Tiempo real . . . . .	113
<b>II Prácticas avanzadas</b>	<b>117</b>
<b>7. Cinemática inversa y la Jacobiana del manipulador</b>	<b>119</b>
7.1. Objetivos . . . . .	119
7.2. Primeros pasos . . . . .	119
7.3. Cinemática inversa . . . . .	120
7.4. Experimentando con el algoritmo . . . . .	122
<b>8. Aplicaciones industriales</b>	<b>125</b>
8.1. Introducción . . . . .	125
8.2. Clasificación de objetos en base a su color . . . . .	125
8.3. Pintura . . . . .	126
8.4. Soldadura . . . . .	127
<b>III Proyecto transversal</b>	<b>129</b>
<b>9. Proyecto evaluable</b>	<b>131</b>
9.1. Introducción . . . . .	131
9.2. Entrega del proyecto y condiciones . . . . .	131
9.3. Descripción del proyecto 1 . . . . .	132
9.4. Código . . . . .	134
9.5. Descripción del proyecto 2 . . . . .	135
9.6. Descripción del proyecto 3 . . . . .	135
9.7. Ayuda . . . . .	137
9.8. Código . . . . .	137
<b>IV Anexos</b>	<b>141</b>
<b>10. Instalación de Coppelia Sim</b>	<b>143</b>
10.1. Introducción . . . . .	143
10.2. Plataformas soportadas . . . . .	143
10.3. Instalación de un entorno virtual de python . . . . .	143
10.4. Instalación de Coppelia . . . . .	144
10.5. Instalación de pyARTE y de las librerías de interfaz . . . . .	145
10.6. Configuración de pycharm . . . . .	145
10.7. Pruebas . . . . .	145



# Lista de Figuras

1.1.	Esquema de las comunicaciones entre pyARTE y Coppelia. Ejemplos de funciones. . . . .	19
1.2.	Menú git integrado en Pycharm . . . . .	21
1.3.	Logo de Pycharm. . . . .	22
1.4.	Menú git de Pycharm. . . . .	22
1.5.	Una vista de la escena <code>scenes/irb140.ttt</code> . . . . .	23
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. . . . .	23
1.7.	Cambiar la posición/orientación de los objetos en la escena. Fuente: Coppelia Sim. . . . .	24
1.8.	a) Menú para el cambio de posición. b) Menú para el cambio de orientación. Fuente: Coppelia Sim. . . . .	24
1.9.	Detalle del árbol de jerarquía de la escena <code>irb140.ttt</code> . Fuente: Coppelia Sim. . . . .	26
1.10.	Menú integrado en Pycharm para ejecutar scripts de python y depurar. . . . .	28
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> . . . . .	29
1.12.	Dos robots añadidos al entorno de Coppelia. . . . .	30
1.13.	El <i>child script</i> del robot UR10. . . . .	31
1.14.	El <i>child script</i> del robot UR10. . . . .	32
1.15.	Una aplicación de pintura. . . . .	33
1.16.	a) Propiedades de un objeto de tipo <i>joint</i> en Coppelia. b) Propiedades dinámicas de una articulación ( <i>joint</i> ). . . . .	35
2.1.	Dos sistemas de referencia con diferente orientación relativa. . . . .	39
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> . . . . .	43
2.3.	Transformación en las coordenadas de un mismo punto en dos sistemas de referencia <i>A</i> y <i>B</i> . . . . .	45
2.4.	Sistemas de referencia en una escena de Coppelia. Fuente: Captura de pantalla sobre Coppelia Sim. . . . .	56
2.5.	Sistemas de referencia en una escena de Coppelia. Fuente: Captura de pantalla de Coppelia Sim. . . . .	58
3.1.	Sistemas de DH sobre el robot IRB 140 de ABB <sup>©</sup> . Fuente: modificado desde el manual oficial del robot publicado en <a href="http://www.abb.com">www.abb.com</a> . . . . .	68
3.2.	Cambio en el punto de vista 3D de una figura de Matlab. . . . .	69

3.3.	El robot IRB140 de ABB <sup>©</sup> en la librería Matlab <sup>©</sup>	69
3.4.	La aplicación GUI <code>teach</code> de la librería ARTE.	72
3.5.	Entorno robótico industrial configurado con <code>teach</code> .	74
3.6.	Cotas en mm para el robot.	75
3.7.	El robot del ejercicio en ARTE.	76
3.8.	El robot <code>serial2</code> con parámetros DH incorrectos.	77
3.9.	El robot <code>serial2</code> con sus cotas principales en mm.	77
4.1.	Un robot plano de 3 GDL.	81
4.2.	El robot IRB140 de ABB <sup>©</sup>	85
4.3.	La aplicación GUI <code>teach</code> de la librería ARTE.	87
5.1.	Dos vistas del extremo del robot donde se aprecian los TCP de la pinza y de la ventosa.	91
5.2.	Una trayectoria recta con posiciones y orientaciones sobre ella.	99
5.3.	Una trayectoria recta con posiciones y orientaciones sobre ella. Se muestran, además, las soluciones de la cinemática inversa para cada posición y orientación $T_i$ .	99
5.4.	La trayectoria 1 del ejercicio. Varias trayectorias rectas con indicación de los puntos de interés sobre ellas.	102
5.5.	La trayectoria 1 del ejercicio. Varias trayectorias rectas con indicación de los puntos de interés sobre ellas.	103
6.1.	Una aplicación de paletizado con Coppelia.	106
6.2.	Dos vistas de las transformaciones de interés en la aplicación de paletizado.	114
6.3.	Posiciones $(x, y, z)$ para el paletizado de 27 piezas en un arreglo de 3x3.	115
6.4.	Pinza del robot colisionando con piezas.	115
6.5.	Aumenta o decelara la velocidad de simulación de Coppelia.	115
7.1.	El robot UR10.	120
8.1.	Una aplicación de clasificación en base al color.	126
8.2.	Un IRB140 en una aplicación de pintura.	127
8.3.	Un IRB140 en una aplicación de soldadura TIG/MIG.	128
9.1.	Una captura de la escena del Proyecto 1.	133
9.2.	Una captura del proyecto 3 en Coppelia.	136

# Lista de Tablas

3.1.	Parámetros DH del robot ABB IRB140.	67
4.1.	Parámetros de DH del robot plano de 3GDL.	83
4.2.	Parámetros DH del robot ABB IRB140.	85
5.1.	Límites articulares del robot IRB140.	94
6.1.	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 $Y$ .	109



# Preámbulo

Este documento de prácticas se divide en dos partes fundamentales:

- Parte I: En esta parte se recogen una serie de prácticas que tratan sobre la simulación de robots manipuladores.
- Parte II: En esta parte se recogen ideas para el proyecto transversal que el estudiante puede utilizar libremente.

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

- Matlab<sup>1</sup>.
- Python<sup>2</sup>.
- La librería ARTE<sup>3</sup> (A Robotics Toolbox for Education). Esta librería funciona bajo el entorno Matlab.
- El simulador robótico Coppelia Sim <sup>4</sup>.
- La librería PyARTE<sup>5</sup>, que está basada en python y utiliza el simulador Coppelia Sim.

Se anota, a continuación, un índice del presente documento:

- **Parte I: Prácticas.**
  - **Práctica 1: Iniciación a Coppelia Sim.** Esta práctica introduce al alumno en el manejo de la simulación en Coppelia Sim. Incluye la simulación de un robot UR5.
    - **Objetivos de aprendizaje:**
      - ◊ Iniciar al estudiante en el uso del simulador Coppelia Sim<sup>6</sup>.
      - ◊ Conocer las principales características de un simulador de robots.

---

<sup>1</sup>[www.mathworks.com](http://www.mathworks.com)

<sup>2</sup>[www.python.org](http://www.python.org)

<sup>3</sup><https://arvc.umh.es/arte>

<sup>4</sup>[www.coppeliasim.com](http://www.coppeliasim.com)

<sup>5</sup><https://github.com/4rtur1t0/pyARTE>

<sup>6</sup>[www.coppeliarobotics.com](http://www.coppeliarobotics.com)

- ◊ Conocer la interfaz de programación en python del simulador e introducir la librería pyARTE y su capacidad para manejar el simulador Coppelia Sim.

- **Actividades:**

- ◊ 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.

**Herramientas: Coppelia Sim y pyARTE.**

• **Práctica 2: Representación de transformaciones en Python.**

Esta práctica persigue que el estudiante se familiarice con las diferentes herramientas para la representación de la posición y orientación, así como de las transformaciones entre diferentes sistemas de referencia. Esta práctica explora el uso de matrices homogéneas, matrices de rotación y ángulos de Euler, así como la conversión entre estas representaciones.

- **Objetivos de aprendizaje:**

- ◊ Comprender la necesidad de utilizar sistemas de referencia para comandar la posición y orientación del extremo del robot.
- ◊ Aplicar los conocimientos sobre matrices de transformación homogénea y matrices de rotación en un entorno de simulación.
- ◊ Utilizar sistemas de referencia locales para la realización de aplicaciones de paletizado y envasado.

- **Actividades:**

- ◊ Definir en python matrices de transformación homogénea y matrices de rotación.
- ◊ Realizar conversiones entre matrices de rotación y ángulos de Euler.
- ◊ 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.

### **Herramientas: Python y PyARTE.**

- **Práctica 3: Cinemática directa con ARTE.** La práctica tiene como objetivo que el estudiante se familiarice con el entorno de Matlab y ARTE. El estudiante deberá entender el modelado cinemático de los robots de tipo serie usando los parámetros de DH (Denavit-Hartenberg). Para ello, se muestran diferentes ejemplos de robots y se ayuda al estudiante con la representación gráfica de estos sistemas de referencia de DH.

- **Objetivos de aprendizaje:**

- ◊ El estudiante deberá ser capaz de definir la cinemática de un robot mediante los parámetros Denavit-Hartenberg de un manipulador robótico.
    - ◊ Comprender la solución de la cinemática directa de un robot manipulador (en posición y orientación) usando parámetros de DH.
  - **Actividades:** Durante la práctica, se realizarán, entre otras, las siguientes actividades:
    - ◊ Visualizar los sistemas de referencia de DH sobre un robot.
    - ◊ Calcular transformaciones sucesivas basadas en parámetros de DH.
    - ◊ Configurar los parámetros DH de un robot de la librería ARTE.

### **Herramientas: Matlab y ARTE.**

- **Práctica 4: Cinemática inversa con ARTE.** Con esta práctica se busca que el estudiante sea capaz de entender correctamente las soluciones de la cinemática inversa en posición/orientación en robots de tipo serie. Para ello, se muestran diferentes ejemplos de robots y se proponen diferentes actividades. En esta práctica se persiguen los siguientes **objetivos de aprendizaje**:

- Resolver las ecuaciones de la cinemática inversa de cualquier manipulador de tipo serie hasta 6 GDL y con muñeca esférica.
  - Ser capaz de escribir estas ecuaciones en el entorno Matlab para un robot de su elección.
  - Poder generar trayectorias articulares que hagan que el extremo siga un movimiento rectilíneo en el espacio cartesiano.

Para alcanzar estos objetivos, se plantean las siguientes **actividades**:

- Calcular con la librería soluciones particulares a la cinemática inversa de un robot plano de 3GDL y un robot industrial de 6GDL.
  - Programar scripts que permitan comprender las relaciones entre la cinemática directa en inversa en robots de tipo serie.
  - Programar scripts para el seguimiento de trayectorias rectas y circulares en el espacio.

### **Herramientas: Matlab y ARTE.**

- **Práctica 5: Cinemática inversa y planificación de trayectorias en Coppelia Sim.** La práctica plantea los siguientes **objetivos de aprendizaje**:

- Soluciones de la cinemática inversa en un simulador robótico completo: colisiones y autocolisiones.
- Cálculo de trayectorias en el espacio de trabajo del robot.
- Instrucciones de movimiento en robots industriales: `moveAbsJ`, `moveJ`, `moveL`.

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.
- Programación de trayectorias usando instrucciones de movimiento propias de lenguajes de programación de robots industriales.

- **Práctica 6: Una aplicación de paletizado con Coppelia Sim.** La práctica desarrolla una aplicación de paletizado en Coppelia Sim usando un robot IRB140. 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.
- Explicar al estudiante las formas posibles de especificar posiciones y orientaciones relativas a distintos sistemas de referencia.

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

- Permitir el desarrollo de una aplicación de paletizado en un entorno de simulación.
- Interacción del robot con objetos del entorno: p. e. asir piezas con una pinza o bien utilizando una ventosa de vacío.

### **Herramientas: Coppelia Sim y pyARTE.**

- **La Jacobiana del manipulador.** Plantea el uso de la Jacobiana del manipulador para resolver la cinemática inversa de manipuladores sin solución cerrada. **Herramientas: Matlab y ARTE.**

- **Parte II:** Propuestas de proyectos. En esta segunda parte se indican algunas ideas sobre proyectos que resultan realizables con las técnicas de simulación presentadas.

- **Anexos:**

- **Instalación de Coppelia.** Se resume en este capítulo los pasos necesarios para la instalación de pyARTE, Coppelia Sim y los paquetes necesarios de Python.

En relación con cada práctica, se anotan, seguidamente, los objetivos de aprendizaje y las actividades principales a realizar:

■ **Parte I: Prácticas.**

- **Representación de transformaciones.** Objetivos:

- Ser capaz de representar la traslación y la rotación de sistemas de referencia mediante matrices de transformación homogénea, coordenadas cartesianas y ángulos de Euler.

**Herramientas: Python y PyARTE.**

- **Cinemática directa** con ARTE

**Herramientas: Matlab y ARTE.**

- **Cinemática inversa** con ARTE.

**Herramientas: Matlab y ARTE.**

- **Iniciación a Coppelia Sim.**

**Herramientas: Coppelia Sim y pyARTE.**

- Una aplicación de paletizado con Coppelia Sim.

**Herramientas: Coppelia Sim y pyARTE.**

- La Jacobiana del manipulador.

**Herramientas: Matlab y ARTE.**

- Instalación de Coppelia.

■ **Parte II:** Propuestas de proyectos.



# **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 Sim<sup>1</sup>.
- 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)<sup>2</sup>.

---

<sup>1</sup>[www.coppeliarobotics.com](http://www.coppeliarobotics.com)

<sup>2</sup>[www.ros.org](http://www.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 pyARTE<sup>3</sup>. 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 ARTE<sup>4</sup> 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).

---

<sup>3</sup><https://github.com/4rtur1t0/pyARTE>

<sup>4</sup>[www.arvc.umh.es/arte](http://www.arvc.umh.es/arte)

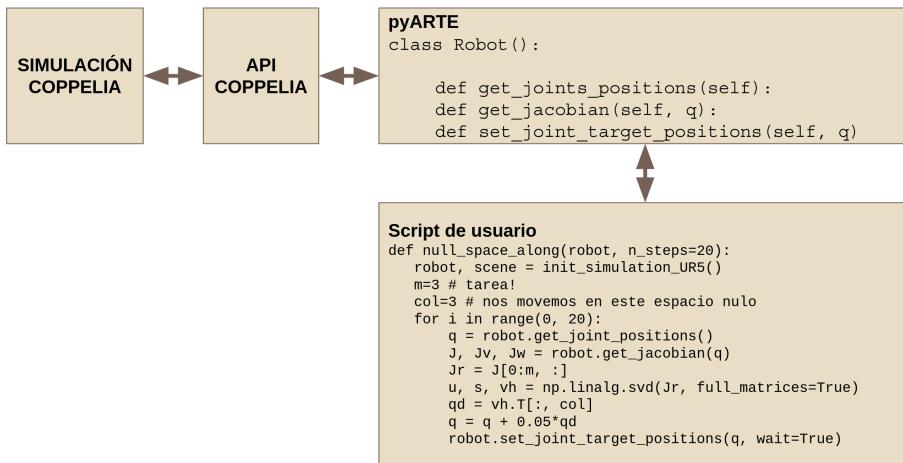


Figura 1.1: Esquema de las comunicaciones entre pyARTE y Coppelia. Ejemplos de funciones.

## 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. Estas librerías (también denominadas APIs) se encuentran en el directorio de instalación de Coppelia. En concreto, la API para python se puede encontrar en:

`CoppeliaSim/programming/remoteApiBindings/python/python`

Además, debe incluirse la librería correspondiente para cada sistema operativo: Linux, Mac o Windows. En el caso de utilizar Ubuntu, la librería del sistema se encuentra en:

`CoppeliaSim/programming/remoteApiBindings/lib/Ubuntu20_04/remoteApi.so`  
En el Capítulo 10 se resumen todos los pasos necesario para la instalación del simulador y la librería pyARTE en un sistema operativo Ubuntu.

## 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 10. 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.

Todos los anteriores son nombres comerciales de ABB, Universal Robots y Kuka. Además, la librería pyARTE maneja los modelos proporcionados junto con el simulador Coppelia Sim a los que se les ha realizado ligeras modificaciones.

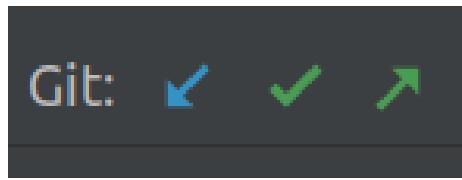


Figura 1.2: Menú git integrado en Pycharm

## 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 10. 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.

## 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 icono azul de la Figura 1.4 y seleccione la opción “merge”.



Figura 1.3: Logo de Pycharm.

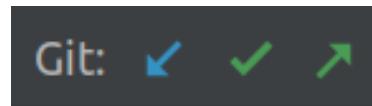


Figura 1.4: Menú git de Pycharm.

## 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:

- 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:

- *Camera pan*: translada la escena.

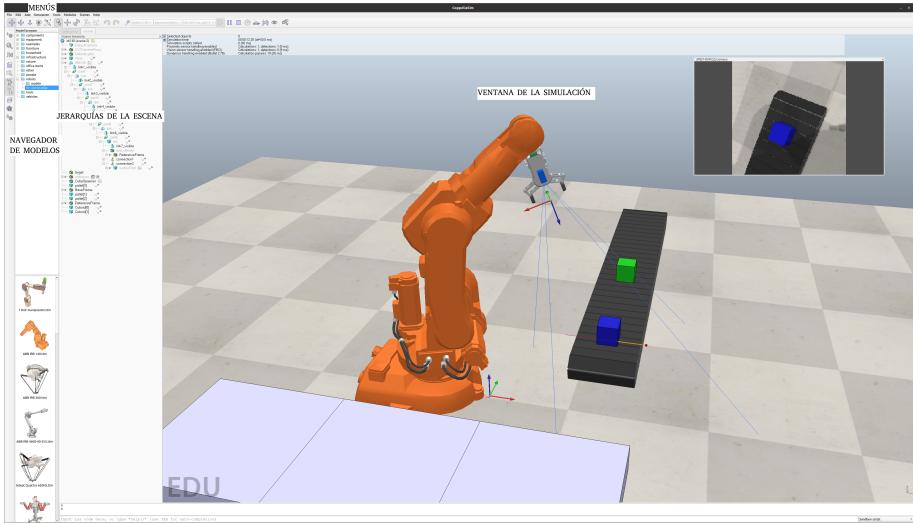


Figura 1.5: Una vista de la escena `scenes/irb140.ttt`.



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 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:

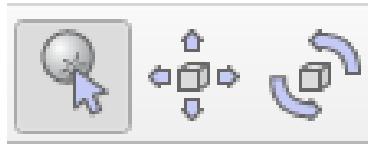


Figura 1.7: Cambiar la posición/orientación de los objetos en la escena. Fuente: Coppelia Sim.

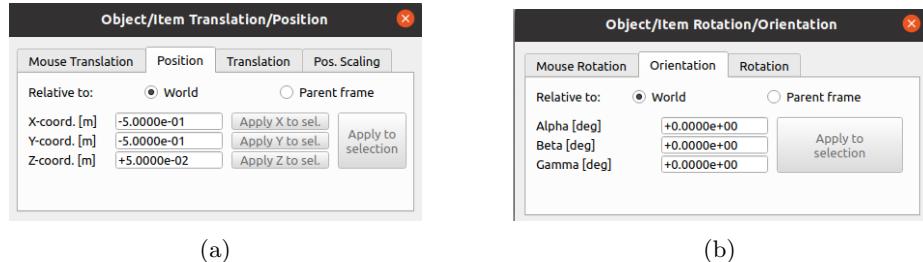


Figura 1.8: a) Menú para el cambio de posición. b) Menú para el cambio de orientación. Fuente: Coppelia Sim.

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

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

#### 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. 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.simulation import Simulation

if __name__ == "__main__":
    simulation = Simulation()
    clientID = simulation.start()
    robot = RobotABBIRB140(clientID=clientID)
    robot.start()

    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])
    q3 = np.array([np.pi/8, 0, -np.pi/4, 0, 3*np.pi/2, 0])

    robot.moveAbsJ(q1, precision=True)
    simulation.wait(100)
    robot.moveAbsJ(q2, precision=True)
    simulation.wait(100)
    robot.moveAbsJ(q3, precision=True)
    simulation.wait(100)

simulation.stop()
```

En concreto, las dos líneas siguientes crean un objeto de la clase `Simulation` y realizan la conexión con Coppelia. El método `start` realiza la conexión entre el script de python y el programa Coppelia. La variable `clientID` es un entero que identifica la conexión dentro del script. Coppelia permite una única conexión de forma simultánea.

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

Seguidamente, las dos líneas siguientes crean un objeto de la clase `RobotABBIRB140` y lo inicializan.

```
robot = RobotABBIRB140(clientID=clientID)
robot.start()
```

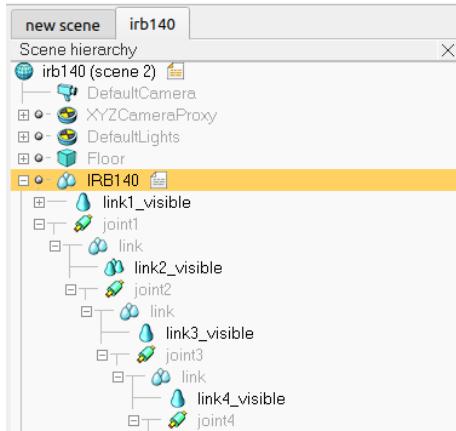


Figura 1.9: Detalle del árbol de jerarquía de la escena `irb140.ttt`. Fuente: Coppelia Sim.

**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(clientID=clientID)
robot2.start(base_name='/IRB140_2')
```

Esto permite tener varios robots conectados en una misma simulación de Coppelia. Seguidamente, las líneas siguientes definen tres 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])
q3 = np.array([np.pi/8, 0, -np.pi/4, 0, 3*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` y `q3`). 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 instante 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.

