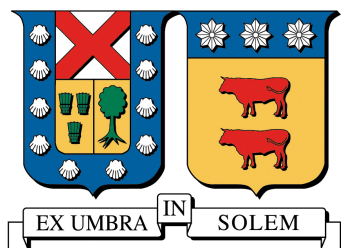


UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE ELECTRÓNICA
VALPARAÍSO - CHILE



**“MODELADO Y SIMULACIÓN CINEMÁTICA
3D DEL ROBOT INDUSTRIAL RV-2AJ”**

JAVIER IGNACIO QUIROGA DE LA ROZA

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELECTRÓNICO
MENCIÓN CONTROL E INSTRUMENTACIÓN

PROFESOR GUÍA:

SR. MANUEL OLIVARES SALINAS.

PROFESOR CORREFERENTE:

SR. AGUSTÍN GONZÁLEZ VALENZUELA.

Marzo 2022

Dedicado a mis padres.

RESUMEN EJECUTIVO

El uso de robots es y seguirá siendo una pieza clave en el desarrollo industrial, es por ello, contar con un modelo fidedigno de estos es clave, sobre todo para realizar simulaciones de las operaciones a realizar antes de implementarlas.

En esta memoria de titulación se obtiene el modelado y simulación del robot industrial RV-2AJ para realizar el seguimiento a trayectorias punto a punto. Para ello se ha desarrollado una aplicación en Matlab App Designer, con la cual es posible simular y manipular el robot a través de una conexión por red al controlador del robot programado en lenguaje Melfa Basic IV .

Se obtiene el modelo cinemático directo a través de la convención de Denavit–Hartenberg y se obtiene su modelo cinemático inverso en base a la estructura geométrica del robot y también por métodos numéricos en base a optimización para luego generar un modelo virtual del robot en 3D.

Estos modelos son utilizados para generar trayectorias en base a dibujos en formato SVG, para que el robot pueda replicarlas dibujándolas sobre una hoja. Estas trayectorias son validadas mediante el software CIROS Studio el cual cuenta con herramientas para la simulación de robots industriales. Todo esto se empaqueta en una aplicación realizada en Matlab con la que se manipula y transmite las trayectorias al controlador del robot mediante la conexión en una red de computadores a través de TCP/IP

Como resultado es posible manipular el robot sin la necesidad de definir puntos mediante su Teach box lo cual permite realizar aplicaciones como por ejemplo, reproducir con el EF del robot el trazado de una figura bidimensional a partir de una imagen en formato SVG.

ABSTRACT

The use of robots is and will continue to be a key element in industrial development, which is why having a reliable model of these is key, especially to carry out simulations of those to be carried out before implementing them.

In this thesis, the modeling and simulation of the RV-2AJ industrial robot are obtained to track point-to-point trajectories. For this, an application has been developed in Matlab App Designer, with which it is possible to simulate and manipulate the robot through a network connection to the robot controller programmed in Melfa Basic IV language.

The direct kinematic model is obtained through the Denavit-Hartenberg convention and its inverse kinematic model is obtained based on the geometric structure of the robot and also by numerical methods based on optimization to then generate a virtual model of the robot in 3D.

These models are used to generate trajectories based on drawings in SVG format, so that the robot can replicate them by drawing them on a sheet. These trajectories are validated using the CIROS Studio software, which has tools for simulating industrial robots. All this is packaged in an application made in Matlab with which the trajectories are manipulated and transmitted to the robot controller by connecting to a computer network through TCP/IP.

As a result, it is possible to manipulate the robot without the need to define points through its Teach box which allows applications such as reproducing with the robot's EF, the tracing of a two-dimensional figure from an image in format.

GLOSARIO

Articulación(Joint) Componentes móviles del robot que causan un movimiento relativo entre los eslabones.

Eslabón(Link) Elemento estructural rígido con el que se forma un brazo robótico. Se conectan mediante articulaciones.

EF(End Effector) En español Efecto final, corresponde al ultimo eslabón de un brazo robótico. Generalmente es un dispositivo periférico cuyo propósito es interactuar con el entorno.

Cadena Cinemática Estructura que forma la unión entre eslabones y articulaciones. Puede ser abierta o cerrada.

DOF Grados de libertad por sus siglas en ingles, numero que describe el total de desplazamientos independientes o aspectos del movimiento. Para robots descritos por cadenas cinemáticas abiertas, generalmente corresponde a numero de articulaciones.

Workspace Conjunto de posiciones a las que el EF puede acceder.

Matriz Jacobiana Matriz cuyos elementos son las derivadas parciales de primer orden de una función vectorial de varias variables.

Teach box/Teach Pendant Terminal de enseñanza en español, dispositivo de control que permite grabar manualmente las posiciones utilizadas para programar los movimientos de un robot. Teach se refiere a indicarle manualmente a donde dirigirse al robot.

TCP/IP Transmission Control Protocol/Internet Protocol (Protocolo de control de transmisión/Protocolo de Internet). TCP/IP es un conjunto de reglas estandarizadas que permiten a los equipos comunicarse en una red.

Paquete Bloque de datos y de información de control que conforman una transacción entre un sistema principal y su red.

Socket Punto final de un enlace de comunicación bidireccional entre dos programas que se ejecutan en la red.

Matlab Abreviatura de MATrix LABoratory, sistema de cómputo numérico que ofrece un entorno de desarrollo integrado (IDE) con un lenguaje de programación propio (lenguaje M).

Callback Usado en Matlab. Función que se ejecuta en respuesta a alguna acción predefinida por el usuario, como un botón.

RVC Robotics, Vision & Control, conjunto de herramientas de robótica para Matlab .

CIROS Studio Computer Integrated Robot Simulation, software para la simulación de entornos de fabricación automatizada en 3D que cuenta con modelos de robots industriales.

URDF Universal Robot Description Format, formato XML que se utiliza para describir todos los elementos físicos de un robot muy utilizado por el middleware ROS(Robot Operating System).

Índice de Contenidos

1. Introducción	1
1.1. Problema a resolver	1
1.2. Herramientas usadas	1
1.3. Resumen de Capítulos	2
2. Modelo Cinemático Directo	3
2.1. Descripción de posición y orientación de un punto en el espacio	3
2.2. Rotación alrededor de un Eje	4
2.3. Cadena Cinemática	8
2.4. Parámetros Denavit–Hartenberg	10
2.5. Espacio de trabajo	14
3. Cinemática diferencial	18
3.1. Jacobiano Geométrico	18
3.2. Jacobiano Analítico	21
3.3. Jacobiano con parámetros Mitsubishi	23
3.4. Singularidades	24
4. Modelo Cinemático Inverso	25
4.1. Método Geométrico	25
4.2. Método Numérico	32
4.3. Resultados	33
5. Validación por simulación	36
5.1. RVC Toolbox	36
5.2. Melfa Basic IV	37
5.3. Conexión entre Matlab Y Robot	39
6. Generación Automática de Trayectorias	45
6.1. Generación de trayectoria a partir de imágenes	45
6.2. Transmisión de puntos de trayectoria	49
6.3. Pizarra de escritura	52
6.4. Resultados obtenidos	54
7. Resultados y Conclusiones	59
7.1. Trabajo futuro	59
A. Funciones Cinemática Inversa	61
A.1. Cinemática Inversa geométrica	61
A.2. Cinemática inversa numérica con J_M	63
A.3. Cinemática inversa numérica con J_a	65

B. Configuración RT Toolbox2 para conexión	67
C. Código Melfa Basic IV completo	70
D. Dibujos	72
E. SVG a Datos	88
E.1. Programa en Matlab	88
E.2. Función Workspace circular	91
E.3. Código Python	91
Bibliografía	105

Índice de Tablas

2.1. Rango de articulaciones RV-2AJ	10
2.2. Parámetros Denavit–Hartenberg RV-2AJ	13
2.3. Parámetros Denavit–Hartenberg RV-2AJ con valores reales	13
5.1. Flags según configuración de pose del Robot	38
6.1. RMSE coordenadas cartesianas dibujo brazo	56
6.2. RMSE coordenadas articulares dibujo brazo	56
6.3. RMSE coordenadas cartesianas dibujo de Helicoide Exponencial	58
D.2. RMSE coordenadas cartesianas dibujo de Mujer con sombrero	74
D.1. RMSE coordenadas articulares dibujo de Mujer con sombrero	74
D.3. RMSE coordenadas articulares dibujo de Búho	77
D.4. RMSE coordenadas cartesianas dibujo de Búho	77
D.5. RMSE coordenadas articulares dibujo de Gato	80
D.6. RMSE coordenadas cartesianas dibujo de Gato	80
D.7. RMSE coordenadas articulares dibujo de Mapamundi	83
D.8. RMSE coordenadas cartesianas dibujo de Mapamundi	83
D.9. RMSE coordenadas articulares dibujo de Abraham Lincoln	85
D.10. RMSE coordenadas cartesianas dibujo de Mujer con Cigarro	87

Índice de Figuras

2.1. Rotación de un marco de referencia en un punto	4
2.2. Rotacion alrededor eje $\hat{\mathbf{X}}_A$	5
2.3. Rotaciones Roll, Pitch y Yaw	6
2.4. Relación entre punto y marco de referencia	8
2.5. Articulaciones robot RV-2AJ	10
2.6. Definición de parámetros Denavit–Hartenberg	11
2.7. Parámetros Denavit–Hartenberg para el robot RV-2AJ	12
2.8. Workspace 2D sin garra con $q_1 = 0$	15
2.9. Workspace 2D con garra y con $q_1 = 0$	16
2.10. Workspace 3D sin garra	16
2.11. Workspace 3D con garra	17
3.1. Ejes del EF x(rojo),y(verde),z(azul) y rotaciones a través de ellos	19
3.2. Relación entre velocidad lineal y angular respecto al eje Z	20
4.1. Calculo de q_1	26
4.2. Análisis geométrico de las coordenadas articulares q_2, q_3 y q_4	26
4.3. Análisis geométrico de las coordenadas articulares para q_2	28
4.4. Análisis geométrico para obtención de p'_{y3} y p'_{x3}	29
4.5. Distintas configuraciones dada posición y orientación	31
4.6. Evolución de las coordenadas articulares usando cinemática inversa con $J_M(q)$	34
4.7. Evolución de las coordenadas articulares usando cinemática inversa con $J_a(q)$	35
5.1. Modelo 3D simple con toolbox RVC	36
5.2. Ejemplo de movimiento continuo CNT	39
5.3. Interfaz de aplicación diseñada	41
5.4. Interfaz Cinemática Inversa numérica	42
5.5. Interfaz para Iniciar conexión con servidor TCP/IP	43
5.6. Distintas configuraciones obtenidas a partir de la aplicación en Matlab	44
6.1. Ejemplo de curva de Bezier	45
6.2. Imagen a dibujar SVG	46
6.3. Región de dibujo en vista al plano XY	47
6.4. Dibujo de un brazo donde los puntos indican saltos entre trayectorias	48
6.5. Comprobacion de trayectoria en CIROS con saltos	49
6.6. Diagrama de secuencia para el dibujo	51
6.7. Interfaz de conexion de aplicacion en Matlab	52
6.7. Ejemplo en Pizarra de dibujo de la aplicación	53
6.8. Comparación de datos cartesianos enviados y recibidos	54
6.9. Comparación de datos cartesianos enviados y recibidos Zoom	55
6.10. Comparación de datos cartesianos enviados y recibidos Zoom sin desfase	55
6.11. Comparación de resta de datos recibidos con enviados	56

6.12. Comparación dibujos de un brazo	57
6.13. Comparación de datos Cartesianos Helicoide Exponencial	58
B.1. Vista de RT Toolbox2 para configurar la linea de comunicación	67
B.2. Configuración de parámetros	68
B.3. Configuración de parámetros	69
D.1. Comparación dibujos de Mujer con sombrero	72
D.2. Comparación de datos Cartesianos Mujer con sombrero	73
D.3. Comparación de datos Articulares Mujer con sombrero	73
D.4. Comparación dibujos de Búho	75
D.5. Comparación de datos Cartesianos Búho	76
D.6. Comparación de datos Articulares Búho	76
D.7. Comparación dibujos de Gato	78
D.8. Comparación de datos Cartesianos Gato	79
D.9. Comparación de datos Articulares Gato	79
D.10. Comparación dibujos de Mapamundi	81
D.11. Comparación de datos Cartesianos Mapamundi (A fijo)	82
D.12. Comparación de datos Articulares Mapamundi	82
D.13. Comparación dibujos de Abraham Lincoln	84
D.14. Comparación de datos Articulares Abraham Lincoln	85
D.15. Comparación dibujos de Mujer con Cigarro	86
D.16. Comparación de datos Cartesianos Mujer con Cigarro	87

1 | Introducción

1.1. Problema a resolver

El Robot RV-2AJ de la empresa Mitsubishi, es un robot industrial de 5 grados de libertad muy utilizado tanto en la industria como en fines educativos, donde su principal forma de uso consiste en grabar manualmente puntos en el espacio con su Teach Box y realizar tareas secuenciales y repetitivas en base a instrucciones simples en lenguaje Melfa Basic IV.

El objetivo general de este proyecto es automatizar la obtención de puntos del EF del robot industrial RV-2AJ para generar diversas trayectorias, es decir, encontrar un modo para que el robot haga un seguimiento a puntos a definir sin la necesidad de grabar cada punto de la trayectoria con el Teach box. Para ello se estudia la descripción del modelo cinemático directo e inverso que relaciona las 5 coordenadas articulares con la posición y orientación del EF para poder describir cada punto a generar y posteriormente enviar al robot.

Para comunicarle al robot los puntos que debe seguir es que se propone enviarlos a través de la red de computadores del departamento de electrónica. Para ello se estudia el método de vincular el computador y programa que mandará los datos ,con el controlador del robot que los recibe.

Las trayectorias son obtenidas a partir de imágenes en formato SVG, el cual es un formato de imágenes vectoriales de las cuales es posible extraer los puntos para generarla.

Los puntos obtenidos deben ser procesados y enviados en un formato en que el programa del controlador del robot en Melfa Basic IV sea capaz de interpretarlo correctamente. En la industria es muy importante tener un modelo del proceso para depurar errores, es por ello que antes de usar el robot físico, se comprueba con un robot virtual lo cual se hace con el programa CIROS Studio el cual tiene la capacidad de simular el robot y su programa en Melfa Basic IV.

El programa seleccionado para hacer todos los análisis matemáticos es Matlab en conjunto con la toolbox RVC ,con el cual se plantea crear una aplicación con la cual será posible hacer una simulación del robot y también manipular el robot real para enviarle las trayectorias.

1.2. Herramientas usadas

Para este proyecto se dispone del Software Matlab R2021b, en conjunto con la toolbox Robotics, Vision & Control en su décima versión [1]. También se cuenta con el Software CIROS Studio que cuenta con avanzadas opciones de simulación para la industria 4.0 [3] Por otro lado , en el laboratorio de control industrial del departamento de electrónica de la Universidad Técnica Federico Santa María, se cuenta con el robot industrial RV-2AJ de Mitsubishi, el cual cuenta con una pinza eléctrica como EF. Posee el controlador CR1-571 [4] y el Teach box R28TB con la cual se puede manipular el robot. En los computadores del laboratorio se dispone del software RT toolbox2 de Mitsubishi con el cual se crean y se cargan los programas en lenguaje Melfa Basic IV al controlador CR1-571 mediante un enlace de red.

1.3. Resumen de Capítulos

En el capítulo 2 se introduce a los conceptos de posición y orientación y su descripción matemática para poder obtener finalmente la cinemática inversa del robot RV-2AJ. Se presentan los conceptos de matrices de rotación y de transformación homogénea con los cuales se pueden describir marcos de referencia a partir de otros.

Luego explica el concepto de los parámetros de Denavit-Hartenberg, con los cuales es posible describir cualquier cadena cinemática para luego obtener estos parámetros para el robot en estudio y describir su cinemática directa para obtener los valores de la posición y orientación a partir de las coordenadas articulares. Se describen también las distintas formas que existen para representar la orientación, una de ellas es la representación usada por Mitsubishi. Por último se obtiene el Workspace para determinar las limitaciones de alcance del robot.

En el capítulo 3 se estudia la cinemática diferencial, haciendo énfasis en el Jacobiano el cual relaciona la velocidad de las articulaciones con las velocidades lineales y angulares del EF. El Jacobiano es obtenido a partir de los parámetros de Denavit-Hartenberg, al cual se le denomina Jacobiano Geométrico y luego se obtiene el Jacobiano analítico el cual será útil para la cinemática inversa. También se obtiene un tercer Jacobiano usando la representación de orientación de Mitsubishi. Por último, a partir de los jacobianos se obtienen las singularidades las cuales representan puntos críticos para la manipulabilidad del robot. El jacobiano geométrico ha sido obtenido en [16], sin embargo, solo para q_2, q_3 y q_4 .

En el capítulo 4 se obtiene la cinemática Inversa la cual sirve para obtener las coordenadas articulares dado el punto y orientación del EF. Esta es obtenida mediante dos métodos; uno geométrico y otro numérico. El primer método es un método exacto el cual está basado en la estructura del robot y sus dimensiones. Esta se obtiene en base a tres métodos de describir la orientación y se puede obtener 1 de las 4 soluciones para una posición y orientación. El método numérico es más impreciso ya que se basa en optimización por minimización, el cual está basado en el Jacobiano. Este fue obtenido con el Jacobiano analítico y el de Mitsubishi. La cinemática inversa del robot RV-2AJ ha sido obtenida en [12][13][14][15] sin embargo en ninguna calculan su forma completa.

En el capítulo 5 se describen algunas de las funcionalidades de la toolbox RVC y algunas instrucciones a usar en el programa en Melfa Basic IV. Luego se describe el formato de los datos a enviar y se muestra el código en Melfa y la aplicación creada en el entorno *App Designer* de Matlab, con la cual es posible enviar posiciones únicas para que el robot se mueva mediante el protocolo TCP/IP.

En el capítulo 6 se obtienen los puntos de trayectoria a partir de imágenes en formato SVG, en donde estos son procesados y son comprobados en simulación usando el software *Ciros Studio*. Tanto la aplicación como el código en Melfa son actualizados y se describe el proceso para enviar los paquetes de datos en red hacia el controlador del robot. Por último se obtienen los resultados de los dibujos junto con el error obtenido.

2 | Modelo Cinemático Directo

El problema cinemático de un robot corresponde a obtener las relaciones entre las posiciones de cada articulación del robot y la posición con orientación de el extremo del robot. La cinemática directa tiene como objetivo describir la posición y orientación del efector final (EF, End Effector por sus siglas en inglés) del brazo robótico respecto a un marco de referencia inercial en función de sus articulaciones. Para ello es necesario introducir las distintas formas de representar una posición y orientación en el espacio.

2.1. Descripción de posición y orientación de un punto en el espacio

La posición de un punto en el espacio respecto a un marco de referencia $\{\mathbf{A}\}$ puede ser descrito usando la forma vectorial:

$${}^A\mathbf{p} = p_x\hat{\mathbf{X}}_A + p_y\hat{\mathbf{Y}}_A + p_z\hat{\mathbf{Z}}_A \quad (2.1)$$

Donde $(\hat{\mathbf{X}}_A, \hat{\mathbf{Y}}_A, \hat{\mathbf{Z}}_A)$ son vectores unitarios de los ejes del marco base $\{\mathbf{A}\}$. También se puede usar la forma matricial:

$${}^A\mathbf{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \quad (2.2)$$

Para definir la orientación, se considera un marco de referencia $\{\mathbf{B}\}$ el cual puede estar en cualquier punto del espacio, con lo cual los ejes del marco $\{\mathbf{B}\}$ se describen como:

$$\hat{\mathbf{X}}_B = r_{11}\hat{\mathbf{X}}_A + r_{21}\hat{\mathbf{Y}}_A + r_{31}\hat{\mathbf{Z}}_A \quad (2.3)$$

$$\hat{\mathbf{Y}}_B = r_{12}\hat{\mathbf{X}}_A + r_{22}\hat{\mathbf{Y}}_A + r_{32}\hat{\mathbf{Z}}_A \quad (2.4)$$

$$\hat{\mathbf{Z}}_B = r_{13}\hat{\mathbf{X}}_A + r_{23}\hat{\mathbf{Y}}_A + r_{33}\hat{\mathbf{Z}}_A \quad (2.5)$$

Donde r_{ij} son escalares. A partir de lo anterior se puede llevar a su forma matricial creando la matriz de rotación del marco $\{\mathbf{A}\}$ al $\{\mathbf{B}\}$

$${}^A R_B = \begin{bmatrix} \hat{\mathbf{X}}_B & \hat{\mathbf{Y}}_B & \hat{\mathbf{Z}}_B \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \quad (2.6)$$

Un punto en el espacio tridimensional tiene en total 6 grados de libertad (DOF) ;3 para su posición $\mathbf{P}_{x,y,z}$ y 3 grados de libertad para su orientación, los cuales corresponde a una rotación por cada eje, sin embargo la matriz de rotación posee 9 elementos, por lo que se necesitan $(9-3)=6$ restricciones las cuales son características propias de los marcos de referencia, las cuales son:

- $\hat{\mathbf{X}}_B, \hat{\mathbf{Y}}_B, \hat{\mathbf{Z}}_B$ son vectores unitarios

$$r_{11}^2 + r_{21}^2 + r_{31}^2 = 1 \quad (2.7)$$

$$r_{12}^2 + r_{22}^2 + r_{32}^2 = 1 \quad (2.8)$$

$$r_{13}^2 + r_{23}^2 + r_{33}^2 = 1 \quad (2.9)$$

- $\hat{\mathbf{X}}_B, \hat{\mathbf{Y}}_B, \hat{\mathbf{Z}}_B$ son vectores ortogonales

$$\hat{\mathbf{X}}_B \cdot \hat{\mathbf{Y}}_B = r_{11}r_{12} + r_{21}r_{22} + r_{31}r_{32} = 0 \quad (2.10)$$

$$\hat{\mathbf{X}}_B \cdot \hat{\mathbf{Z}}_B = r_{12}r_{13} + r_{22}r_{23} + r_{32}r_{33} = 0 \quad (2.11)$$

$$\hat{\mathbf{Y}}_B \cdot \hat{\mathbf{Z}}_B = r_{11}r_{13} + r_{21}r_{23} + r_{31}r_{33} = 0 \quad (2.12)$$

Por lo tanto la matriz de rotación R es ortonormal por lo que se cumple que $R^T R = I$ con I la matriz identidad y multiplicando lo anterior por R^{-1} se tiene que su inversa es igual a su transpuesta es decir:

$${}^A R_B^T = {}^A R_B^{-1} = {}^B R_A. \quad (2.13)$$

Luego si el marco de referencia está definido por la regla de la mano derecha ($\hat{\mathbf{X}}_B \times \hat{\mathbf{Y}}_B = \hat{\mathbf{Z}}_B$) entonces su determinante es $\det(R)=1$.

Los términos escalares r_{ij} de la matriz de rotación (2.6) se pueden reescribir notando que los componentes de cada vector es la proyección de ese vector en la dirección del marco base $\{\mathbf{A}\}$, es decir como el producto punto de los vectores unitarios de cada marco, quedando expresado como:

$${}^A R_B = \begin{bmatrix} \hat{\mathbf{X}}_B \cdot \hat{\mathbf{X}}_A & \hat{\mathbf{Y}}_B \cdot \hat{\mathbf{X}}_A & \hat{\mathbf{Z}}_B \cdot \hat{\mathbf{X}}_A \\ \hat{\mathbf{X}}_B \cdot \hat{\mathbf{Y}}_A & \hat{\mathbf{Y}}_B \cdot \hat{\mathbf{Y}}_A & \hat{\mathbf{Z}}_B \cdot \hat{\mathbf{Y}}_A \\ \hat{\mathbf{X}}_B \cdot \hat{\mathbf{Z}}_A & \hat{\mathbf{Y}}_B \cdot \hat{\mathbf{Z}}_A & \hat{\mathbf{Z}}_B \cdot \hat{\mathbf{Z}}_A \end{bmatrix} \quad (2.14)$$

El conjunto de matrices de rotación de 3×3 conforman el **grupo ortonormal especial** $SO(3)$ el cual cumple con que sus columnas son ortonormales y su determinante igual a 1. La multiplicación de 2 matrices del grupo también pertenece al grupo.