## 1.10. Mueva el robot

Nos disponemos, a continuación a mover el robot 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, 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 1.10.1: 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).
- Un cuaternion  $Q$ .

Considere utilizar esta información durante los siguientes apartados.

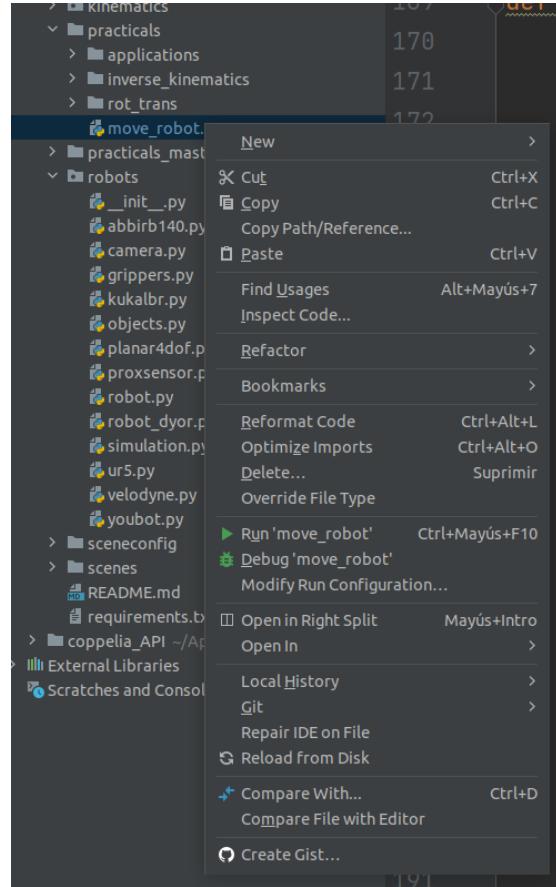


Figura 1.10: Menú integrado en Pycharm para ejecutar scripts de python y depurar.

#### Ejercicio 1.10.2: 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 a la izquierda del robot.

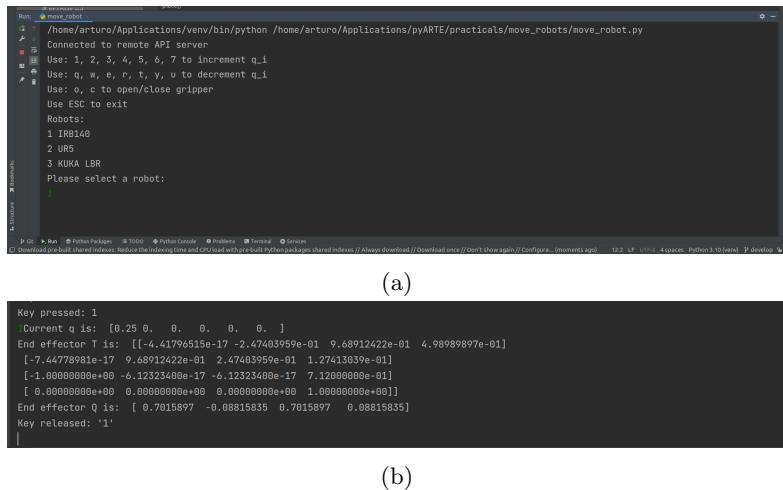


Figura 1.11: a) Seleccione alguno de los robots presentes en la escena de Coppelia. b) Salida por pantalla de la aplicación `move_robot.py`.

### Ejercicio 1.10.3: 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).

### 1.10.1. Añadir 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 hexápodo móvil, por ejemplo (Figura 1.12). Si inicia la simulación, observará que los robots realizan por sí mismos una serie movimientos.

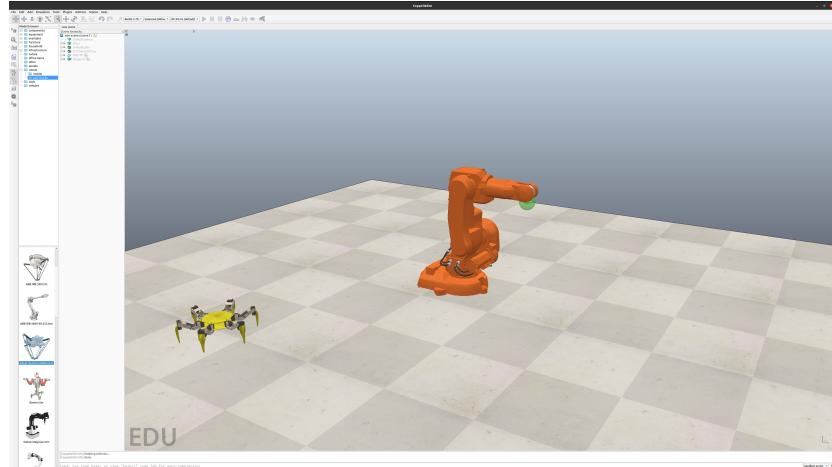


Figura 1.12: Dos robots añadidos al entorno de Coppelia.

#### Ejercicio 1.10.4: 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.

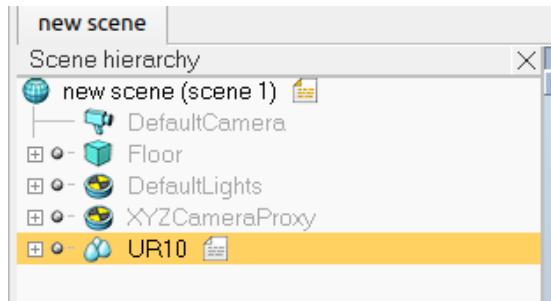


Figura 1.13: El *child script* del robot UR10.

#### Ejercicio 1.10.5: Child scripts

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

- Abra la escena `scenes/ur10.ttt`.
- Inicie la simulación. Observará que el robot UR10 se mueve.
- Edite el *child script* asociado al robot UR10. 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 los target points especificados en el script (`targetPos1`, `targetPos2` y `targetPos3`) que se encuentran en la función `coroutineMain()`. Observe el resultado.
- Elimine el *child script*: botón derecho, Edit-Remove-Associated Child Script.
- Vuelva a ejecutar la simulación. Deberá observar que, ahora, en la simulación el robot no realiza ningún movimiento.

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 function sysCall_init()
2     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 -- This is a threaded script, and is just an example!
15
16 function movCallback(config,vel,accel,handles)
17     for i=1,#handles,1 do
18         if sim.getJointMode(handles[i])~=sim.jointmode_force and sim.isDynamicallyEnabled(handles[i]) then
19             sim.setJointTargetPosition(handles[i],config[i])
20         else
21             sim.setJointPosition(handles[i],config[i])
22         end
23     end
24 end
25
26 function moveToConfig(handles,maxVel,maxAccel,maxJerk,targetConf)
27     local currentConf={}
28     for i=1,#handles,1 do
29         currentConf[i]=sim.getJointPosition(handles[i])
30     end
31     sim.moveToConfig(-1,currentConf,nil,nil,maxVel,maxAccel,maxJerk,targetConf,nil,movCallback,handles)
32 end
33
34 function coroutineMain()
35     local jointHandles={-1,-1,-1,-1,-1,-1}
36     for i=1,6 do
37         jointHandles[i]=sim.getObjectHandle("IRB140_joint"..i)
38     end
39
40     local vel=120
41     local accel=40
42     local jerk=80
43     local maxVel=(vel*math.pi/180,vel*math.pi/180,vel*math.pi/180,vel*math.pi/180,vel*math.pi/180,vel*math.pi/180)
44     local maxAccel=(accel*math.pi/180,accel*math.pi/180,accel*math.pi/180,accel*math.pi/180,accel*math.pi/180,accel*math.pi/180)
45     local maxJerk=(jerk*math.pi/180,jerk*math.pi/180,jerk*math.pi/180,jerk*math.pi/180,jerk*math.pi/180,jerk*math.pi/180)
46
47     local targetPos1=(90*math.pi/180,90*math.pi/180,-90*math.pi/180,90*math.pi/180,90*math.pi/180,90*math.pi/180)
48     moveToConfig(jointHandles,maxVel,maxAccel,maxJerk,targetPos1)
49
50     local targetPos2=(-90*math.pi/180,45*math.pi/180,90*math.pi/180,135*math.pi/180,90*math.pi/180,90*math.pi/180)
51     moveToConfig(jointHandles,maxVel,maxAccel,maxJerk,targetPos2)
52
53     local targetPos3=(0,0,0,0,0,0)
54     moveToConfig(jointHandles,maxVel,maxAccel,maxJerk,targetPos3)
55 end

```

Figura 1.14: El *child script* del robot UR10.

### Ejercicio 1.10.6: Acciones sobre los elementos

En este ejercicio vamos a cambiar la posición de un elemento en Lua. Para ello, siga los siguientes pasos:

- Abra la escena `scenes/more/irb140_paint_gun.ttt`.
- Se observa un robot IRB140 equipado con una pistola de pintura.
- Inserte un objeto en la escena. File – import Mesh. Puede incluir cualquier objeto en formato STL o mesh. En pyARTE encontrará incluidos algunos de estos modelos. Por ejemplo, navegue y seleccione el fichero `scenes/stl/aston_martin.stl`.
- Observará que se ha creado un objeto que se denomina *Shape*. Cambie el nombre a “Aston”. Deberá tener algo parecido a la Figura 1.15.
- Añada un child script (botón derecho): Add – associated child script – non-threaded – Lua.
- Edite el código del Child script de Lua con el código que se proporciona debajo.
- Pruebe a mover el vehículo en diferentes direcciones XYZ.

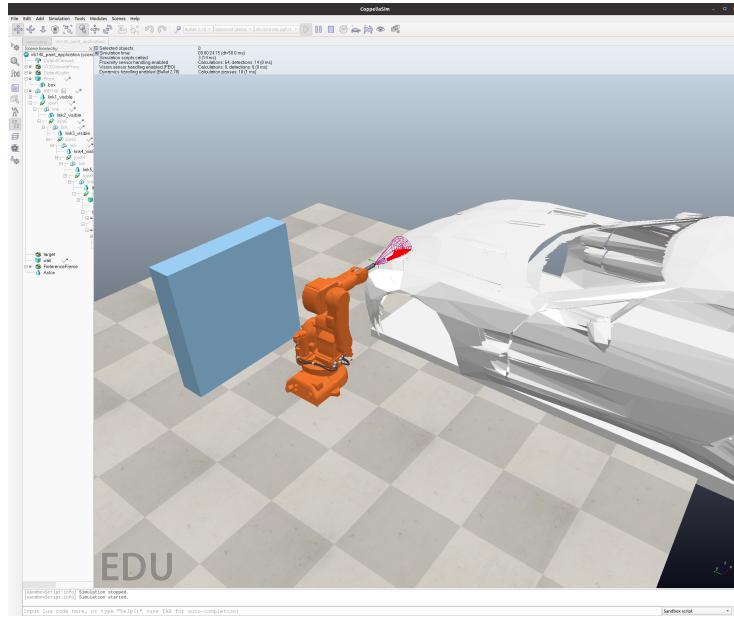


Figura 1.15: Una aplicación de pintura.

Las siguientes funciones aparecen en los scripts de Lua de los objetos de la simulación. En concreto, es interesante tener en cuenta que la variable `h_aston` es un manejador del objeto insertado. A través de este manejador se puede modificar la posición, velocidad y aceleración del objeto, así como aplicarle fuerzas y momentos.

```

function sysCall_init()
    h_aston=sim.getObjectHandle('Aston')
    p=sim.getObjectPosition(h_aston,-1)
end

function sysCall_actuation()
    p[2] = p[2] - 0.001
    sim.setObjectPosition(h_aston, -1, p)
end

```

## 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 1.16a. Si entramos en el menú “Show dynamic properties dialog”, aparecerá el diálogo de la Figura 1.16b. 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”.

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

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`.

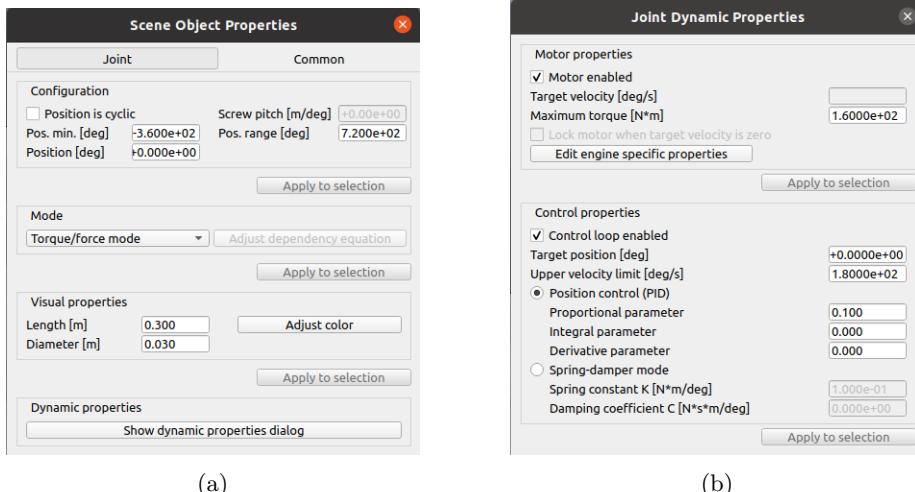


Figura 1.16: a) Propiedades de un objeto de tipo *joint* en Coppelia. b) Propiedades dinámicas de una articulación (*joint*).

#### 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 = []
    errorCode, robotbase = sim.simxGetObjectHandle(self.clientID,
                                                    base_name,
                                                    sim.simx_opmode_oneshot_wait)
    errorCode, q1 = sim.simxGetObjectHandle(self.clientID,
                                           base_name + '/' + joint_name + '1',
                                           sim.simx_opmode_oneshot_wait)
    errorCode, q2 = sim.simxGetObjectHandle(self.clientID,
                                           base_name + '/' + joint_name + '2',
                                           sim.simx_opmode_oneshot_wait)
    errorCode, q3 = sim.simxGetObjectHandle(self.clientID,
                                           base_name + '/' + joint_name + '3',
```

```

        sim.simx_opmode_oneshot_wait)
errorCode, q4 = sim.simxGetObjectHandle(self.clientID,
                                         base_name + '/' + joint_name + '4',
                                         sim.simx_opmode_oneshot_wait)
errorCode, q5 = sim.simxGetObjectHandle(self.clientID,
                                         base_name + '/' + joint_name + '5',
                                         sim.simx_opmode_oneshot_wait)
errorCode, q6 = sim.simxGetObjectHandle(self.clientID,
                                         base_name + '/' + joint_name + '6',
                                         sim.simx_opmode_oneshot_wait)

armjoints.append(q1)
armjoints.append(q2)
armjoints.append(q3)
armjoints.append(q4)
armjoints.append(q5)
armjoints.append(q6)

self.joints = armjoints

```

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

## 1.12. Comentarios finales

Finalmente, puede volver a abrir el script de python que sirvió de introducción `practicals/irb140_first_script.py` y la escena `irb140.ttt`. Puede utilizar cualquier editor de python para ello.

```

1 import numpy as np
2 from robots.abbirb140 import RobotABBIRB140
3 from robots.simulation import Simulation
4
5 if __name__ == "__main__":
6     simulation = Simulation()
7     clientID = simulation.start()
8     robot = RobotABBIRB140(clientID=clientID)
9     robot.start()
10
11    q1 = np.array([-np.pi/4, np.pi/8, np.pi/8, np.pi/4, -np.pi/4, np.pi/4])
12    q2 = np.array([0, 0, 0, 0, 0, 0])
13    q3 = np.array([np.pi/8, 0, -np.pi/4, 0, 3*np.pi/2, 0])
14
15    robot.moveAbsJ(q1, precision=True)
16    simulation.wait(100)
17    robot.moveAbsJ(q2, precision=True)
18    simulation.wait(100)
19    robot.moveAbsJ(q3, precision=True)
20    simulation.wait(100)
21    simulation.stop()

```

Así pues, repasando, vemos que en las líneas 1, 2 y 3 se importan paquetes de python (`numpy`) y de la propia librería pyARTE. En las líneas 6 y 7 se crea un objeto para manejar la simulación y se conecta con el método `start()`. Seguidamente, se crea un objeto de la clase `RobotABBIRB140()` y, en la línea 9 se conecta con Coppelia, obteniéndose todos los manejadores. El resto del script ya se vio, pues define unas posiciones articulares y comanda al robot a ellas.

## 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.

A continuación, en este documento de prácticas encontraréis una primera parte con contenido teórico sobre: matrices de rotación, matrices de transformación homogénea y ángulos de Euler. 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:

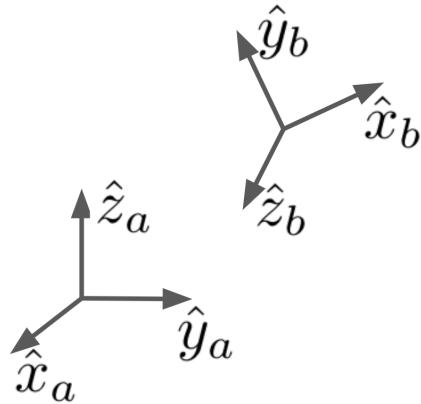


Figura 2.1: Dos sistemas de referencia con diferente orientación relativa.

- 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.
- Definir un punto de soldadura en la carrocería de un vehículo.
- Indicar 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:

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

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  $\alpha$  (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  $\beta$  (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  $\gamma$  (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  $\alpha$  (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  $\beta$  (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  $\gamma$  (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:

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

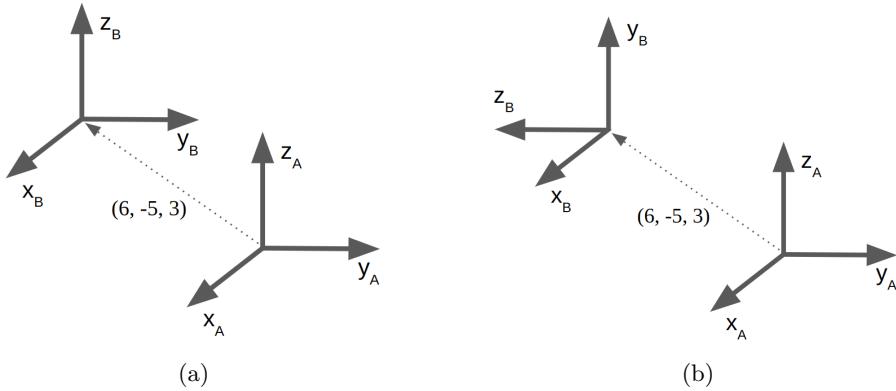


Figura 2.2: a) Translación pura entre dos sistemas  $A$  y  $B$ . b) Rotación seguida de translación entre dos sistemas  $A$  y  $B$ .

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  $(\alpha, \beta, \gamma)$  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:

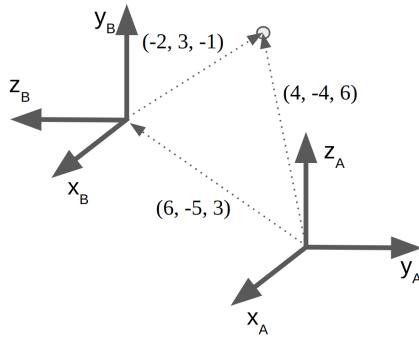


Figura 2.3: Transformación en las coordenadas de un mismo punto en dos sistemas de referencia  $A$  y  $B$ .

- 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  $\alpha$  (rad) sobre el eje X. A continuación, sobre el sistema de referencia resultante se realiza una rotación de  $\beta$  (rad) sobre el eje Y. Finalmente, sobre el sistema resultante realizamos una rotación de  $\gamma$  (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  $(\alpha, \beta, \gamma)$ . 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  $\beta$ :

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

Conocido  $\beta$  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  $\gamma$  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  $\theta$  a partir de su seno y su coseno. Dicha función devolverá el único ángulo  $\theta$  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  $\theta$  del número complejo  $\cos \theta + i \sin \theta$ .

Con todo esto, podemos volver a escribir la solución de  $\gamma$  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  $\gamma'$  si utilizamos el otro valor hallado para  $\beta$ :  $\beta'$ .