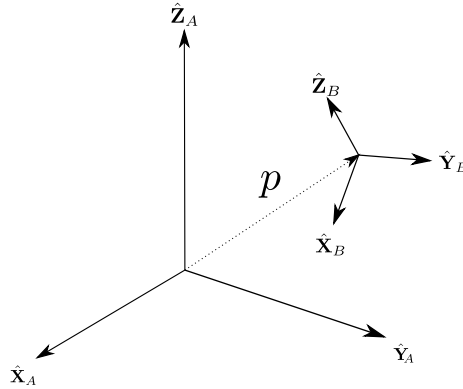


Figura 2.1: Rotación de un marco de referencia en un punto

2.2. Rotación alrededor de un Eje

Uno de los principales motivos para definir matrices de traslación es para rotar un marco base alrededor de uno o mas de sus ejes. Por ejemplo ,con una rotación de θ respecto al eje x se tienen los siguientes productos

punto:

$$\begin{aligned}
 \hat{\mathbf{Y}}_B \cdot \hat{\mathbf{Y}}_A &= \cos \theta & \hat{\mathbf{Z}}_B \cdot \hat{\mathbf{Y}}_A &= -\sin \theta \\
 \hat{\mathbf{Y}}_B \cdot \hat{\mathbf{Z}}_A &= \sin \theta & \hat{\mathbf{Z}}_B \cdot \hat{\mathbf{Z}}_A &= \cos \theta \\
 \hat{\mathbf{X}}_B \cdot \hat{\mathbf{X}}_A &= 1 & & \\
 \hat{\mathbf{Y}}_B \cdot \hat{\mathbf{X}}_A &= 0 & \hat{\mathbf{Z}}_B \cdot \hat{\mathbf{X}}_A &= 0 \\
 \hat{\mathbf{X}}_B \cdot \hat{\mathbf{Y}}_A &= 0 & \hat{\mathbf{X}}_B \cdot \hat{\mathbf{Z}}_A &= 0
 \end{aligned}$$

Por lo tanto usando la matriz (2.14) y reemplazando lo anterior se obtiene la matriz de rotación en $\psi[\text{rad}]$ sobre el eje $\hat{\mathbf{X}}$ como:

$$R_X(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \psi & -\sin \psi \\ 0 & \sin \psi & \cos \psi \end{bmatrix} \quad (2.15)$$

De manera similar se pueden obtener las matrices de rotación sobre el eje $\hat{\mathbf{Y}}$ y $\hat{\mathbf{Z}}$.

$$R_Y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix} \quad (2.16)$$

$$R_Z(\phi) = \begin{bmatrix} \cos \phi & -\sin \phi & 0 \\ \sin \phi & \cos \phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.17)$$

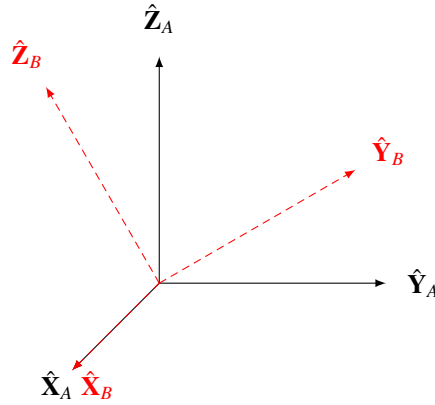


Figura 2.2: Rotación alrededor eje $\hat{\mathbf{X}}_A$

A partir de las matrices de rotación anterior, se pueden hacer secuencias de rotaciones simplemente multiplicando las matrices de rotación en orden. Por ejemplo si se hace una rotación de $\psi = \frac{\pi}{2}$ en $\hat{\mathbf{X}}$ y luego una rotación en $\phi = \frac{\pi}{2}$ en $\hat{\mathbf{Z}}$ esto se traduce como:

$$R_X(\psi = \pi/2)R_Z(\phi = \pi/2) = \begin{bmatrix} \cos \psi & -\sin \psi & 0 \\ \sin \psi & \cos \psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \phi & -\sin \phi & 0 \\ \sin \phi & \cos \phi & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.18)$$

Notar que hay que mantener el orden ya que el producto no es conmutativo

$$R_Z(\phi = \pi/2)R_X(\psi = \pi/2) = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad (2.19)$$

Como se mencionó anteriormente la orientación de un marco en el espacio tiene 3 grados de libertad, por lo que se pueden usar los ángulos (ϕ, θ, ψ) en conjunto con una secuencia de las matrices anteriores. Una de las convenciones mas comunes es realizar una secuencia de rotaciones, como por ejemplo la secuencia **ZYX** en donde los ángulos (ϕ, θ, ψ) se conocen como *Roll*, *Pitch* y *Yaw* [10] o ángulos de Euler, tres parámetros para describir una orientación, relacionado a sus 3 grados de libertad.

Esta secuencia de rotaciones, como se muestra en la figura (2.3), se hace de manera que primero se rota una cantidad ϕ alrededor del eje Z, luego se rota θ alrededor del nuevo eje Y (Y') producto de la rotación anterior, y por ultimo se rota ψ alrededor del nuevo eje X (X''), es decir una rotación **ZY'X''**, por lo que se habla que las rotaciones son a partir de los ejes intrínsecos, en vez de los ejes absolutos de un marco base original. Si se hacen las rotaciones en términos de los ejes absolutos usando los mismos ángulos, pero en el orden XYZ se llega al mismo resultado [7][9].

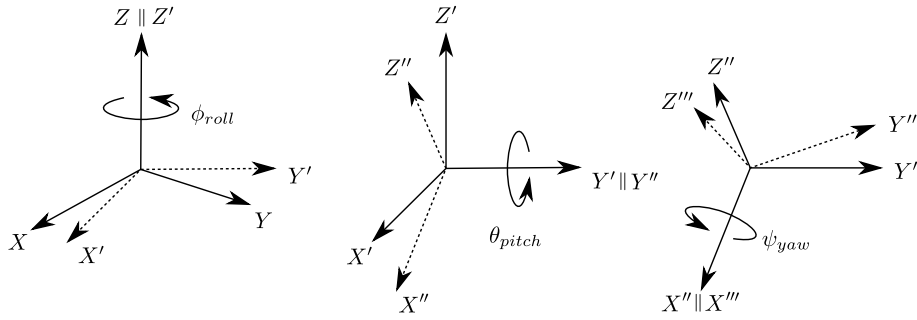


Figura 2.3: Rotaciones Roll, Pitch y Yaw

La matriz resultante de las rotaciones anteriores es la siguiente.

$${}^A R_B = R_Z(\phi_{roll})R_Y(\theta_{pitch})R_X(\psi_{yaw}) = \begin{bmatrix} c_\phi c_\theta & c_\phi s_\theta s_\psi - s_\phi c_\psi & c_\phi s_\theta c_\psi + s_\phi s_\psi \\ s_\phi c_\theta & s_\phi s_\theta s_\psi + c_\phi c_\psi & s_\phi s_\theta c_\psi - c_\phi s_\psi \\ -s_\theta & c_\theta s_\psi & c_\theta c_\psi \end{bmatrix} \quad (2.20)$$

Donde $c_\alpha = \cos \alpha$ y $s_\beta = \sin \beta$.

Luego, para despejar los valores de *Roll*, *Pitch* y *Yaw* a partir de la matriz de rotación se realiza el siguiente procedimiento:

- Si $r_{31} \neq \pm 1$, entonces

$$\theta_{pitch} = \text{atan2}\left(-r_{31}, \sqrt{r_{11}^2 + r_{21}^2}\right) \quad (2.21)$$

$$\phi_{roll} = \text{atan2}(r_{21}, r_{11}) \quad (2.22)$$

$$\psi_{yaw} = \text{atan2}(r_{32}, r_{33}) \quad (2.23)$$

- Si $r_{31} = -1$

$$\theta_{pitch} = \frac{\pi}{2} \quad (2.24)$$

$$\phi_{roll} = 0 \quad (2.25)$$

$$\psi_{yaw} = \text{atan2}(r_{12}, r_{22}) \quad (2.26)$$

$$(2.27)$$

- Si $r_{31} = 1$

$$\theta_{pitch} = -\frac{\pi}{2} \quad (2.28)$$

$$\phi_{roll} = 0 \quad (2.29)$$

$$\psi_{yaw} = -atan2(r_{12}, r_{22}) \quad (2.30)$$

$$(2.31)$$

En general la secuencia ZYX es la mas común para representar rotaciones en robótica, sin embargo se pueden usar otras combinaciones de ejes, como por ejemplo ZYZ que también es común.

Tres parámetros son suficientes para describir una rotación, sin embargo, en particular para robots Mitsubishi de 5 DOF, como el robot RV-2AJ, se tiene un grado de libertad, por lo que describir una orientación de su EF se puede hacer a través de solo 2 parámetros; A y B. El parámetro B representa el ángulo que se forma desde el eje Z del marco base al eje Z del EF, y el parámetro A se relaciona a la rotación respecto al eje Z del EF. Estos parámetros serán discutidos en profundidad mas adelante.

Ya teniendo definida la posición y orientación de un punto, se puede juntar todo en la **Matriz de transformación Homogénea** que describe el marco de referencia {B} respecto al marco {A} o simplemente una configuración de orientación y posición en el espacio:

$${}^A T_B = \left[\begin{array}{ccc|c} {}^A R_B & & & {}^A P_B \\ \hline 0 & 0 & 0 & 1 \end{array} \right] = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.32)$$

En donde ${}^A R_B$ es la rotación desde el marco de referencia {A} al {B} y ${}^A P_B$ es el vector desde el origen del marco {A} (O_A) al origen del marco {B} (O_B), esto representado en la figura (2.4).

Con la Matriz de transformación Homogénea es posible describir un punto de un marco a otro como:

$${}^A P = {}^A T_B {}^B P \quad (2.33)$$

Donde ${}^A P$ y ${}^B P$ son vectores de 4×1 donde el ultimo elemento es un 1

$${}^A P = \begin{bmatrix} p_{xA} \\ p_{yA} \\ p_{zA} \\ 1 \end{bmatrix} \quad (2.34)$$

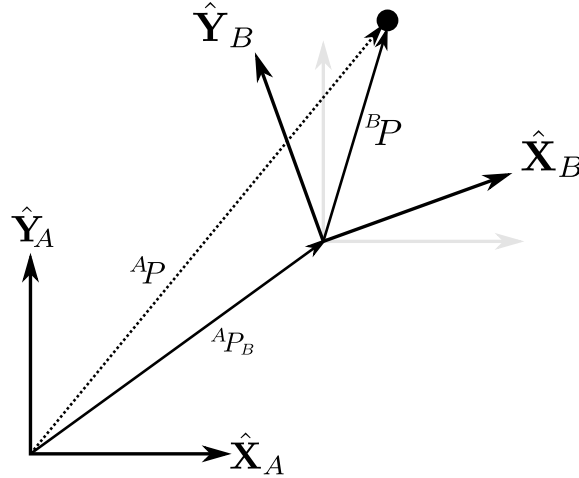


Figura 2.4: Relación entre punto y marco de referencia

El conjunto de matrices de transformación homogénea de 4×4 conforman el **grupo Euclídeo especial** $SE(3)$ en el cual la matriz $R \in SO(3)$ y el vector de posición $P \in \mathbb{R}^3$. Esta matriz es invertible y tiene la forma:

$$T^{-1} = \left[\begin{array}{ccc|c} R & P \\ \hline 0 & 0 & 0 & 1 \end{array} \right]^{-1} = \left[\begin{array}{ccc|c} R^T & -R^T P \\ \hline 0 & 0 & 0 & 1 \end{array} \right]^{-1} \quad (2.35)$$

La multiplicación o composición de 2 matrices de transformación homogénea también es una matriz de transformación homogénea:

$$T_1 T_2 = \left[\begin{array}{ccc|c} R_1 & P_1 \\ \hline 0 & 0 & 0 & 1 \end{array} \right] \left[\begin{array}{ccc|c} R_2 & P_2 \\ \hline 0 & 0 & 0 & 1 \end{array} \right] = \left[\begin{array}{ccc|c} R_1 R_2 & P_1 + R_1 P_2 \\ \hline 0 & 0 & 0 & 1 \end{array} \right] \quad (2.36)$$

A partir de esta matriz se pueden definir entonces rotaciones y traslaciones elementales:

$$Trans_x(a) = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad Trans_y(b) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad Trans_z(c) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.37)$$

$$Rot_x(\psi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c_\psi & -s_\psi & 0 \\ 0 & s_\psi & c_\psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad Rot_y(\theta) = \begin{bmatrix} c_\theta & 0 & s_\theta & 0 \\ 0 & 1 & 0 & 0 \\ -s_\theta & 0 & c_\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad Rot_z(\phi) = \begin{bmatrix} c_\phi & -s_\phi & 0 & 0 \\ s_\phi & c_\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.38)$$

2.3. Cadena Cinemática

Un brazo robótico se conforma por una serie de eslabones(Link) que se conectan entre si a través de articulaciones(Joint). Las articulaciones son generalmente de dos tipo; Rotacionales o Prismáticos. Las Articulaciones rotacionales permiten el movimiento rotacional alrededor del eje articular y las Prismáticas

permiten el movimiento traslacional en dirección del eje articular.

Este tipo de articulación poseen solo un grado de libertad, es decir, para la rotacional solo puede formar 1 ángulo y la prismática solo se mueve en una dirección. De todas maneras existen otros tipos de articulaciones como la cilíndrica, universal y esférica que tienen mas de 1 grados de libertad.

La unión de varios eslabones a través de articulaciones forman una estructura mecánica llamada cadena cinemática, que es básicamente la estructura de un brazo robotico, y esta es caracterizada por sus grados de libertad (DOF). Una forma de generalizar el numero de DOF de un brazo robótico es usar la formula de Grübler que se explica a continuación:

Para un mecanismo de N eslabones, J articulaciones, $m=6$ para un mecanismo espacial ($m=3$ para uno planar), f_i los grados de libertad de la articulación i , y c_i las restricciones de la articulación i donde $f_i + c_i = m$, la formula de Grübler es la siguiente.

$$DOF = \underbrace{m(N-1)}_{\text{DOF si los eslabones no estuvieran unidos}} - \underbrace{\sum_{i=1}^J c_i}_{\text{restricciones de la articulación}} \quad (2.39)$$

$$= m(N-1) - \sum_{i=1}^J (m - f_i) \quad (2.40)$$

$$= m(N-1-J) + \sum_{i=1}^J (f_i) \quad (2.41)$$

Por ejemplo para el robot RV-2AJ, que se mueve en el espacio ($m=6$), tiene $N-1=5$ eslabones, y $J=5$ articulaciones, cada una con $f=1$ grado de libertad, se tiene:

$$\begin{aligned} DOF_{RV-2AJ} &= m(N-1-J) + \sum_{i=1}^J (f_i) \\ &= 6 \times (6-1-5) + 5 \times 1 \\ &= 5 \end{aligned} \quad (2.42)$$

Debido a que, como se mencionó anteriormente, un punto en el espacio tiene 6 grados de libertad total, el robot RV-2AJ va a tener algunas limitaciones para llevar con su EF a algunas configuraciones. Cada articulación tiene un rango de posición angular, que se conoce como coordenada articular. Diferentes configuraciones de coordenadas articulares representan distintas configuraciones del EF. Para el caso del Robot RV-2AJ se tienen 5 articulaciones cuyas coordenadas articulares se denotan como $\mathbf{q} = [q_1 \ q_2 \ q_3 \ q_4 \ q_5]^T$ y tienen los rangos mostrados en la tabla (2.1).

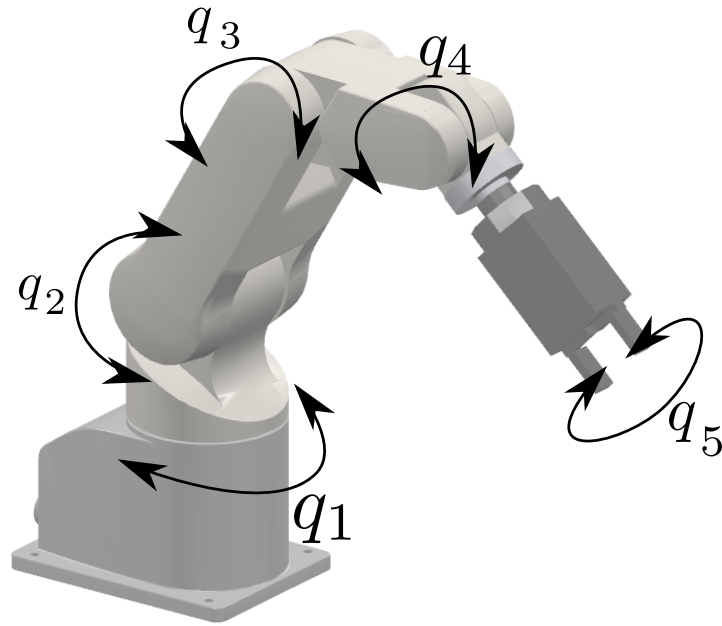


Figura 2.5: Articulaciones robot RV-2AJ

Articulación	Rango(°)
q_1	$[-150, 150]$
q_2	$[-60, 120]$
q_3	$[-110, 120]$
q_4	$[-90, 90]$
q_5	$[-200, 200]$

Tabla 2.1: Rango de articulaciones RV-2AJ

2.4. Parámetros Denavit–Hartenberg

Para describir la cadena cinemática de un brazo robótico y en específico la configuración del EF respecto a su marco de referencia base, se puede usar la composición de varias matrices de transformación homogénea. Por ejemplo para la posición desde un marco de referencia $\{\mathbf{n}\}$ a un marco de referencia $\{\mathbf{0}\}$, con una cadena cinemática que tiene $\mathbf{n}$ articulaciones enumeradas con $\mathbf{j}=1, \dots, \mathbf{n}$, y $\mathbf{n}+1$ eslabones enumerados con $\mathbf{l}=0, \dots, \mathbf{n}$, donde cada articulación $\mathbf{j}$ conecta el eslabon $\mathbf{l}-1$ con el $\mathbf{l}$ y a su vez cada eslabon $\mathbf{l}$ conecta las articulaciones $\mathbf{j}$ a la $\mathbf{j}+1$, considerando que el eslabon $\mathbf{l}=0$ es la base y el $\mathbf{l}=\mathbf{n}$ es el EF

$${}^0P = ({}^0T_1 {}^1T_2 \dots {}^{n-1}T_n) {}^nP \quad (2.43)$$

$$= \left(\prod_{i=1}^n {}^{i-1}T_i \right) {}^nP \quad (2.44)$$

La cinemática directa busca encontrar la posición y orientación del EF de una cadena cinemática a partir de sus coordenadas articulares. Debido a que cada articulación es variable por sus coordenadas articulares, las matrices de transformación homogénea de cada articulación también son variables, por lo que la cinemática directa se describe por la matriz de transformación desde el marco del EF $\{\mathbf{n}\}$ al marco base $\{\mathbf{0}\}$ como:

$${}^0T_n(\mathbf{q}) = {}^0T_1(q_1) {}^1T_2(q_2) \dots {}^{n-1}T_n(q_n) \quad (2.45)$$

$$= \prod_{i=1}^n {}^{i-1}T_i(q_i) \quad (2.46)$$

Para describir cada marco ${}^i T_{i+1}(q_i)$ se ocupa la convención de Denavit-Hartenberg, la cual con solo cuatro parámetros es posible describir cada matriz de la siguiente forma

$${}^{i-1}T_i = Rot_z(\theta_i) Trans_z(d_i) Trans_x(a_i) Rot_x(\alpha_i) \quad (2.47)$$

$$= \begin{bmatrix} c_{\theta_i} & -s_{\theta_i} & 0 & 0 \\ s_{\theta_i} & c_{\theta_i} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ a_i & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c_{\alpha_i} & -s_{\alpha_i} & 0 \\ 0 & s_{\alpha_i} & c_{\alpha_i} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} c_{\theta_i} & -s_{\theta_i}c_{\alpha_i} & s_{\theta_i}s_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i}c_{\alpha_i} & -c_{\theta_i}s_{\alpha_i} & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.48)$$

Los cuatro parámetros $\theta_i, a_i, d_i, \alpha_i$ son suficientes para describir la relación geométrica entre dos marcos de referencia, lo que entra en contradicción con lo establecido anteriormente de que 6 parámetros describen la posición y orientación de un marco de referencia, pero esto se explica porque existen 2 restricciones extras: la primera es que el eje x_i debe ser perpendicular al eje z_{i-1} y la segunda es que el eje x_i debe intersectar con el eje z_{i-1} .

Tres de estos parámetros son constantes y uno es variable. Como se mostró anteriormente con la variable q_i , para articulaciones prismáticas, el parámetro variable es d_i y para rotacionales es θ_i .

Para esta representación, el eje z_i de cada marco es el eje de rotación o traslación de la articulación y este representa la actuación de la articulación q_{i+1} , debido a esto es que cada término θ_i es el mismo que q_i , solo escrito con diferente notación. Estos parámetros expresan las siguientes relaciones mostradas en la figura (2.6).

- a_i : Distancia entre la intersección de los ejes x_i y z_{i-1} a lo largo del eje x_i .
- d_i : Distancia entre la intersección de los ejes x_i y z_{i-1} a lo largo del eje z_{i-1} .
- α_i : Ángulo entre z_{i-1} y z_i en torno a x_i .
- θ_i : Ángulo entre x_{i-1} y x_i en torno a z_i .

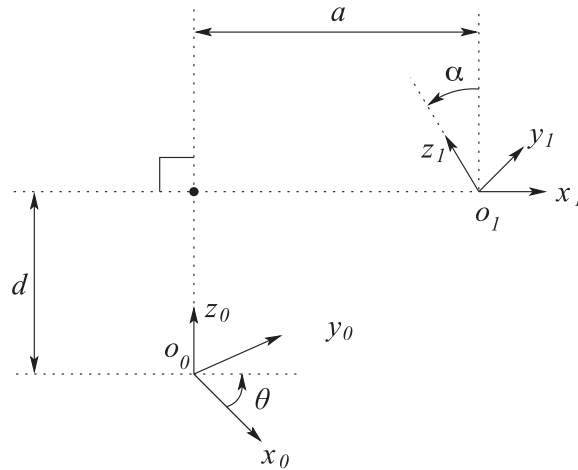


Figura 2.6: Definición de parámetros Denavit–Hartenberg

Para obtener estos parámetros se puede proceder con el siguiente algoritmo obtenido del libro *Robot, Modeling and Control* [6]:

Paso 1 Ubicar los ejes de Giro $z_0 \dots z_{n-1}$

Paso 2 Establecer el marco base $(O_0 x_0 y_0 z_0)$ con la regla de la mano derecha

for $i = 1, \dots, n - 1$ hacer pasos 3 al 5

Paso 3 Establecer el origen O_i donde se intersece z_i con z_{i-1} o si son paralelos, en cualquier punto en z_i

Paso 4 Si z_i y z_{i-1} son paralelos establecer x_i en la línea que cruza O_{i-1} con O_i , si z_i y z_{i-1} se intersectan establecer x_i como la normal del plano formado por z_i y z_{i-1} . Si z_i y z_{i-1} no son coplanares, establecer x_i como el segmento mas corto que unen perpendicularmente los ejes z_i y z_{i-1} .

Paso 5 Establecer y_i de manera que se cumpla la regla de la mano derecha

Paso 6 Establecer el origen del EF O_n con z_n apuntando en saliente.

Paso 7 Crear la tabla con los parámetros $\theta_i, a_i, d_i, \alpha_i$

Paso 8 Crear las matrices ${}^i T_{i+1}(q_i)$ de la ecuación (2.48)

Paso 9 Formar ${}^0 T_n(\mathbf{q})$, la matriz de transformación homogénea del EF al marco base de la ecuación (2.45)

En la figura (2.7) se muestra la obtención de los parámetros Denavit–Hartenberg para el robot RV-2AJ a partir del algoritmo anterior.