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

Finalmente,  $\alpha$  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  $\alpha'$  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()
```

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 figura gráfica.

#### 2.14.2. 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 artelib.rotationmatrix import RotationMatrix, Rx, Ry, Rz
from artelib.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.

### 2.14.3. 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 exemplifica 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`.  
 Modifique el giro `np.pi/4` y observe el resultado en la figura (T3).

## 2.15. La clase Euler

En pyARTE se añade una clase importante: `Euler`. Esta clase permite la conversión sencilla entre ambas las 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.

**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 como dibujar estos elementos en Coppelia, pues visualizarlos 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 2.4, 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 clases `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()
    clientID = simulation.start()
    frame = ReferenceFrame(clientID=clientID)
    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

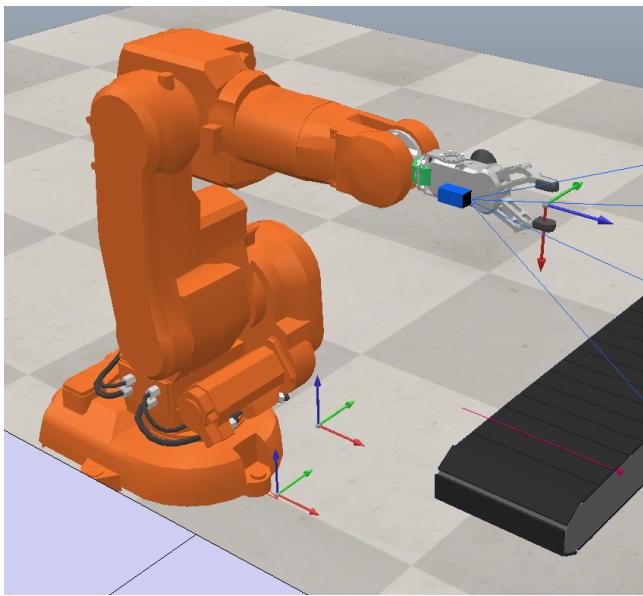


Figura 2.4: Sistemas de referencia en una escena de Coppelia. Fuente: Captura de pantalla sobre Coppelia Sim.

(`frame.set_position`) mediante un `Vector`, una lista o un array de numpy. La 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.

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

- 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 2.5. 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.

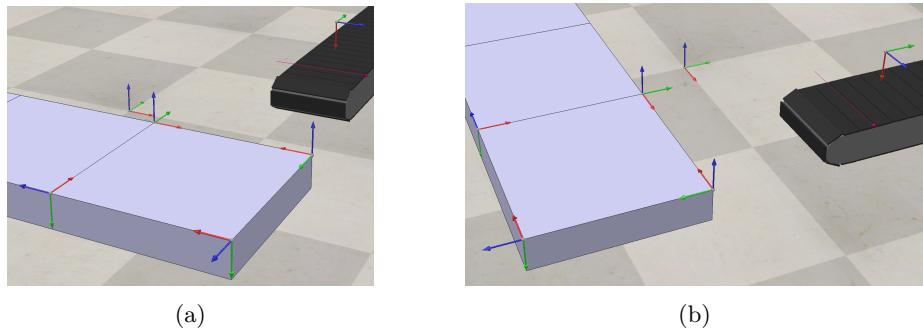


Figura 2.5: Sistemas de referencia en una escena de Coppelia. Fuente: Captura de pantalla de Coppelia Sim.

## 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.

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

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

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

$$\begin{aligned} R_a &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix} & e_a &= (-\pi/2, 0, 0), (\pi/2, \pi, -\pi) \\ R_b &= \begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \end{pmatrix} & e_b &= (\pi/2, 0, \pi/2), (-\pi/2, \pi, -\pi/2) \\ R_c &= \begin{pmatrix} 0 & 0 & 1 \\ 0 & -1 & 0 \\ 1 & 0 & 0 \end{pmatrix} & e_c &= (0, \pi/2, \pi), (\pi, \pi/2, 0) \text{ (degenerado)} \\ R_d &= \begin{pmatrix} 0 & 0 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{pmatrix} & e_d &= (0, -\pi/2, \pi), (\pi, -\pi/2, 0) \text{ (degenerado)} \end{aligned}$$

### 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('Rxzx 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

## Cinemática directa

### 3.1. Introducción

En esta práctica utilizaremos la librería ARTE para Matlab. Esta librería nos permitirá afianzar los conceptos de cinemática directa en posición y orientación y analizar algunos mecanismos manipuladores industriales.

### 3.2. Objetivos

Durante esta práctica se abordan los siguientes objetivos de aprendizaje que se materializan en una serie de actividades:

- **Objetivos de aprendizaje:**

- El estudiante deberá ser capaz de definir la cinemática de un robot mediante los parámetros Denavit-Hartenberg de un manipulador robótico.
- Comprender la solución de la cinemática directa de un robot manipulador (en posición y orientación) usando parámetros de DH.

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

- Visualizar los sistemas de referencia de DH sobre un robot.
- Calcular transformaciones sucesivas basadas en parámetros de DH.
- Configurar los parámetros DH de un robot de la librería ARTE.

En lo relativo al manejo de la librería ARTE en Matlab, abordaremos los siguientes conceptos:

- 1 Instalación de la librería ARTE en Matlab.
- 2 Carga de robots en la librería.
- 3 Visualización de robots en la librería.

4 Uso de la aplicación “teach”: una GUI (Graphical User Interface) con robots de la librería.

5 Modificación de los parámetros de DH de un robot de la librería.

### 3.3. Instalación de la librería

La librería se encuentra instalada en los equipos de prácticas. Se indica, a continuación, cómo instalar la librería en otro equipo, para poder realizar las prácticas en un PC personal, por ejemplo. Instalar la librería en Matlab es sencillo. Simplemente, sigue los siguientes pasos:

- a) Descarga y descomprime la librería desde <http://arvc.umh.es/arte>.
- b) Puedes descomprimir el fichero .zip en el escritorio. También puedes descomprimir la librería en un dispositivo USB para poderlo llevar contigo.
- c) Arranca Matlab.
- d) A continuación, navega hasta tu directorio dentro de Matlab. Por ejemplo, si copiaste la librería al Escritorio, deberías tener algo parecido a esto (Linux):

```
>> pwd  
ans =  
/home/arvc/Desktop/arte
```

Si usas Windows, puedes ver algo como esto:

```
>> pwd  
ans =  
C:\Documents and Settings\arvc\Desktop\arte
```

- e) Seguidamente, en la línea de comandos de Matlab escribe lo siguiente:

```
>> init_lib
```

El texto siguiente confirma que la librería se ha inicializado correctamente.

```
% ARTE (A Robotics Toolbox for Education)  
% Copyright (C) 2012 Arturo Gil Aparicio, arturo.gil@umh.es  
% http://arvc.umh.es/arte  
%  
% This program is free software: you can redistribute it and/or modify  
% it under the terms of the GNU Lesser General Public License as published  
% by the Free Software Foundation, either version 3 of the License, or  
% any later version.  
%  
% This program is distributed in the hope that it will be useful,  
% but WITHOUT ANY WARRANTY; without even the implied warranty of  
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
```

```

%
%   GNU Lesser General Public License for more details.
%
%   You should have received a copy of the GNU Lesser General Public License
%   along with this program. If not, see <http://www.gnu.org/licenses/>.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% To begin with, try loading a robot:
%>> robot = load_robot('ABB','IRB140');
%
% Next, draw it on its zero position:
%>> drawrobot3d(robot,[0 0 0 0 0])
%
% Next, try a different pose:
%>> drawrobot3d(robot,[0 pi/2 -pi/2 0 0 0])
%
% Finally, use the teach pendant to move the robot:
%>> teach
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

Con el comando `init_lib` se inicializa la librería. Matlab añadirá todos los directorios de la librería al path de Matlab para que todas las funciones estén disponibles.

### 3.4. Carga de robots en ARTE

Toda la librería está basada en la estructura de datos que llamaremos “robot”. Esta estructura de datos de Matlab guarda todos los parámetros necesarios para definir un robot y se guarda en el fichero `parameters.m` de cada robot de la librería. Existe la posibilidad de añadir nuevos robots a la librería (aunque no es el objetivo de la práctica de hoy). Para cargar cualquier robot a la librería podéis hacer:

```

>> robot=load_robot('ABB','IRB140')

ans =
/Users/arturogilaparicio/Desktop/arte/robots/ABB/IRB140

Reading link 0
/Users/arturogilaparicio/Desktop/arte/robots/ABB/IRB140/link0.stl
EndOfFile found...
Reading link 1
/Users/arturogilaparicio/Desktop/arte/robots/ABB/IRB140/link1.stl
EndOfFile found...
Reading link 2
/Users/arturogilaparicio/Desktop/arte/robots/ABB/IRB140/link2.stl
EndOfFile found...

[...]

/Users/arturogilaparicio/Desktop/arte/robots/ABB/IRB140/link5.stl
EndOfFile found...
Reading link 6
/Users/arturogilaparicio/Desktop/arte/robots/ABB/IRB140/link6.stl
EndOfFile found...
robot =

```

```

name: 'ABB_IRB140_M2000'
DH: [1x1 struct]
J: []
inversekinematic_fn: [1x33 char]
directkinematic_fn: [1x25 char]
DOF: 6
kind: 'RRRRRR'
maxangle: [6x2 double]
velmax: [6x1 double]
accelmax: [6x1 double]
linear_velmax: 2.5000
T0: [4x4 double]
debug: 0
q: [6x1 double]
qd: [6x1 double]
qdd: [6x1 double]
time: []
q_vector: []
qd_vector: []
qdd_vector: []
last_target: [4x4 double]
last_zone_data: 'fine'
tool0: [1x19 double]
wobj0: []
tool_activated: 0
path: [1x65 char]
graphical: [1x1 struct]
axis: [1x6 double]
has_dynamics: 1
dynamics: [1x1 struct]
motors: [1x1 struct]

```

Con el comando anterior hemos cargado el robot IRB140 del fabricante ABB, guardado en el directorio `arte/robots/ABB/IRB140`. Así pues, en el entorno de Matlab, existirá una variable denominada `robot`. Podemos ver el contenido de esta variable si hacemos:

```
>> robot
```

La función `load_robot` también puede ser llamada sin parámetros:

```
>> robot=load_robot
```

De esta manera, es posible navegar a cualquiera de los directorios de robots de los fabricantes y hacer click en el fichero `parameters.m`. Escribe el comando anterior y navega al directorio `arte/robots`, donde encontrarás un directorio para cada fabricante, incluyendo: ABB, KUKA, Stäubli y muchos más.

Si todo se ha realizado correctamente, debería aparecer una figura con el robot que se acaba de cargar en la librería (Figura 3.3).

#### Ejercicio 3.4.1: Carga de robots en ARTE

**Ejercicio:** Cargue, al menos, tres robots de tres fabricantes diferentes. Utilice el comando resumido:

```
>> robot=load_robot
```

Transformación	$\theta$ (rad)	$d$ (m)	$a$ (m)	$\alpha$ (rad)
$0 \rightarrow 1$	$q_1$	0,352	0,070	$-\pi/2$
$1 \rightarrow 2$	$q_2 - \pi/2$	0	0,360	0
$2 \rightarrow 3$	$q_3$	0	0	$-\pi/2$
$3 \rightarrow 4$	$q_4$	0,380	0	$\pi/2$
$4 \rightarrow 5$	$q_5$	0	0	$-\pi/2$
$5 \rightarrow 6$	$q_6 + \pi$	0,065	0	0

Tabla 3.1: Parámetros DH del robot ABB IRB140.

### 3.5. Parámetros de DH de un robot

En lo sucesivo, utilizaremos el robot IRB140 para el estudio de la cinemática directa. Podemos observar los parámetros de DH de este robot en el fichero:

ARTE/robots/ABB/IRB140/parameters.m.

Dentro de este fichero, debemos fijarnos en las líneas siguientes:

```
robot.DH.theta= '[q(1) q(2)-pi/2 q(3) q(4) q(5) q(6)+pi]';  
robot.DH.d='[0.352 0 0 0.380 0 0.065]';  
robot.DH.a='[0.070 0.360 0 0 0 0]';  
robot.DH.alpha= '[-pi/2 0 -pi/2 pi/2 -pi/2 0]';
```

Donde  $q(1)$ ,  $q(2)$ , ... etc son las coordenadas articulares. Fíjate en los parámetros de DH de este robot, según se indican en la Tabla 3.1 para deducir la forma en que se definen estos parámetros. Ten en cuenta que las variables `robot.DH.theta`, `d`, `a` y `alpha` son cadenas de texto y, por tanto, se trata de funciones escritas en base a las variables articulares  $q(i)$ . Para poder utilizar estas variables es necesario evaluarlas con el valor real de las coordenadas articulares en cada instante. Por ejemplo, el código siguiente calcula la matriz  ${}^0A_1$  para unos valores de las coordenadas articulares:

```
q=[0 0 0 0 0 0]  
theta = eval(robot.DH.theta);  
d=eval(robot.DH.d);  
a=eval(robot.DH.a);  
alpha= eval(robot.DH.alpha);  
A01 = dh(theta(1), d(1), a(1), alpha(1))
```

Nótese que `eval` evalúa la función de texto para convertirla en un vector numérico de Matlab. Los parámetros de DH de la Tabla 3.1, cuando  $q = [0, 0, 0, 0, 0, 0]$ , corresponden con la colocación de los sistemas de referencia de la Figura 3.1. **Importante:** La función `dh` calcula la matriz de transformación de los parámetros DH. En concreto:

```
function A=dh(theta, d, a, alpha)  
A=[cos(theta) -cos(alpha)*sin(theta) sin(alpha)*sin(theta) a*cos(theta);  
sin(theta) cos(alpha)*cos(theta) -sin(alpha)*cos(theta) a*sin(theta);  
0 sin(alpha) cos(alpha) d;  
0 0 0 1];
```

### 3.6. Cinemática directa del robot IRB140

Nos interesa ahora calcular la posición y orientación del extremo del robot cuando las posiciones articulares son  $q = [0 0 0 0 0 0]$  (*id est*, la posición inicial).

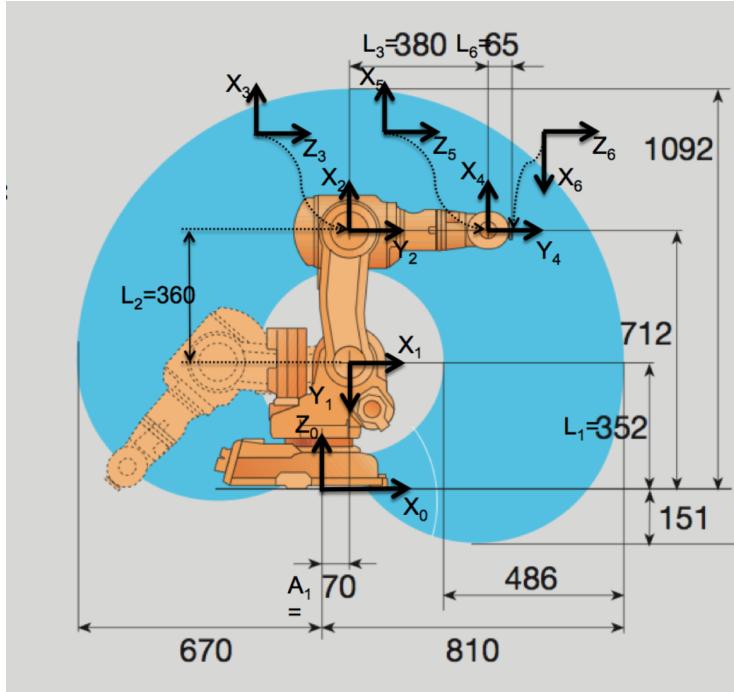


Figura 3.1: Sistemas de DH sobre el robot IRB 140 de ABB<sup>©</sup>. Fuente: modificado desde el manual oficial del robot publicado en [www.abb.com](http://www.abb.com).

La función que calcula la cinemática directa se denomina `directkinematic(robot, q)`. Esta función recibe dos parámetros: la propia variable `robot` y las coordenadas articulares de éste. Haga lo siguiente:

```
>> robot=load_robot('ABB', 'IRB140');
>> q = [0 0 0 0 0 0];
>> T=directkinematic(robot, q)
T =
    0.0    0.0    1.0    0.5150
    0.0    1.0    0.0    0.0
   -1.0    0.0    0.0    0.7120
    0    0.0    0    1.0
>> drawrobot3d(robot, q)
```

Como resultado, la matriz  $T$  representa la posición y orientación del sistema de referencia 6 con respecto al sistema de referencia de la base  $X_0Y_0Z_0$  cuando todas las coordenadas son nulas (Figura 3.1).

La función `drawrobot3d` realiza una representación gráfica del robot (Figura 3.3). Utiliza en esta figura los controles de punto de vista de Matlab (Figura 3.2). Después de llamar a `drawrobot3d(robot, q)` deberías comprobar que la posición y orientación de  $T$  coincide con lo que se observa en la Figura 3.3. Es decir, comprueba que:

- La posición 3D es la que indica  $T$ .
- La orientación es la indicada por la submatriz de rotación de  $T$ .



Figura 3.2: Cambio en el punto de vista 3D de una figura de Matlab.

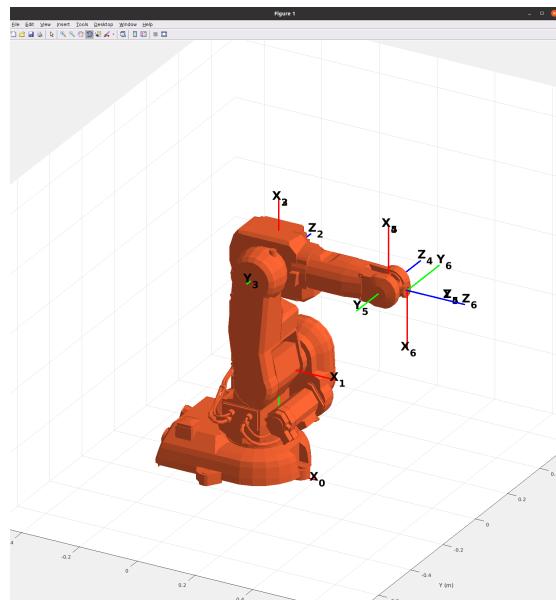


Figura 3.3: El robot IRB140 de ABB<sup>©</sup> en la librería Matlab<sup>©</sup>

#### Ejercicio 3.6.1: Cinemática directa

Calcule la matriz  $T$  para las siguientes posiciones articulares. Dibuje el robot (`drawrobot3d`) para cada posición y compruebe la posición y orientación alcanzadas.

- $q = [\pi/2 \ 0 \ 0 \ 0 \ 0 \ 0]$ .
- $q = [\pi/2 \ 0 \ \pi/2 \ 0 \ 0 \ 0]$ .
- $q = [0 \ 0 \ 0 \ \pi/2 \ 0 \ 0]$ .
- $q = [0 \ 0 \ 0 \ \pi/2 \ 0 \ -\pi/2]$ .
- $q = [\pi/2 \ 0 \ 0 \ \pi/2 \ \pi/2 \ \pi/2]$ .

Es habitual guardar la variable articular  $q$  en una variable de Matlab y hacer, por ejemplo:

```
>> robot=load_robot('ABB', 'IRB140');
>> q = [pi/8 pi/8 pi/8 pi/8 pi/8];
>> T=directkinematic(robot, q)
>> drawrobot3d(robot, q)
```

### 3.7. Transformaciones intermedias

Recordando los conceptos de cinemática directa vistos en las sesiones de teoría, sabemos que:

$$T = {}^0A_1{}^1A_2{}^2A_3 \dots {}^5A_6$$

Cada una de las matrices  ${}^{i-1}A_i$  es función de una fila de parámetros de DH (y del valor actual de  $q$ ). Se propone ahora observar que las transformaciones intermedias también son correctas.

**Muy importante:** en esta práctica se busca que el/la estudiante sea capaz de visualizar las posiciones y orientaciones relativas entre pares de transformaciones consecutivas. Para ello, use la figura de Matlab de drawrobot3d. Para poder hacer esto, usaremos la función dh que permite calcular cada matriz D-H intermedia de la siguiente manera:

```
>> A01 = dh(robot, [0 0 0 0 0], 1)
A01 =
    1.0000      0      0     0.0700
    0     0.0000    1.0000      0
    0    -1.0000    0.0000    0.3520
    0      0      0     1.0000

>> A12 = dh(robot, [0 0 0 0 0], 2)
A12 =
    0.0000    1.0000      0     0.0000
   -1.0000    0.0000      0    -0.3600
    0      0    1.0000      0
    0      0      0     1.0000

>> A23 = dh(robot, [0 0 0 0 0], 3)
A23 =
    1.0000      0      0      0
    0     0.0000    1.0000      0
    0    -1.0000    0.0000      0
    0      0      0     1.0000
```

Es decir: el último índice (el número), hace referencia a la fila de la tabla de DH donde están almacenados los parámetros DH del robot. Así, sabemos que en la primera fila de una tabla de DH se define  ${}^0A_1$ , en la fila 2 se define  ${}^1A_2$ ... etc.

#### Ejercicio 3.7.1: Cinemática directa: transformaciones intermedias

Calcule las transformaciones intermedias:

```
>> robot=load_robot('ABB', 'IRB140');
>> q = [0 0 0 0 0];
>> drawrobot3d(robot, q)
>> T1=directkinematic(robot, q)
>> A01 = dh(robot, q, 1);
>> A12 = dh(robot, q, 2);
>> A23 = dh(robot, q, 3);
>> ...
>> T2 = A01*A12*A23...
```

¿Qué relación existe entre T1 y T2?

### Ejercicio 3.7.2: Visualizando las transformaciones intermedias

Seguidamente, se propone al estudiante que observe con detalle las transformaciones intermedias  $i^{-1}A_i$ . Para ello, haga:

```
>> robot.graphical.draw_transparent=1  
>> q = [pi/2 pi/2 pi/2 pi/2 pi/2 pi/2]  
>> drawrobot3d(robot, q)  
>> A01 = dh(robot, q, 1)  
>> A12 = dh(robot, q, 2)  
>> A23 = dh(robot, q, 3)
```

Observando la figura de Matlab del robot (el robot es, ahora, transparente), deberá:

- Observar el sistema  $X_1Y_1Z_1$  en coordenadas del sistema  $X_0Y_0Z_0$ . Compruebe esta transformación con A01. ¿La posición y orientación definidas por A01 son las que se muestra en la gráfica?
- Fíjese en el sistema  $X_1Y_1Z_1$  en coordenadas del sistema  $X_2Y_2Z_2$ . Compare esta transformación con A21 = inv(A12).
- Mire el sistema  $X_3Y_3Z_3$  en coordenadas del sistema  $X_2Y_2Z_2$ . Relacione esta transformación con la matriz A23.
- Plantee el sistema  $X_4Y_4Z_4$  en coordenadas del sistema  $X_0Y_0Z_0$ . Compruebe esta transformación mediante A04=A01\*A12\*A23\*A34.
- Por último, ensayando un ejercicio que resulta frecuente en los exámenes de Robótica, deberá ser capaz de observar el sistema  $X_0Y_0Z_0$  en coordenadas del sistema  $X_6Y_6Z_6$ .
- Halle y observe A60.

## 3.8. Uso de la aplicación teach

Para facilitar la vida del estudiante, ARTE cuenta con una aplicación gráfica en el entorno de Matlab. La aplicación emula una “paletera de programación” de un robot industrial, pues muestra la misma información que este tipo de dispositivos. Para lanzar la aplicación gráfica, por favor, escriba:

```
>> robot=load_robot  
>> teach
```

La función `load_robot`, cuando se llama sin argumentos, permite cargar cualquier robot de la librería: seleccione el fichero `parameters.m` de cualquiera de los robots bajo el directorio `arte/robots`.