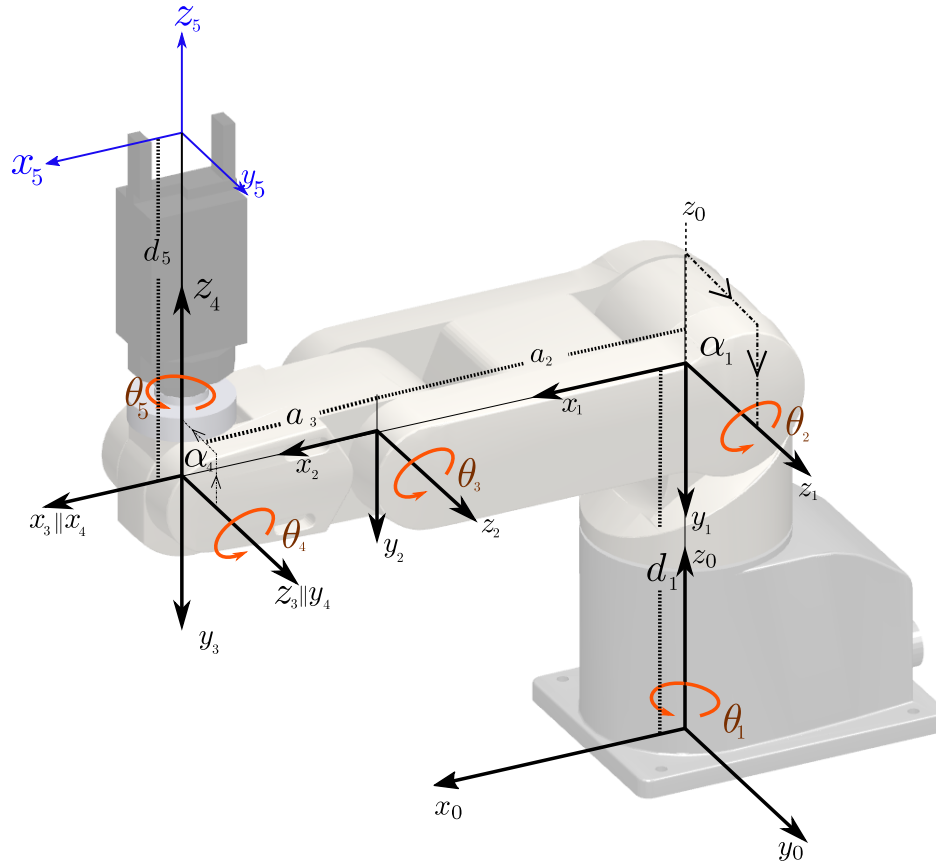


Figura 2.7: Parámetros Denavit–Hartenberg para el robot RV-2AJ

En la documentación del robot [4], este debe tener una postura tal que para las coordenadas articulares $\mathbf{q} = [q_1, q_2, q_3, q_4, q_5] = [\theta_1, \theta_2, \theta_3, \theta_4, \theta_5] = [0, 0, 0, 0, 0]$ el robot esté totalmente extendido para alcanzar la

mayor altura con su EF, por lo que hay que agregar un offset o desfase entre los valores de θ_2 y θ_4 . Teniendo en consideración eso la tabla generada es la siguiente

Articulación	$\theta_i(^{\circ})$	$\alpha_i(^{\circ})$	a(m)	d(m)
1	θ_1	α_1	0	d_1
2	$\theta_2 - \theta_{2off}$	0	a_2	0
3	θ_3	0	a_3	0
4	$\theta_4 + \theta_{4off}$	α_4	0	0
5	θ_5	0	0	d_5

Tabla 2.2: Parámetros Denavit–Hartenberg RV-2AJ

Reemplazando con los valores reales

Articulación	$\theta_i(^{\circ})$	$\alpha_i(^{\circ})$	a(m)	d(m)
1	θ_1	-90°	0	0.3
2	$\theta_2 - 90^{\circ}$	0	0.25	0
3	θ_3	0	0.16	0
4	$\theta_4 + 90^{\circ}$	90°	0	0
5	θ_5	0	0	0.072+0.21(garra)

Tabla 2.3: Parámetros Denavit–Hartenberg RV-2AJ con valores reales

Luego las matrices ${}^i T_{i+1}(q_i)$ en base a (2.48) son las siguientes:

$${}^0 T_1(q_1) = \begin{bmatrix} c_{q_1} & 0 & -s_{q_1} & 0 \\ s_{q_1} & 0 & c_{q_1} & 0 \\ 0 & -1 & 0 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.49)$$

$${}^1 T_2(q_2) = \begin{bmatrix} s_{q_2} & c_{q_2} & 0 & a_2 s_{q_2} \\ -c_{q_2} & s_{q_2} & 0 & -a_2 c_{q_2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.50)$$

$${}^2 T_3(q_3) = \begin{bmatrix} c_{q_3} & -s_{q_3} & 0 & a_3 c_{q_3} \\ s_{q_3} & c_{q_3} & 0 & a_3 s_{q_3} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.51)$$

$${}^3 T_4(q_4) = \begin{bmatrix} -s_{q_4} & 0 & c_{q_4} & 0 \\ c_{q_4} & 0 & s_{q_4} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.52)$$

$${}^4 T_5(q_5) = \begin{bmatrix} c_{q_5} & -s_{q_5} & 0 & 0 \\ s_{q_5} & c_{q_5} & 0 & 0 \\ 0 & 0 & 1 & d_5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.53)$$

Luego:

$${}^0 T_5(\mathbf{q}) = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.54)$$

Donde los valores de r_{ij} y p_{xyz} , usando la notación $C_{q_1 q_2 \dots q_n} = \cos(q_1 + q_2 + \dots + q_n)$, son:

$$\begin{aligned}
 r_{11} &= C_{q_1} C_{q_2 q_3 q_4} C_{q_5} - S_{q_1} S_{q_5} \\
 r_{21} &= S_{q_1} C_{q_2 q_3 q_4} C_{q_5} + C_{q_1} S_{q_5} \\
 r_{31} &= -S_{q_2 q_3 q_4} C_{q_5} \\
 r_{12} &= -C_{q_1} C_{q_2 q_3 q_4} S_{q_5} - S_{q_1} C_{q_5} \\
 r_{22} &= -S_{q_1} C_{q_2 q_3 q_4} S_{q_5} + C_{q_1} C_{q_5} \\
 r_{32} &= S_{q_2 q_3 q_4} S_{q_5} \\
 r_{13} &= C_{q_1} S_{q_2 q_3 q_4} \\
 r_{23} &= S_{q_1} S_{q_2 q_3 q_4} \\
 r_{33} &= C_{q_2 q_3 q_4} \\
 p_x &= C_{q_1} (a_2 S_{q_2} + a_3 S_{q_2 q_3} + d_5 S_{q_2 q_3 q_4}) \\
 p_y &= S_{q_1} (a_2 S_{q_2} + a_3 S_{q_2 q_3} + d_5 S_{q_2 q_3 q_4}) \\
 p_z &= d_1 + a_2 C_{q_2} + a_3 C_{q_2 q_3} + d_5 C_{q_2 q_3 q_4}
 \end{aligned} \tag{2.55}$$

$${}^0T_5(\mathbf{q}) = \begin{bmatrix} C_{q_1} C_{q_2 q_3 q_4} C_{q_5} - S_{q_1} S_{q_5} & -C_{q_1} C_{q_2 q_3 q_4} S_{q_5} - S_{q_1} C_{q_5} & C_{q_1} S_{q_2 q_3 q_4} & C_{q_1} (a_2 S_{q_2} + a_3 S_{q_2 q_3} + d_5 S_{q_2 q_3 q_4}) \\ S_{q_1} C_{q_2 q_3 q_4} C_{q_5} + C_{q_1} S_{q_5} & -S_{q_1} C_{q_2 q_3 q_4} S_{q_5} + C_{q_1} C_{q_5} & S_{q_1} S_{q_2 q_3 q_4} & S_{q_1} (a_2 S_{q_2} + a_3 S_{q_2 q_3} + d_5 S_{q_2 q_3 q_4}) \\ -S_{q_2 q_3 q_4} C_{q_5} & S_{q_2 q_3 q_4} S_{q_5} & C_{q_2 q_3 q_4} & d_1 + a_2 C_{q_2} + a_3 C_{q_2 q_3} + d_5 C_{q_2 q_3 q_4} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.56}$$

De esta matriz es ya posible encontrar la posición y orientación del EF para cualquier combinación de coordenadas articulares, donde también se pueden extraer los parámetros de rotación en términos de Roll ,Pitch y Yaw de las ecuaciones (2.21), (2.22), (2.23).

Por ultimo , como se comentó anteriormente, los robots Mitsubishi de 5 DOF usan los parámetros de rotación A y B en vez de Roll ,Pitch y Yaw, donde B corresponde al ángulo que forma el eje Z del EF con el eje Z del marco base y A es una función que describe la rotación sobre el eje Z del EF, y se calculan con las siguientes ecuaciones.

$$B = q_2 + q_3 + q_4 \tag{2.57}$$

$$A = q_5 + q_1 \cos(B) \tag{2.58}$$

2.5. Espacio de trabajo

Las coordenadas articulares al estar limitadas en rangos acotados, hacen que al aplicar la cinemática directa, las posiciones cartesianas también se encuentren acotadas en el espacio. Este espacio o volumen acotado se le denomina Espacio de trabajo o “**Workspace**”.

Para determinar cual es su forma ,dados los limites de cada articulación mostrados en la tabla (2.1), es que se crea un programa en Matlab, donde se fija $q_1 = q_5 = 0$ y con el resto de articulaciones se crea un conjunto de puntos que se guardan en una grilla de 3 dimensiones con el comando *ndgrid* de Matlab. Esta grilla contiene combinaciones de q_2, q_3 y q_4 de manera que abarquen lo mas completamente posible el plano que se genera con $q_1 = q_5 = 0$.

Luego ,a estos conjuntos de articulaciones se aplica la cinemática directa para obtener ,en este caso , los valores de X y Z ya que Y=0 con $q_1 = 0$. Obtenidos los puntos en X y Z se aplica la función *boundary* la cual

calcula los índices de los puntos que se encuentran en la frontera del conjunto de puntos. Así, para el Robot RV-2AJ sin considerar la garra se obtiene la siguiente imagen del Workspace planar.

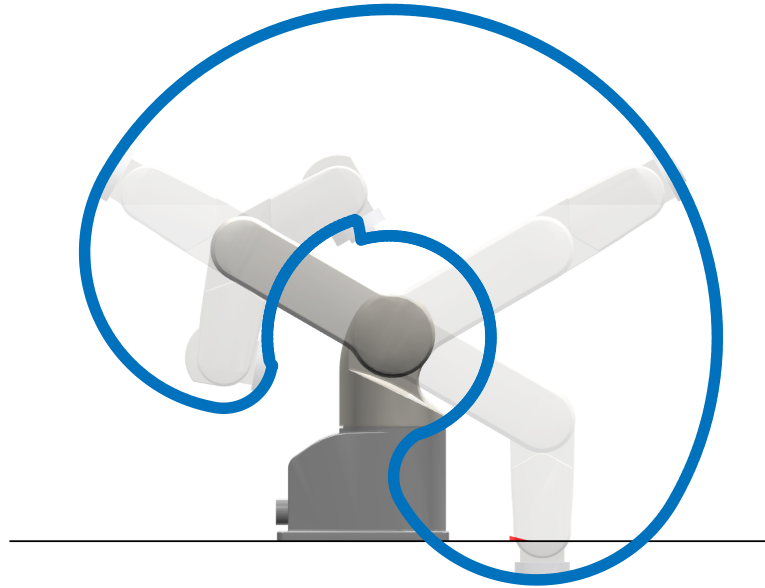


Figura 2.8: Workspace 2D sin garra con $q_1 = 0$

Se puede observar en la imagen algunas posiciones del robot en su límite, además se puede observar que algunas partes del workspace está dentro de la misma estructura del robot, por lo que se debe tener especial cuidado en esa zona para no hacer chocar el robot consigo mismo o con una mesa si hubiera en $Z=0$.

Se hace lo mismo pero ahora considerando la garra del robot, cuyo largo es de $0.21[m]$, igual a la del robot real del laboratorio de control industrial.

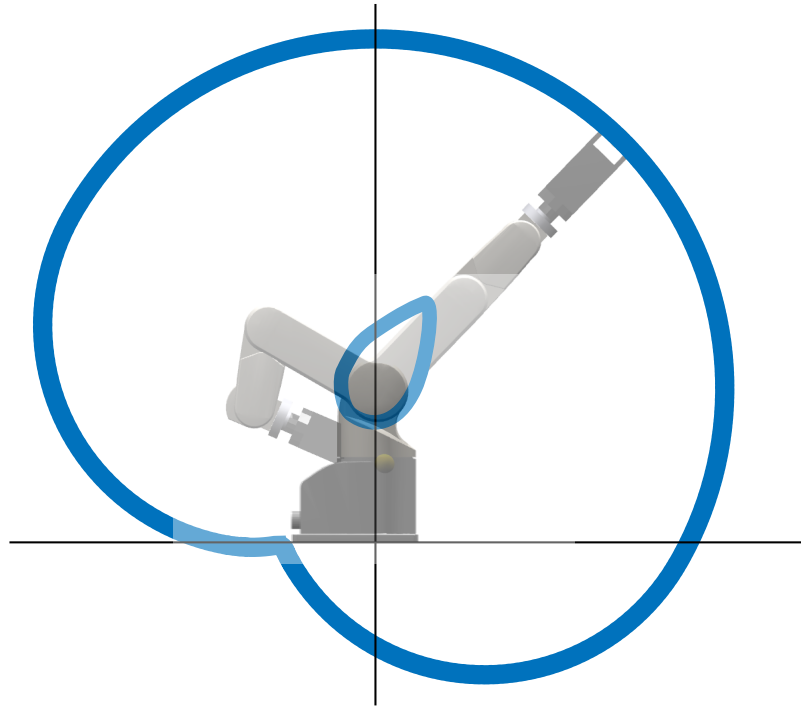


Figura 2.9: Workspace 2D con garra y con $q_1 = 0$

Se puede observar que al tener la garra el workspace es mas grande, sin embargo existen mas configuraciones de las articulaciones que hacen que el robot choque con si mismo.

Para obtener el workspace en 3 dimensiones se puede girar lo obtenido en los rangos de q_1 obteniéndose las siguientes figuras.

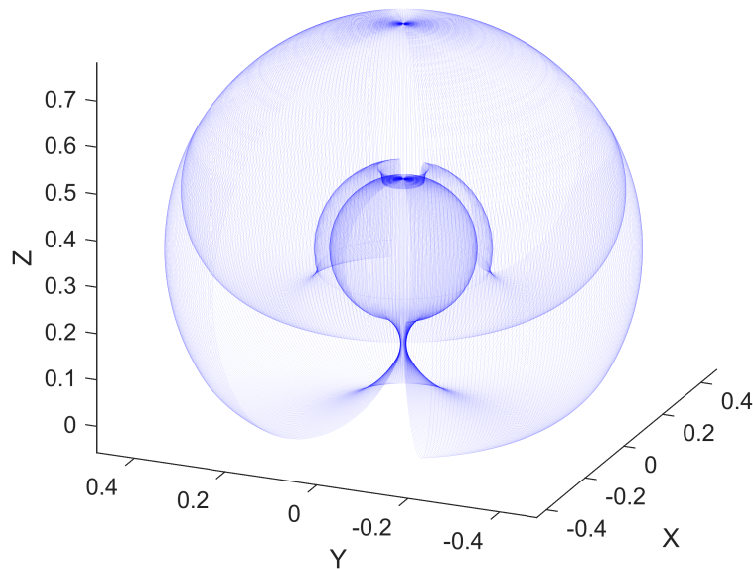


Figura 2.10: Workspace 3D sin garra

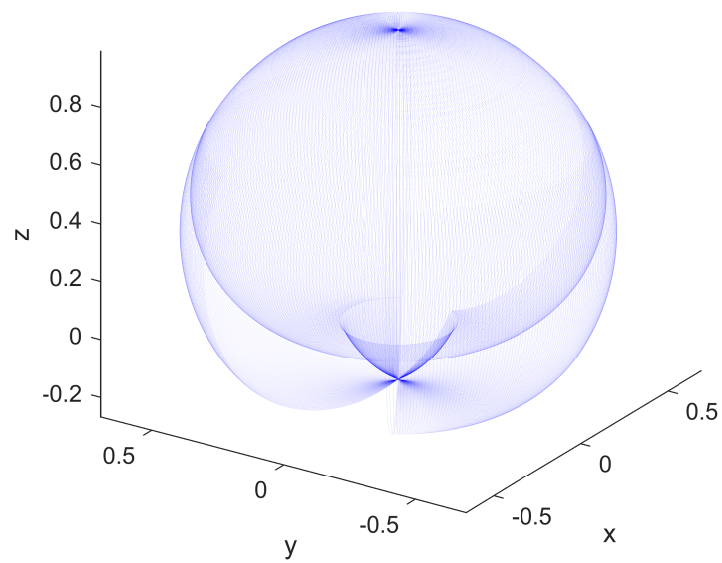


Figura 2.11: Workspace 3D con garra

En sí, el workspace representa solo los puntos a los que se puede llevar el EF del robot, mas no considera su orientación.

3 | Cinemática diferencial

Así como el problema cinemático relaciona la posición de las articulaciones con la posición y orientación del end-effector, la cinemática diferencial relaciona las velocidades lineales y angulares (fig 3.1) del EF con las velocidades de las articulaciones a través del Jacobiano

3.1. Jacobiano Geométrico

El Jacobiano es una matriz que contiene las derivadas parciales de posición y orientación respecto a cada articulación, por lo tanto, para un robot de n articulaciones que se mueve en el espacio describiendo una posición y orientación, el Jacobiano es de dimensión $6 \times n$.

$$\mathbf{J}(\mathbf{q}) = \begin{bmatrix} \frac{\partial \mathbf{f}(\mathbf{q})}{\partial q_1} & \dots & \frac{\partial \mathbf{f}(\mathbf{q})}{\partial q_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1(\mathbf{q})}{\partial q_1} & \dots & \frac{\partial f_1(\mathbf{q})}{\partial q_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m(\mathbf{q})}{\partial q_1} & \dots & \frac{\partial f_m(\mathbf{q})}{\partial q_n} \end{bmatrix} \quad (3.1)$$

$$\mathbf{v}_n = \begin{bmatrix} \dot{p}_n \\ \omega_n \end{bmatrix} = \begin{bmatrix} \dot{x}_n \\ \dot{y}_n \\ \dot{z}_n \\ \omega_{xn} \\ \omega_{yn} \\ \omega_{zn} \end{bmatrix} = \begin{bmatrix} J_1 & \dots & J_n \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \vdots \\ \dot{q}_n \end{bmatrix} = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}} \quad (3.2)$$

$$\mathbf{J} = \begin{bmatrix} J_P \\ J_O \end{bmatrix} \quad (3.3)$$

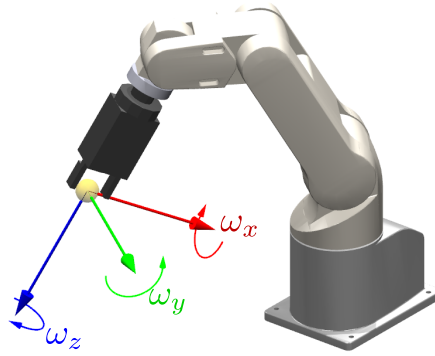


Figura 3.1: Ejes del EF x(rojo),y(verde),z(azul) y rotaciones a través de ellos

Debido a que la cinemática directa describe la posición y orientación en términos de las articulaciones, el jacobiano también puede describir separadamente la velocidad lineal y angular del EF. Una forma bastante eficiente para robots con articulaciones rotacionales es describir cada columna del jacobiano con la matriz (3.4) que describe la contribución a la velocidad lineal (J_{Pi}) y angular (J_{Oi}) de cada articulación a partir de las matrices que se obtienen a través de los parámetros de Denavit–Hartenberg obtenidas en (2.50) a (2.53) y donde cada elemento de J_i se muestra en (3.5).

$$J_i = \begin{bmatrix} J_{Pi} \\ J_{Oi} \end{bmatrix} = \begin{bmatrix} {}^0Z_{i-1} \times ({}^0p_n - {}^0p_{i-1}) \\ {}^0Z_{i-1} \end{bmatrix} \quad (3.4)$$

$${}^0T_n(q) = {}^0T_1(q) \cdot {}^1T_2(q) \cdot \dots \cdot {}^{n-1}T_n(q) = \left[\begin{array}{ccc|c} {}^0X_n & {}^0Y_n & {}^0Z_n & {}^0p_n \\ \hline 0 & 0 & 0 & 1 \end{array} \right] \quad (3.5)$$

El elemento $J_{Oi} = {}^0Z_{i-1}$ de dimensión 3×1 , corresponde al eje de rotación de cada articulación respecto al origen $\{0\}$ que se obtiene sabiendo que $\omega = Z\dot{q}$, es decir la velocidad angular es la misma velocidad de la articulación q al girar en su eje Z .

El elemento $J_{Pi} = {}^0Z_{i-1} \times ({}^0p_n - {}^0p_{i-1})$ de dimensión 3×1 proviene de saber que la velocidad lineal (v) es el producto cruz entre la velocidad angular (ω) y el vector del origen al punto de rotación (r) es decir, $v = \omega \times r$, en donde para alguna articulación i , $\omega_i = {}^0Z_{i-1}$ y la distancia entre el origen y una articulación i es $r = {}^0p_n - {}^0p_{i-1}$

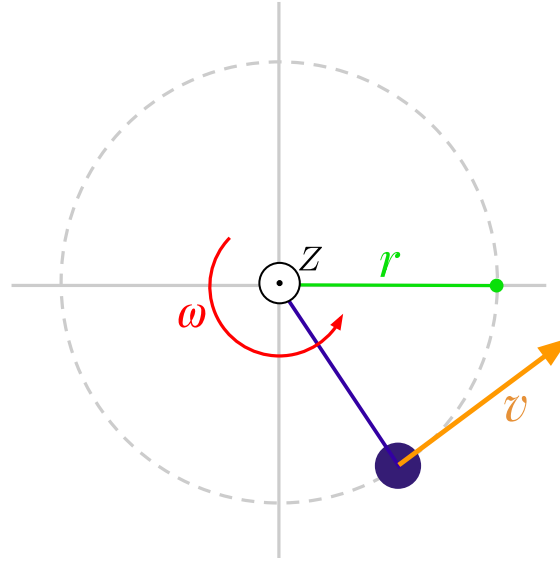


Figura 3.2: Relación entre velocidad lineal y angular respecto al eje Z

Entonces para obtener el Jacobiano primero se obtienen las matrices de transformación homogénea de cada articulación respecto a la base.

$${}^0T_1(q) = \begin{bmatrix} c_{q_1} & 0 & -s_{q_1} & 0 \\ s_{q_1} & 0 & c_{q_1} & 0 \\ 0 & -1 & 0 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.6)$$

$${}^0T_2(q) = \begin{bmatrix} c_{q_1}s_{q_2} & c_{q_1}c_{q_2} & -s_{q_1} & a_2c_{q_1}s_{q_2} \\ s_{q_1}s_{q_2} & s_{q_1}c_{q_2} & c_{q_1} & a_2s_{q_1}s_{q_2} \\ c_{q_2} & -s_{q_2} & 0 & d_1 + a_2c_{q_2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.7)$$

$${}^0T_3(q) = \begin{bmatrix} c_{q_1}s_{q_2q_3} & c_{q_1}c_{q_2q_3} & -s_{q_1} & c_{q_1}(a_2s_{q_2} + a_3s_{q_2q_3}) \\ s_{q_1}s_{q_2q_3} & s_{q_1}c_{q_2q_3} & c_{q_1} & s_{q_1}(a_2s_{q_2} + a_3s_{q_2q_3}) \\ c_{q_2q_3} & -s_{q_2q_3} & 0 & d_1 + a_2c_{q_2} + a_3c_{q_2q_3} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.8)$$

$${}^0T_4(q) = \begin{bmatrix} c_{q_1}c_{q_2q_3q_4} & -s_{q_1} & c_{q_1}s_{q_2q_3q_4} & c_{q_1}(a_2s_{q_2} + a_3s_{q_2q_3}) \\ s_{q_1}c_{q_2q_3q_4} & c_{q_1} & s_{q_1}s_{q_2q_3q_4} & s_{q_1}(a_2s_{q_2} + a_3s_{q_2q_3}) \\ c_{q_2q_3} & -s_{q_2q_3} & 0 & d_1 + a_2c_{q_2} + a_3c_{q_2q_3} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.9)$$

$${}^0T_5(q) = \begin{bmatrix} c_{q_1}c_{q_2q_3q_4}c_{q_5} - s_{q_1}s_{q_5} & -c_{q_1}c_{q_2q_3q_4}s_{q_5} - s_{q_1}c_{q_5} & c_{q_1}s_{q_2q_3q_4} & c_{q_1}(a_2s_{q_2} + a_3s_{q_2q_3} + d_5s_{q_2q_3q_4}) \\ s_{q_1}c_{q_2q_3q_4}c_{q_5} + c_{q_1}s_{q_5} & -s_{q_1}c_{q_2q_3q_4}s_{q_5} + c_{q_1}c_{q_5} & s_{q_1}s_{q_2q_3q_4} & s_{q_1}(a_2s_{q_2} + a_3s_{q_2q_3} + d_5s_{q_2q_3q_4}) \\ -s_{q_2q_3q_4}c_{q_5} & s_{q_2q_3q_4}s_{q_5} & c_{q_2q_3q_4} & d_1 + a_2c_{q_2} + a_3c_{q_2q_3} + d_5c_{q_2q_3q_4} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.10)$$

Luego al aplicar la formula (3.4) se construye el jacobiano.

$$J = [J_1 \quad J_2 \quad J_3 \quad J_4 \quad J_5] \quad (3.11)$$

$$= \begin{bmatrix} -s_{q_1}(a_3 s_{q_2 q_3} + a_2 s_{q_2} + d_5 s_{q_2 q_3 q_4}) & c_{q_1}(a_3 c_{q_2 q_3} + a_2 c_{q_2} + d_5 c_{q_2 q_3 q_4}) & c_{q_1}(a_3 c_{q_2 q_3} + d_5 c_{q_2 q_3 q_4}) & d_5 c_{q_1} c_{q_2 q_3 q_4} & 0 \\ c_{q_1}(a_3 s_{q_2 q_3} + a_2 s_{q_2} + d_5 s_{q_2 q_3 q_4}) & s_{q_1}(a_3 c_{q_2 q_3} + a_2 c_{q_2} + d_5 c_{q_2 q_3 q_4}) & s_{q_1}(a_3 c_{q_2 q_3} + d_5 c_{q_2 q_3 q_4}) & d_5 s_{q_1} c_{q_2 q_3 q_4} & 0 \\ 0 & -a_3 s_{q_2 q_3} - a_2 s_{q_2} - d_5 s_{q_2 q_3 q_4} & -a_3 s_{q_2 q_3} - d_5 s_{q_2 q_3 q_4} & -d_5 s_{q_2 q_3 q_4} & 0 \\ 0 & -s_{q_1} & -s_{q_1} & -s_{q_1} & c_{q_1} s_{q_2 q_3 q_4} \\ 0 & c_{q_1} & c_{q_1} & c_{q_1} & s_{q_1} s_{q_2 q_3 q_4} \\ 1 & 0 & 0 & 0 & c_{q_2 q_3 q_4} \end{bmatrix} \quad (3.12)$$

A este jacobiano en particular se le denomina “Jacobiano Geométrico” y una consideración importante es que la parte de la orientación es respecto a las rotaciones absolutas del marco de referencia del EF respecto a la base, sin embargo, como se vio con anterioridad, las rotaciones se escriben en términos de la sucesión de rotaciones intrínsecas, en este caso ZYX con Roll, Pitch y Yaw , por lo que se debe hallar una forma de transformar el Jacobiano geométrico a un Jacobiano que se relacione con las velocidades angulares intrínsecas $[\dot{\psi}, \dot{\theta}, \dot{\phi}]^T$, el cual se denomina “Jacobiano Analítico”

3.2. Jacobiano Analítico

El jacobiano analítico se puede obtener a partir del Jacobiano geométrico ,pero primero es necesario definir como se obtiene la derivada respecto al tiempo de una matriz de rotación y también definir la matriz antisimétrica.

Se sabe que la inversa de la matriz de rotación es igual a la transpuesta

$$R(t)R(t)^T = I \quad (3.13)$$

Al derivar cada lado, lo anterior da como resultado:

$$\dot{R}(t)R(t)^T + R(t)\dot{R}(t)^T = 0 \quad (3.14)$$

Una matriz antisimétrica tiene la siguiente forma definiendo un vector $\omega = [\omega_x \omega_y \omega_z]^T$.

$$S(\omega) = \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix} \quad (3.15)$$

Esta matriz tiene las siguientes propiedades

$$S^T + S = 0 \quad (3.16)$$

$$S(\omega)p = \omega \times p \quad (3.17)$$

$$RS(\omega)R^T = S(R\omega) \quad (3.18)$$

Entonces se puede definir S como el primer termino de (3.14) .

$$S = \dot{R}(t)R(t)^T \quad (3.19)$$

En donde la transpuesta de S está dada por:

$$S^T = (\dot{R}(t)R(t)^T)^T = R(t)\dot{R}(t)^T \quad (3.20)$$

Lo que cumple con (3.17)

Luego multiplicando ambos lados de (3.19) por $R(t)$ se logra obtener:

$$\dot{R}(t) = S R(t) \quad (3.21)$$

Entonces para las rotaciones elementales en X,Y y Z se obtienen las siguientes matrices S ,donde $\hat{\mathbf{X}}, \hat{\mathbf{Y}}, \hat{\mathbf{Z}}$ son los vectores unitarios de un vector de 3 dimensiones.:

$$\begin{aligned} \dot{R}_X(\psi)R_X(\psi)^T &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & -\sin \psi & -\cos \psi \\ 0 & \cos \psi & -\sin \psi \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \psi & \sin \psi \\ 0 & -\sin \psi & \cos \psi \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} = S(\hat{\mathbf{X}}) \\ \dot{R}_Y(\theta)R_Y(\theta)^T &= \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} = S(\hat{\mathbf{Y}}) \\ \dot{R}_Z(\phi)R_Z(\phi)^T &= \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = S(\hat{\mathbf{Z}}) \end{aligned}$$

Luego para encontrar la relación entre las velocidades angulares absolutas $\omega = [\omega_x, \omega_y, \omega_z]^T$ y las velocidades angulares intrínsecas $[\dot{\psi}, \dot{\theta}, \dot{\phi}]^T$ se obtiene la derivada de la matriz de rotación en términos de Roll, Pitch y Yaw usando la propiedad (3.18)

$$\begin{aligned} \dot{R}_{rpy} &= \dot{R}_z R_y R_z + R_z \dot{R}_y R_z + R_z R_y \dot{R}_z \\ &= S(\dot{\phi} \hat{\mathbf{Z}}) R_z R_y R_z + R_z [S(\dot{\theta} \hat{\mathbf{Y}}) R_y] R_z + R_z R_y [S(\dot{\psi} \hat{\mathbf{X}}) R_z] \\ &= [S(\dot{\phi} \hat{\mathbf{Z}}) + S(R_z \dot{\theta} \hat{\mathbf{Y}}) + S(R_z R_y \dot{\psi} \hat{\mathbf{X}})] R \\ &= S(\omega) R \end{aligned}$$

En donde el vector ω corresponde a:

$$\begin{aligned} \omega &= \dot{\phi} \hat{\mathbf{Z}} + R_z \dot{\theta} \hat{\mathbf{Y}} + R_z R_y \dot{\psi} \hat{\mathbf{X}} \\ &= \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \dot{\phi} + \begin{bmatrix} c_\phi & -s_\phi & 0 \\ s_\phi & c_\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \dot{\theta} + \begin{bmatrix} c_\theta c_\phi & -s_\phi & s_\theta c_\phi \\ c_\theta s_\phi & c_\phi & s_\theta s_\phi \\ -s_\theta & 0 & c_\theta \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \dot{\psi} \\ &= \begin{bmatrix} 0 & -s_\phi & c_\theta c_\phi \\ 0 & c_\phi & c_\theta s_\phi \\ 1 & 0 & -s_\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = B(\phi, \theta, \psi) \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = B(\alpha) \dot{\alpha} \end{aligned} \quad (3.22)$$

De esta manera se relaciona las velocidades absolutas con las velocidades intrínsecas. Luego, reemplazando (3.22) en (3.2) se obtiene finalmente el Jacobiano Analítico $J_a(q)$.

$$J(q) \dot{q} = \begin{bmatrix} \dot{d} \\ \omega \end{bmatrix} = \begin{bmatrix} \dot{d} \\ B(\alpha) \dot{\alpha} \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & B(\alpha) \end{bmatrix} \begin{bmatrix} \dot{d} \\ \dot{\alpha} \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & B(\alpha) \end{bmatrix} J_a(q) \dot{q} \quad (3.23)$$

$$\mathbf{J}_a(q) = \begin{bmatrix} I & 0 \\ 0 & B^{-1}(\alpha) \end{bmatrix} J(q) \quad (3.24)$$

En donde la inversa de $B(\alpha)$ es:

$$B^{-1}(\alpha) = \begin{bmatrix} \frac{s_\theta c_\phi}{c_\theta} & \frac{s_\theta s_\phi}{c_\theta} & 1 \\ -s_\phi & c_\phi & 0 \\ \frac{c_\phi}{c_\theta} & \frac{s_\phi}{c_\theta} & 0 \end{bmatrix} \quad (3.25)$$

Nótese que la inversa de $B(\alpha)$ deja de ser invertible si su determinante es cero, lo cual sucede cuando $\cos(\theta) = 0$. Esto se conoce como singularidad representacional.

Entonces la relación de velocidades con el Jacobiano Analítico queda como:

$$\begin{bmatrix} \dot{x}_n \\ \dot{y}_n \\ \dot{z}_n \\ \dot{\phi}_{roll} \\ \dot{\theta}_{pitch} \\ \dot{\psi}_{yaw} \end{bmatrix} = J_a(q)\dot{q} \quad (3.26)$$

La única diferencia con la geométrica es la parte de orientación.

$$J_{a_o}(q) = \begin{bmatrix} 1 & \sin(\phi - q_1)t_\theta & \sin(\phi - q_1)t_\theta & \sin(\phi - q_1)t_\theta & c_\phi^2 c_{q_2 q_3 q_4} + c_\phi c_{q_1} s_{q_2 q_3 q_4} t_\theta + s_\phi (c_{q_2 q_3 q_4} s_{phi} + s_{q_1} s_{q_2 q_3 q_4} t_\theta) \\ 0 & \cos(\phi - q_1) & \cos(\phi - q_1) & \cos(\phi - q_1) & -\sin(\phi - q_1) s_{q_2 q_3 q_4} \\ 0 & \sin(\phi - q_1)/c_\theta & \sin(\phi - q_1)/c_\theta & \sin(\phi - q_1)/c_\theta & \cos(\phi - q_1) s_{q_2 q_3 q_4} / c_\theta \end{bmatrix} \quad (3.27)$$

Donde $t_\theta = \tan(\theta)$ y los ángulos ϕ y θ o roll y pitch, se obtienen a través de las ecuaciones (2.22) y (2.21).

3.3. Jacobiano con parámetros Mitsubishi

Una tercera forma de jacobiano es propuesta, y es usando los dos parámetros de rotación (A y B) de los robots Mitsubishi de 5 DOF. Está fue obtenida simplemente calculando las derivadas parciales como en (3.1)

$$J_M(q) = \begin{bmatrix} -s_{q_1}(a_3 s_{q_2 q_3} + a_2 s_{q_2} + d_5 s_{q_2 q_3 q_4}) & c_{q_1}(a_3 c_{q_2 q_3} + a_2 c_{q_2} + d_5 c_{q_2 q_3 q_4}) & c_{q_1}(a_3 c_{q_2 q_3} + d_5 c_{q_2 q_3 q_4}) & d_5 c_{q_1} c_{q_2 q_3 q_4} & 0 \\ c_{q_1}(a_3 s_{q_2 q_3} + a_2 s_{q_2} + d_5 s_{q_2 q_3 q_4}) & s_{q_1}(a_3 c_{q_2 q_3} + a_2 c_{q_2} + d_5 c_{q_2 q_3 q_4}) & s_{q_1}(a_3 c_{q_2 q_3} + d_5 c_{q_2 q_3 q_4}) & d_5 s_{q_1} c_{q_2 q_3 q_4} & 0 \\ 0 & -a_3 s_{q_2 q_3} - a_2 s_{q_2} - d_5 s_{q_2 q_3 q_4} & -a_3 s_{q_2 q_3} - d_5 s_{q_2 q_3 q_4} & -d_5 s_{q_2 q_3 q_4} & 0 \\ c_{q_2 q_3 q_4} & -q_1 c_{q_2 q_3 q_4} & -q_1 c_{q_2 q_3 q_4} & -q_1 c_{q_2 q_3 q_4} & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \quad (3.28)$$

$$\begin{bmatrix} \dot{x}_n \\ \dot{y}_n \\ \dot{z}_n \\ \dot{A} \\ \dot{B} \end{bmatrix} = J_M(q)\dot{q} \quad (3.29)$$

Nótese que solo cambian las filas relacionadas a la orientación entre los tres Jacobianos, y que en particular el jacobiano $J_M(q)$ es de dimensión 5x5 lo cual al ser una matriz cuadrada es posiblemente invertible.

3.4. Singularidades

Una singularidad en el Jacobiano implica que dada una velocidad v en alguna posición del EF dado q , no exista velocidad $\dot{q}$ que cumpla con (3.1). Esto ocurre cuando el rango de la matriz jacobiana es menor al número de coordenadas espaciales. Para el caso de los jacobianos obtenidos, el rango es menor o igual a 5 y las coordenadas espaciales mínimas al ser el robot de 5 DOF son 5.

Las singularidades generalmente ocurren cuando el EF está en los límites de su espacio de trabajo o la posición no es alcanzable, pero pueden existir singularidades incluso estando en una posición dentro de su espacio de trabajo. En las singularidades o muy cerca de ellas puede ocurrir que para velocidades acotadas del EF se produzcan velocidades articulares no acotadas.

Estas singularidades se producen cuando el determinante del Jacobiano es cero.

Para el caso del Jacobiano J_M , la matriz es cuadrada de 5x5 por lo tanto es posible obtener su determinante.

$$\det(J_M) = -a_2 a_3 s_{q_3} (a_2 s_{q_2} + a_3 s_{q_2 q_3} + d_5 s_{q_1 q_2 q_3}) = 0 \quad (3.30)$$

Para el caso del Jacobiano geométrico la matriz es de 6x5 por lo tanto no se puede obtener su determinante, sin embargo, se puede quitar una fila correspondiente al giro alrededor del eje x, en donde para la configuración del robot en la figura (3.1), su giro no provoca movimiento.

$$\det(J_{5 \times 5}) = a_2 a_3 c_{q_1 q_2 q_3} c_{q_1} s_{q_3} (a_2 s_{q_2} + a_3 s_{q_2 q_3} + d_5 s_{q_1 q_2 q_3}) = 0 \quad (3.31)$$

En ambos se ve que aparece el término entre paréntesis lo que se hace 0 cuando $q_2 = q_3 = q_4 = 0$ es decir cuando el robot está en posición vertical. Y claramente, cuando el robot está totalmente extendido q_3 debe ser 0, lo que ocasiona que $s_{q_3} = 0$ y el Jacobiano se hace singular al estar en el límite del espacio de trabajo.

4 | Modelo Cinemático Inverso

Con la cinemática directa se resolvió el problema convertir las coordenadas articulares de un robot al sistema cartesiano de posición y orientación del EF. Este problema tuvo solución única, es decir, dado un conjunto de coordenadas articulares definidas, estos describen una posición y orientación única. Sin embargo, un problema más complejo, es de determinar cuáles son las coordenadas articulares dado una posición y orientación. Este es el problema de la Cinemática Inversa o “Inverse Kinematics”(IK). Este problema es más complejo debido a que puede no tener una solución en forma cerrada, y pueden existir múltiples, infinitas e incluso no tener solución debido a la estructura del robot.

Por lo general existen tres formas de abordar este problema: Analítica, Geométrica y Numéricamente. Para el Robot RV-2AJ se resuelve por el método geométrico y numérico

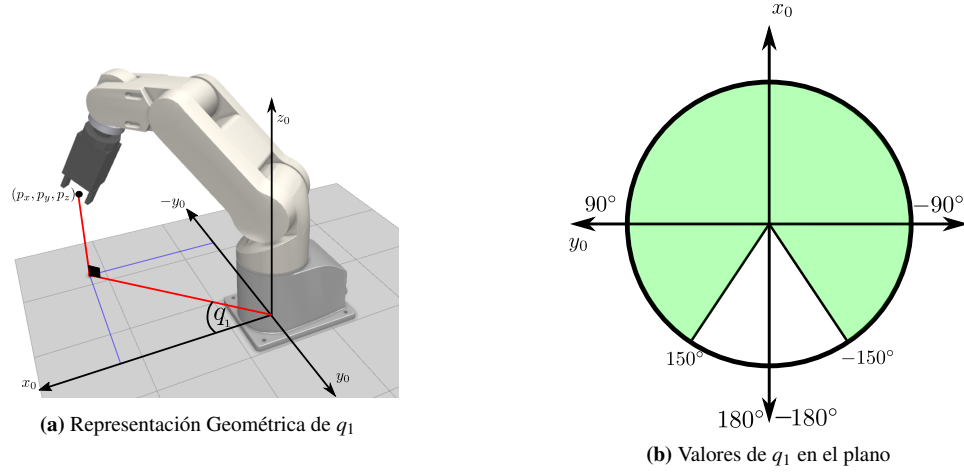
4.1. Método Geométrico

Un punto (p_x, p_y, p_z) cualquiera del EF proyecta un punto en el plano x_o, y_o es decir en $z_0 = 0$ como se muestra en la figura (4.1a). Con esto se puede calcular la primera coordenada articular q_1 usando la función arcotangente de dos parámetros **atan2(y,x)** que devuelve un valor en el intervalo $[-180^\circ, 180^\circ]$ según el valor de x e y acorde a la figura (4.1b), en donde también se ve en verde el rango de los valores de q_1 $[-150^\circ, 150^\circ]$

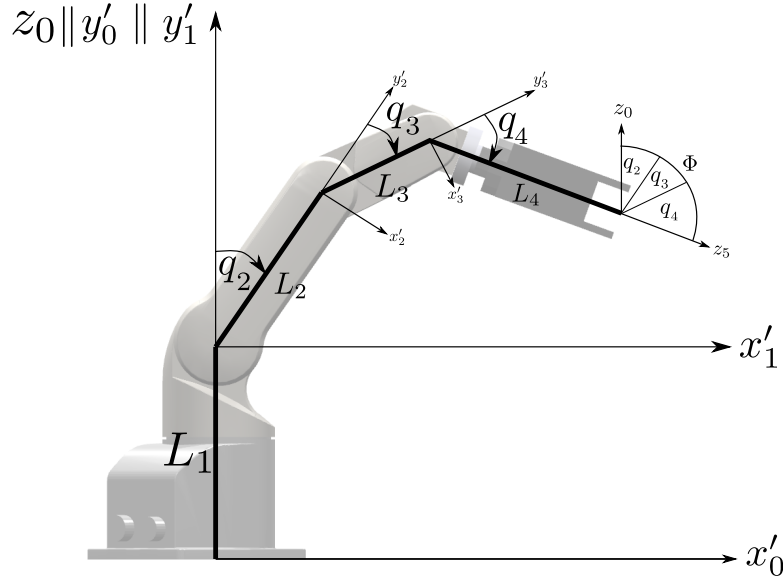
$$q_1 = \text{atan2}(p_y, p_x) \quad (4.1)$$

Además, se puede obtener la distancia entre el origen O_0 y la proyección del EF en el plano $z_0 = 0$ representado por la línea roja en la figura (4.1a) como el radio.

$$r = \sqrt{p_x^2 + p_y^2} \quad (4.2)$$

Figura 4.1: Cálculo de q_1

Luego para calcular q_2, q_3 y q_4 se crea un plano extra independiente del valor de q_1 que estará alineado al robot. También se toma como antecedente que, como orientación, se tiene el valor de Φ , el cual es un valor angular que representa la suma de las coordenadas articulares q_2, q_3 y q_4 y que se interpreta como el ángulo de inclinación que forma el eje z_0 del marco base con el eje z_5 del marco de referencia del EF, es como el ángulo polar cuando se estudian coordenadas esféricas. Los largos de cada eslabón están representados por la letra L . Todo esto muestra en la figura (4.2)

Figura 4.2: Análisis geométrico de las coordenadas articulares q_2, q_3 y q_4

Sean (p'_{x1}, p'_{y1}) , (p'_{x2}, p'_{y2}) , (p'_{x3}, p'_{y3}) los puntos de los orígenes de los sistemas coordenados $x'_1y'_1$, $x'_2y'_2$ y $x'_3y'_3$ que se ubican en las articulaciones del robot de la figura (4.3) donde hay diferenciar estos sistemas respecto a los que describen los parámetros de Denavit–Hartenberg, ya que estos sistemas coordenados “primos” (') es para hacer más fácil el análisis.

Cabe destacar también que, para este análisis, estos ángulos tienen positividad respecto a la dirección en que se forman; q_2 , q_3 y q_4 son positivos en el semiplano derecho de cada marco de referencia, es decir, cuando giran en sentido horario crecen positivamente.

Se sabe que (p'_{x1}, p'_{y1}) respecto al sistema coordenado $x'_0y'_0$ es:

$$(p'_{x1}, p'_{y1}) = (0, L_1) \quad (4.3)$$

Luego por trigonometría y usando como base el sistema coordenado $x'_1 y'_1$ se tienen las siguientes relaciones:

$$p'_{x2} = L_2 S_{q_2} \quad (4.4)$$

$$p'_{y2} = L_2 C_{q_2} \quad (4.5)$$

$$\begin{aligned} p'_{x3} &= p'_{x2} + L_3 S_{q_2 q_3} \\ &= L_2 S_{q_2} + L_3 S_{q_2 q_3} \end{aligned} \quad (4.6)$$

$$\begin{aligned} p'_{y3} &= p'_{y2} + L_3 C_{q_2 q_3} \\ &= L_2 C_{q_2} + L_3 C_{q_2 q_3} \end{aligned} \quad (4.7)$$

Lo anterior se puede manipular algebraicamente sumando los cuadrados de p'_{x3} y p'_{y3} para obtener q_3

$$(p'_{x3})^2 + (p'_{y3})^2 = L_2^2 S_{q_2}^2 + L_3^2 S_{q_2 q_3}^2 + 2L_2 L_3 S_{q_2} S_{q_2 q_3} + L_2^2 C_{q_2}^2 + L_3^2 C_{q_2 q_3}^2 + 2L_2 L_3 C_{q_2} C_{q_2 q_3} \quad (4.8)$$

$$= L_2^2 + L_3^2 + 2L_2 L_3 (C_{q_2} C_{q_2 q_3} + S_{q_2} S_{q_2 q_3}) \quad (4.9)$$

Luego usando la identidad trigonométrica $\cos(A - B)$ con $A = q_2 + q_3$ y $B = q_3$ para el termino en paréntesis:

$$\begin{aligned} C_{q_2} C_{q_2 q_3} + S_{q_2} S_{q_2 q_3} &= \cos(A - B) \\ &= \cos(q_2 + q_3 - q_3) \\ &= \cos(q_3) = C_{q_3} \end{aligned}$$

$$(p'_{x3})^2 + (p'_{y3})^2 = L_2^2 + L_3^2 + 2L_2 L_3 C_{q_3} \quad (4.10)$$

Se despeja lo anterior para obtener q_3 , obteniendo la siguiente expresión:

$$q_3 = \arccos \left[\frac{(p'_{x3})^2 + (p'_{y3})^2 - (L_2^2 + L_3^2)}{2L_2 L_3} \right] \quad (4.11)$$

Se podría haber llegado al mismo resultado usando el teorema del coseno de la ecuación (4.12), ya que la expresión $(p'_{x3})^2 + (p'_{y3})^2 = c$ conforma una recta que pasa por los orígenes O'_1 y O'_3 formando un triángulo con los eslabones $L_2 = a$ y $L_3 = b$ donde a y b forman el ángulo Γ el cual es suplementario con q_3 formando la relación $\cos(\Gamma) = \cos(180 - q_3) = -\cos(q_3)$ llegando a la misma expresión.

$$c^2 = a^2 + b^2 - 2ab \cos \Gamma \quad (4.12)$$

De lo anterior también se puede rescatar que si el argumento del $\arccos()$ es mayor a 1, el resultado es un numero complejo, lo que indica que no existe solución debido a que, por las limitaciones físicas del robot, este no podría alcanzar ese punto ya que estaría fuera del Workspace. En este caso habría una singularidad ya que si $q_3 = \arccos(1) = 0$ se produce la singularidad como se mencionó en el capítulo pasado.

Luego, para obtener q_2 se considera el análisis de la figura (4.3.)

Sea γ el ángulo que se forma desde el eje x'_1 a la recta en azul que intersecta el origen $O'_1(p'_{x1}, p'_{y1})$ con el punto (p'_{x3}, p'_{y3}) , y β , el ángulo que se forma desde el eslabón L_2 a la recta azul.

Tal como se mencionó anteriormente para los ángulos q_2 , q_3 y q_4 , estos ángulos tienen positividad respecto a donde crecen ; es decir tal como se muestra en la figura (4.3), β crece positivamente en sentido horario y γ crece en sentido antihorario.