Cuando llamamos a la función `teach`, la siguiente aplicación gráfica debería aparecer (Figura 3.4). Esta aplicación gráfica de Matlab permite al alumno experimentar de forma sencilla con multitud de robots en la librería y los siguientes conceptos teóricos:

- La cinemática directa del robot en posición/orientación.

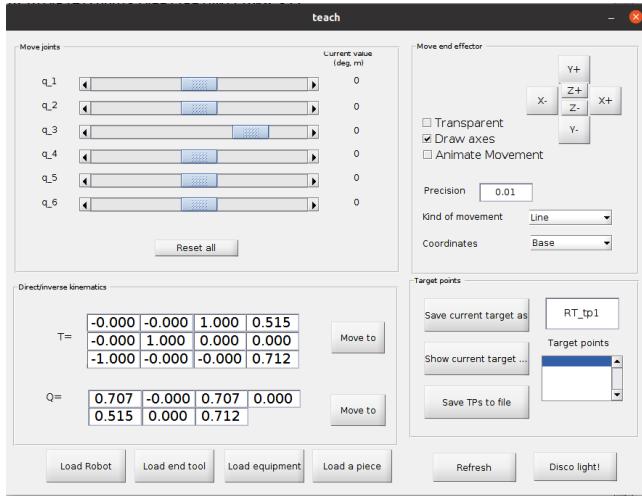


Figura 3.4: La aplicación GUI **teach** de la librería ARTE.

- Uso de los sistemas de referencia de DH sobre el robot.
- La cinemática inversa del robot.
- Representación de la orientación con Cuaterniones.

#### Ejercicio 3.8.1: Cinemática directa con teach

Realice las siguientes actividades usando **teach**:

- Modifique el valor de  $q_1, q_2, \dots, q_6$  y observe la posición y orientación que calcula, en cada instante, la aplicación **teach**. La posición/orientación se muestra como T (matriz homogénea). Deberá mover los controles de tipo *slider* de la aplicación para modificar los valores articulares. Interprete los resultados. **Cuidado:** En esta aplicación gráfica las variables articulares se muestran en grados sexagesimales. En el resto de la librería se utilizan radianes.
- La sección de la aplicación titulada “Move end effector” utiliza la cinemática inversa del robot para mover el extremo del mismo. Esta parte será tratada en una práctica posterior.
- Utilice el control “Transparent”, que permite dibujar al robot de forma transparente. Esto permite una mejor visualización de los sistemas de referencia sobre el robot.
- Utilice el control “Draw axes”. Esto permite visualizar mejor al robot sin ver los sistemas de referencia.

### Ejercicio 3.8.2: Transformaciones intermedias con teach

Realice las siguientes actividades:

- Modifique el valor de  $q_1, q_2, \dots, q_6$  y observe la posición y orientación que calcula, en cada instante, la aplicación **teach**.
- Ponga el robot en modo transparente y con los sistemas de referencia de DH visibles.
- Cambie el valor de  $q_1$  y observe el movimiento relativo entre los sistemas  $X_0Y_0Z_0$  y  $X_1Y_1Z_1$ . Observe sobre qué eje está rotando  $X_1Y_1Z_1$  respecto de  $X_0Y_0Z_0$ .
- Repita la experiencia con  $q_2, q_3, \dots, q_6$ .
- Cuando se mueve  $q_3$ ... ¿qué sistemas de referencia están cambiando? ¿Sobre qué vector se está rotando?

### Ejercicio 3.8.3: Carga de elementos con teach

Realice las siguientes actividades:

- Experimente con el botón “Load robot” para cargar otros robots de diferentes modelos y fabricantes. Navegue a la carpeta: **arte/robots** y cargue el fichero **parameters.m**.
- Experimente con el botón “Load end tool” para cargar elementos terminales en el robot. Podrás cargar pinzas para robots, ventosas de succión, herramientas de soldadura, entre otros “end effectors”. Encontrarás elementos terminales en **arte/robots/equipment/end\_tools**
- Haga pruebas con el botón “Load equipment”. En esta sección podrás añadir elementos accesorios para representar la celda de trabajo del robot. Encontrarás estos elementos en **arte/robots/equipment/**.
- Utilice el botón “load piece” para cargar una pieza de trabajo. Una pieza de trabajo podrá ser asida por el extremo del robot para simular un proceso industrial típico. Encontrarás piezas en **arte/robots/equipment/cylinders** y **arte/robots/equipment/objects**.
- El botón “Refresh” se debe utilizar, en ocasiones, para actualizar la última vista del robot.

**Nota:** este último ejercicio es libre, un posible resultado de esta actividad se presenta en la Figura 3.5, donde se ha cargado:

- Robot: **robots/ABB/IRB140**.
- End tool: **robots/equipment/end\_tools/paint\_gun2**

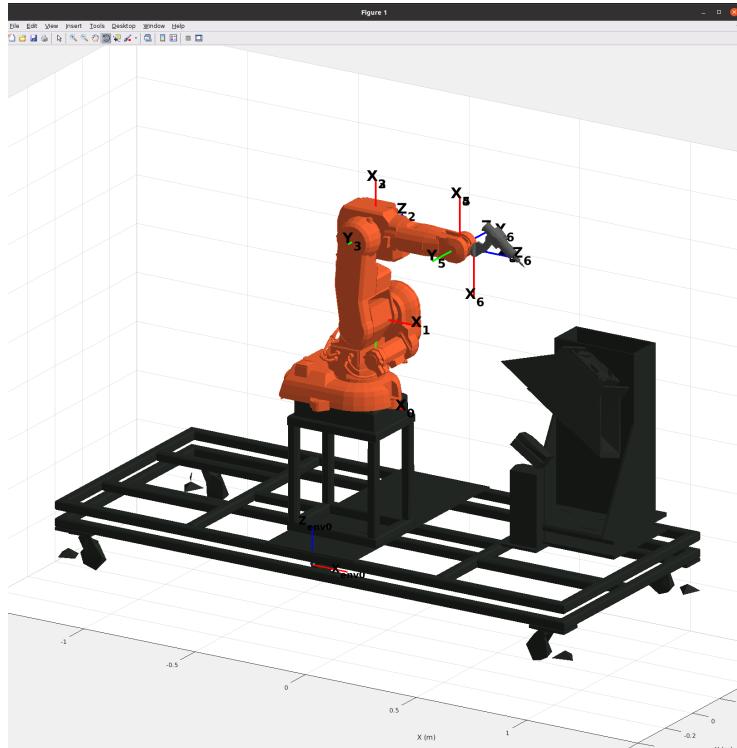


Figura 3.5: Entorno robótico industrial configurado con teach.

- Equipment: robots/equipment/bumper\_cutting

### 3.9. Ejercicios finales

Se plantean, finalmente, dos ejercicios que permitirán al estudiante afianzar los conceptos relativos a los parámetros de DH.

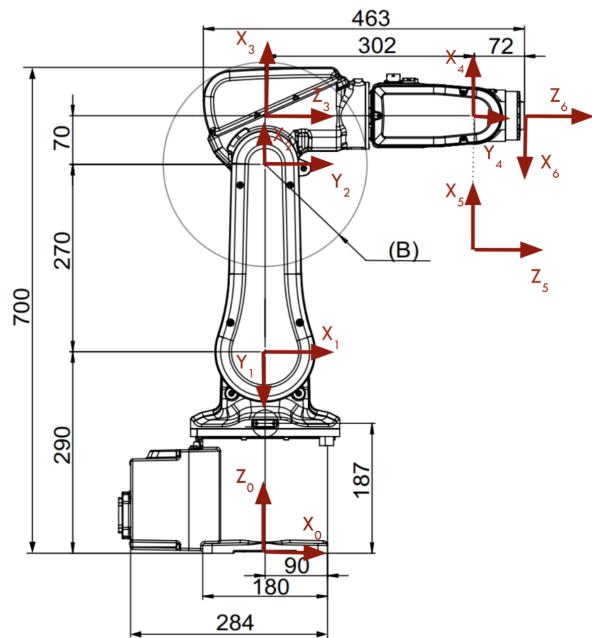


Figura 3.6: Cotas en mm para el robot.

### Ejercicio 3.9.1: Parámetros de DH I

Edite el fichero

`arte/robots/example/serial/parameters.m`. Descubrirá que el robot no tiene los parámetros de DH definidos. **Ejercicio:** Escriba los parámetros de DH del robot para que los sistemas aparezcan como en la Figura 3.6. A continuación, cargue el robot y use la función `drawrobot3d` para verlo:

```
>> robot = load_robot('example', 'serial')
>> drawrobot3d(robot)
```

Si ha escrito bien los parámetros de DH, el robot aparecerá correctamente, según se muestra en la Figura 3.7. En caso contrario, deberá editar, de nuevo, los parámetros de DH.

**IMPORTANTE:** deberá volver a cargar el robot (`load_robot`) después de editar y guardar el fichero `parameters.m`.

Finalmente, utilice `teach` para observar el movimiento del robot.

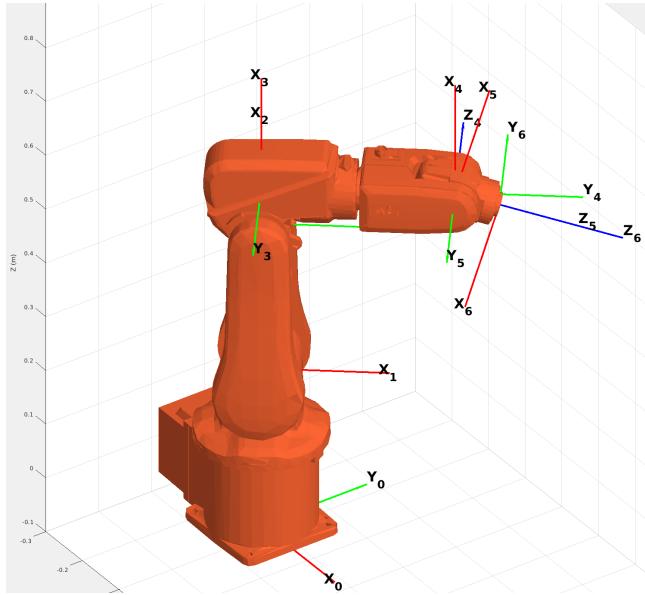


Figura 3.7: El robot del ejercicio en ARTE.

### Ejercicio 3.9.2: Parámetros de DH II

Abra el fichero

`arte/robots/example/serial2/parameters.m`. Descubrirá que los parámetros DH son incorrectos, con lo que el robot se representa de forma incorrecta(Figura 3.8). Con ayuda de la Figura 3.9, escriba los parámetros correctos ( $G=1075$  mm,  $D=1142,5$  mm)

**Ejercicio:** Escriba los parámetros de DH del robot. En este caso, es necesario que Ud. plantea una colocación de los sistemas de DH sobre el robot, rellene una tabla con los parámetros DH y, finalmente, añada estos parámetros al fichero de parámetros del robot.

**IMPORTANTE:** la colocación de los sistemas de coordenadas de DH sobre el robot no es única. Así pues, deberá probar con diferentes colocaciones.

A continuación, cargue el robot y use la función `drawrobot3d` para verlo:

```
>> robot = load_robot('example', 'serial2')
>> teach
```

Si ha escrito bien los parámetros de DH, el robot aparecerá correctamente. En caso contrario, deberá editar, de nuevo, los parámetros de DH.

**IMPORTANTE:** deberá volver a cargar el robot (`load_robot`) después de editar y guardar el fichero `parameters.m`.

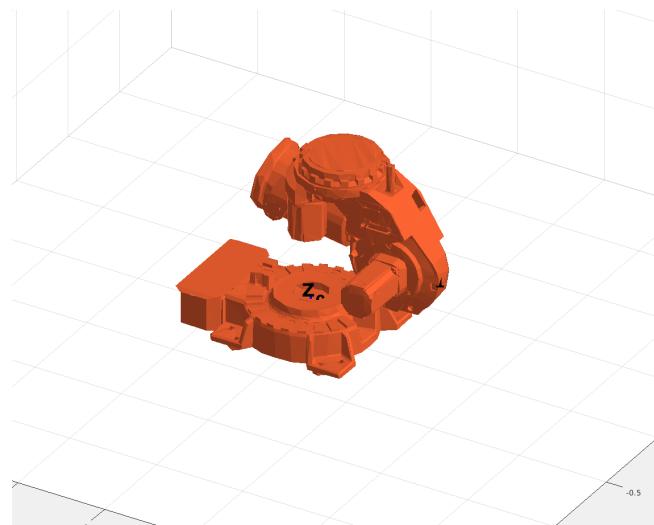


Figura 3.8: El robot `serial2` con parámetros DH incorrectos.

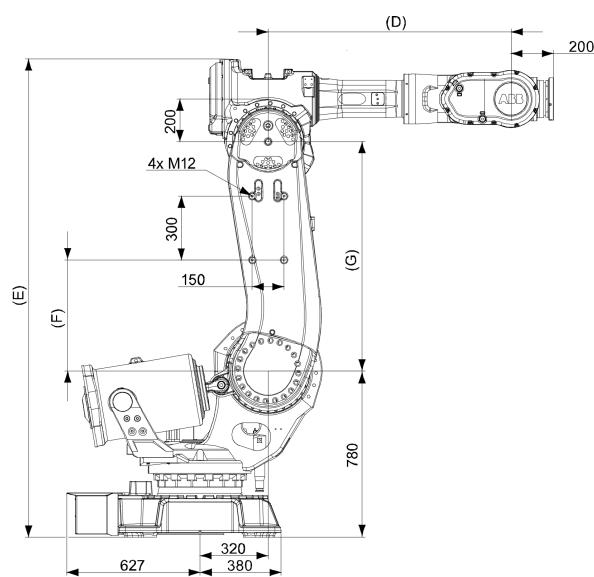


Figura 3.9: El robot `serial2` con sus cotas principales en mm.



## Capítulo 4

# Cinemática inversa

### 4.1. Objetivos

Con esta práctica se busca que el estudiante sea capaz de entender correctamente las soluciones de la cinemática inversa en posición/orientación en robots de tipo serie. Para ello, se muestran diferentes ejemplos de robots y se proponen diferentes actividades:

#### Objetivos de aprendizaje:

- Resolver las ecuaciones de la cinemática inversa de cualquier manipulador de tipo serie hasta 6 GDL y con muñeca esférica.
- Ser capaz de escribir estas ecuaciones en el entorno Matlab para un robot de su elección.
- Poder generar trayectorias articulares que hagan que el extremo siga un movimiento rectilíneo en el espacio cartesiano.

#### Actividades:

- Calcular con la librería soluciones particulares a la cinemática inversa de un robot plano de 3GDL y un robot industrial de 6GDL.
- Programar scripts que permitan comprender las relaciones entre la cinemática directa en inversa en robots de tipo serie.

### 4.2. El problema cinemático inverso

En la mayoría de aplicaciones robóticas, se conoce la posición (y orientación) de:

- Una pieza que debe asir el robot.
- Un punto de soldadura que hay que realizar.
- Una caja que hay que paletizar.

Así pues, muchas veces estaremos interesados en calcular los ángulos de las articulaciones que permiten que el extremo del robot alcance una posición y orientación determinadas. Conviene aclarar aquí algunos conceptos:

- **Cinemática del robot:** Se refiere al estudio del movimiento del robot sin atender a las fuerzas o momentos que generan ese movimiento. Así, la cinemática del robot considera el estudio de posiciones, velocidades y aceleraciones de todos los eslabones del robot. Especial interés se pone en el estudio de la posición, velocidad y aceleración **del extremo del robot**.
- **Cinemática directa:** Se denomina así el estudio cinemático del robot cuando se consideran conocidos los valores de las articulaciones. Se distingue:
  - **Cinemática directa en posición:** cuando se conoce la posición articular  $q$  y se desea calcular la posición y orientación del extremo.
  - **Cinemática directa en velocidad:** cuando se conoce  $q$  y las velocidades articulares  $\dot{q}$  y se desea calcular la velocidad del extremo.
- **Cinemática inversa:** Se denomina así el estudio cinemático del robot cuando se consideran conocidos la posición, orientación, velocidad y aceleración del extremo. Se distingue:
  - **Cinemática inversa en posición:** cuando se conoce la posición y orientación del extremo (por ejemplo, se conoce  $T$ )  $q$  y se desea calcular los valores de las articulaciones  $q$ .
  - **Cinemática inversa en velocidad:** cuando se conoce las velocidad lineal y angular del extremo ( $\vec{v}, \vec{\omega}$ ) y se desea calcular las velocidades articulares  $\dot{q}$  cuando el robot se encuentra en una posición articular  $q$ .

### 4.3. Soluciones cerradas de la cinemática inversa

Resolver el problema cinemático inverso no siempre es trivial. En los robots que se analizan en esta práctica es posible escribir un conjunto de ecuaciones cerradas que utilizan, como datos de partida, la posición y orientación especificadas por una matriz homogénea  $T$ . La solución de esta cinemática inversa depende de la topología del robot (es decir, la configuración de sus ejes). En general, podemos decir que, en el caso de robots de 6 GDL, para que exista una solución cerrada de la cinemática inversa, el robot debe contar con una muñeca de tipo esférico.

En esta práctica veremos:

- Las soluciones de la cinemática inversa de un robot plano de 3 GDL.
- Las soluciones de la cinemática inversa de un robot de 6 GDL industrial con muñeca esférica.

#### 4.4. Cinemática inversa de un robot plano de 3 GDL

Se presenta un brazo robótico de 3 GDL (Figura 4.1). La longitud de los eslabones es  $L_1=1\text{m}$ ,  $L_2=1\text{m}$  y  $L_3=1\text{m}$ . A continuación, definiremos:

- La cinemática directa del mecanismo que define la posición y orientación del sistema  $X_3Y_3$  en coordenadas del sistema 0.
- La matriz de transformación  $T = {}^0A_1{}^1A_2{}^2A_3$  como función de las coordenadas articulares.
- La cinemática inversa del mecanismo. Dada una matriz de transformación homogénea  $T$  conocida, escribiremos las ecuaciones que permiten calcular las coordenadas articulares que llevan al brazo a esa posición y orientación.

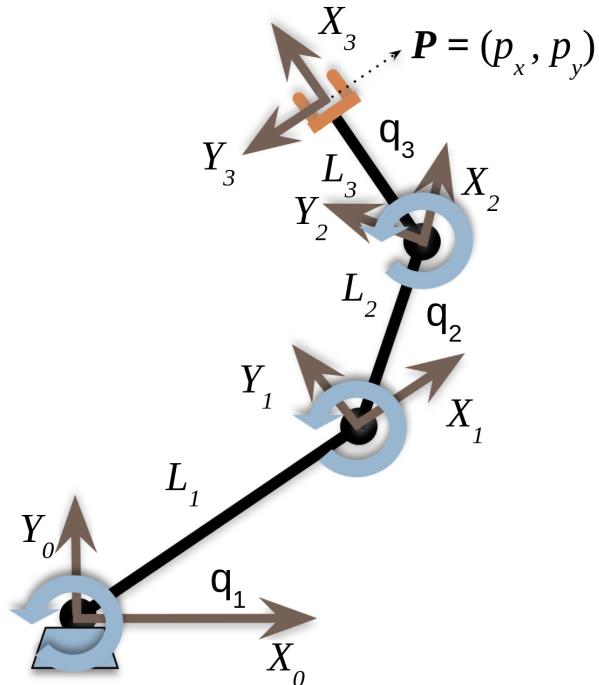


Figura 4.1: Un robot plano de 3 GDL.

En la Figura 4.1 se presenta una ubicación de los sistemas de DH. Se presenta en la Tabla 4.1 los parámetros de DH. En base a estos parámetros, podemos calcular la matriz de transformación como  $T = {}^0A_1{}^1A_2{}^2A_3$ :

$$T = \begin{pmatrix} \cos(\phi) & -\sin(\phi) & 0 & L_1 \cos(q_1) + L_2 \cos(q_1 + q_2) + L_3 \cos(q_1 + q_2 + q_3) \\ \sin(\phi) & \cos(\phi) & 0 & L_1 \sin(q_1) + L_2 \sin(q_1 + q_2) + L_3 \sin(q_1 + q_2 + q_3) \\ 0 & 0 & 1 & \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

con  $\phi = q_1 + q_2 + q_3$ . Por ejemplo, para una posición articular de ejemplo (Figura 4.1), tendremos  $q = (\pi/4, \pi/4, \pi/4)$ , y la matriz  $T$  vale:

$$T = \begin{pmatrix} -\sqrt{2}/2 & -\sqrt{2}/2 & 0 & 0 \\ \sqrt{2}/2 & -\sqrt{2}/2 & 0 & 2,4142 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Seguidamente, nos proponemos resolver la cinemática inversa del mecanismo. Por tanto, suponemos conocida la matriz  $T$  del extremo del robot:

$$T = \begin{pmatrix} \vec{x}_3 & \vec{y}_3 & \vec{z}_3 & \vec{p} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

siendo  $\vec{p}$  la posición del extremo del robot. Comenzamos obteniendo la posición del punto  $P$  de la Figura 4.1 que denominaremos  $\vec{p}_m$ .

$$\vec{p}_m = \vec{p} - L_3 \vec{x}_3$$

Conocido  $\vec{p}_m$  se plantea la cinemática inversa de un brazo de 2GDL. Puede obtener el lector, la solución, por métodos algebraicos o geométricos. Se indica, a continuación, una solución:

$$\begin{aligned} q_1 &= \arctan 2(p_{my}, p_{mx}) \pm \arccos \frac{R^2 + L_1^2 - L_2^2}{2L_1 R} \\ q_2 &= \pm(\arccos \frac{R - L_1^2 - L_2^2}{2L_1 L_2} - \pi) \end{aligned}$$

con  $R^2 = p_{mx}^2 + p_{my}^2$ . Una vez halladas  $q_1$  y  $q_2$ , deseamos calcular la coordenada articular  $q_3$ . Para ello nos fijamos en la primera columna de  $T$  y hallamos

$$\phi = \arctan 2(\sin \phi, \cos \phi)$$

Conocido  $\phi = q_1 + q_2 + q_3$  y dada una posible solución  $(q_1, q_2)$ , existe un único valor posible de  $q_3$ :

$$q_3 = \phi - q_1 - q_2$$

En resumen, existen 2 posibles soluciones diferentes  $(q_1, q_2, q_3)$  y  $(q_1^*, q_2^*, q_3^*)$  que se denominan: codo arriba y codo abajo. Las soluciones se deben arreglar de la siguiente manera:

■ CODO ABAJO:

$$\begin{aligned} q_1 &= \arctan 2(p_{my}, p_{mx}) + \arccos \frac{R^2 + L_1^2 - L_2^2}{2L_1 R} \\ q_2 &= \arccos \frac{R - L_1^2 - L_2^2}{2L_1 L_2} - \pi \\ q_3 &= \phi - q_1 - q_2 \end{aligned}$$

■ CODO ARRIBA:

$$\begin{aligned} q_1^* &= \arctan 2(p_{my}, p_{mx}) - \arccos \frac{R^2 + L_1^2 - L_2^2}{2L_1 R} \\ q_2^* &= \pi - \arccos \frac{R - L_1^2 - L_2^2}{2L_1 L_2} \\ q_3^* &= \phi - q_1^* - q_2^* \end{aligned}$$

Transformación	$\theta$ (rad)	$d$ (m)	$a$ (m)	$\alpha$ (rad)
$0 \rightarrow 1$	$q_1$	0	$L_1$	0
$1 \rightarrow 2$	$q_2$	0	$L_2$	0
$2 \rightarrow 3$	$q_3$	0	$L_3$	0

Tabla 4.1: Parámetros de DH del robot plano de 3GDL.