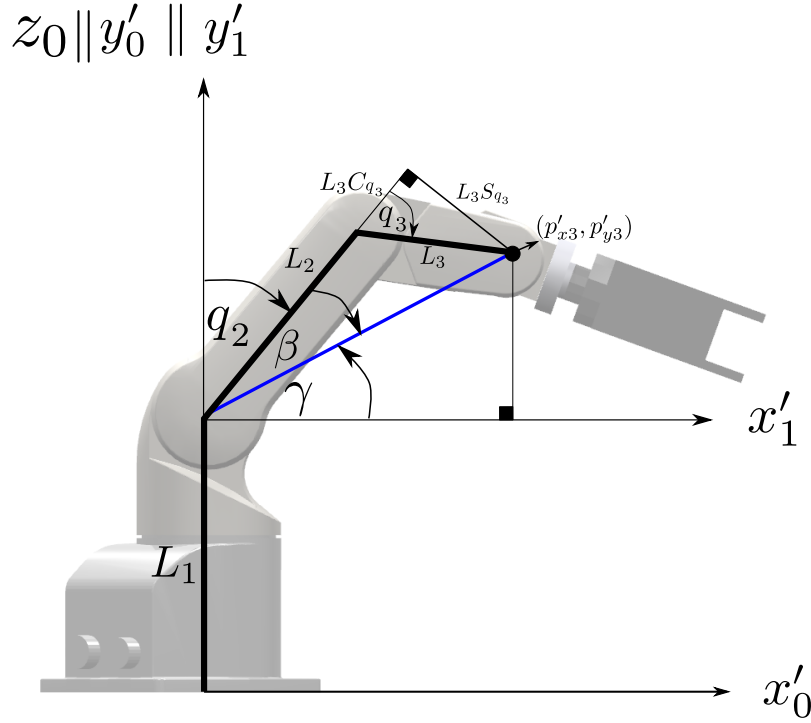


Figura 4.3: Análisis geométrico de las coordenadas articulares para q_2

Observando la figura anterior, se puede notar que la suma de los ángulos q_2 , β y γ forma un ángulo recto, entonces:

$$q_2 = 90^\circ - \gamma - \beta \quad (4.13)$$

Ahora es solo cuestión de encontrar β y γ .

Para γ se puede observar que se puede usar la misma técnica que se usó para encontrar q_1 , a partir de la función arcotangente de dos parámetros (fig 4.1b) usando el punto (p'_{x3}, p'_{y3}) .

$$\gamma = \text{atan2}(p'_{y3}, p'_{x3}) \quad (4.14)$$

Luego para β , este forma parte del triángulo rectángulo señalado en la figura (4.3), cuyos catetos ,opuesto(CO) y adyacente(CA) a β son:

$$CO_\beta = L_3 S_{q_3} \quad (4.15)$$

$$CA_\beta = L_2 + L_3 C_{q_3} \quad (4.16)$$

Luego por trigonometría se obtiene

$$\tan(\beta) = \frac{CO_\beta}{CA_\beta} = \frac{L_3 S_{q_3}}{L_2 + L_3 C_{q_3}} \quad (4.17)$$

Luego para la inversa se puede seguir usando la función arcotangente de dos parámetros.

$$\beta = \text{atan2}((L_3 S_{q_3}), (L_2 + L_3 C_{q_3})) \quad (4.18)$$

Finalmente se obtiene q_2 como:

$$q_2 = 90^\circ - \gamma - \beta \quad (4.19)$$

$$= 90^\circ - \text{atan2}(p'_{y3}, p'_{x3}) - \text{atan2}((L_3 S_{q_3}), (L_2 + L_3 C_{q_3})) \quad (4.20)$$

Luego se obtiene p'_{y3} y p'_{x3} en función del punto del EF (p_x, p_y, p_z) . Para p'_{x3} recordamos de la ecuación (4.2) que describe el radio y consideramos la proyección del eslabón L_4 en el plano x'_0 , y para p'_{y3} se obtiene la relación geométrica entre L_1 , p'_{y3} , p_z y la proyección del eslabón L_4 al eje z_0 .

$$p'_{x3} = r - L_4 \sin(180 - \Phi) \quad (4.21)$$

$$= \sqrt{p_x^2 + p_y^2} - L_4 S_\Phi$$

$$p'_{y3} = (p_z - L_1) + L_4 \cos(180 - \Phi) \quad (4.22)$$

$$= (p_z - L_1) - L_4 C_\Phi \quad (4.23)$$

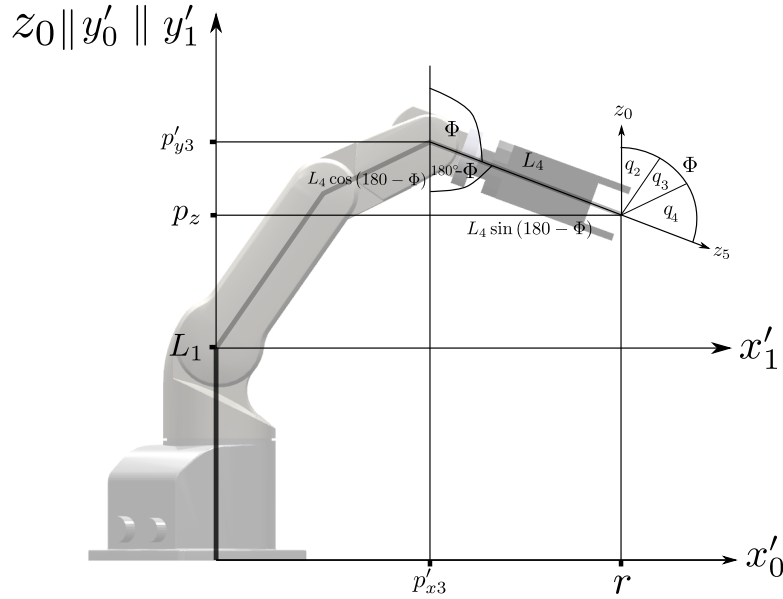


Figura 4.4: Análisis geométrico para obtención de p'_{y3} y p'_{x3}

Como se mencionó en un principio, el ángulo de inclinación Φ que proporciona información sobre la orientación del punto, cumple con:

$$\Phi = q_2 + q_3 + q_4 \quad (4.24)$$

Por lo tanto ya habiendo obtenido q_2 y q_3 se tiene que q_4 es

$$q_4 = \Phi - q_2 - q_3 \quad (4.25)$$

La última coordenada articular que falta es q_5 sin embargo para una posición (p_x, p_y, p_z) , esta no afecta ya que es el giro del EF del robot y no cambia su posición. Esto se puede ver también en la expresión de la matriz de transformación homogénea del robot (2.56) en donde se ve que q_5 no afecta al vector de posición, solo al de orientación.

Continuando con la cinemática inversa, ahora es necesario estudiar cual es el efecto de la orientación solicitada sobre las coordenadas articulares.

El uso de las matrices de rotación es la forma más completa de describir una orientación, sin embargo, existen distintas maneras de describirla. La orientación del EF se puede dar como la matriz completa de rotación perteneciente al conjunto $SO(3)$, o como se estudió, por los tres parámetros *Roll*, *Pitch* y *Yaw*. Pero debido a que el robot RV-2AJ cuenta solo con 5 grados de libertad, existe un grado de limitación que se aplica justamente a la orientación, lo que implica que el robot no puede alcanzar algunas orientaciones, incluso si la posición está correcta.

Por eso es que, para este robot, la orientación se puede describir solo con dos parámetros. Uno de estos ya se describió anteriormente, el ángulo de inclinación Φ , luego el siguiente parámetro puede ser el mismo q_5 , por lo que la función de la cinemática inversa puede tener la forma:

$$[q_1, q_2, q_3, q_4, q_5]^T = IK(p_x, p_y, p_z, \Phi, q_5) \quad (\text{forma simple}) \quad (4.26)$$

Esta es la forma más simple para la cinemática inversa, sin embargo, si se usa como entrada una matriz de transformación homogénea o *Roll*, *Pitch* y *Yaw*, se debe extraer de ellos Φ y q_5

$$[q_1, q_2, q_3, q_4, q_5]^T = IK({}^0T_n) \quad (\text{Con Matriz SE}(3)) \quad (4.27)$$

$$= IK(p_x, p_y, p_z, {}^0R_n) \quad (\text{Con Posición y Matriz rotación } SO(3)) \quad (4.28)$$

$$= IK(p_x, p_y, p_z, \phi_{roll}, \theta_{pitch}, \psi_{yaw}) \quad (\text{Con Posición y RPY}) \quad (4.29)$$

Si se da con *Roll*, *Pitch* y *Yaw* se debe convertir a la matriz de rotación (2.20), para los otros casos ya se tiene la matriz de rotación.

Para extraer Φ , se puede observar que de los valores de la matriz de transformación de este robot de las ecuaciones en (2.55), r_{33} contiene Φ

$$\begin{aligned} r_{33} &= C_{q_2 q_3 q_4} \\ &= C_{\Phi} \end{aligned} \quad (4.30)$$

$$\Phi = \arccos(r_{33}) \quad (4.31)$$

Luego para q_5 , observando los mismos valores anteriores, se toma r_{31} y r_{32} .

$$\begin{aligned} r_{31} &= -S_{q_2 q_3 q_4} C_{q_5} \\ r_{32} &= S_{q_2 q_3 q_4} S_{q_5} \\ \frac{r_{32}}{r_{31}} &= \frac{S_{q_5}}{-C_{q_5}} \\ q_5 &= \text{atan2}(r_{32}, -r_{31}) \end{aligned} \quad (4.32)$$

Otra forma de representar la orientación con los dos parámetros A y B de Mitsubishi introducidos anteriormente en (2.57) y (2.58), en donde se puede observar que B es igual a Φ y A es función de q_5 por lo que se puede despejar obteniendo $q_5 = A - q_1 \cos(B)$.

$$[q_1, q_2, q_3, q_4, q_5]^T = IK(p_x, p_y, p_z, A, B) \quad (\text{Con rotación Mitsubishi}) \quad (4.33)$$

Ya obtenidas las coordenadas articulares para una posición y orientación dada, se debe considerar que para la cinemática inversa, si solo se entrega como información el punto en el espacio, sin ninguna orientación, se pueden obtener infinitas soluciones de coordenadas articulares, y en el mismo sentido, si se entrega como información el punto y la orientación, no va a existir una solución única, de hecho para este robot, existen hasta 4 distintas configuraciones que se muestran en la figura (6.2).

La configuración es "de frente/espalda", describe si el EF está en la dirección mostrado por la flecha azul en las figuras en (6.2), la cual se puede interpretar como la normal al plano que generan los ejes de giro

de $q_1(z_0)$ con $q_2(z_1)$, por lo que cambia de pendiente de q_1 . Entonces si el EF está en dirección a la flecha, su configuración es “de frente”, y caso contrario es “de espalda”.

La configuración de “Codo arriba/abajo”, describe la posición de la articulación J3, que genera q_3 , respecto a la recta que une las articulaciones J2 y J4 mostrado por la recta naranja de las figuras en (6.2). Para los casos “de frente”, si la articulación está por sobre la recta su configuración es “Codo arriba”, caso contrario es “Codo abajo” y para los casos “de espalda”, si la articulación está por sobre la recta su configuración es “Codo abajo”, caso contrario es “Codo arriba”. Esto es debido a que para que sea “Codo arriba”, q_3 debe ser positivo y para “Codo abajo” q_3 es negativo. Si $q_3 = 0$ se puede considerar como “Codo arriba” o “Codo abajo”.

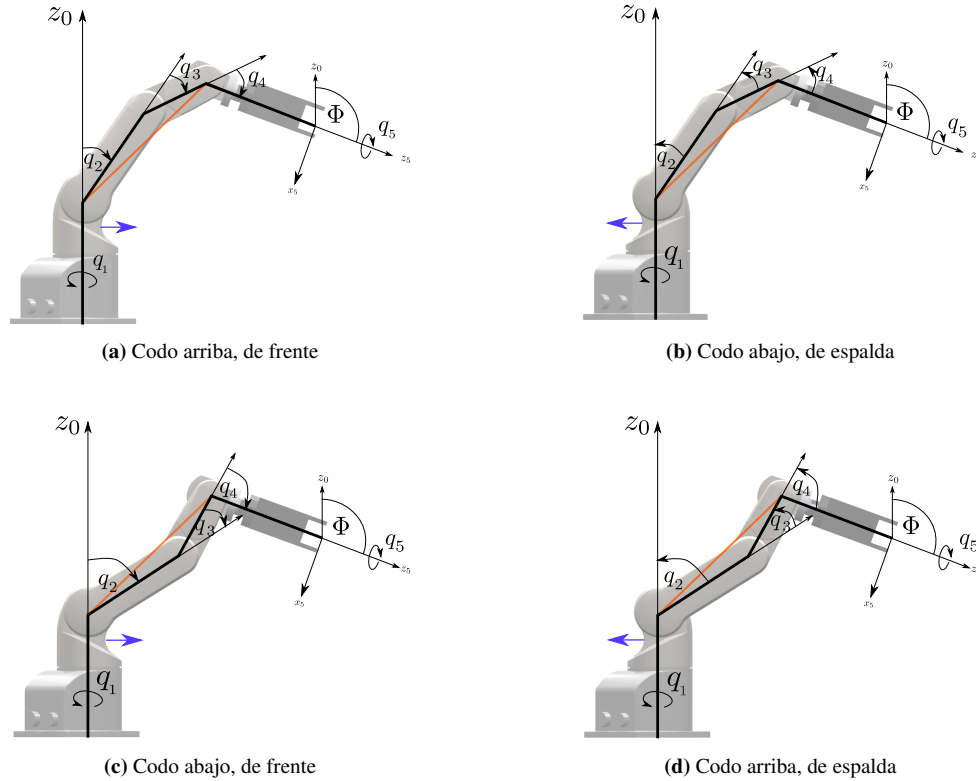


Figura 4.5: Distintas configuraciones dada posición y orientación

Nótese que en las figuras se dejó representado por la flecha en q_2, q_3 y q_4 la dirección de giro positiva, en donde aumenta el valor angular, en las figuras (4.5a) y (4.5c) el eje z de giro de cada articulación entra a la hoja y en las figuras (4.5d) y (4.5b) el eje z sale de la hoja cumpliéndose con la regla de la mano derecha en todas las articulaciones.

Para obtener la cinemática inversa dada alguna de estas configuraciones se observa que para los casos “Codo arriba/abajo” dada la configuración “de frente”, las coordenadas articulares q_2 y q_4 cambian su magnitud, sin embargo q_3 solo cambia de signo con “Codo abajo” por lo que este se calcula cambiando el signo en la ecuación (4.11), y luego β , q_2 y q_4 se calculan igual con las ecuaciones (4.18), (4.13) y (4.25) respectivamente. Para los casos “de frente/espalda”, se puede observar que su efecto es cambiar el signo de los ángulos q_2 , q_3 y q_4 , para mantener una misma postura de “Codo arriba/abajo”. Esto es porque además el ángulo q_1 debe hacer un giro en 180° , el cual también afecta a la orientación del EF por lo que q_5 también debe hacer un giro en 180° .

Todas estas configuraciones están limitadas a los rangos de movimiento de cada articulación, por ejemplo, si q_1 se encuentra entre $[-30^\circ, 30^\circ]$ y se desea cambiar su configuración a “de espalda” no se podrá, ya que girarlo en 180° lo dejará fuera de su rango $[-150^\circ, 150^\circ]$. El programa completo en Matlab se

encuentra en el anexo A

4.2. Método Numérico

Debido a que para algunos tipos de robots pueda ser muy difícil obtener una expresión analítica de la cinemática inversa, o simplemente no exista una solución cerrada, es que existen los métodos numéricos. En particular se estudia el método de optimización por descenso de Newton-Raphson, el cual busca converger a una solución usando el Jacobiano por medio de la minimización del error.

Sea x_d el vector que contiene la posición y orientación deseada, q_d el vector de coordenadas articulares que se desea encontrar y $f(q)$ la función de la cinemática directa. Se debe encontrar q_d que satisfaga con:

$$x_d - f(q_d) = 0 \quad (4.34)$$

Expandiendo la función de la cinemática directa en una serie de Taylor de primer orden dada una estimación inicial q_0 cercana a q_d se obtiene la siguiente linealización :

$$x_d = f(q_d) = f(q_0) + \left. \frac{\partial f}{\partial q} \right|_{q_0} (q_d - q_0) \quad (4.35)$$

En donde el termino derivativo es el Jacobiano el cual va a depender del tipo de formato de la orientación como se explicó anteriormente.

Entonces la expresión se puede reescribir como:

$$J(q_0)(q_d - q_0) = x_d - f(q_0) \quad (4.36)$$

Luego asumiendo que $J(q_0)$ es cuadrada e invertible se llega a:

$$(q_d - q_0) = J^{-1}(q_0)(x_d - f(q_0)) \quad (4.37)$$

Para encontrar el valor deseado de las coordenadas articulares q^d se deben dar una serie de estimaciones $q_0, q_1, \dots$, las cuales al ir siendo orientadas por el jacobiano se converge a un valor lo más cercano posible a x_d .

$$q_k = q_{k-1} + \alpha_k J^{-1}(q_{k-1})(x_d - f(q_{k-1})) \quad (4.38)$$

Donde lo anterior se puede agregar a un “loop” incrementando k en cada iteración hasta que el error $e = (x_d - f(q_{k-1}))$ sea lo suficientemente pequeño. Esto se puede controlar con un criterio de parada usando la norma vectorial $\|e\| \leq \eta$ con $\eta > 0$. Además, se agrega el termino α que es el tamaño de paso, el cual puede ser ajustado convenientemente para asegurar la convergencia.

En el caso en que el Jacobiano no sea invertible ya sea porque no es una matriz cuadrada o presenta una singularidad, se puede ocupar la Pseudoinversa de Moore-Penrose, en donde si el Jacobiano tiene más columnas que filas, se calcula como:

$$J^\dagger = J^T (J J^T)^{-1} \quad (4.39)$$

Si tiene mas filas que columnas entonces:

$$J^\dagger = (J^T J)^{-1} J^T \quad (4.40)$$

De esta manera se puede calcular la cinemática inversa con el Jacobiano analítico $J_a(q)$ usando la pseudoinversa de (4.40) donde la posición y orientación deseada x_d es:

$$x_{da} = \begin{bmatrix} x_d \\ y_d \\ z_d \\ \phi_{roll_d} \\ \theta_{pitch_d} \\ \psi_{yaw_d} \end{bmatrix} \quad (4.41)$$

O se puede usar el Jacobiano con los parámetros de Mitsubishi $J_M(q)$ simplemente usando la inversa ya que es una matriz cuadrada de 5x5, donde la posición y orientación deseada x_d es:

$$x_{dM} = \begin{bmatrix} x_d \\ y_d \\ z_d \\ A_d \\ B_d \end{bmatrix} \quad (4.42)$$

Para escoger la estimación inicial, en el caso de usar $J_M(q)$, se debe escoger tal que no genere alguna singularidad, lo cual se puede determinar usando el determinante obtenido anteriormente en (3.30). Por ejemplo, una mala estimación inicial es eligiendo $q_3 = 0$ o cercano lo que puede provocar que, al ejecutar el código en Matlab, arroje un error o que empiece a iterar indefinidamente sin encontrar solución.

Para el caso de usar la pseudoinversa con $J_a(q)$ es más difícil determinar cuándo pueda ocurrir una singularidad ya que la expresión es demasiado compleja por lo que como regla general se debe usar como estimación inicial una configuración relativamente cercana a lo que se quiera y que no esté en los límites del workspace.

4.3. Resultados

Como ejemplo se calcula la cinemática inversa con el método geométrico usando RPY y con el método numérico con $J_a(q)$ y $J_M(q)$ para un mismo destino.

$$x_{da} = \begin{bmatrix} 275,82 \\ 48,63 \\ 637,77 \\ 100 \\ 40 \\ 90 \end{bmatrix} \quad x_{dM} = \begin{bmatrix} 275,82 \\ 48,63 \\ 637,77 \\ 50 \\ 90 \end{bmatrix} \quad (4.43)$$

Con las fórmulas obtenidas usando el método geométrico se calculan las coordenadas articulares definiendo que tengan la configuración “codo arriba” y “de frente” y usando los parámetros Mitsubishi A y B. obteniendo el siguiente resultado esperado.

$$q = \begin{bmatrix} 10 \\ 20 \\ 30 \\ 40 \\ 50 \end{bmatrix} \quad (4.44)$$

Luego se calcula usando el Jacobiano con los parámetros Mitsubishi donde usando como estimación

inicial $q_0 = [0 \ 10 \ 10 \ 90 \ 0]^T$ y un tamaño de paso $\alpha = 0,05$ obteniendo el siguiente resultado

$$q_{d_M} = \begin{bmatrix} 9,9932 \\ 19,9757 \\ 30,0131 \\ 40,0320 \\ 49,9520 \end{bmatrix} \quad (4.45)$$

Su evolución para encontrar los valores se ve a continuación en donde se detuvo en la iteración 135 al alcanzar la norma del error del criterio de parada $\eta = 10^{-3}$

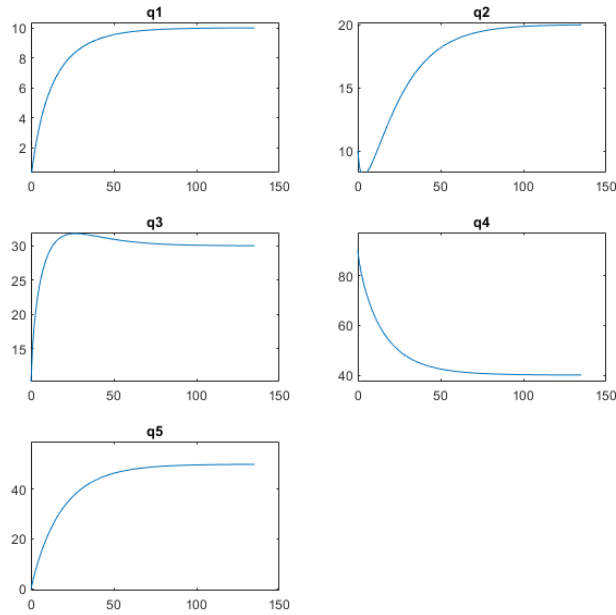


Figura 4.6: Evolución de las coordenadas articulares usando cinemática inversa con $J_M(q)$

Luego usando la misma estimación inicial, pero con el Jacobiano analítico $J_a(q)$ se obtiene el siguiente resultado, con 152 iteraciones usando el mismo criterio de parada y tamaño de paso.

$$q_{d_a} = \begin{bmatrix} 10,0091 \\ 19,9957 \\ 29,9847 \\ 40,0513 \\ 49,9878 \end{bmatrix} \quad (4.46)$$

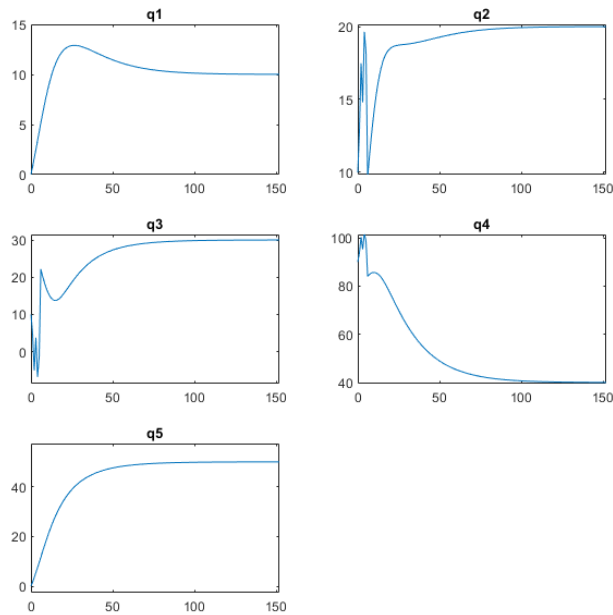


Figura 4.7: Evolución de las coordenadas articulares usando cinemática inversa con $J_a(q)$

Se puede observar que los valores obtenidos no son exactos, pero se acercan bastante, y si se quieren valores más precisos basta con modificar η lo que conllevaría más tiempo de computación. Otra acotación es que el método numérico no distingue entre las configuraciones “codo arriba/abajo” y “de frente/espalda” por lo que se podrían obtener cualquiera de las 4 configuraciones, siempre que sean posibles, dependiendo de la estimación inicial.

En el anexo A se encuentran los códigos en Matlab para obtener la cinemática Inversa Geométrica y numérica con los jacobianos $J_a(q)$ y $J_M(q)$

5 | Validación por simulación

5.1. RVC Toolbox

Para poder simular y verificar los modelos cinemático directo e inverso es que usa Matlab en conjunto con la Toolbox RVC [2] (Robotics, Vision and Control), la cual cuenta con una serie de funciones que hacen posible realizar cálculos de robótica.

Con esta Toolbox se pueden generar objetos eslabones ('Link') para juntarlos en una cadena cinemática ('serialLink') según los parámetros de Denavit-Hartenberg y los límites de las articulaciones. En particular para el robot RV-2AJ esta cadena se construye con el siguiente código.

```
L(1) = Link('d', d1, 'a', 0, 'alpha', af1, 'qlim', [-150*deg 150*deg]);
L(2) = Link('d', 0, 'a', a2, 'alpha', 0, 'qlim', [-60*deg 120*deg] ...
, 'offset', -90*deg);
L(3) = Link('d', 0, 'a', a3, 'alpha', 0, 'qlim', [-110*deg 120*deg]);
L(4) = Link('d', 0, 'a', 0, 'alpha', af2, 'qlim', [-90*deg 90*deg] ...
, 'offset', 90*deg);
L(5) = Link('d', d5, 'a', 0, 'alpha', 0, 'qlim', [-200*deg 200*deg]);
RV_2aj=SerialLink(L,'name','RV_2aj');
```

Con esta Toolbox se puede simular un modelo simple en 3D dándole como entrada las coordenadas articulares en radianes.

```
RV_2aj.plot([-30 45 60 30 10]*pi/180)
```

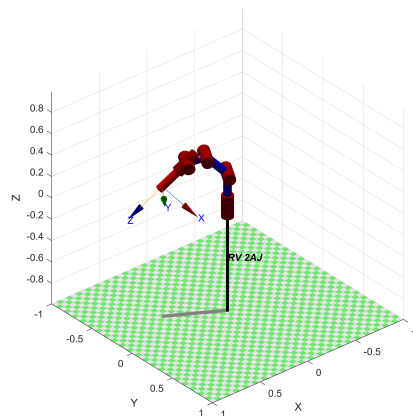


Figura 5.1: Modelo 3D simple con toolbox RVC

Para efecto de los cálculos, esta toolbox cuenta funciones para formar matrices de rotación y de transformación homogénea.

Por ejemplo, cuenta con las funciones $rpy2r(r,p,y)$ y $rpy2tr(r,p,y)$ las cuales entregan una matriz de rotación(SO(3)) y una matriz de transformación homogénea(SE(3)) respectivamente con los parámetros *roll*, *pitch* y *yaw*.

$$rpy2r(r, p, y) = Rot_z(y)Rot_y(p)Rot_x(r)$$

Sin embargo, esta toolbox define al revés roll con yaw, donde en la ecuación (2.20) se definió como $Rot_z(r)Rot_y(p)Rot_x(y)$. Estas diferencias se deben a que en diversos libros y referencias usan distintas definiciones para los ángulos de Euler.

De todas maneras, se pueden usar las funciones $rotz(r)$, $roty(p)$ y $rotx(y)$ que entregan matrices de rotación las que se pueden transformar a una matriz de transformación homogénea con $r2t()$.

También, para definir translaciones se cuenta con la función $transl(x,y,z)$ que entrega una matriz de transformación homogénea con una matriz identidad como su rotación.

Entonces, para generar una matriz de transformación homogénea en base a posición y orientación, se puede usar:

```
T=transl(px,py,pz)*r2t(rotz(r)*roty(p)*rotx(y))
T=transl(px,py,pz)*rpy2tr(y,p,r,'deg') %'deg' para angulos en grados
```

Para obtener *roll*, *pitch* y *yaw* de una matriz se pueden usar las ecuaciones en (2.22) o usar la función de la toolbox $tr2rpy(T)$ sin embargo como están intercambiados roll con yaw se obtiene el resultado en orden con el comando $flip()$ de Matlab.

```
RPY=flip(tr2rpy(T1)) %RPY=[r,p,y]
```

Esta toolbox también permite obtener la cinemática directa del robot ,obteniendo una matriz de transformación homogénea T dado un vector q con el comando $fkine(q)$.

```
T=RV_2aj.fkine(q)
```

5.2. Melfa Basic IV

Melfa Basic IV es un lenguaje de programación usado para los robots Mitsubishi el cual está basado en el lenguaje de alto nivel *Basic* el cual se basa en una secuencia de comandos para describir movimientos del robot, operaciones lógicas, operaciones matemáticas, operaciones de entrada y salida y también cuenta con variables para monitoreo de eventos o atributos del robot.

La ejecución del programa funciona leyendo línea a línea ,y cuando termina la última ,el programa retorna al inicio generando un ciclo continuo, lo que es útil para procesos continuos y repetitivos. Además, cada línea empieza con un número que debe ir ascendiendo a medida que se avanza en el código.

A continuación, se describen algunos de los comandos disponibles que se utilizan en este trabajo.

DEF JNT JA Define una variable JA con un conjunto de coordenadas articulares del robot. Estas variables deben empezar con la letra J de “Joint”.

Para el caso del robot RV-2AJ , se considera que J4 no existe (J4=0), la cual sería la articulación para obtener los 6 DOF, por lo que para describir una variable articular esta se define como

$$JA = (J1, J2, J3, 0, J5, J6)$$

Donde $[J1, J2, J3, J5, J6] = [q_1, q_2, q_3, q_4, q_5]$ en grados. En JA se puede guardar la variable interna J_CURR, la cual contiene la información de las coordenadas articulares actuales del robot.

DEF POS P0 Define una variable P0 que contiene información de posición, orientación y tipo de configuración. Estas variables deben empezar con la letra P de “Position”. La estructura de la variable es

$$P0 = (p_x, p_y, p_z, A, B, 0)(f, 0)$$

Donde p_x, p_y, p_z es la posición en milímetros, A y B son los parámetros de orientación de Mitsubishi vistos anteriormente en grados y f corresponde a un flag relacionado a la configuración, en donde la configuración “codo Arriba/Abajo” o, con la nomenclatura de Mitsubishi, “Above/Below” (A/B), corresponde al primer bit 0/1, y la configuración “de Frente/Espalda” o “Right/Left” (R/L) es el segundo bit 0/1. El bit menos significativo es 0, el existiría si el robot tuviera 6 DOF. En la siguiente tabla se ve el flag convertido de binario a decimal.

R/L = 0/1	A/B = 0/1	-	f
0	0	0	0
0	1	0	2
1	0	0	4
1	1	0	6

Tabla 5.1: Flags según configuración de pose del Robot

En PO se puede guardar la variable interna P_CURR, la cual contiene la información de la posición actual del robot

DEF <INTE,FLOAT,DOUBLE>M0 Define una variable numérica que puede ser un entero de 16 bits (INTE), un punto flotante de simple precisión (FLOAT) o doble precisión (DOUBLE).

OVRD <> Especifica la velocidad del robot en formato porcentual del 1 % al 100 %.

DLY <> Genera un retardo en segundos en el momento en que se ejecute este comando. El valor mínimo es de 0.05 segundos.

GOSUB (RETURN) Llama a una subrutina que puede estar en un número de línea o asociada a una etiqueta usando el símbolo asterisco (*). La subrutina es un código que se ejecuta cuando se use GOSUB el cual salta a la línea donde está la etiqueta. Al final de la subrutina debe ir RETURN para devolver a la línea siguiente de GOSUB para retornar y continuar con el código.

MOV <> Mueve el robot a un punto mediante interpolación de las articulaciones. El controlador del robot calcula la trayectoria óptima para llegar al punto especificado por lo que está es desconocida para el operador. También se le puede especificar una distancia de acercamiento, con el que el robot se moverá al punto deseado pero desplazado en el eje Z de la garra. Por ejemplo, MOV P1,-50.

MVS <> Mueve el robot a un punto mediante interpolación línea, por lo que la trayectoria será una línea recta entre el punto de partida y el especificado. Este movimiento está limitado por la misma estructura del robot, por lo que para algunos movimientos podría arrojar un error. Al igual que MOV, también se puede especificar una distancia de acercamiento.

CNT 0/1 Activa o desactiva el movimiento continuo entre puntos. Al tenerlo activado el robot se dirige al punto, pero puede no pasar por este y solo acercarse a una posición vecina. Esto provoca que, al no moverse directamente a los puntos, el robot no se detenga en cada uno de ellos lo que reduce el tiempo en completar la trayectoria

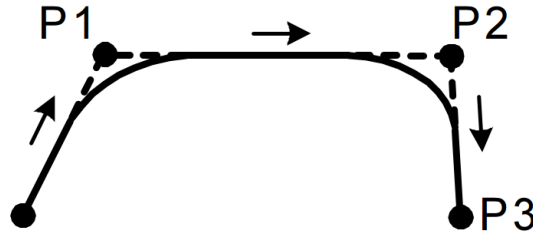


Figura 5.2: Ejemplo de movimiento continuo CNT

OPEN Abre la línea de comunicación que puede ser por el puerto serial con el estándar RS232-C, o por comunicación TCP/IP. La línea de comunicación es manejada como un archivo de texto el cual recibirá y enviará datos. Existen 8 líneas de comunicación disponibles y el archivo también se especifica como un número del 1 al 8. Para abrir la línea de comunicación 4 usando el archivo 1 se escribe:

OPEN"COM4 :'" AS#1

Se puede usar la variable binaria interna M_OPEN(1) informa si el archivo 1 está abierto. Para cerrar el archivo 1 su línea de comunicación se usa CLOSE #1 y todos los datos de comunicación son borrados.

INPUT Recibe datos de entrada a un archivo, el cual puede ser una línea de comunicación y los guarda en una variable definida. El formato del dato debe ser un carácter o una línea de texto en formato ASCII. Se pueden recibir varias separadas por una coma. Por ejemplo, para recibir una variable numérica y de posición articular que fue enviada como "100 , (0,10,20,0,30,40)" y guardarlas en las variables M1 y J1 teniendo abierta la línea de comunicación con el archivo #1, se ejecuta el siguiente comando :

INPUT#1,M1,J1

Donde M1 será 100 y J1 será (0,10,20,0,30,40). El programa esperará en esta instrucción hasta que se envíe el dato en formato ASCII que cuente con un terminador de línea de Retorno de carro(CR) y salto de línea(LF).

PRINT Envía datos de salida a un archivo, el cual puede ser una línea de comunicación. El formato del dato de salida es un carácter o una línea de texto en formato ASCII con un terminador de línea de Retorno de carro(CR).

5.3. Conexión entre Matlab Y Robot

Para cargar un programa en lenguaje Melfa Basic IV al controlador CR1-571 del robot, se usa el software RT ToolBox2 de Mitsubishi. La conexión entre el controlador y el computador con el software se hace a través de la red de computadores del departamento de electrónica.

Para generar la conexión entre Matlab y el Controlador, en Matlab se crea un objeto socket para establecer la conexión por TCP/IP. El objeto socket en Matlab se configura como Servidor mientras que el punto de conexión del controlador será de tipo Cliente. Para generar el socket en Matlab se usa el comando *tcpserver*

```
server=tcpsrvr(direccion,puerto);
```

Donde la dirección es la dirección IP del computador con Matlab.

El controlador es configurado como Cliente usando RT toolbox2 en donde se especifica el puerto 4 como la línea de conexión. Mas detalles de la configuración en el anexo B.

Como primera prueba se realiza en Melfa un programa que constantemente lee coordenadas articulares las cuales son enviadas por la conexión TCP/IP entre Matlab y el controlador y el robot se mueve a ellas.

```
1 DEF JNT JT
2 OPEN "COM4:" AS #1
3 WHILE 1
4 INPUT #1, JT
5 MOV JT
6 WEND
```

Código 5.1: Movimiento articular

Lo mismo se puede hacer con coordenadas cartesianas

```
1 DEF POS PT
2 OPEN "COM4:" AS #1
3 WHILE 1
4 INPUT #1, PT
5 MOV PT
6 WEND
```

Código 5.2: Movimiento cartesiano

Para enviar los datos en formato *string* desde Matlab se usa el comando *write* donde *to_write* es la línea de texto con los formatos vistos anteriormente dependiendo si se envía una variable de posición articular o cartesiana.

```
write(server,to_write,"string")
```

Para integrar todo lo visto hasta el momento, se construye una aplicación en Matlab con la herramienta *App Designer*, con la cual, mediante varias herramientas como botones, campos de texto, deslizadores, gráficos, entre otros, se diseña una interfaz para ver una simulación del robot usando la toolbox RVC y simular la cinemática directa e inversa.

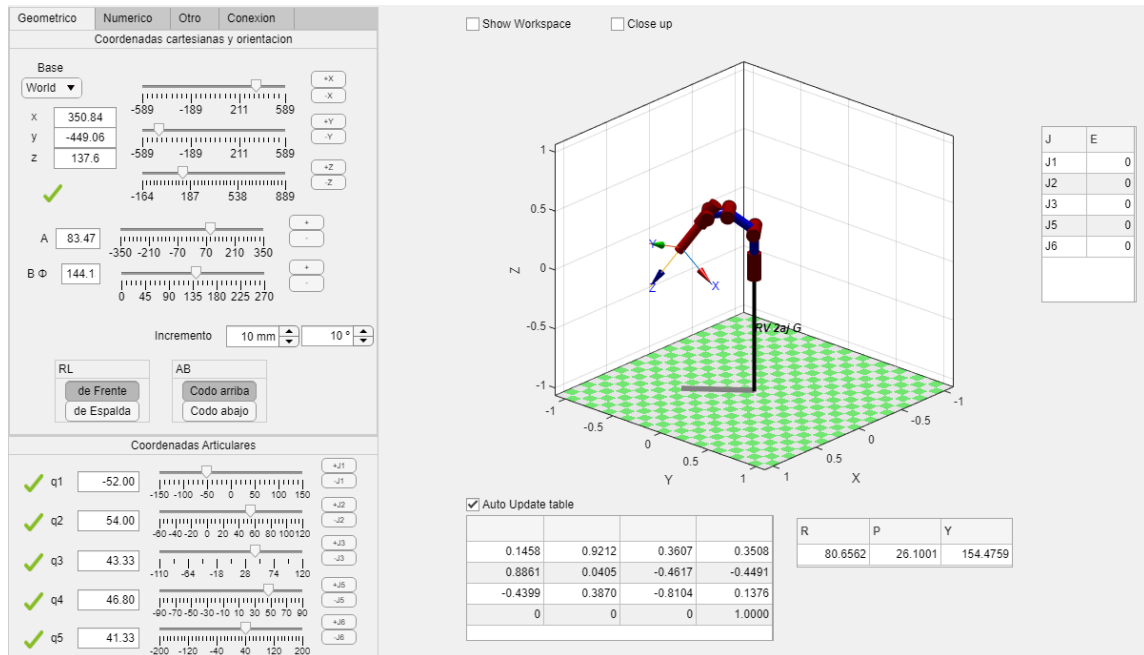


Figura 5.3: Interfaz de aplicación diseñada

La aplicación cuenta con deslizadores, campos de texto y botones para controlar tanto la cinemática directa como la inversa. También cuenta con campos de texto para modificar el incremento y decremento tanto en milímetros como grados si es que se usan los botones para modificar las coordenadas. Cuenta con botones para intercambiar entre las configuraciones “Codo Arriba/Abajo” y “de Frente/Espalda”. También se puede intercambiar la base o el marco de referencia inercial para el control de los botones de incremento o decremento en xyz, pudiendo ser *World* o *EF* con esta última usando el marco de referencia del EF utilizando la inversa de la matriz de transformación homogénea como se muestra en (2.35), en donde si se aprieta el botón +Z, el robot se mueve en dirección Z respecto al EF, y los campos de texto se actualizan con los valores respecto al marco base.

También se muestra una tabla que contiene la matriz de transformación homogénea y al lado los valores de Roll, Pitch y Yaw.

Los objetos para modificar la posición del robot se actualizan automáticamente por lo que, si se modifica el valor de alguna articulación, también se modificarán los valores de los objetos asociados a las coordenadas cartesianas y viceversa.

También cuenta con unos indicadores que muestran un visto (✓) o una cruz (✗) en caso de que alguno de los valores se encuentre fuera de sus rangos, por ejemplo, si se quiere cambiar a una configuración “de Espalda” y esta no es posible, se indicará cuáles son las coordenadas articulares que no permiten llegar a tal configuración, o si se desea ir a un punto fuera del Workspace indicará con una cruz el indicador en las coordenadas cartesianas. Cada vez que el símbolo cruz esté presente y se estén manipulando algunos valores, estos cambiarán mas no modificarán al robot, ni será posible enviar valores por comunicación hasta que los valores sean modificados para que estén en los valores posibles del robot. Cada uno de los objetos de la aplicación responde a eventos mediante *Callbacks* para realizar las acciones solicitadas.

A la aplicación también se le agregó el método numérico para la cinemática inversa usando el Jacobiano con los parámetros de orientación de Mitsubishi, en donde para realizar pruebas, se tiene el botón *Guardar posición* el cual guarda la posición actual como la deseada o se puede configurar manualmente escribiendo en los campos de texto, en donde para la orientación se puede ingresar tanto como los ángulos RPY, los parámetros AB o usar q_5 en vez de A. La estimación inicial será la posición actual del robot por lo que para ver su efecto es recomendable modificar las coordenadas articulares para llevar el robot a otra posición.

En la figura 5.4 se puede observar un trayecto entre la estimación inicial y la posición del EF producto de la serie de estimaciones. El programa también indica la cantidad de iteraciones que fueron necesarias para llegar a la posición indicada.

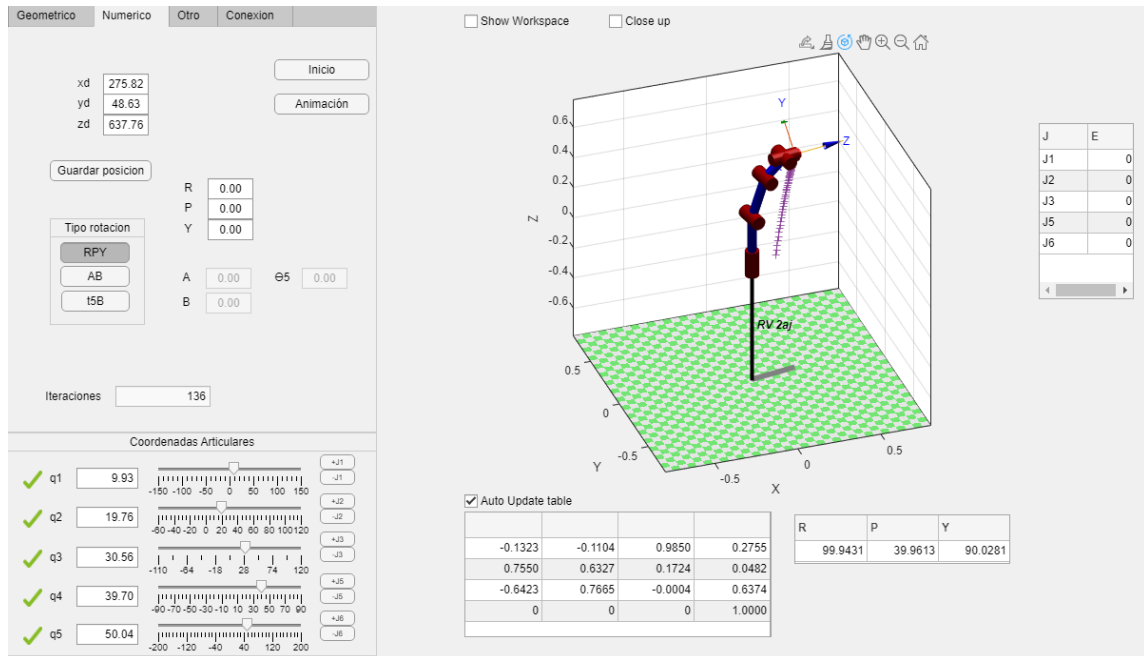


Figura 5.4: Interfaz Cinemática Inversa numérica

Finalmente, para mover el robot usando la aplicación para enviar instrucciones de posición, se inicia primero el servidor con el botón de la figura 5.5, luego se aprieta en el botón “Iniciar programa” y se puede seleccionar el modo para enviar variables de posición cartesiana o articular. Para esto al código en Melfa Basic IV se le incorporan los códigos 5.1 y 5.2 como subrutinas.

```

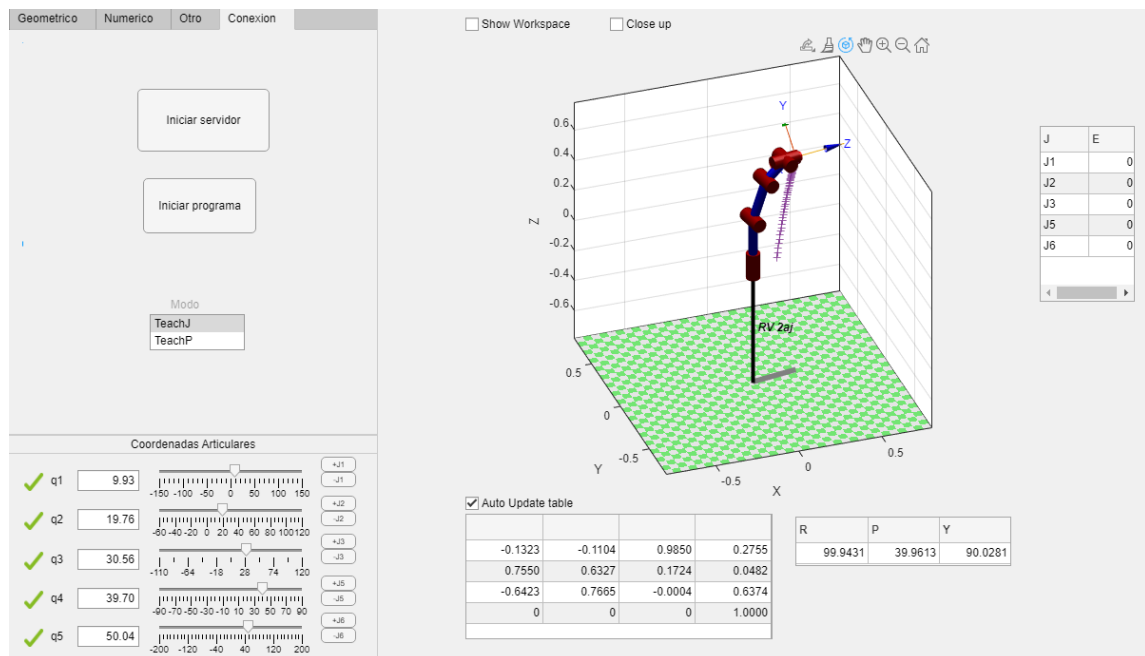
1  DEF POS P0          'var posicion cartesiana
2  DEF JNT J0          'var posicion articular
3  DEF INTE M          'var de Modo (articular =1 cartesiano=2)
4  '*****
5  *RETRY              'genera loop infinito hasta que se inicie el servidor
6  OPEN "COM4:" AS #1
7  IF M_OPEN(1) <> 1 THEN GOTO *RETRY
8  '*****
9  INPUT #1,M          'se recibe el modo inicial M=1
10 *MODE
11 SELECT M            'segun modo va a subrutina cartesiano o articular
12 CASE 1
13 GOSUB *JTCH         'Joint Teach
14 BREAK
15 CASE 2
16 GOSUB *PTCH         'Position Teach
17 BREAK
18 END SELECT
19 GOTO *MODE
20 END
21 '*****
22 '*****SUBROUTINAS*****
23 *JTCH
24 WHILE 1
25 INPUT #1,JT,M        'se lee la posicion y el modo
26 IF M <> 1 THEN *OJT   'si se cambia el modo vuelve a seleccion de modo

```

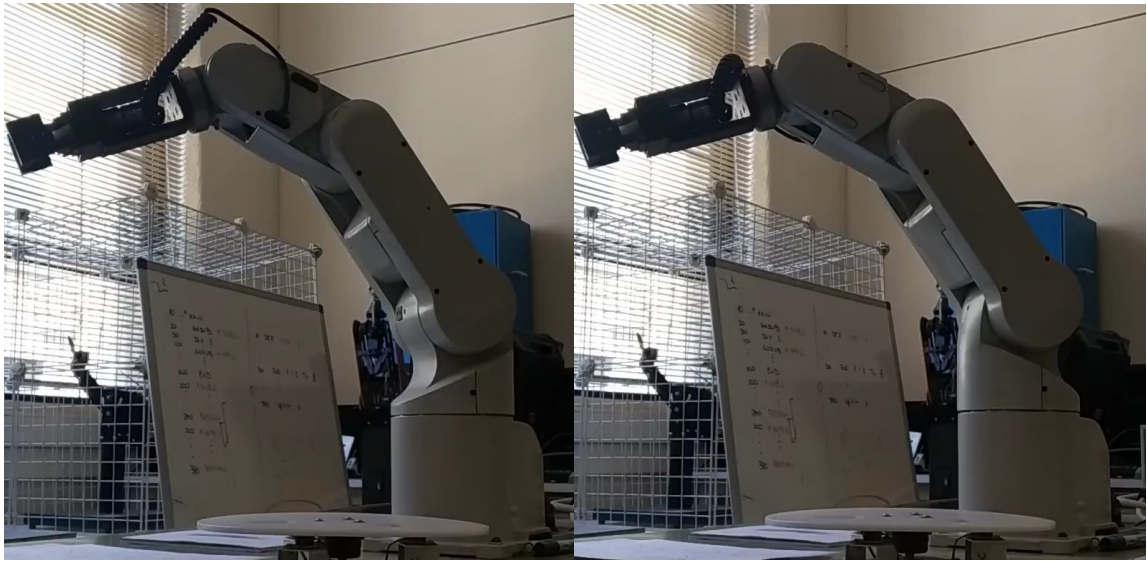
```

27 MOV JT
28 WEND
29 *OJT
30 RETURN
31 '*****
32 *PTCH
33 WHILE 1
34 INPUT #1,PT,M      'se lee la posicion y el modo
35 IF M <> 2 THEN *OPT 'si se cambia el modo vuelve a seleccion de modo
36 MOV PT
37 WEND
38 *OPT
39 RETURN

```

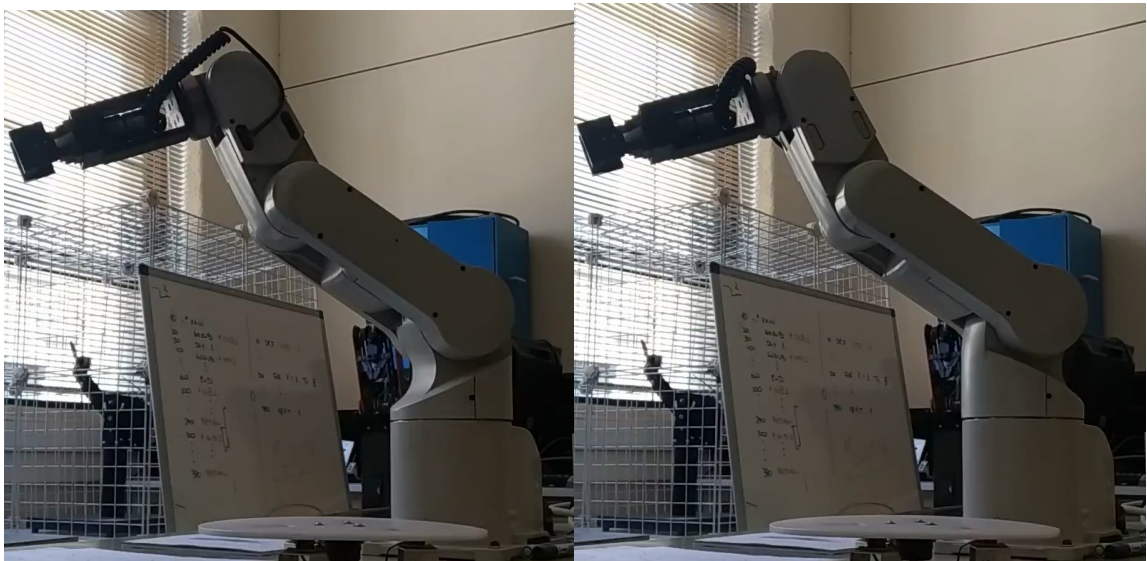
Código 5.3: Código para mover el robot usando la aplicacion**Figura 5.5:** Interfaz para Iniciar conexión con servidor TCP/IP

Con el robot real se prueban las 4 configuraciones para un punto y orientación específica, mostradas en las siguientes imágenes.