## 4.5. Cinemática inversa de un robot plano de 3 GDL en ARTE

Se puede experimentar con el mismo robot del apartado anterior utilizando la librería ARTE, pues el modelo cinemático está ya integrado en la librería. Para ello, puedes hacer:

```
>> robot = load_robot('example', '3dofplanar')
>> q = [pi/3 pi/3 pi/3];
>> T = directkinematic(robot, q)
T =
-1.0000   -0.0000       0   -1.0000
 0.0000   -1.0000       0    1.7321
  0       0    1.0000       0
  0       0       0    1.0000
>> qi = inversekinematic(robot, T)
Computing inverse kinematics for the Example 3DOF planar arm robot
qi =
  2.0944    1.0472
 -1.0472    1.0472
  2.0944    1.0472

>> T1 = directkinematic(robot, qi(:,1))
T1 =
-1.0000   -0.0000       0   -1.0000
 0.0000   -1.0000       0    1.7321
  0       0    1.0000       0
  0       0       0    1.0000
>> T2 = directkinematic(robot, qi(:,2))
T2 =
-1.0000   -0.0000       0   -1.0000
 0.0000   -1.0000       0    1.7321
  0       0    1.0000       0
  0       0       0    1.0000
>> drawrobot3d(robot, qi(:,1))
>> drawrobot3d(robot, qi(:,2))
```

Note que, en el ejemplo anterior, se utiliza la función `directkinematic` para calcular  $T$ , dada  $q = (\pi/3, \pi/3, \pi/3)$ . Seguidamente, se calculan la cinemática inversa con `inversekinematic`, lo que genera dos soluciones (arregladas por columnas). A continuación, se comprueba que ambas soluciones permiten obtener la misma matriz  $T$  original. Finalmente, el comando `drawrobot3d` permite representar las dos posibles soluciones del robot.

### Ejercicio 4.5.1: Cinemática inversa

Observe atentamente las ecuaciones de la cinemática inversa del robot de 3GDL plano. Encontrará la función en: `arte/robots/example/3dofplanar`. Dada una matriz T de posición y orientación, la función `inversekinematic` calcula las soluciones de la cinemática inversa de acuerdo con las ecuaciones presentadas anteriormente.

```
>> qinv = inversekinematic(robot, T)
```

### Ejercicio 4.5.2: Cinemática inversa

Ejecute las siguientes líneas de código Matlab:

- ¿Qué relación observa entre q y qinv? Justifíquelo.
- ¿Qué relación observa entre T1 y T2? Justifíquelo.

```
>> robot = load_robot('example', '3dofplanar')
>> q = [pi/4 pi/4 pi/4]
>> T = directkinematic(robot, q)
>> qinv = inversekinematic(robot, T)
>> T1 = directkinematic(robot, qinv(:,1))
>> T2 = directkinematic(robot, qinv(:,2))
>> drawrobot3d(robot, qinv(:,1))
>> drawrobot3d(robot, qinv(:,2))
```

## 4.6. Solución de la cinemática inversa para un robot de 6 GDL

Se presenta, en la Figura 4.2, un robot ABB IRB 140. Seguidamente, se indica en la Tabla 4.2 sus parámetros de DH. Estos parámetros están definidos en el fichero `parameters.m` que se encuentra en `arte/robots/ABB/IRB140`. Edite este fichero y observe las siguientes líneas:

```
robot.DH.theta= '[q(1) q(2)-pi/2 q(3) q(4) q(5) q(6)+pi]';
robot.DH.d='[0.352 0 0 0.380 0 0.065]';
robot.DH.a='[0.070 0.360 0 0 0 0]';
robot.DH.alpha= '[-pi/2 0 -pi/2 pi/2 -pi/2 0]';
robot.J=[];
robot.inversekinematic_fn = 'inversekinematic_irb140(robot, T)';
```

Nótese que no existe una solución cerrada y universal para la cinemática inversa de todos los brazos manipuladores. Así pues, es necesario proporcionar las ecuaciones específicas de este robot llamando internamente a la función `inversekinematic_irb140(robot, T)` que se encuentra en el directorio `arte/robots/ABB/IRB140`. La llamada a esta función se puede realizar a través de un “wrapper”. En este caso, la función `inversekinematic` se encarga de hacerlo. Las ecuaciones para resolver la cinemática inversa de este mecanismo se han estudiado durante las lecciones de teoría y, también, se encuentran detalladas en las hojas de problemas. El estudiante debería repasar estas ecuaciones para familiarizarse con el concepto de cinemática inversa. La llamada básica para calcular la cinemática inversa del robot es:

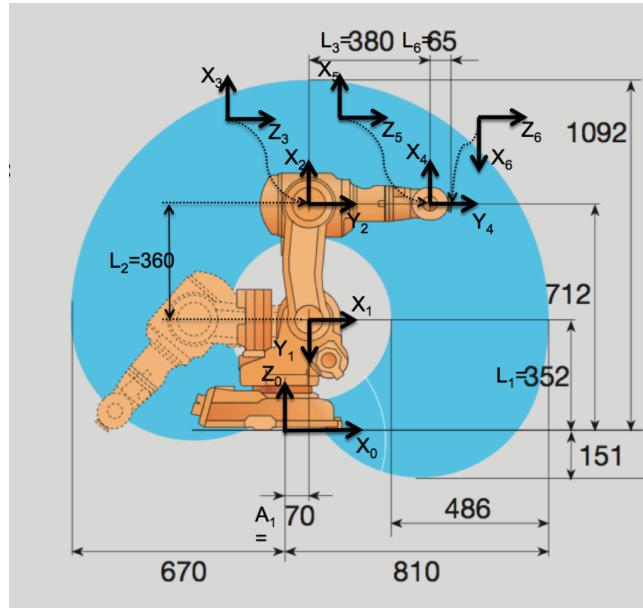


Figura 4.2: El robot IRB140 de ABB<sup>©</sup>

Transformación	$\theta$ (rad)	$d$ (m)	$a$ (m)	$\alpha$ (rad)
$0 \rightarrow 1$	$q_1$	0.352	0.070	$-\pi/2$
$1 \rightarrow 2$	$q_2 - \pi/2$	0	0.360	0
$2 \rightarrow 3$	$q_3$	0	0	$-\pi/2$
$3 \rightarrow 4$	$q_4$	0.380	0	$\pi/2$
$4 \rightarrow 5$	$q_5$	0	0	$-\pi/2$
$5 \rightarrow 6$	$q_6 + \pi$	0.065	0	0

Tabla 4.2: Parámetros DH del robot ABB IRB140.

```
>> init_lib
>> robot = load_robot('ABB', 'IRB140')
>> T = % Especifique una posición y orientación en T
>> qinv = inversekinematic(robot, T)
```

Se puede demostrar que, en un robot industrial de 6 GDL, dada una matriz de posición y orientación  $T$  existen 8 soluciones  $q_i$ , con  $i = 1, 2, 3, \dots, 8$ . Si la función  $T = f(q)$  hace referencia a la cinemática inversa del robot, entonces:

- Debe ocurrir que si  $q_i$  es una solución de la cinemática inversa, entonces  $T_i = f(q_i)$  y que  $T = T_i, \forall q_i$ .
- Además, debe ocurrir que  $q - q_i = \vec{0}$  para algún  $i$ .

### Ejercicio 4.6.1: Cinemática inversa

Realice las siguientes actividades:

- Defina una matriz  $T$  para el robot.
- Calcule los valores articulares de la cinemática inversa. ¿Cuántas soluciones existen en el caso más general?
- Compruebe, para cada una de las soluciones obtenidas de la cinemática inversa que, en efecto, el robot alcanza la misma posición y orientación definidas por  $T$ .
- Visualice todas las soluciones halladas por la librería:

```
>> init_lib
>> robot = load_robot('ABB', 'IRB140')
>> T = % Especifique una posición y orientación en T
>> qinv = inversekinematic(robot, T)
>> drawrobot3d(robot, qinv(:,1))
>> drawrobot3d(robot, qinv(:,2))
>> drawrobot3d(robot, qinv(:,3))
>> ...
>> drawrobot3d(robot, qinv(:,8))
```

- Entre estas soluciones, identifique aquellas que se denominaron “codo abajo”, “codo arriba”, “muñeca abajo” y “muñeca arriba”.

### Ejercicio 4.6.2: Cinemática inversa

Realice un script en Matlab que compruebe que todas las soluciones de la cinemática inversa para una posición y orientación dadas por  $T$  son válidas. Se proporciona, a continuación, un pequeño esquema de este código.

```
close all
n_solutions = 8;
q=[0.2 -0.2 0.3 0.1 0 0.1]
robot=load_robot('ABB', 'IRB140');
adjust_view(robot)
drawrobot3d(robot, q)
T = directkinematic(robot, q)
qinv = inversekinematic(robot, T)
for i=1:n_solutions
    Ti = directkinematic(robot, qinv(:,i))
    %¿T es igual a Ti para todo i?
end
for i=1:n_solutions
    e = q - qinv(:,i)
    %¿Hay alguna solución en qinv igual a q?
end
```

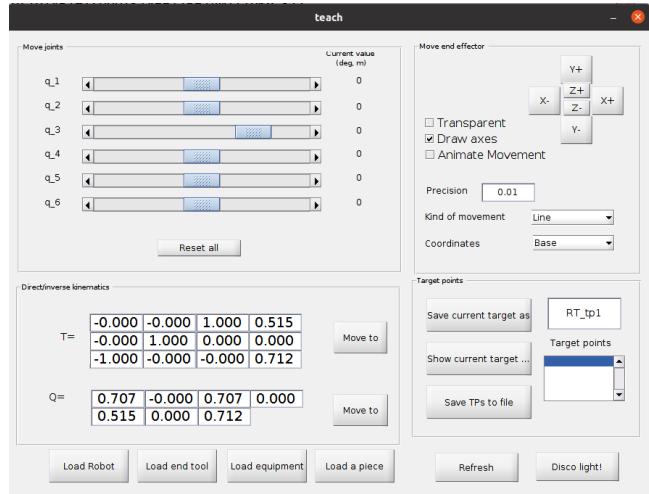


Figura 4.3: La aplicación GUI `teach` de la librería ARTE.

## 4.7. Uso de la aplicación `teach`

Se estudia, a continuación, el uso de la aplicación `teach` para el estudio de la cinemática inversa. Para ello, en la línea de comandos de Matlab, escriba:

```
>> init_lib
>> robot=load_robot('ABB', 'IRB140')
>> teach
```

La siguiente aplicación gráfica debería aparecer (Figura 4.3).

### Ejercicio 4.7.1: Cinemática inversa

La aplicación `teach` también considera que el robot se puede mover en términos de cinemática inversa.

- Modifique la posición y orientación especificadas en la matriz  $T$ . Presione el botón “Move to”. Al presionar sobre este botón, Matlab hallará las soluciones de la cinemática inversa y dibujará al robot en la primera de estas soluciones (arbitrariamente, utiliza la primera columna de todas las soluciones posibles).
- Utilice los botones X+, X-, Y+, Y-, Z+, Z-. Esto permite mover el extremo del robot de diferentes maneras. Por ejemplo:
  - Siguiendo líneas rectas en el coordenadas de la base (a orientación constante), a lo largo de los ejes  $X_0$ ,  $Y_0$  y  $Z_0$ . Para ello, seleccione “line” y “base”.
  - Siguiendo líneas rectas en el coordenadas del extremo (a orientación constante), a lo largo de los ejes  $X_6$ ,  $Y_6$  y  $Z_6$ . Para ello, seleccione “line” y “end effector”.
  - Realizando rotaciones sobre los ejes de la base  $X_0$ ,  $Y_0$  y  $Z_0$  (a posición constante). Para ello, seleccione “rotate” y “base”.
  - Realizando rotaciones sobre los ejes del extremo  $X_6$ ,  $Y_6$  y  $Z_6$  (a posición constante). Para ello, seleccione “rotate” y “end effector”.
  - El parámetro “precision” permite aumentar o disminuir el movimiento entre posiciones y orientaciones consecutivas.

## Capítulo 5

# Cinemática inversa y planificación de trayectorias en Coppelia

### 5.1. Introducción

La práctica plantea los siguientes **objetivos de aprendizaje**:

- Soluciones de la cinemática inversa en un simulador robótico completo: colisiones y autocolisiones.
- Cálculo de trayectorias en el espacio de trabajo del robot.
- Instrucciones de movimiento en robots industriales: `moveAbsJ`, `moveJ`, `moveL`.

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.
- Programación de trayectorias usando instrucciones de movimiento propias de lenguajes de programación de robots industriales.

### 5.2. Cinemática directa del ABB IRB140

#### 5.2.1. Manejo básico del robot IRB140

Pretendemos, en este apartado, mostrar los movimientos básicos del robot IRB 140 de ABB en simulación. Para esto, realice el ejercicio siguiente:

### Ejercicio 5.2.1: Coger piezas en cinemática directa

Realice las actividades siguientes:

- Abra la escena `scenes/irb140.ttt`.
- Abra el script de python `practicals/move_robot.py` en PyCharm. Puede ejecutar el script de python para mover el robot utilizando el teclado. Use las teclas 1-2-3-4-5-6 para incrementar cada articulación y q-w-e-r-t-y para decrementar su valor. Use las teclas “o” y “c” para abrir/cerrar la pinza del robot.
- Recoja una pieza de la cinta transportadora y déjela en la superficie de paletizado.

La actividad anterior busca que el estudiante se familiarice con el robot. Debe quedar patente que controlar el robot en **cinemática directa** es complejo y, por tanto, será necesario crear un script que automatice la tarea de recoger y dejar piezas.

### 5.2.2. El TCP de la herramienta

Es importante mencionar, aquí, el concepto de TCP (*Tool Center Point*). El TCP de un robot se define como la transformación relativa entre el sistema de referencia de su último eslabón y el sistema de referencia de su herramienta. Esta transformación se puede definir mediante una matriz de transformación homogénea  $T_{TCP}$ . Esta transformación es constante  $T_{tcp}$ , pues la pinza es solidaria al extremo del robot.

En un programa de control de un robot (por ejemplo, escrito en RAPID, Karel o python) estamos interesados en especificar la posición y orientación de la herramienta. Denominaremos esta posición y orientación como  $T_{target}$ . Así pues, en un robot de 6 GDL tendremos que:

$$T_{target} = {}^0A_1{}^1A_2{}^2A_3{}^3A_4{}^4A_5{}^5A_6T_{tcp} = {}^0A_6T_{tcp} \quad (5.1)$$

Obviamente, para la cinemática inversa, necesitaremos especificar la posición y orientación del sistema  $X_6Y_6Z_6$ . Por tanto, calculamos:

$${}^0A_6 = T_{target}T_{tcp}^{-1} \quad (5.2)$$

La función de cinemática inversa, calculará las coordenadas articulares que le permiten colocar su último eslabón en  ${}^0A_6$ . De esta manera, no es necesario implementar una función de cinemática inversa para cada pinza o efecto final que utilice el robot. Únicamente se precisa definir correctamente la transformación  $T_{tcp}$  cada vez que se equipe al robot con un nuevo elemento terminal.

### 5.2.3. Definición del TCP

Nos fijamos ahora en la Figura 5.1 para definir la transformación  $T_{tcp}$ . En este robot, definiremos un TCP para la pinza de dos dedos (RG2) y un TCP diferente cuando queramos trabajar con la ventosa de vacío.

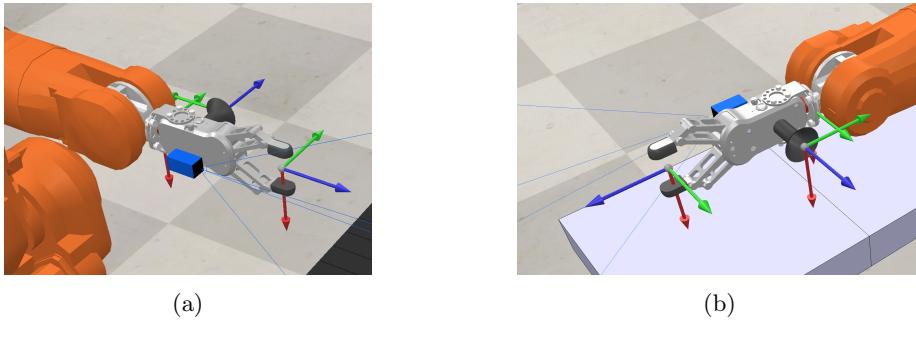


Figura 5.1: Dos vistas del extremo del robot donde se aprecian los TCP de la pinza y de la ventosa.

**Importante:** Las transformaciones  $T_{tcp}$  se definen en el sistema de referencia del extremo del robot. En nuestro caso, en coordenadas del sistema  $X_6Y_6Z_6$ . Para la pinza, tendremos:

$$T_{tcp} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.19 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Para la ventosa, tendremos:

$$T_{tcp} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0.065 \\ 0 & -1 & 0 & 0.11 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Esta transformación se puede definir usando un objeto de tipo `HomogeneousMatrix`. Para fijar un TCP, debemos hacer (como se indica ya en el script), si usamos la pinza RG2:

```
robot.set_TCP(HomogeneousMatrix(Vector([0, 0, 0.19]), RotationMatrix(np.eye(3))))
```

Si usamos la ventosa de vacío:

```
robot.set_TCP(HomogeneousMatrix(Vector([0, 0.065, 0.11]), Euler([-np.pi/2, 0, 0])))
```

### 5.3. Cinemática inversa del robot IRB140

Las ecuaciones para resolver la cinemática inversa de este robot, se han definido durante las sesiones de teoría. Para facilitar el desarrollo de la práctica, las ecuaciones se han programado y probado en pyARTE. Puedes revisarlas y comprobarlas en la clase del robot. La clase que define el robot está en: `robots/abbirb140.py` y la función se denomina `inversekinematics`

Debemos recordar que, durante las clases de teoría, se indicó que existían 8 soluciones diferentes para la cinemática inversa en posición y orientación. Sin embargo, en este apartado vamos a probar estas ecuaciones de la cinemática

inversa desde un punto de vista práctico y orientado a una aplicación. En concreto, dada una posición y orientación del extremo del robot, debemos tener en cuenta que:

- Existen posiciones y orientaciones  $T$  del extremo del robot en las que encontramos 8 soluciones válidas para la cinemática inversa del robot.
- Existen otras posiciones y orientaciones  $T$  del extremo del robot en las que encontramos menos de 8 soluciones válidas. En concreto, **observamos que hay menos de 8 columnas** en la solución de la cinemática inversa.
- Podemos encontrar el caso en el que no exista ninguna solución válida. Por ejemplo, si  $T$  está fuera del espacio de trabajo del robot.
- De entre las soluciones válidas, algunas pueden estar fuera del rango de las articulaciones. La solución puede ser, matemáticamente, correcta, pero el robot no puede alcanzarla por los límites físicos que imponga alguna de las articulaciones.
- De entre las soluciones válidas, no todas son alcanzables por el robot, pues en ocasiones, el robot colisiona con el entorno o consigo mismo (autocolisiones).
- Finalmente, si las articulaciones 4 y 6 tienen un rango extendido ( $\pm 400$  grados), entonces, existen muchas más soluciones posibles. Esta es la opción `extended=True` de la función.

Así, pues, hablaremos de:

- $n_m$ : número de soluciones matemáticamente válidas.
- $n_a$ : número de soluciones matemáticamente válidas y dentro del rango de las articulaciones.
- $n_c$ : número de soluciones matemáticamente válidas, dentro del rango de las articulaciones y que pueden ser alcanzadas en Coppelia (esto es, no colisionan con ningún objeto).

Obviamente, tenemos que:  $n_m \geq n_a \geq n_c$ .

#### Ejercicio 5.3.1: Soluciones válidas de la cinemática inversa

Abra y ejecute el fichero `practicals/kinematics/irb140_ikine.py`.

El script `irb140_ikine.py` presenta el siguiente código:

```
target_position = Vector([0.4, 0.0, 0.8])
target_orientation = Euler([0, np.pi / 2, 0])
q = inverse_kinematics(robot=robot,
                       target_position=target_position,
                       target_orientation=target_orientation)

view_all_solutions(robot, q)
q = filter_within_range(robot, q)
move_robot(robot, q)
```

La función `view_all_solutions` muestra por pantalla todas las soluciones soluciones que son matemáticamente correctas. Es necesario eliminar las soluciones que no están dentro del rango articular. Para ello se proporciona la función `filter_within_range`. Finalmente, la función `move_robot` mueve el robot en Coppelia. Es de utilidad, en este caso, la función `robot.check_joints()` que comprueba si todos los ángulos están dentro de los límites especificados por el fabricante del robot. En concreto, los límites articulares del robot IRB140 se especifican en la Tabla 5.1.

Al ejecutar el script `irb140_ikine.py`, veremos en la consola del programa:

```
TODAS LAS SOLUCIONES DE LA CINEMÁTICA INVERSA:
[[ 0.    0.    0.    0.    3.14   3.14   3.14   3.14]
 [-1.   -1.   1.93  1.93 -2.25 -2.25  0.32  0.32]
 [ 1.12  1.12 -4.26 -4.26  0.85  0.85 -3.99 -3.99]
 [-3.14  0.    0.   -3.14 -3.14  0.    -0.   3.14]
 [ 0.13 -0.13  2.33 -2.33  1.74 -1.74  0.54 -0.54]
 [ 3.14 -0.    0.   -3.14  0.   -3.14 -3.14  0.   ]]
HAY 8 SOLUCIONES VÁLIDAS MATEMÁTICAMENTE
*****
Solución: 0
Valores articulares q: [ 0.   -0.55  0.17 -0.   0.38  0. ]
Posición y orientación alcanzadas (T):
[[ 0.    0.    1.    0.3]
 [ 0.    1.    0.    0. ]
 [-1.   0.    0.    0.8]
 [ 0.    0.    0.    1. ]]
[...]
ELIMINANDO SOLUCIONES FUERA DE RANGO ARTICULAR:
SE HAN ENCONTRADO 8 SOLUCIONES EN RANGO ARTICULAR DEL TOTAL DE 8 SOLUCIONES VALIDAS
[[ 0.    0.    0.    0.    3.14   3.14   3.14   3.14]
 [-0.39 -0.39  1.01  1.01 -1.09 -1.09  0.04  0.04]
 [-0.21 -0.21 -2.93 -2.93 -0.48 -0.48 -2.66 -2.66]
 [-0.   3.14  0.   -3.14 -3.14  0.   -3.14  0.   ]
 [ 0.6  -0.6  1.92 -1.92  1.58 -1.58  0.52 -0.52]
 [ 0.   -3.14  0.   -3.14  0.   -3.14 -0.   3.14]]
COMANDANDO AL ROBOT A LAS SOLUCIONES
```

En este caso, la salida por pantalla indica que existen tantas soluciones matemáticamente válidas como columnas de la matriz  $q$ . El programa muestra que la posición y orientación alcanzadas ( $T$ ) son iguales. En este caso del ejemplo, tenemos:  $n_m = 8$ ,  $n_a = 8$  y  $n_c = 8$ .

#### Ejercicio 5.3.2: Soluciones válidas de la cinemática inversa

Utilice el script `irb140_ikine.py`. Realice las siguientes tareas:

- Modifique la posición `target_position=[0.5, 0, 0.9]`.
- Observe la información que se muestra por pantalla.
- Observe que, para cada columna  $q_i$ , **la solución  $q_i$  cambia, pero  $T$  es siempre la misma**.

$q_i$	Mín. ángulo (grados)	Máx. ángulo (grados)
1	-180	180
2	-90	110
3	-230	50
4	-200	200
5	-115	115
6	-400	400

Tabla 5.1: Límites articulares del robot IRB140.

#### Ejercicio 5.3.3: Soluciones en rango de la cinemática inversa

Utilice el script `practicals/kinematics/irb140_ikine.py`. Realice las siguientes tareas:

- Modifique la posición `target_position= [0.5, 0.0, 0.9]`.
- Observe la información que muestra el script. ¿Cuántas soluciones  $n_m$ ,  $n_a$  y  $n_c$ ?  $n_c$  son las soluciones a las que llega Coppelia sin colisionar.
- Cambie la posición `target_position= [0.4, 0.0, 0.5]` y cuente, de nuevo,  $n_m$ ,  $n_a$ ,  $n_c$ .

#### Ejercicio 5.3.4: Soluciones extendidas

Utilice el script `practicals/kinematics/irb140_ikine.py`. Realice las siguientes tareas:

- Modifique la opción `extended=False` a `extended=True` en la función `robot.inversekinematics`. Con esta opción, las articulaciones 4 y 6 tienen un rango de  $\pm 400$  grados y existen muchas más combinaciones de la cinemática inversa.
- Ejecute el script y observe estas soluciones.

Finalmente, el estudiante debe comprender que, si la solución es matemáticamente factible y, además, todas las articulaciones están dentro del rango, solamente entonces, se llama a la función:

```
robot.set_joint_target_positions(qi, precision=True)
```

Con esta función, el script de python conecta con Coppelia y le dice que el robot debe comandarse a esos valores articulaciones. La posición `qi`, normalmente, es posible alcanzarla, salvo el caso en el que el robot colisione con algún objeto del entorno. En este caso, se mostrará el mensaje:

```
ERROR, joint position could not be achieved, try increasing max_iterations
Errors (q)
[ 3.09944153e-06  1.43057484e-01 -8.44021828e-06  2.62260437e-06
 -6.24126065e-06 -9.53674317e-07]
Total error is: 0.14305748473552307
```

#### Ejercicio 5.3.5: Colisiones en Coppelia

Utilice el script `practicals/irb140_ikine.py`. Realice las siguientes tareas:

- Modifique la posición `target_position=[0.7, 0, 0]` y `target_orientation=[0, np.pi/2, 0]`.
- Observe al robot colisionar con el entorno. No se alcanza las soluciones comandadas.

## 5.4. Generación de trayectorias en el espacio de trabajo del robot

En las prácticas anteriores, nos hemos dado cuenta de que, al comandar al robot desde una posición articular  $q_a$  a una posición articular  $q_b$ , el **extremo del robot no sigue una trayectoria recta** en el espacio de trabajo.

Este tipo de trayectorias son **muy frecuentes** en aplicaciones robóticas. Las trayectorias rectas las podemos encontrar, por ejemplo en:

- La trayectoria de aproximación del robot para coger una pieza.
- Trayectorias para realizar soldaduras en planchas de metal.
- Cuando el robot debe realizar cortes en piel, metal,... etc.

Las trayectorias rectas están implementadas en el lenguaje de programación de los robots industriales. Por ejemplo, en el lenguaje RAPID<sup>©</sup> de ABB se incluye la instrucción `MoveL` que planifica una trayectoria recta desde la posición y orientación actuales del robot hasta la posición y orientación definidas en la instrucción.

Durante esta práctica, vamos a desarrollar un algoritmo que nos va a permitir:

- Llevar el extremo del robot desde una posición y orientación dadas por  $T_A$  hasta una posición y orientación  $T_B$ . La orientación en el punto  $A$  y  $B$  será la misma. Esta trayectoria la denominamos: “trayectoria recta a orientación constante”.
- Llevar el extremo del robot desde una posición y orientación dadas por  $T_A$  hasta una posición y orientación  $T_B$ . La posición en el punto  $A$  y  $B$  será la misma. Esta trayectoria la denominamos: “trayectoria posición constante”. En este caso, se busca una transición suave en la orientación del extremo del robot, que se mantiene en la misma posición.
- El tercer caso es una combinación de los dos movimientos anteriores. En este caso más general, se debe realizar una interpolación entre las posiciones y orientaciones indicadas en  $A$  y  $B$ .

En una primera parte de la práctica, nos dedicaremos a generar las trayectorias en el espacio de trabajo del robot y observarlas en Coppelia Sim. En una segunda parte de esta práctica, veremos de qué manera podemos hacer que el extremo del robot siga estas trayectorias.

### 5.4.1. Interpolación de posiciones y orientaciones en el espacio de la tarea

En este apartado damos un marco teórico para la generación de posiciones y orientaciones en el espacio de la tarea. Supongamos, inicialmente, que deseamos generar un conjunto de posiciones entre un punto inicial  $\vec{p}_a$  y  $\vec{p}_b$  expresados en coordenadas de la base del robot. Comenzamos considerando que la velocidad lineal a la que se mueve el extremo del robot es  $\vec{v}_e$  con  $|\vec{v}_e| = v_{max}$ . Así pues, 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  $\delta t$  (s). En particular,  $\delta t = 0.05(s)$  es el intervalo estándar de simulación en Coppelia. En consecuencia, debemos planificar  $n$  puntos sobre la recta:

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

Si redondeamos al siguiente entero:

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

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

$$\vec{p}_i = (1 - t) \cdot \vec{p}_a + t \cdot \vec{p}_b \quad (5.3)$$

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

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

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**:

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

Pues la matriz resultado debe ser orthonormal. Y la suma de dos matrices orto-normales no produce una matriz ortonormal.

Una posible forma de realizar esta interpolación entre dos orientaciones diferentes se puede realizar con los ángulos de Euler. Así, suponga que la orientación en el inicio de la trayectoria está dada por los ángulos de Euler  $e_a = (\alpha_a, \beta_a, \gamma_a)$  y  $e_b = (\alpha_b, \beta_b, \gamma_b)$  representa la orientación al final de la trayectoria. Entonces, una orientación intermedia entre las dos puede ser generada a través de la fracción  $t$  como:

$$e_i = (1 - t) \cdot e_a + t \cdot e_b \quad (5.4)$$

En este caso, un ángulo de Euler intermedio  $e_i$  sí tiene sentido y representa una orientación válida.

### 5.4.2. Generación de trayectorias rectas a orientación constante

Nos centramos, ahora, en formas de generar una trayectoria recta en el espacio de trabajo del robot. Es decir, debemos entender que se generarán un conjunto de posiciones sobre una recta. La orientación del extremo será la misma en el inicio y el final de la trayectoria. El código siguiente os puede ayudar a realizar esta tarea:

```
def n_movements_pos(pA, pB):
    vmax = 0.3 # m/s
    delta_time = 0.05 # 50 ms
    total_time = np.linalg.norm(np.array(pB) - np.array(pA)) / vmax
    n = total_time / delta_time
    n = np.ceil(n)
    return int(n)

def n_movements_orient(eA, eB):
    wmax = 1 # rad/s
    delta_time = 0.05 # 50 ms
    total_time = np.linalg.norm(np.array(eB) - np.array(eA)) / wmax
    n = total_time / delta_time
    n = np.ceil(n)
    return int(n)

def path_planning_line(pA, oA, pB, oB):
    pA = pA.array
    pB = pB.array
    oA = oA.abg
    oB = oB.abg
    n1 = n_movements_pos(pB, pA)
    n2 = n_movements_orient(oB, oA)
    n = max(n1, n2)
    t = np.linspace(0, 1, n)
    positions = []
    orientations = []
    #
    # ;COMPLETE EL CÓDIGO!
    #
    return positions, orientations
```

Nótese que **n1** se calcula en base a la distancia euclídea entre  $\vec{p}_a$  y  $\vec{p}_b$ . El valor **n2** se aproxima en base a la distancia euclídea en el espacio de los ángulos de Euler.

#### Ejercicio 5.4.1: Rectas a orientación constante

Abra el script `practicals/path_planning/path_planning.py`. Realice las siguientes tareas:

- Termine el código de la función `path_planning_line`. Deberá generar **posiciones** sobre la trayectoria usando la Ecuación (5.3). La orientación sobre todos los puntos de la trayectoria será igual a **oA** (orientación inicial).
- Pruebe el script y observe cómo se mueve el sistema de referencia en Coppelia (a orientación constante). Varíe los puntos de inicio y de final de la trayectoria.

#### Ejercicio 5.4.2: Solución completa

Abra el script `practicals/path_planning/path_planning.py`. Realice las siguientes tareas:

- Termine el código de la función `path_planning_line`. Deberá interpolar los puntos **y las orientaciones** de acuerdo con las Ecuaciones (5.3) y (5.4).
- Pruebe el script y observe cómo se mueve el sistema de referencia en Coppelia. Varíe los puntos de inicio y de final de la trayectoria. Varíe la orientación inicial y final.
- Realice una trayectoria a posición constante y orientación variable.

#### 5.4.3. Seguimiento de trayectorias por el robot

Suponga que, en un determinado momento, el robot está en una posición articular  $q_a$ . La posición y orientación del extremo se obtiene con la función `directkinematics`:

```
Ta = robot.directkinematic(qa)
```

Pensemos, ahora, que deseamos que el robot siga una trayectoria recta desde  $T_a$  hasta  $T_b$ , según se muestra en la Figura 5.2. Llamaremos  $T_i$  a la siguiente posición y orientación en la trayectoria.

Ahora considere que, a cada punto de la trayectoria especificada por  $T_i$  aplicamos nuestro conocido algoritmo de cinemática inversa. En general, dado  $T_i$  obtendremos 8 soluciones de la cinemática inversa que llamaremos  $q_i = \{q_i^1, q_i^2, q_i^3, \dots, q_i^8\}$ . Para el siguiente punto de la trayectoria,  $T_{i+1}$ , obtendremos otras 8 soluciones diferentes  $q_{i+1} = \{q_{i+1}^1, q_{i+1}^2, q_{i+1}^3, \dots, q_{i+1}^8\}$ . Esta idea se muestra en la Figura 5.3.

Finalmente, **para calcular una trayectoria recta en el espacio**, se calculan un conjunto de posiciones articulares  $q_i$  de manera que:

- Ninguna articulación sobrepase sus valores articulares máximos.
- Suponga que el robot parte de  $q_a$ . Suponga que en siguiente punto de la trayectoria  $q_i$  ha obtenido 8 soluciones  $q_i = \{q_i^1, q_i^2, q_i^3, \dots, q_i^8\}$ . Se elige la solución en  $i$  de manera que la distancia  $|q_a - q_i^j|_2$  sea mínima.
- Se repite el proceso para el resto de puntos de la trayectoria.

Todo este proceso de cálculo de trayectorias y seguimiento de trayectorias lo realiza la función `moveL` de la librería. A continuación, se presentan el resto de instrucciones de movimiento típicas en un lenguaje de programación de robots.

#### 5.4.4. Instrucciones de movimiento en un robot

La mayoría de lenguajes de programación de robots cuentan con instrucciones de movimiento que comandan al robot con el objetivo de realizar tareas. Con pequeñas diferencias, todos los lenguajes de robots (p. e. RAPID, Karel) cuentan con las siguientes instrucciones de movimiento:

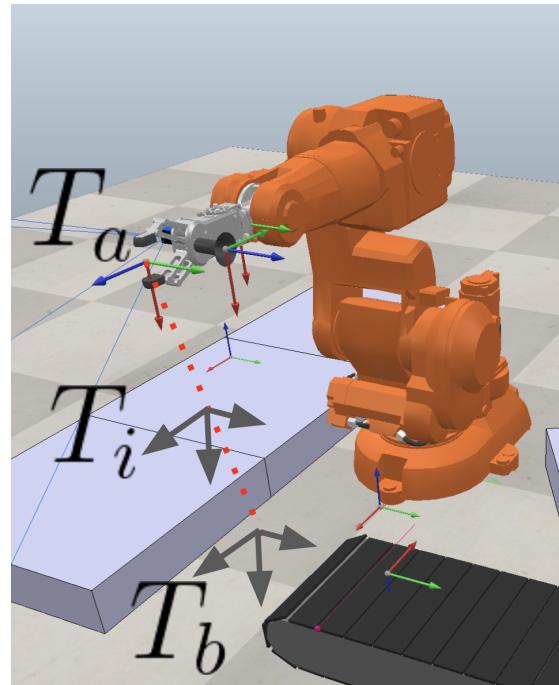


Figura 5.2: Una trayectoria recta con posiciones y orientaciones sobre ella.

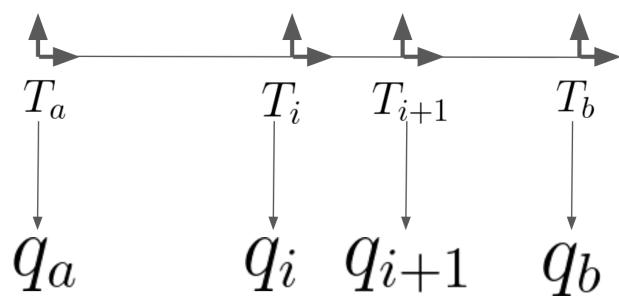


Figura 5.3: Una trayectoria recta con posiciones y orientaciones sobre ella. Se muestran, además, las soluciones de la cinemática inversa para cada posición y orientación  $T_i$ .

- **MoveAbsJ**: suponiendo que el robot se encuentra en la posición articular  $q_a$ , el robot debe dirigirse a la posición articular  $q_b$  especificada en la instrucción. Se realiza una planificación suave entre  $q_a$  y  $q_b$ . Se sigue un camino mínimo entre  $q_a$  y  $q_b$ .
- **MoveJ**: suponiendo que el robot se encuentra en la posición articular  $q_a$ , el robot debe dirigirse a la posición articular  $q_b$  especificada por un *target point*. Es decir,  $q_b$  es una solución de la cinemática inversa del *target point*. Se realiza una planificación suave entre  $q_a$  y  $q_b$ .
- **MoveL**: suponiendo que el robot se encuentra en la posición articular  $q_a$ , el robot debe dirigirse a una posición y orientación que se especifican en la instrucción y el extremo del robot debe seguir una línea recta en el espacio. Se realiza una interpolación entre la orientación del extremo en el punto  $a$  y el  $b$ . El algoritmo, generalmente, implementa una solución similar a la mostrada en el Apartado 5.4.2.
- **MoveC**: suponiendo que el robot se encuentra en la posición articular  $q_a$ , el robot debe dirigirse a una posición y orientación que se especifican en la instrucción. El extremo del robot debe seguir una trayectoria circular. Se realiza una interpolación entre la orientación del extremo en el punto  $a$  y el  $b$  que pase por el punto  $c$ . Esta instrucción no se encuentra implementada en pyARTE, pues no resulta una trayectoria habitual en aplicaciones industriales.

La diferencia entre las instrucciones **MoveAbsJ** y **MoveJ** es:

- En **MoveAbsJ** se especifican los valores articulares  $q_b$ .
- En **MoveJ** se especifica una posición y orientación  $T$ . Generalmente, se le dice al robot que elija la solución  $q_b$  que esté más cerca de  $q_0$ .

#### 5.4.5. Programas de ejemplo

En el fichero `path_planning140_traj1.py` (trayectoria 1) se observa el siguiente código.

```
if __name__ == "__main__":
    simulation = Simulation()
    clientID = simulation.start()
    robot = RobotABBIRB140(clientID=clientID)
    robot.start()
    robot.set_TCP(HomogeneousMatrix(Vector([0, 0, 0.19]), RotationMatrix(np.eye(3)))

    q0 = np.array([0, 0, 0, 0, -np.pi / 2, 0])
    tp1 = Vector([0.6, -0.5, 0.8])
    to1 = Euler([0, np.pi / 2, 0])
    tp2 = Vector([0.6, 0.5, 0.5])
    to2 = Euler([0, np.pi / 2, 0])

    robot.moveAbsJ(q0, precision=True)
    robot.moveJ(target_position=tp1, target_orientation=to1, precision=True)
    robot.moveL(target_position=tp2, target_orientation=to2, precision=False)
    robot.moveL(target_position=tp1, target_orientation=to1, precision=False)
    robot.moveAbsJ(q0, precision=True)
```

```
# Stop arm and simulation
simulation.stop()
robot.plot_trajectories()
```

En el código, se observa:

- El robot se dirige a una posición inicial especificada por `q0` con `MoveAbsJ`.
- Despues, se debe dirigir a un *target point* especificado por `tp1` y `to1` usando una instrucción `moveJ`.
- Despues, se debe dirigir a otro *target point* especificado por `tp2` y `to2` usando una instrucción `moveL`.
- Vuelta a `tp1` y `to1`.
- Y se comanda, otra vez a `q0`.

**Nota:** es muy importante que se tenga en cuenta que en las instrucciones de movimiento se hace referencia al siguiente destino que debe alcanzar el robot: el estado actual será el resultado de la instrucción anterior. Así pues, todo programa de robot debe tener una instrucción de inicio de tipo `moveAbsJ` para asegurar una posición inicial conocida!

#### Ejercicio 5.4.3: Trayectoria 1

Edite los *target points* del fichero `irb140-traj1.py`. El robot deberá seguir la trayectoria de la Figura 5.4. Para resolver el ejercicio, deberá:

- Elegir un valor de `q0` adecuado.
- Escribir los puntos de destino adecuados.

En la Figura 5.4 los puntos que definen la trayectoria del TCP del robot. Especificamos, a continuación, los puntos de interés de esta trayectoria.

$$\begin{aligned}\vec{p}_a &= (0.45, 0, 0.967)(m) & \vec{p}_b &= (0.6, -0.5, 0.8)(m) \\ \vec{p}_c &= (0.6, -0.5, 0.3)(m) & \vec{p}_d &= (0.6, 0.5, 0.3)(m) \\ \vec{p}_e &= (0.6, 0.5, 0.8)(m) & \vec{p}_f &= (0.6, 0, 0.8)(m)\end{aligned}$$

La orientación se especifica mediante matrices de orientación. La orientación en *a* y en el resto de la trayectoria es:

$$\begin{aligned}R_a &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ R_b = R_c = R_d = R_e = R_f &= \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{pmatrix}\end{aligned}$$

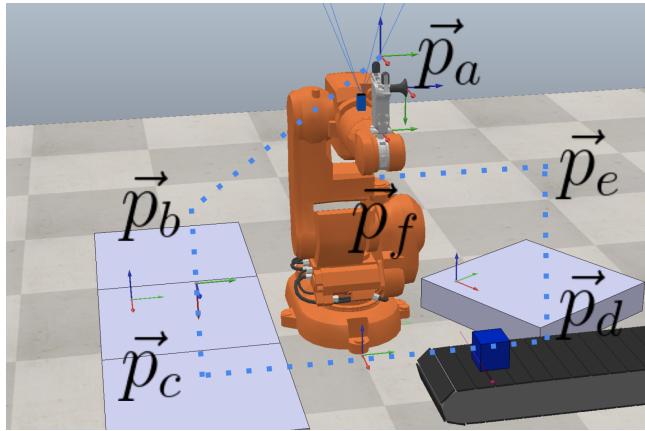


Figura 5.4: La trayectoria 1 del ejercicio. Varias trayectorias rectas con indicación de los puntos de interés sobre ellas.

Ejecute el script para ver el seguimiento que hace el robot sobre la trayectoria.

#### Ejercicio 5.4.4: Trayectoria 2

Edite los *target points* del fichero `irb140_traj2.py`. El robot deberá seguir la trayectoria de la Figura 5.5. Para resolver el ejercicio, deberá:

- Elegir un valor de `q0` adecuado.
- Escribir las variables `target_positions` y `target_orientations`.