(a) Codo arriba, de frente

(b) Codo abajo, de espalda



(c) Codo abajo, de frente

(d) Codo arriba, de espalda

Figura 5.6: Distintas configuraciones obtenidas a partir de la aplicación en Matlab

6 | Generación Automática de Trayectorias

6.1. Generación de trayectoria a partir de imágenes

Ya sabiendo como enviar una única posición a través de la aplicación, el siguiente paso es generar una trayectoria definida. Para esto se propone generar trayectorias a partir de dibujos simples para que el robot lo dibuje. Para ello se usan imágenes en formato SVG (Scalable Vector Graphics) el cual es un formato de imágenes vectoriales, por lo que no se basa en píxeles, sino en objetos geométricos como rectas, curvas, polígonos, etc. los cuales son generalmente formados por curvas de Bézier

Estas curvas son muy usadas para la creación de elementos gráficos por computador. Son curvas paramétricas y se construyen a través de 4 puntos: Dos puntos o nodos de finales, y un punto de control emparejado a cada nodo.

Los puntos de control especifican la forma de la curva de modo que la recta que une el nodo con el punto de control es tangente a la curva que une los nodos. La ecuación que describe la curva se muestra a continuación.

$$\mathbf{B}(t) = (1 - t)^3 \mathbf{P}_0 + 3t(1 - t)^2 \mathbf{P}_1 + 3t^2(1 - t) \mathbf{P}_2 + t^3 \mathbf{P}_3, t \in [0, 1] \quad (6.1)$$

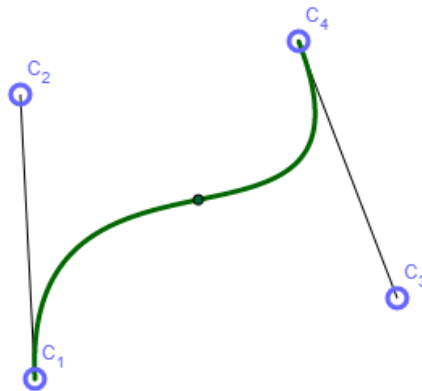


Figura 6.1: Ejemplo de curva de Bezier

En internet es posible encontrar una gran cantidad de dibujos en formato SVG, pero si uno desea crear su dibujo una buena herramienta es el software *Inkscape* el cual además permite convertir imágenes en formato tradicional, como JPG o PNG, a SVG con un cierto margen de error.

Para la selección de imágenes se buscan dibujos de “Arte Lineal” el cual se basa en curvas continuas para generar formas.

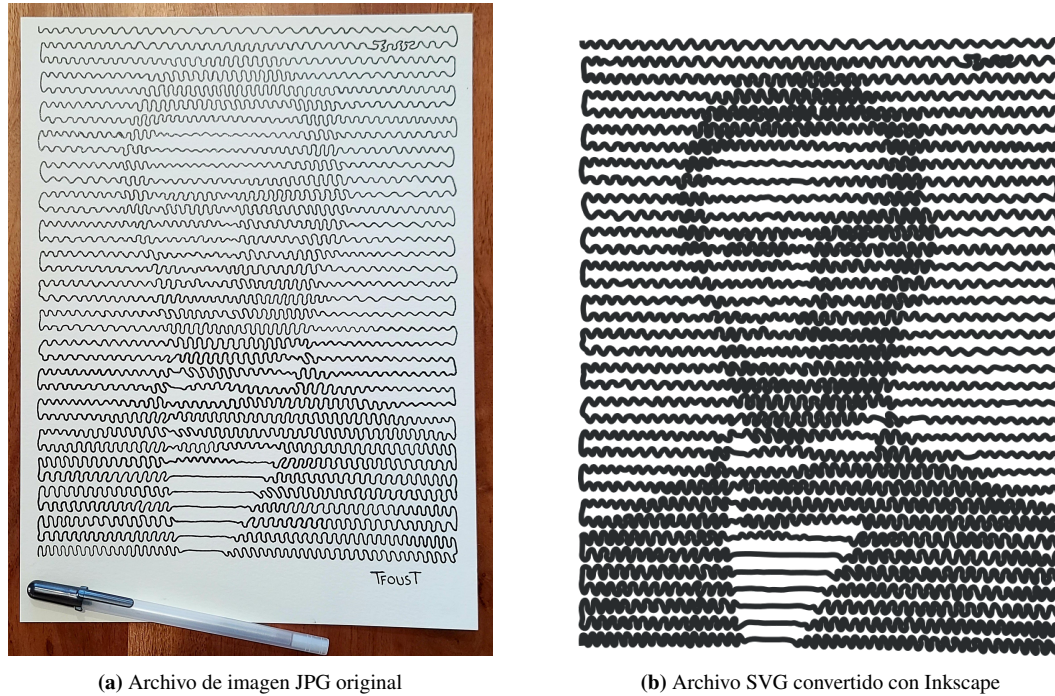


Figura 6.2: Imagen a dibujar SVG

Para convertir una imagen SVG a una trayectoria, se usa una versión modificada del proyecto de Python **svg.path** [5] cuyo principal uso es obtener las especificaciones de las líneas y trayectorias de un archivo SVG y extraer un conjunto de puntos equiespaciados que describe la trayectoria. Este programa se ejecuta por Matlab ya que este cuenta con funcionalidades para ejecutar programas en Python.

```
data=py.svgpath.svg_draw(svgfile,size_board,mm_x_pixel);
```

Donde las entradas son *svgfile* es el nombre del archivo, *size_board* es un vector que contiene las dimensiones del dibujo el cual se establece de 27x15 cm, que cabe en una hoja de oficio, y *mm_x_pixel* especifica cada cuantos milímetros se generan los puntos consecutivos de una trayectoria, dependiendo del dibujo se especifica entre 1 a 5 milímetros.

Como salida se entrega una tupla de Python con el conjunto de las coordenadas en X e Y de los puntos y la cantidad de puntos generados a partir de la imagen. Además, esta función escala las imágenes SVG que pueden tener distintas dimensiones, para que los puntos obtenidos estén en escala 1:1 respecto al *size_board* establecido.

Los puntos generados deben ser procesados para que estén en un área de dibujo definida dentro del Workspace. Se desea dibujar con el robot de manera que el eje Z del EF esté apuntando perpendicularmente a la hoja, es decir con $\Phi = B = 180^\circ$.

Dejando fija la orientación B se tiene menos libertad de trabajo en el Workspace y dando un valor en Z en donde tendrá que dibujar, el Workspace tendrá forma de corona circular.

Para determinar los límites de la corona dado B y Z fijos se aplica la función cinemática Inversa, en donde como entrada se le entregan los valores fijos y un vector de X entre 0 y 600mm con Y=0, donde esta función además de entregar como salida las coordenadas articulares, arroja si el valor es posible o no y así se puede

determinar los valores de X posibles y por ende los radios de los círculos que forman la corona. Los códigos de Python y Matlab para la obtención y procesamiento de los puntos se encuentran en el anexo E

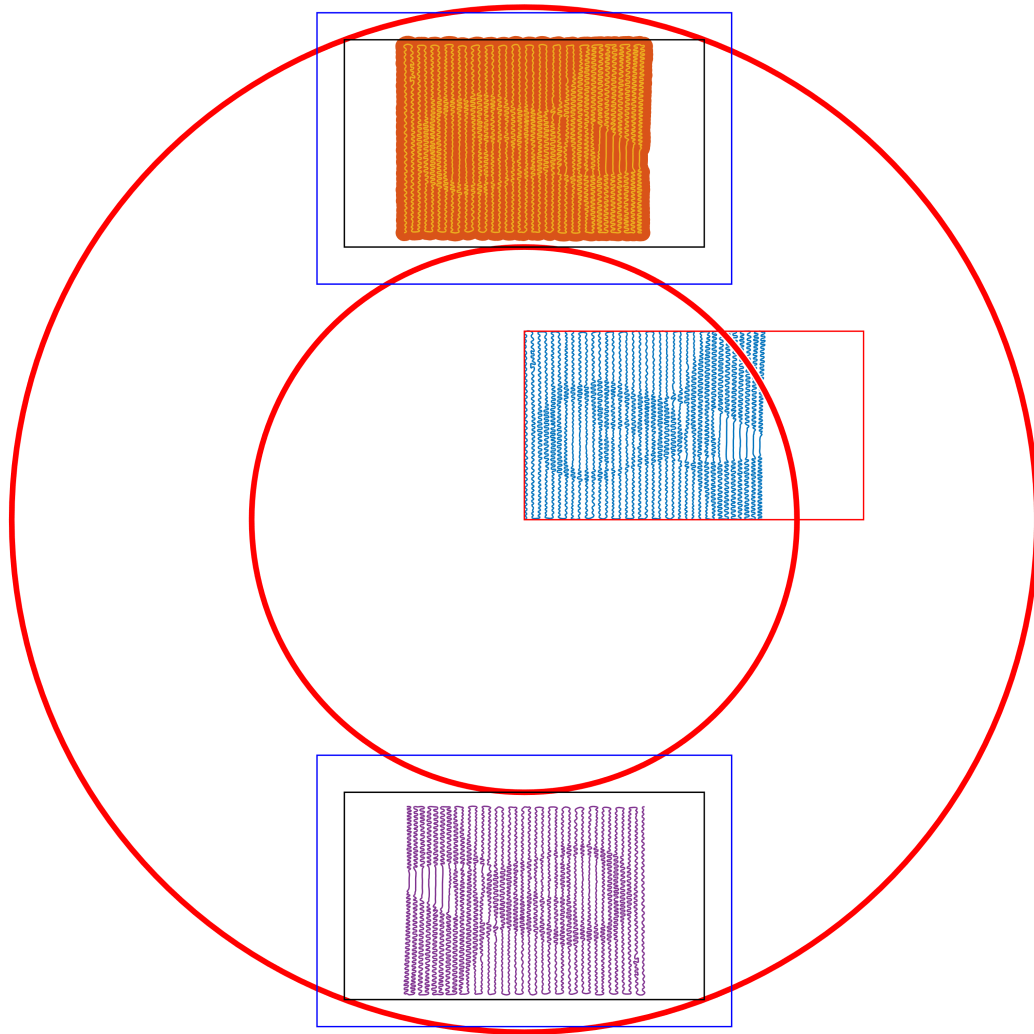


Figura 6.3: Región de dibujo en vista al plano XY

En la figura 6.3 se observan los círculos que forman el Workspace dado una orientación $B=180^\circ$ y un $Z=50$, este último considerando que el EF se encuentra en la garra el cual sostiene un lápiz que sobresale 50 milímetros considerando que la hoja está en $Z=0$.

Al centro la figura están los puntos generados por la función *svgpath* con la dimensión *size_board* en su rectángulo. Arriba se muestran los puntos trasladados para hacerlos calzar dentro de la corona circular, donde el rectángulo pequeño indica una región rectangular en el que puede estar el dibujo y el rectángulo grande en azul tiene las dimensiones de una hoja de oficio en la cual se dibujará. Luego se rota en 180° mostrándose en el dibujo de abajo de la figura, ya que no se puede ocupar ese sector de la mesa en donde está el robot real.

En el ejemplo anterior los puntos forman una sola trayectoria continua, sin embargo, es posible que el dibujo sean varias trayectorias, por lo que para evitar que el robot dibuje una línea entre dos trayectorias se debe realizar un salto, es decir, se debe levantar el lápiz para luego bajarlo donde comienza la nueva línea.

Para esto se guarda el índice del último punto de la línea continua, el cual será enviado como un valor de control al programa en Melfa para levantar el brazo ó cuando sea necesario. El índice se obtiene comparando la distancia entre puntos consecutivos, en donde si la distancia es un poco mayor a la variable *mm_x_pixel* se considera que se debe hacer un salto entre esos puntos.

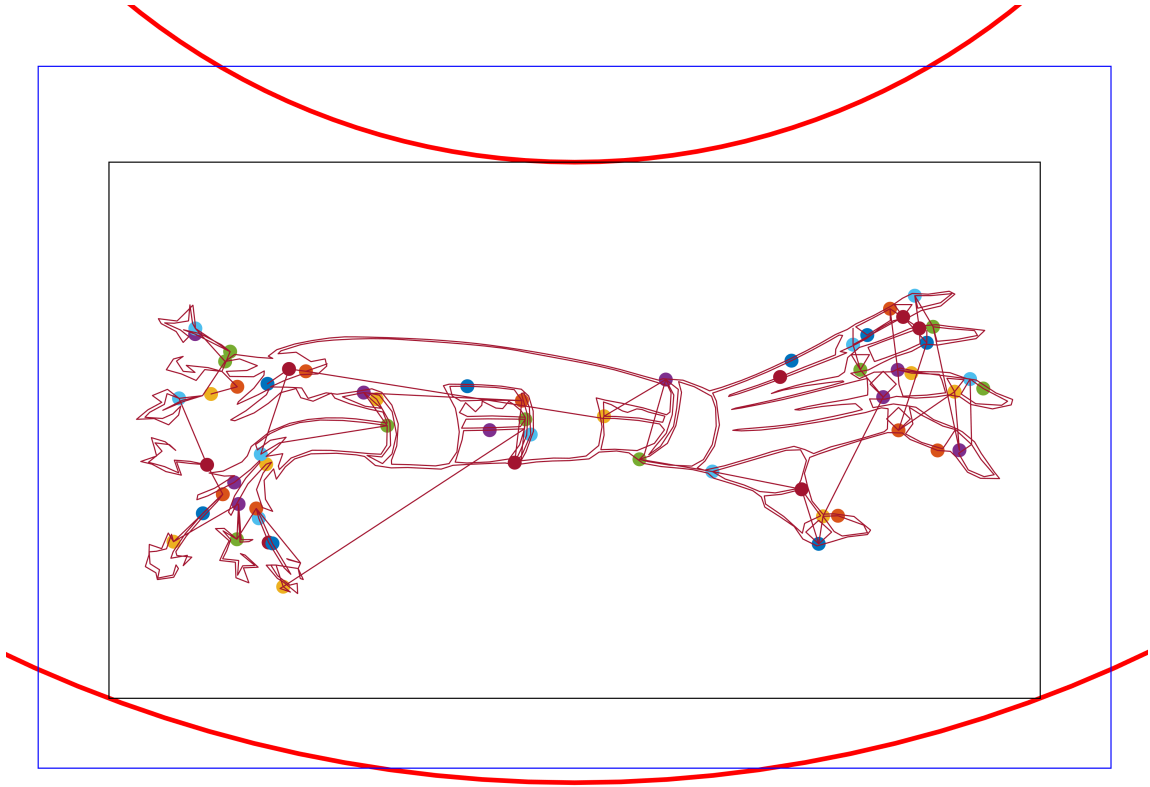


Figura 6.4: Dibujo de un brazo donde los puntos indican saltos entre trayectorias

Para comprobar que los puntos de trayectoria están correctos, se envían a la función de Cinemática Inversa la cual comprueba que los puntos son accesibles. También se comprueba que están correctos usando el software *CIROS Studio*, el cual es una herramienta muy avanzada para simulación de entornos de trabajo con robots industriales.

Este software tiene la capacidad de simular programas en MELFA BASIC IV, sin embargo, no cuenta con la capacidad de simular enlaces de comunicación.

Entonces para comprobar los puntos primero se genera un archivo de texto en formato “.POS” el cual es un archivo que contiene una lista de posiciones cartesianas las cuales son accesibles por el programa y controlador simulado. El archivo tiene la siguiente estructura.

```
DEF POS P (1)=(-77.68,-271.38,0.00,-90.00,180.00,0.00) (6,0)
DEF POS P (2)=(-80.22,-270.38,0.00,-90.00,180.00,0.00) (6,0)
DEF POS P (3)=(-82.78,-269.43,0.00,-90.00,180.00,0.00) (6,0)
.
.
.
```

Luego para moverse entre esos puntos se define un arreglo con el comando Melfa *DIM* con el cual se iterarán las posiciones con un ciclo for, y se añade internamente la funcionalidad de que, si la distancia entre puntos consecutivos es mayor a un límite, el robot sube linealmente 20 milímetros especificando la distancia de acercamiento, se mueve sobre la siguiente posición y luego baja lineal.

```

DEF INTE X
DEF INTE M
DEF INTE D
D=12                                'distancia entre puntos para salto
M=369                                'numero de puntos de la trayectoria
CNT 1                                'se activa el modo continuo
DIM P(369)                           'Arreglo de posiciones

MOV P(1)                             'se mueve a posicion 1
FOR X=2 TO M                          'inicio ciclo for
IF DIST(P(X-1),P(X))>D THEN          'si la distancia entre puntos es >12 salta
MVS P_CURR,-20                       'se mueve de la posicion actual hacia arriba
MOV P(X),-20                         'se mueve a arriba del punto
MVS P(X)                             'baja al punto para continuar el dibujo
ELSE
MOV P(X)                             'si no hay salto continua el dibujo
ENDIF
NEXT X                                'siguiente posicion
END

```

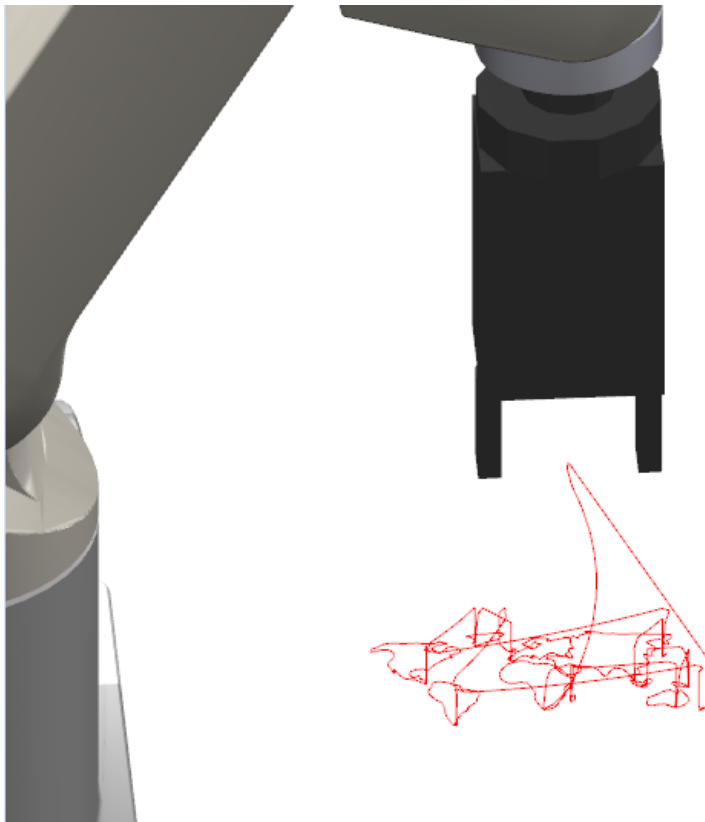


Figura 6.5: Comprobacion de trayectoria en CIROS con saltos

6.2. Transmisión de puntos de trayectoria

Anteriormente se manipuló el Robot mediante el envío de paquetes con el protocolo TCP/IP, pero ahora se deben enviar muchos puntos que caben en el orden de los cientos o miles. Realizando pruebas de testeo para el envío de posiciones se concluye que no se pueden enviar más de 15 valores de posición ya que el controlador deja de funcionar. Esto puede ser posible porque el buffer que posee

el controlador no tiene mucha capacidad.

Es por ello que, para evitar problemas, se realiza la siguiente secuencia;

- Se envía el primer punto del servidor (Matlab) al cliente (controlador)
- El robot se mueve al punto
- El cliente envía la posición alcanzada por el robot al servidor
- El servidor la recibe como acuse de recibo y envía el siguiente punto.

En Matlab es posible configurar un “Callback” para la comunicación.

```
configureCallback(server, "terminator", callbackFcn)
```

Se configura de manera que cada vez que exista un terminador de retorno de carro(CR) disponible para leer desde el cliente, se llame a una función especificada, la cual lee y guarda el dato de posición y luego envía el siguiente. En principio no es necesario mandar el dato de posición, basta con enviar un string vacío, no obstante, se elige esta opción para después comparar los datos enviados con los recibidos.

Las posiciones cartesianas de la trayectoria también son convertidas a coordenadas articulares mediante la función de la cinemática inversa, sin embargo, Melfa Basic IV no permite utilizar los comandos MOV o MVS con distancia de acercamiento para levantar en Z con coordenadas articulares para describir un salto, por lo que se agregan unos puntos extras que ejecutan la misma función.

La estructura de comunicación para el envío de los datos del dibujo se muestra en el diagrama de secuencia de la figura 6.6.