En este último caso, las variables `target_positions` y `target_orientations` son listas, de tal manera que se puede comandar al robot haciendo un bucle de tipo `for`.

```
robot.moveAbsJ(q0, precision=True)
for i in range(len(target_positions)):
    robot.moveL(target_position=target_positions[i],
                target_orientation=target_orientations[i],
                precision=False)
```

Los puntos de la trayectoria son:

$$\begin{aligned} \vec{p}_a &= (-0.5, -0.5, 0.3)(m) & \vec{p}_b &= (0.5, -0.5, 0.3)(m) \\ \vec{p}_c &= (0.5, 0.5, 0.3)(m) & \vec{p}_d &= (-0.5, 0.5, 0.3)(m) \\ R_a = R_b &= R_c = R_d = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \end{aligned}$$

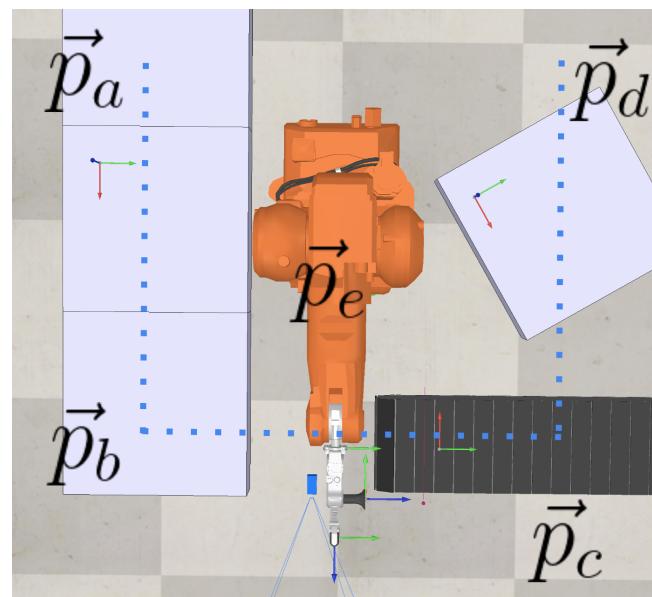


Figura 5.5: La trayectoria 1 del ejercicio. Varias trayectorias rectas con indicación de los puntos de interés sobre ellas.



# Capítulo 6

## Una aplicación de paletizado con Coppelia

### 6.1. 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 6.1.

### 6.2. 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.

### 6.3. Material proporcionado

Para la práctica se proporciona al estudiante el siguiente material:

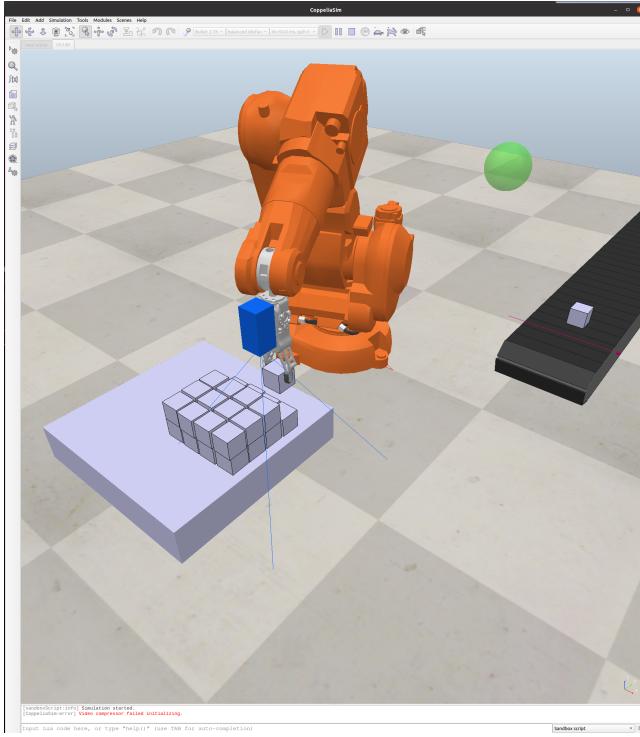


Figura 6.1: Una aplicación de paletizado con Coppelia.

- 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`.

## 6.4. 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`.

## 6.5. Transformaciones

Si observamos la Figura 6.2, 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 6.2 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:

$$T_{target} = {}^0T_m {}^mT_p \quad (6.1)$$

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

$${}^0A_6T_{tcp} = T_{target} = {}^0T_m {}^mT_p \quad (6.2)$$

**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:

$${}^0A_6 = T_{target}T_{tcp}^{-1} \quad (6.3)$$

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  $T_{target} = {}^0T_m {}^mT_p$ .

## 6.6. 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  $\epsilon$  (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 6.3 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 6.1: 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  $Y$ .

## 6.7. 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`.

### 6.7.1. 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.

### 6.7.2. 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 6.7.1: Solución inicial

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

#### 6.7.3. 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).

#### 6.7.4. 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 (6.1).

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 6.7.2: 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  $2 \times 3 \times 2$ ,  $4 \times 2 \times 2$ .
- 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  $2 \times 2$  encuentre la mayor altura que puede paletizar.

### Ejercicio 6.7.3: 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  $T_{target} = {}^0T_m {}^mT_p$ .

## 6.8. Selección de la herramienta

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 6.4). 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.

## 6.9. 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.

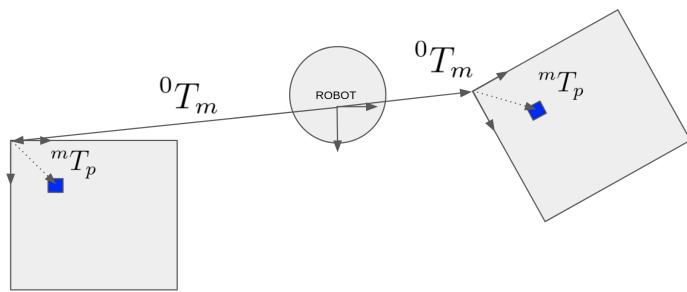
#### Ejercicio 6.9.1: Ventosas

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

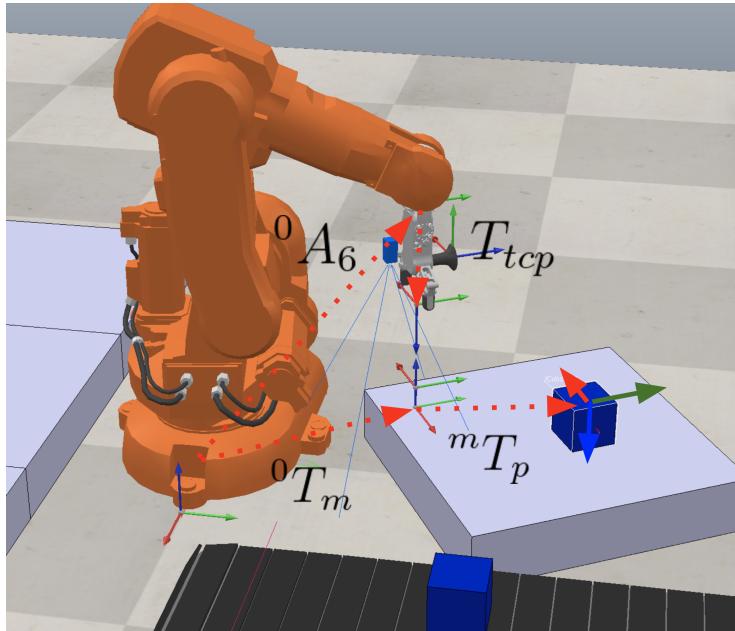
- Será necesario que cambie el TCP de la herramienta. Deberá modificar ligeramente, la posición y orientación de los target points en la función `pick` y `place`.
- Reduzca ahora la variable `piece_gap` para maximizar el rendimiento volumétrico del pallet.

## 6.10. 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 6.5 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 6.2: Dos vistas de las transformaciones de interés en la aplicación de paletizado.

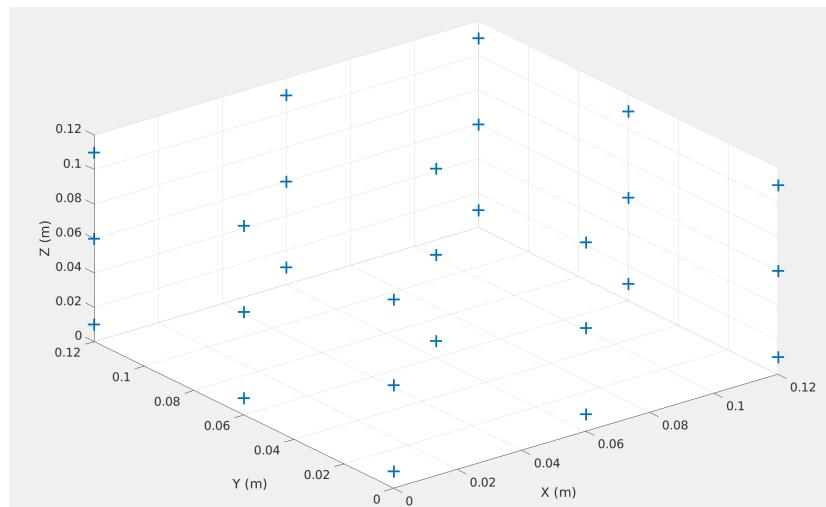


Figura 6.3: Posiciones ( $x, y, z$ ) para el paletizado de 27 piezas en un arreglo de 3x3.

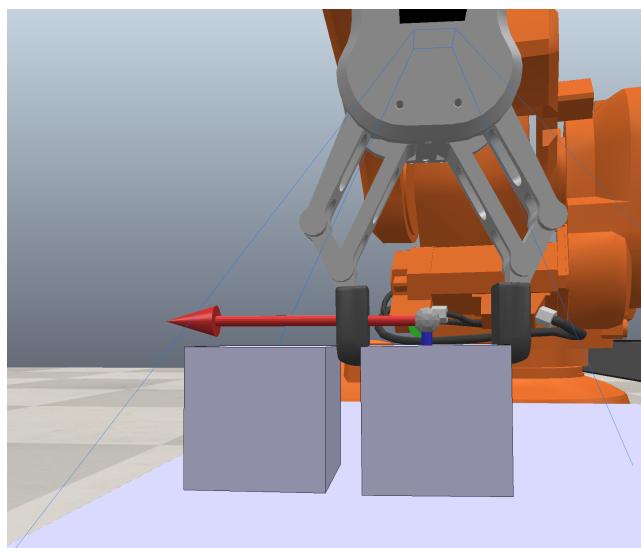


Figura 6.4: Pinza del robot colisionando con piezas.

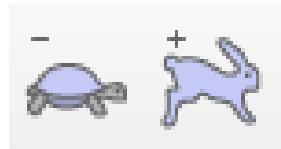


Figura 6.5: Aumenta o decelara la velocidad de simulación de Coppelia.



## **Parte II**

# **Prácticas avanzadas**



## Capítulo 7

# Cinemática inversa y la Jacobiana del manipulador

### 7.1. Objetivos

En esta sesión práctica, el estudiante se enfrentará a las soluciones cinemáticas para:

- Robots de muñeca no esférica.
- Robots redundantes.

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

- Utilizar un método basado en la Jacobiana para obtener soluciones a la cinemática inversa de robots manipuladores.
- Resolver las ecuaciones de la cinemática inversa de cualquier manipulador de tipo serie incluyendo mecanismos redundantes y muñecas no esféricas.

### 7.2. Primeros pasos

Analizamos en esta sección inicial los parámetros del robot UR10. El robot UR10, fabricado por Universal Robots tiene una estructura de robot antropomórfico, terminado con una muñeca no esférica. Añada los parámetros de DH del robot en [arte/robots/practicals/UR10/parameters.m](#):

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

A continuación cargue el robot y dibújelo. El resultado se observa en la Figura 7.1:

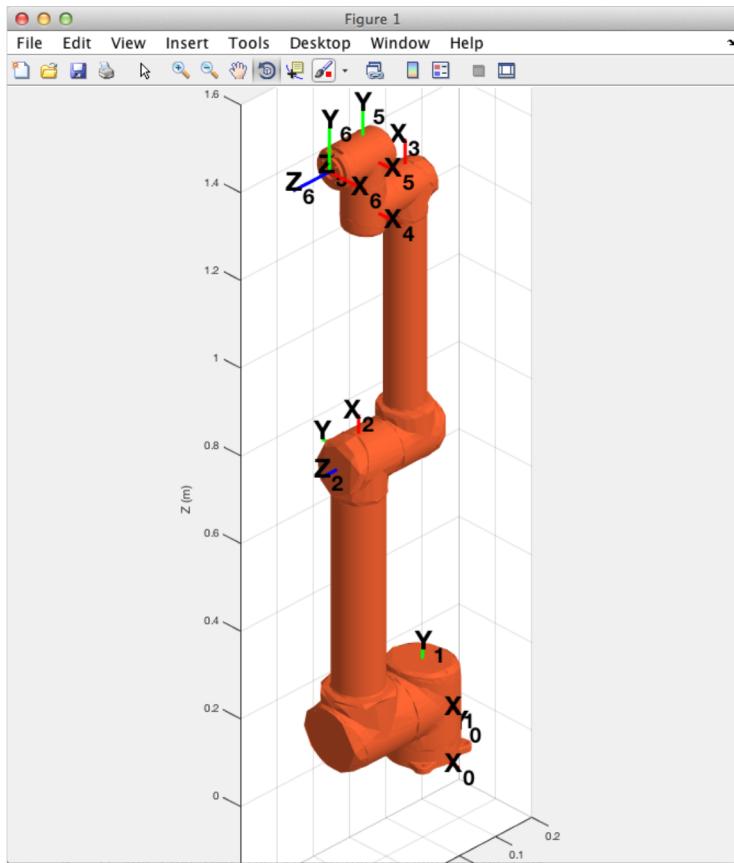


Figura 7.1: El robot UR10.

```
>> pwd
ans =
'/Users/arturogilaparicio/Desktop/arte'
>> init_lib
>> robot=load_robot('practicals','UR10')
>> T=directkinematic(robot, [0 0 0 0 0 0])
>> drawrobot3d(robot, [0 0 0 0 0 0])
```

Seguidamente, puede utilizar el comando teach para mover el robot utilizando los sliders:

```
>> teach
```

### 7.3. Cinemática inversa

Hasta el momento hemos resuelto la cinemática inversa de robots con muñeca esférica. La posición de la muñeca se define como el punto de corte de los ejes asociados a las últimas coordenadas articulares. En un robot industrial de 6GDL, tendremos  $\vec{q} = (q_1, q_2, q_3, q_4, q_5, q_6)$ . Las últimas tres coordenadas articulares ( $q_4, q_5, q_6$ ) tienen asociados los vectores de DH ( $z_3, z_4, z_5$ ). Si estos tres vectores

se cruzan en un punto (la muñeca,  $p_m$ ), entonces la posición de esta muñeca permanecerá invariante ante cambios de  $(q_4, q_5, q_6)$ . En estos robots, generalmente, somos capaces de calcular la posición  $p_m$  de la muñeca como:

$$p_m = p - L_6 * z_6$$

donde  $p$  es la posición cartesiana del extremo efector en coordenadas de la base y  $z_6 = z_5$  define la orientación del extremo. Este hecho simplifica enormemente el cálculo de la cinemática inversa, pues la posición  $p_m$  depende únicamente de  $\vec{q} = (q_1, q_2, q_3)$  y los últimos tres giros se pueden calcular para fijar la orientación del extremo.

El robot UR10, cuenta con una muñeca **no esférica**. Se debe observar que los vectores  $(z_3, z_4, z_5)$  no se cortan en un único punto. Como consecuencia, la posición y orientación del extremo están acopladas. En este tipo de robots, se emplea generalmente una solución de la cinemática inversa basada en la Jacobiana del manipulador.

El algoritmo iterativo considera lo siguiente como datos de partida:

- Que en el instante  $i$  el extremo del robot se encuentra en la posición actual  $\vec{p}_i = (p_x, p_y, p_z)_i$  y se desea ir a la posición final  $\vec{p}_f = (p_x, p_y, p_z)_f$ .
- Que en el instante  $i$  el extremo del robot tiene la orientación dada por el cuaternión  $Q_i = (q_w, q_x, q_y, q_z)_i$  y se desea llevar al robot a la orientación final  $Q_f = (q_w, q_x, q_y, q_z)_f$

Recordemos que, en cualquier instante, la Jacobiana del Manipulador relaciona las velocidades articulares y la velocidad lineal/angular del extremo como:

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

donde  $\dot{q}$  son las velocidades articulares. Por otra parte,  $\vec{v}$  y  $\vec{w}$  son las velocidad lineal y angular del extremo (en coordenadas de la base).

Si la matriz  $J$  es cuadrada e invertible, entonces, podemos hacer:

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

Esto permite calcular las velocidades articulares del robot de manera que las velocidades del extremo sean las deseadas. Considere que, en cualquier instante de tiempo existe una función que permite calcular la velocidad lineal y angular  $(\vec{v}, \vec{w})$  del extremo que permite acercar al robot hacia la solución deseada en posición y orientación. En este caso, el algoritmo se resume en los siguientes pasos:

- Comprobar si el robot ha alcanzado la posición y orientación deseadas. Si el error en posición y orientación está por debajo de un factor de tolerancia, entonces finaliza el algoritmo.
- Calcular las velocidad lineal y angular que acercan la  $\vec{p}_i$  y  $Q_i$  hacia  $\vec{p}_f$  y  $Q_f$ . En el caso de la velocidad lineal, es:

$$\vec{v} = \vec{p}_f - \vec{p}_i$$

- Calcular la velocidad articular como función de  $\vec{v}$  y  $\vec{w}$ .

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

- Integrar de forma aproximada las coordenadas articulares. La aproximación más sencilla que se puede hacer para esta integración es:

$$q_{i+1} = q_i + \dot{q}\Delta t$$

donde  $\Delta t$  es un intervalo de tiempo de integración. En código Matlab, esta integración es:

```
q = q + step_time*qd
```

donde  $qd$  son las velocidades articulares y `step_time` es el “paso” del algoritmo.

## 7.4. Experimentando con el algoritmo

Se proporciona al alumno una solución de la cinemática inversa basada en la Jacobiana del manipulador. Esta solución se encuentra en:

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

Además, se proporciona al alumno un script para probar esta cinemática inversa, en:

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

Con las siguientes actividades se busca que el estudiante se familiarice con este tipo de algoritmos y entienda sus principales fortalezas y debilidades. Comenzamos describiendo ambos scripts:

- `test_kinematics_ur10.m`: En este script se calcula una matriz  $T$  a partir de una posición articular  $q$  arbitraria:

```
T = directkinematic(robot, q)
```

Seguidamente, en la línea:

```
qinv = inversekinematic(robot, T, q0)
```

Se llama a la cinemática inversa. Nótese que se parte de una semilla para el algoritmo  $q0$  que se debe entender como la posición actual del robot. Finalmente se comprueba si la solución hallada por la cinemática inversa `qinv` consigue llegar a la posición y orientación indicadas por  $T$ . Para facilitar la comparación, se realiza la resta  $T-T_reach$

```
T_reach = directkinematic(robot, qinv)
```

#### Ejercicio 7.4.1: Error máximo

Encuentre Lea los ficheros de parámetros del robot que se encuentran en:

- `arte/robots/example/3dofplanar/parameters.m`: parámetros del robot.
- `arte/robots/example/3dofplanar/inversekinematic_3dofplanar.m`: ecuaciones de la cinemática inversa del robot.

Deberá comprender los parámetros cinemáticos del fichero `parameters.m` y las ecuaciones de la cinemática inversa.



## Capítulo 8

# Aplicaciones industriales

### 8.1. 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.

### 8.2. 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 8.1). 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).

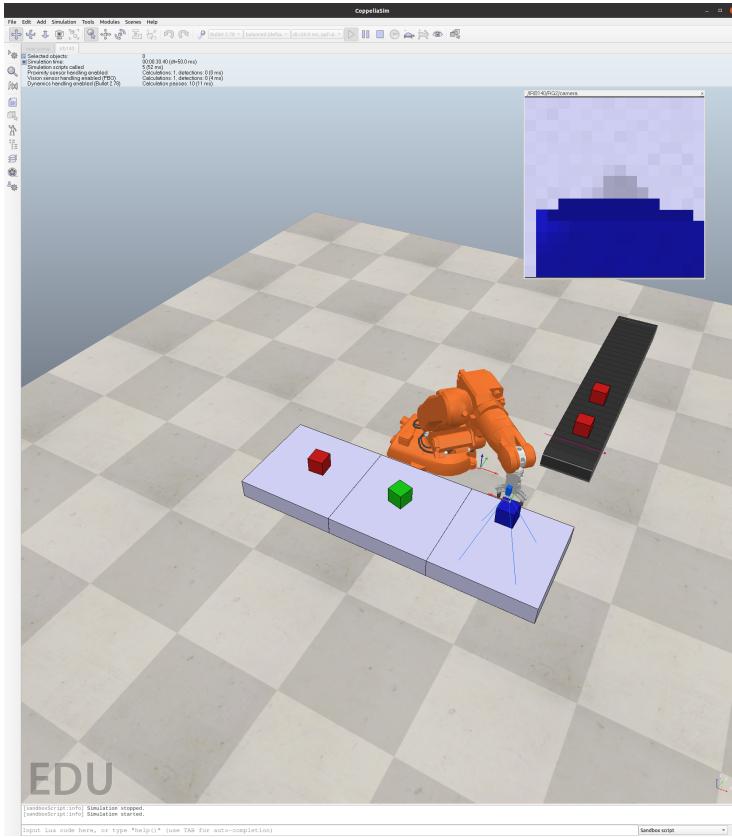


Figura 8.1: Una aplicación de clasificación en base al color.

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.

### 8.3. Pintura

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

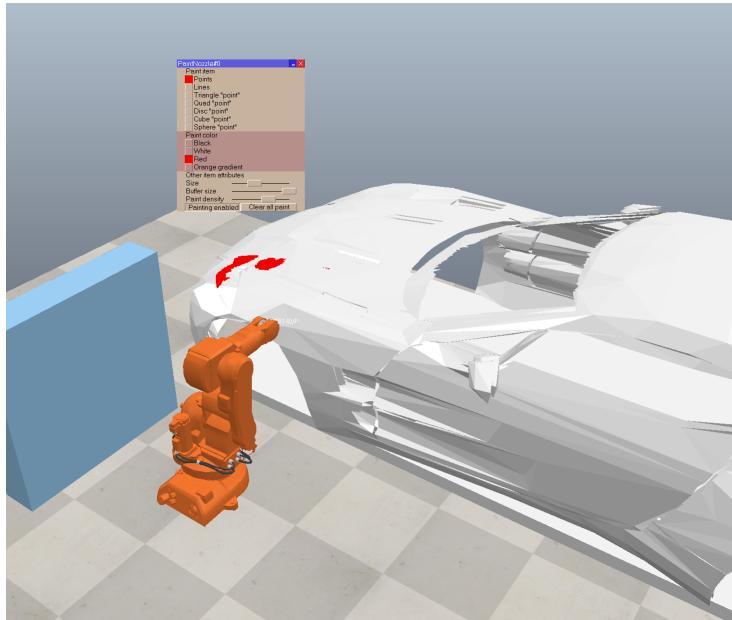


Figura 8.2: Un IRB140 en una aplicación de pintura.

- Escena: `irb140_paint_application.ttt`. Se proporciona una escena que incluye un robot IRB140 y una carrocería de un vehículo (Figura 8.2). 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.

## 8.4. Soldadura

Se propone la simulación de una aplicación de soldadura (Figura 8.3). 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.

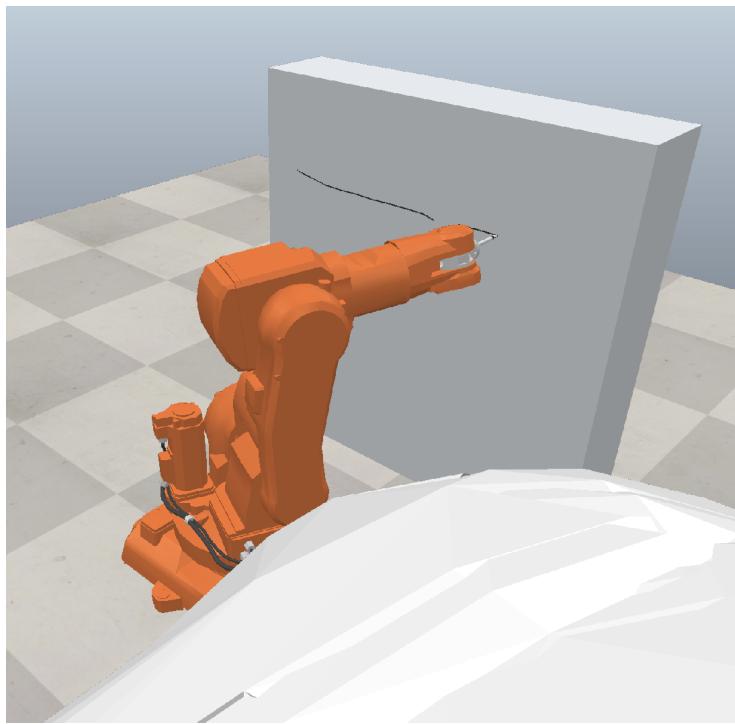


Figura 8.3: Un IRB140 en una aplicación de soldadura TIG/MIG.

- 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.

## **Parte III**

# **Proyecto transversal**



# Capítulo 9

## Proyecto evaluable

### 9.1. Introducción

Se propone a los estudiantes que realicen un proyecto basado en la librería pyARTE y en las prácticas realizadas. El resultado de este trabajo constituirá parte de la evaluación de la asignatura, debiendo ser un proyecto ser original e inédito.

### 9.2. Entrega del proyecto y condiciones

El proyecto realizado se entrega para la evaluación de la parte práctica de la asignatura y, por ende, forma parte de un porcentaje de la nota final de la asignatura. Se consideran las siguientes condiciones para este proyecto transversal:

- Los trabajos se realizarán por equipos de dos estudiantes y deberán ser enteramente originales e inéditos.
- En los trabajos se valora el resultado final y la calidad del código desarrollado por los estudiantes.

Para la entrega del proyecto, se deberá adjuntar la siguiente documentación:

- Un script de Python 3.8/3.9/3.10 funcional que utilice la librería pyARTE (código proporcionado durante las prácticas) y que resuelva el problema planteado. El repositorio original lo encontraréis en:  
<https://github.com/4rtur1t0/pyARTE.git>
- El código anterior deberá funcionar con la escena proporcionada durante la práctica.
- En caso de utilizar algún repositorio o librería diferente de las vistas durante las prácticas, se mencionará esta librería y referenciará convenientemente.
- Un **vídeo** grabando la simulación de Coppelia (botón grabar en Coppelia).

#### Criterios de evaluación:

- El robot es capaz de recoger las piezas con precisión usando la ventosa.

- El robot es capaz de dejar las piezas en posiciones y orientaciones precisas para el paletizado.
- El paletizado se realiza alineado con los ejes principales de los sistemas de referencia ubicados sobre los palés.
- Se procesan las piezas indicadas.
- El código está bien estructurado y es legible.
- El código permite reducir el tiempo entre piezas, p.e. WAIT\_TIME=10.

### 9.3. Descripción del proyecto 1

#### **Material proporcionado:**

- Se proporciona la siguiente escena de Coppelia: `scenes/more/irb40_project1.ttt`. En la Figura 9.1 se observa una vista general de la escena.
- También, como punto de partida, se cuenta con toda la librería pyARTE y el script `applications/irb40_project1.py`.

En la escena, se observan los siguientes elementos:

- Una cinta transportadora que se mueve a velocidad constante. Nótese que existe un sensor que detiene la cinta cuando se detecta una pieza en uno de los extremos.
- Existe un script de Lua (`CubeSpawner`) que genera cubos en posiciones y orientaciones aleatorias. El color de las piezas es, también, aleatorio, siendo la gama de colores: R (rojo), G (verde) o B (azul). Las piezas azules aparecen en grupos de dos, siempre.
- El robot está equipado con una cámara en su extremo.
- Se proporciona al estudiante una función denominada `find_color` que realiza las siguientes acciones:
  - Sitúa la cámara del robot sobre la pieza.
  - Captura una imagen y calcula su color medio.
  - Devuelve un resultado 'R', 'G' o 'B' con el color más cercano al calculado.

#### **Funcionamiento deseado:**

- Cuando se detecte una pieza con el sensor de la cinta transportadora, se deberá iniciar el proceso de recogida de la pieza.
- Conocido su color, el robot deberá coger la pieza sin detener la cinta transportadora (ni el tiempo de simulación).

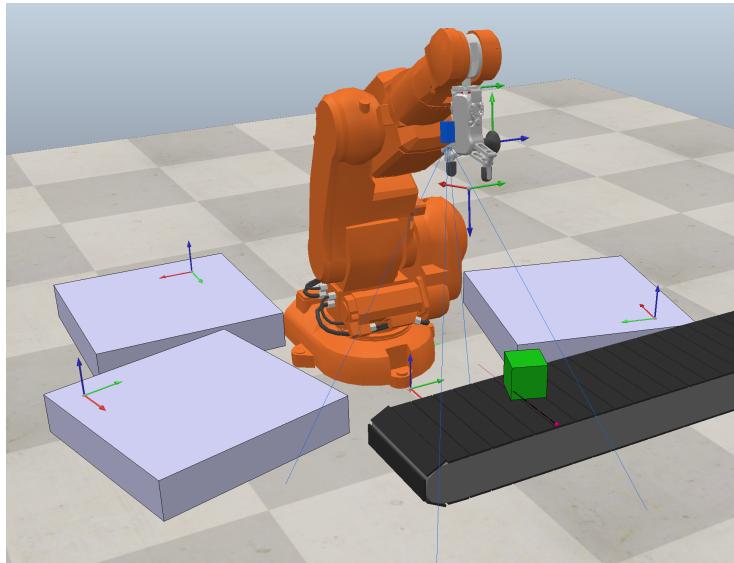


Figura 9.1: Una captura de la escena del Proyecto 1.

- El robot deberá colocar la pieza de forma ordenada en base a su color en alguno de los pallets proporcionados. Nótese que los palés están ubicados en diferentes posiciones y orientaciones. Cada palé contendrá piezas de un único color.
- La disposición de las piezas utilizará los mismos principios vistos en la práctica de paletizado. Se desea utilizar arreglos de 2x3x2 piezas en las direcciones X, Y y Z, respectivamente, de acuerdo con los sistemas de referencia que se encuentran ubicados en cada palé.
- Todas las piezas tienen un tamaño de 0,08 m y se desea un espaciado entre piezas de 0,005 m.
- La operación de recoger y dejar la pieza se deberá realizar con la ventosa que existe adherida a la pinza del robot.
- La aplicación finalizará cuando se hayan paletizado 30 piezas.
- Se deberá programar el robot para que sea capaz de procesar todas las piezas generadas con un tiempo `WAIT_TIME = 14` o menor.

En la librería existen algunas funciones que puede utilizar el estudiantado:

- Un método `camera.get_color_name`. Devuelve el color medio de la escena (R, G o B) como un carácter de texto. Para ver un ejemplo de uso, se puede observar la función `get_color()`
- Se proporciona al estudiante una función que devuelve la posición y orientación de la pieza en el sistema de coordenadas de la base del robot. Esta función simula la existencia de una cámara que permite calcular la posición de la pieza. La función `get_object_transform` devuelve una matriz de transformación homogénea.

Nota: la función `get_object_transform` necesita conocer el índice del objeto que se denomina 'Cuboid', 'Cuboid0', 'Cuboid1'... etc. Estos nombres se generan de forma automática cada vez que el script de Lua `CubeSpawner` genera un nuevo cuboide. De ahí que precise conocer un índice global de la pieza que se está procesando.

## 9.4. Código

Se proporciona una estructura inicial del código. El/la estudiante deberá completar las funciones denominadas `pick` y `place`. La función principal del programa se presenta, a continuación:

```
def pick_and_place():
    simulation = Simulation()
    clientID = simulation.start()
    robot = RobotABBIRB140(clientID=clientID)
    robot.start()
    conveyor_sensor = ProxSensor(clientID=clientID)
    conveyor_sensor.start()
    camera = Camera(clientID=clientID)
    camera.start()
    gripper = SuctionPad(clientID=clientID)
    gripper.start()
    robot.set_TCP(HomogeneousMatrix(Vector([0, 0.065, 0.105]),
                                      Euler([-np.pi / 2, 0, 0])))
    q0 = np.array([0, 0, 0, np.pi / 2, 0])
    robot.moveAbsJ(q_target=q0, precision=False)

    piece_index = 0
    color_indices = np.array([0, 0, 0])
    n_pieces = 30
    for i in range(n_pieces):
        print('PROCESSING PIECE: ', i)
        while True:
            if conveyor_sensor.is_activated():
                break
            simulation.wait()

        color = find_color(robot, camera, piece_index)
        pick(robot, gripper, piece_index)
        place(robot, gripper, color, color_indices)
        robot.moveAbsJ(q_target=q0, precision=False)

        # Next piece! Update indices
        piece_index += 1
        if color == 'R':
            color_indices[0] += 1
        elif color == 'G':
            color_indices[1] += 1
        else:
            color_indices[2] += 1
    simulation.stop()
```

La función `find_color` se proporciona completa al alumno y devuelve el color de la pieza que se encuentra detenida en el extremo de la cinta transportadora. La función necesita conocer el número de pieza que se está procesando. Dentro de la función `find_color` encontraréis la función:

```
T_piece = get_object_transform(clientID=robot.clientID,
                               base_name='Cuboid',
```

```
piece_index=Piece_Index)
```

Esta función permite obtener una matriz de transformación homogénea con la posición y orientación de la pieza  $i$  que se desea colocar en un palé. En el caso de las piezas azules, primero se devuelve la transformación de la pieza superior y, a continuación, de la inferior.

Dada una matriz homogénea  $T$ , el método siguiente resulta de interés, pues muestra en Coppelia la posición y orientación de  $T$ , por ejemplo, representando un *target point*.

```
robot.show_target_points([T.pos()], [T.R()])
```

Funciones a completar en el proyecto:

- **pick**: Esta función deberá utilizar la función `get_object_transform` para obtener la posición y orientación de la pieza. La orientación de la pinza se deberá calcular teniendo en cuenta la orientación de la pieza que se desea aprehender.
- **place**: Esta función deberá ubicar las piezas en tres palés diferentes en función de su color. Para ello, se debe mantener una cuenta separada de las piezas de cada color. En el código proporcionado, se almacena en el array `color_indices` el número de piezas rojas (R), verdes (G) y azules (B).

## 9.5. Descripción del proyecto 2

### Material proporcionado:

Se generan piezas que caen unas sobre otras, de tal manera que es necesario calcular un orden en que se recogen.

## 9.6. Descripción del proyecto 3

### Material proporcionado:

- Se proporciona la siguiente escena de Coppelia: `more/irb40_project3.ttt`
- También, como punto de partida, se cuenta con toda la librería pyARTE y el script `applications/irb40_project.py`.

En la escena, se observan los siguientes elementos:

- Una cinta transportadora que se mueve a velocidad constante. Nótese que existe un sensor que detecta el paso de una pieza, pero no detiene la cinta.
- Las piezas caen a una zona de recogida que simula una superficie no uniforme. Por tanto, las piezas caerán en posiciones y orientaciones aleatorias.
- Existe un script de Lua (`CubeSpawner`) que genera cubos en posiciones y orientaciones aleatorias sobre la cinta transportadora. El color de las piezas es, también, aleatorio, siendo la gama de colores: R (rojo), G (verde) o B (azul).

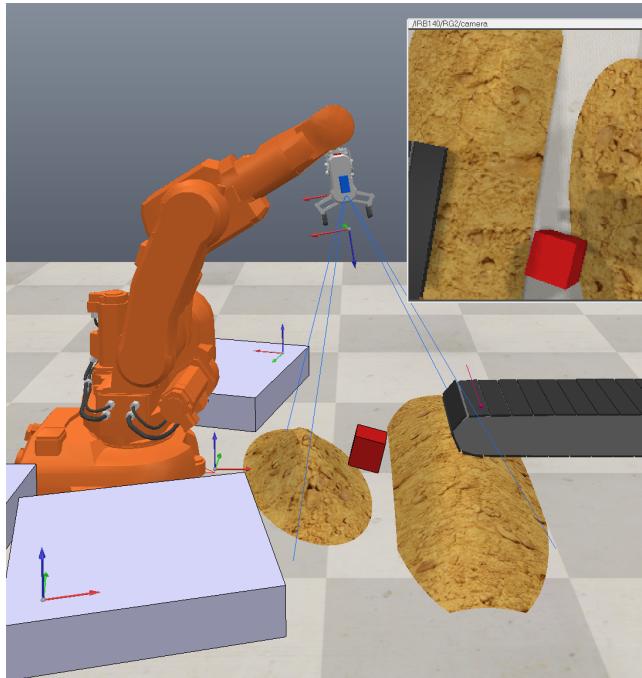


Figura 9.2: Una captura del proyecto 3 en Coppelia.

- El robot está equipado con una cámara en su extremo.
- Se proporciona al estudiante una función denominada `find_color` que realiza las siguientes acciones:
  - Sitúa la cámara del robot sobre la pieza.
  - Captura una imagen y calcula su color medio.
  - Devuelve un resultado 'R', 'G' o 'B' con el color más cercano al calculado.

Esta función se deberá modificar para observar la pieza de forma conveniente.

#### **Funcionamiento deseado:**

- Cuando se detecte una pieza con el sensor de la cinta transportadora, se deberá iniciar el proceso de recogida de la pieza.
- A diferencia del proyecto 1, en este caso las piezas caen y toman posiciones y orientaciones pseudo-aleatorias.
- El script deberá elegir la cara del cubo con mejor orientación para realizar la inspección del color de la pieza y el proceso de recogida.
- El robot deberá colocar la pieza de forma ordenada en base a su color en alguno de los pallets proporcionados. Nótese que los palés están ubicados

en diferentes posiciones y orientaciones. Cada palé contendrá piezas de un único color.

- La disposición de las piezas utilizará los mismos principios vistos en la práctica de paletizado. Se desea utilizar arreglos de 2x3x2 piezas en las direcciones X, Y y Z, respectivamente, de acuerdo con los sistemas de referencia que se encuentran ubicados en cada palé.
- Todas las piezas tienen un tamaño de 0,08 m y se desea un espaciado entre piezas de 0,005 m.
- La operación de recoger y dejar la pieza se deberá realizar con la ventosa que existe instalada en la pinza del robot.
- La aplicación finalizará cuando se hayan paletizado 30 piezas.
- Se deberá programar el robot para que sea capaz de procesar todas las piezas generadas con un tiempo WAIT\_TIME = 14 o menor.

En la librería existen algunas funciones que puede utilizar el estudiantado:

- Un método `camera.get_color_name`. Devuelve el color medio de la escena (R, G o B) como un carácter de texto. Para ver un ejemplo de uso, se puede observar la función `get_color()`
- Se proporciona al estudiante una función que devuelve la posición y orientación de la pieza en el sistema de coordenadas de la base del robot. Esta función simula la existencia de una cámara que permite calcular la posición de la pieza. La función `get_object_transform` devuelve una matriz de transformación homogénea.

## 9.7. Ayuda

Debe tenerse en cuenta que la función `get_object_transform` devuelve información esencial para recoger la pieza, pues:

- Permite conocer el centro de masas del Cuboide.
- Permite conocer la orientación de un sistema de referencia ubicado en su centroide.
- Nótese que, a partir de esa información, es posible conocer la posición del punto central de cada cara del cuboide, pudiéndose elegir aquélla cara que tenga un punto central más elevado. Igualmente, se puede calcular fácilmente varios *target points* para asir la pieza en cada una de sus caras.

## 9.8. Código

Se proporciona una estructura inicial del código. El/la estudiante deberá completar las funciones denominadas `pick` y `place`. La función principal del programa se presenta, a continuación:

```

def pick_and_place():
    simulation = Simulation()
    clientID = simulation.start()
    robot = RobotABBIRB140(clientID=clientID)
    robot.start()
    conveyor_sensor = ProxSensor(clientID=clientID)
    conveyor_sensor.start()
    camera = Camera(clientID=clientID)
    camera.start()
    gripper = SuctionPad(clientID=clientID)
    gripper.start()
    robot.set_TCP(HomogeneousMatrix(Vector([0, 0.065, 0.105]), Euler([-np.pi / 2, 0, 0])))
    q0 = np.array([0, 0, 0, 0, np.pi / 2, 0])
    robot.moveAbsJ(q_target=q0, precision=False)

    piece_index = 0
    color_indices = np.array([0, 0, 0])
    n_pieces = 30
    for i in range(n_pieces):
        print('PROCESSING PIECE: ', i)
        while True:
            if conveyor_sensor.is_activated():
                break
            simulation.wait()

        color = find_color(robot, camera, piece_index)
        pick(robot, gripper, piece_index)
        place(robot, gripper, color, color_indices)
        robot.moveAbsJ(q_target=q0, precision=False)

        # Next piece! Update indices
        piece_index += 1
        if color == 'R':
            color_indices[0] += 1
        elif color == 'G':
            color_indices[1] += 1
        else:
            color_indices[2] += 1
    simulation.stop()

```

La función `find_color` se proporciona completa al alumno y devuelve el color de la pieza que se encuentra detenida en el extremo de la cinta transportadora. La función necesita conocer el número de pieza que se está procesando. Dentro de la función `find_color` encontraréis la función:

```

T_piece = get_object_transform(clientID=robot.clientID,
                               base_name='Cuboid',
                               piece_index=piece_index)

```

Esta función permite obtener una matriz de transformación homogénea con la posición y orientación de la pieza  $i$  que se desea colocar en un palé. En el caso de las piezas azules, primero se devuelve la transformación de la pieza superior y, a continuación, de la inferior.

Dada una matriz homogénea  $T$ , el método siguiente resulta de interés, pues muestra en Coppelia la posición y orientación de  $T$ , por ejemplo, representando un *target point*.

```
robot.show_target_points([T.pos()], [T.R()])
```

Funciones a completar en el proyecto:

- **pick:** Esta función deberá utilizar la función `get_object_transform` para obtener la posición y orientación de la pieza. La orientación de la pinza se deberá calcular teniendo en cuenta la orientación de la pieza que se desea aprehender.
- **place:** Esta función deberá ubicar las piezas en tres palés diferentes en función de su color. Para ello, se debe mantener una cuenta separada de las piezas de cada color. En el código proporcionado, se almacena en el array `color_indices` el número de piezas rojas (R), verdes (G) y azules (B).



## **Parte IV**

## **Anexos**



# Capítulo 10

## Instalación de Coppelia Sim

### 10.1. 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.

### 10.2. 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.

### 10.3. 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
```

## 10.4. Instalación de Coppelia

Coppelia permite descargar distribuciones del simulador desde:

[www.coppeliarobotics.com/downloads](http://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.

## 10.5. Instalación de pyARTE y de las librerías de interfaz

Clone la librería pyARTE:

```
$ cd /home/usuario/Escritorio  
$ git clone https://github.com/4rtur1t0/pyARTE.git
```

Clone las librerías de interfaz de Coppelia con python:

```
$ cd /home/usuario/Escritorio  
$ git clone https://github.com/4rtur1t0/coppelia_API.git
```

Dentro del directorio clonado (`coppelia_API`) encontraremos, entre otros, un fichero `sim.py`. Este fichero es el punto de partida de la librería para su importación.

## 10.6. 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/Escritorio/venv/bin/python`. Esto le dice a pycharm dónde se encuentra el intérprete de python.
- Añada las librerías de interfaz: File - Settings - Project - Project Structure - add root content - añada el directorio que contiene la API: `/home/usuario/Escritorio/coppelia_API`.  
Estas librerías las proporciona la empresa desarrolladora de Coppelia para diferentes lenguajes: C/C++, python y Matlab. En nuestro caso, utilizamos las librerías de python para Linux.

## 10.7. 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.