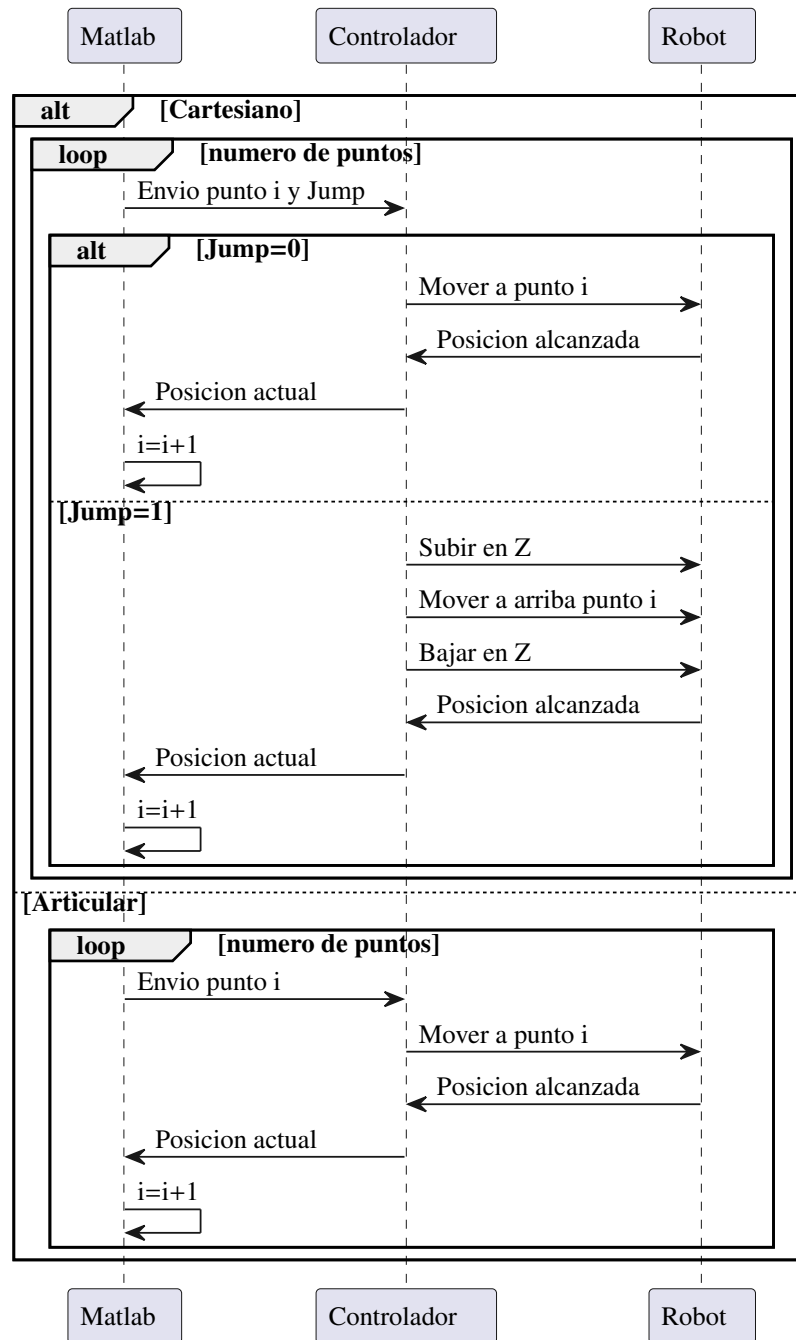


Figura 6.6: Diagrama de secuencia para el dibujo

Se actualiza la aplicación para añadir la funcionalidad de que el robot dibuje a partir de los puntos generados de imágenes SVG.



Figura 6.7: Interfaz de conexion de aplicacion en Matlab

Se añade una lista desplegable “Dibujo” para seleccionar algunos de las imágenes disponibles, también se añade un campo numérico para indicar la altura Z en donde el lápiz toca a la hoja a dibujar. Este valor se puede obtener usando el Teach Box del robot y anotando el valor en Z, o usando la misma aplicación en modo “TeachP” pero usando los botones $\pm Z$ ajustando un incremento muy pequeño para evitar colisiones. Además, se añade la lista desplegable “Tipo de Coordenada” en donde se pueden seleccionar 3 opciones: La primera es modo “XYZ A fijo” en donde el parámetro A de orientación será 0, por lo que la coordenada q_5 se moverá en función de la ecuación (2.58) y se mandan los datos en formato cartesiano. La segunda opción es “XYZ A Variable” donde A se determina por la misma ecuación (2.58), fijando $q_5 = 0$, y la tercera opción es “Joint” en donde se mandan los valores en formato de coordenadas articulares.

Los puntos ya obtenidos y procesados en X e Y se obtienen de un archivo contenedor de datos “.Mat” , y para el caso de seleccionar el modo “Joint”, las coordenadas articulares son obtenidas por cinemática inversa usando como Z el valor del campo de texto y A fijo de 90° .

El robot empieza a dibujar cuando se apriete el botón “Dibujar” y se puede detener en cualquier momento con el botón “Stop”, el cual lleva al robot a una posición segura en altura. Se puede reanudar el proceso apretando de nuevo el botón “Stop”.

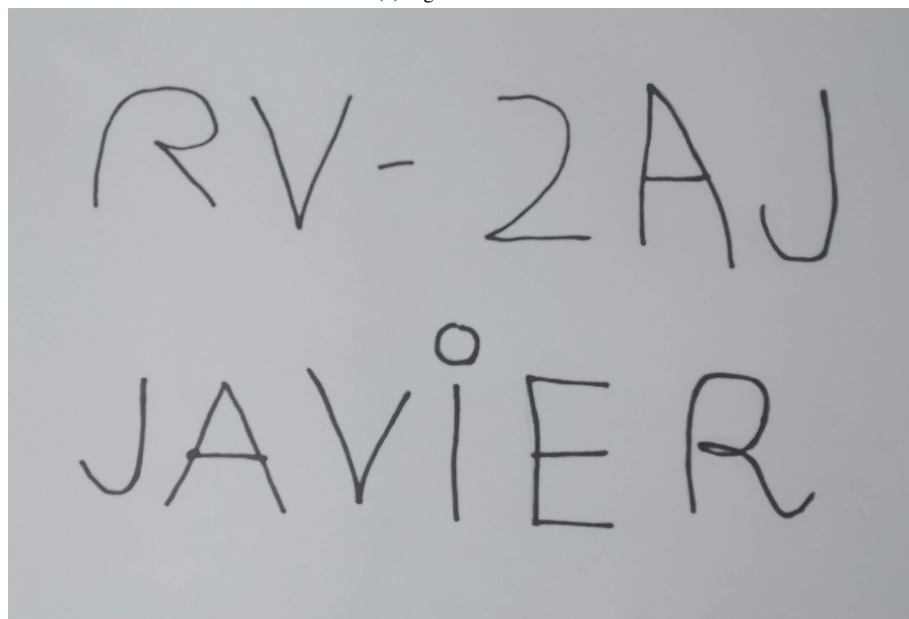
6.3. Pizarra de escritura

Se añade también a la aplicación una pizarra de dibujo simple, con la cual se puede dibujar cualquier forma. Funciona mediante “Callbacks” donde los eventos son si se aprieta o se suelta el clic y si se está moviendo el cursor. Por lo tanto, mientras esté apretado y se mueva el cursor, se registran los valores como puntos y se registra cada vez que se suelta para definir un salto . Los para finalizar se aprieta en el botón verde “Done” donde los datos guardados se transmiten al igual como si fueran los datos de un dibujo SVG usando

automáticamente la opción “XYZ A fijo”.



(a) Figura en Pizarra



(b) Figura resultante

Figura 6.7: Ejemplo en Pizarra de dibujo de la aplicación

Por otra parte, el código en Melfa Basic IV es actualizado para integrar estas nuevas funciones. Este se divide en cuatro modos principales, los cuales son “JTCH”(Joint Teach) para mandar coordenadas articulares individuales, “PTCH”(Position Teach) para mandar coordenadas cartesianas individuales, “DRWJ”(Draw with Joint) para dibujar según coordenadas articulares y “DRWP”(Draw with Position) para dibujar según coordenadas cartesianas.

Para los modos “JTCH” y “PTCH” se añade también que retornen la posición alcanzada, para luego calcular

el error, el cual se muestra en la figura (5.3) en el costado derecho, donde dependiendo del modo mostrará el error de cada coordenada articular, o de cada parámetro de posición y orientación. El código completo se muestra en el Anexo C

6.4. Resultados obtenidos

Con los modos de manipulación de puntos individuales de la aplicación “Teach J” y “Teach P” se observaron los errores en alcanzar los puntos. Estos son calculados como la resta entre el valor enviado menos el recibido, y en todos los casos se obtuvo error=0.00. Debido que los datos enviados y recibidos tienen una precisión de dos decimales, el verdadero valor calculado internamente por la aplicación cuenta con más decimales por lo que no es posible obtener información del error más allá de los 2 decimales.

Por parte de los dibujos, se recolectaron los datos recibidos ,con los cuales se pueden comparar visualmente con los enviados. En el siguiente ejemplo se ven los datos para el dibujo de un brazo en donde el eje X de cada gráfico corresponde al número del punto y el eje Y a su valor en milímetros o grados dependiendo de la variable, donde en azul se muestra el valor enviado y en rojo el recibido. La mayor diferencia que se observa es en Z, en donde en rojo se ven los valores cuando se produce un salto, ya que esta instrucción de salto se envía como un parámetro aparte a las coordenadas.

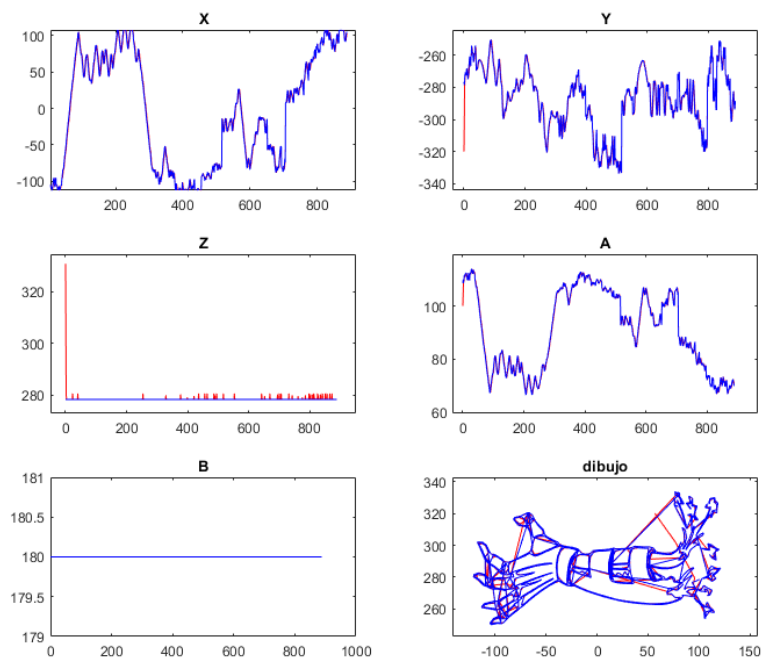


Figura 6.8: Comparación de datos cartesianos enviados y recibidos

Al hacer un acercamiento a la figura se observa que se produce un desfase entre los datos enviados y recibidos, esto es debido a que el controlador con el programa Melfa, empieza a mandar los valores mientras se está moviendo el robot y no cuando llega a la posición de destino por lo que se produce un desfase de un punto entre los datos.

Además, debido a que se activó el modo *CNT 1* para hacer el dibujo más continuo, es posible que existan pequeños errores ya que el robot puede no pasar por el punto exacto, sino por su vecindario. En las siguientes figuras se muestran los valores comparados con acercamiento y ajustados para quitando el desfase.

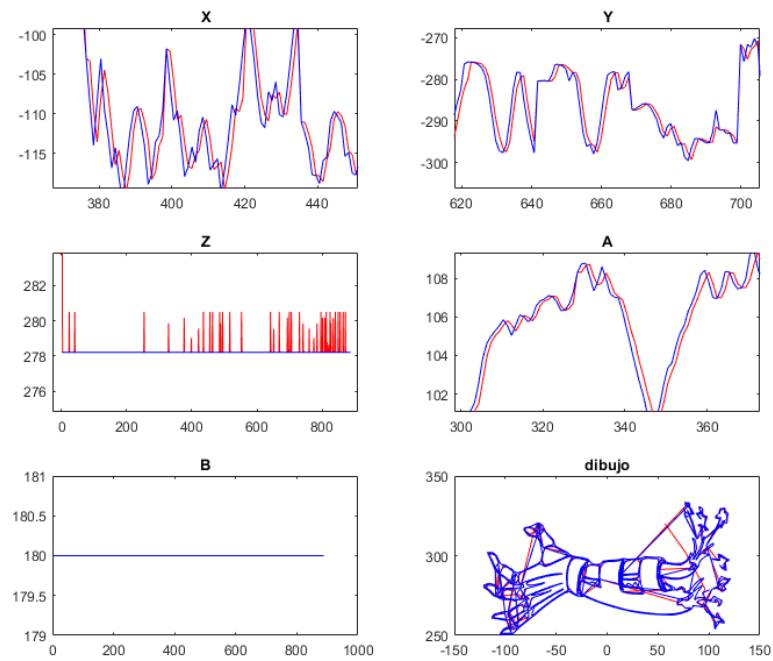


Figura 6.9: Comparación de datos cartesianos enviados y recibidos Zoom

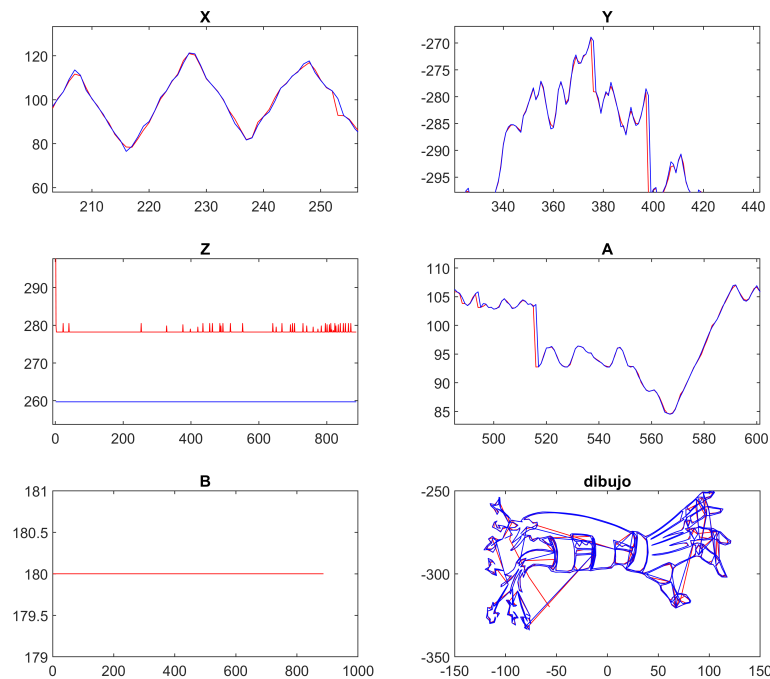


Figura 6.10: Comparación de datos cartesianos enviados y recibidos Zoom sin desfase

Luego para medir numéricamente el error se hace mediante el cálculo del error cuadrático medio (RMSE) el cual es generalmente usado para medir el error entre dos conjuntos de datos.

$$\text{RMSE} = \sqrt{\frac{\sum_{n=1}^N (x_{E,n} - x_{R,n})^2}{N}} \quad (6.2)$$

Donde $x_{E,n}$ son los datos enviados, $x_{R,n}$ son los recibidos y N es el número de puntos

Para el dibujo del brazo se tienen los siguientes errores cuadráticos medio tanto de los datos en coordenadas articulares, como en cartesianas.

Coordenada articular	RMSE
X	4.055364 mm
Y	2.796914 mm
Z	2.009045 mm
A	0.812699 °
B	0 °

Tabla 6.1: RMSE coordenadas cartesianas dibujo brazo

Coordenada articular	RMSE(°)
J1	1.018967 °
J2	0.843466 °
J3	1.702457 °
J5	1.498422 °
J6	1.018967 °

Tabla 6.2: RMSE coordenadas articulares dibujo brazo

Se aprecia que el error puede ser grande, sobre todo en X, pero se explica debido a la desincronización que hay entre el dato leído y el enviado por el controlador. Para ver esto se observan los valores de X en conjunto con la resta entre el enviado y el recibido con los datos sincronizados y no sincronizados.

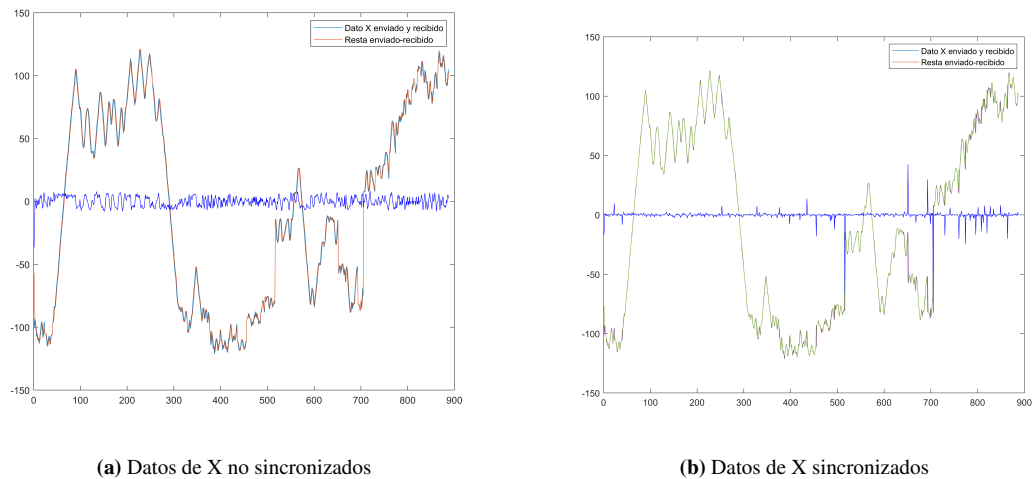


Figura 6.11: Comparación de resta de datos recibidos con enviados

Observando los gráficos se puede observar que la resta en los datos no sincronizados es más errática, de allí su valor de 4.055364. Por otro lado, con los valores sincronizados uno esperaría obtener un error menor, lo que se alcanza a ver en algunos puntos, sin embargo, debido a los saltos en el dibujo es que se generan unas alzas inesperadas en la resta y calculando el RMSE con esos valores se llega a un valor de 4.5913, que es incluso mayor al valor sin sincronizar.

De todas maneras, estos valores del error son solo significativos de analizar en profundidad si en el dibujo obtenido se hubiera obtenido algo con un error apreciable, lo cual no fue así

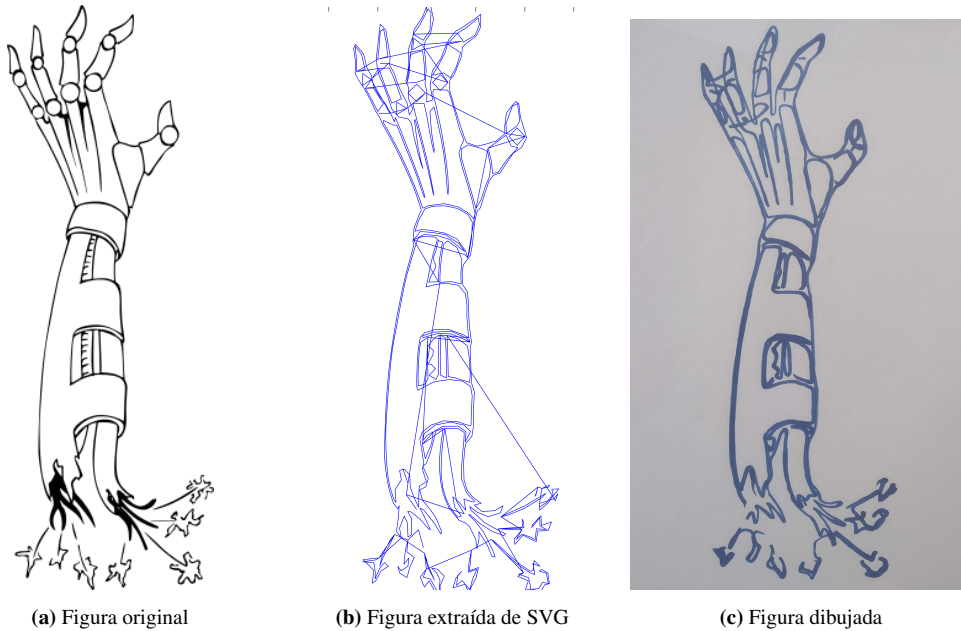


Figura 6.12: Comparación dibujos de un brazo

Nótese del dibujo anterior que la figura extraída no es exactamente igual a la original, esto es debido a la precisión con la se obtiene la imagen, que es directamente relacionado a la variable *mm_x_pixel* que para este dibujo fue de 5, con una cantidad total de puntos de 888.

Por otro lado, comparando los tiempos de varios dibujos se obtuvo que la velocidad promedio en términos de puntos es de aproximadamente 10 puntos por segundo, por lo que el robot obtuvo el dibujo anterior en 1 minuto con 29 segundos.

En el anexo D se muestran más resultados de otras imágenes.

Es posible también generar trayectorias en base a funciones matemáticas, como curvas paramétricas en 3D (robótica paramétrica).

A modo de ejemplo se genera una trayectoria de una **Helicoide Exponencial**, con sus puntos generados por el siguiente código en Matlab.

```
z = linspace(280,445,200);
x = (exp(-z./100) .* sin(1000*z)) * 100;
y = (exp(-z./100) .* cos(1000*z)) * 100 - 250;
```

Esta función fue diseñada para que esté en los rangos del workspace del robot, ajustando tanto su rango en Z, como las constantes de la función.

A partir de lo anterior se obtuvieron los siguientes gráficos que muestran la comparativa entre los datos enviados y recibidos.

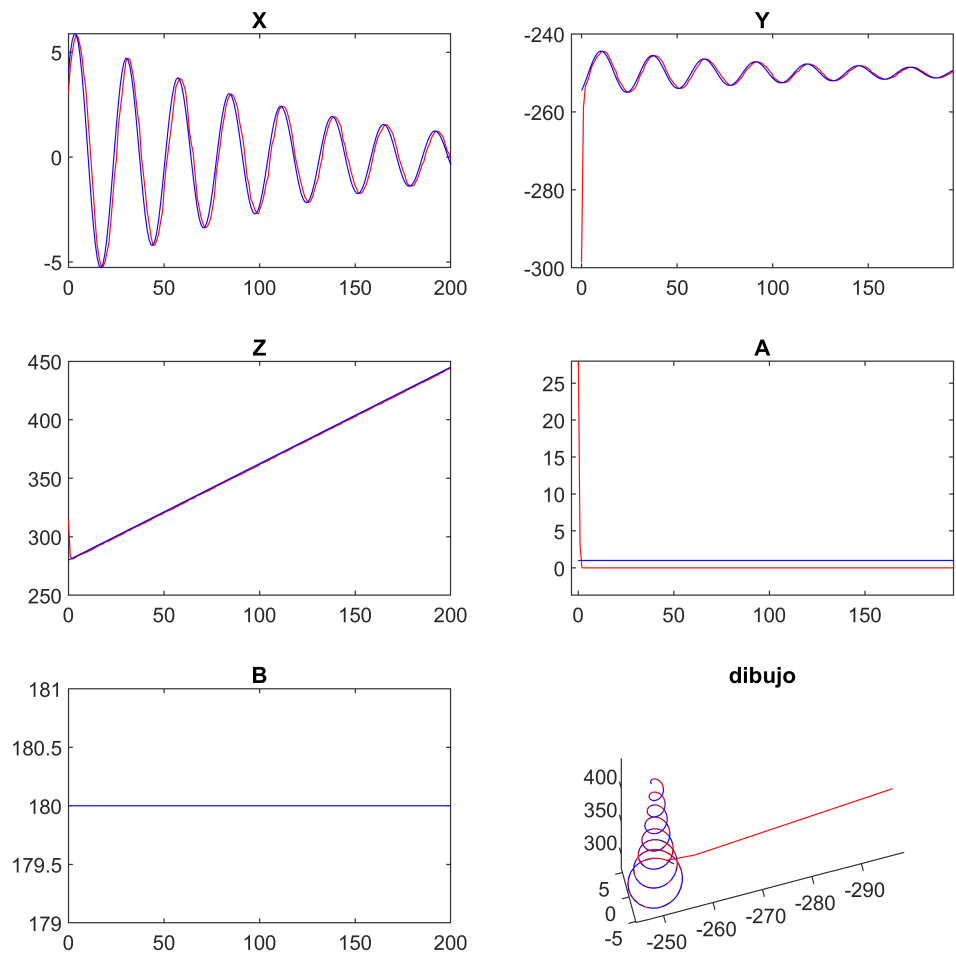


Figura 6.13: Comparación de datos Cartesianos Helicoide Exponencial

El RMSE obtenido se muestra a continuación.

Coordenada Cartesiana	RMSE
X	0.1453 mm
Y	0.1345 mm
Z	0.2349 mm
A	0 °
B	0 °

Tabla 6.3: RMSE coordenadas cartesianas dibujo de Helicoide Exponencial

En el siguiente [link](#) se puede ver el video de la demostración de la Helicoide Exponencial.

7 | Resultados y Conclusiones

En el presente trabajo se logró obtener de manera satisfactoria los modelos cinemáticos directo e inverso para el robot RV-2AJ y su simulación mediante una aplicación en Matlab y se completó con éxito el seguimiento a diversas trayectorias, obtenidas de dibujos, a través de la transmisión de puntos descritos en su forma cartesiana y articular, hacia el controlador del robot por medio de la red de computadores del departamento de electrónica.

Una de las principales dificultades fue crear una aplicación en Matlab que se comunicara con el controlador del robot programado en Melfa Basic IV, de manera que aceptara distintos modos de operación haciendo entender dos lenguajes muy distintos a través de comunicación de datos.

El hecho de transmitir datos de forma continua también fue una problemática, debido a que en un principio se planteó enviar todos los puntos de trayectoria de una sola vez, y que el controlador los guardara, los leyera, y ordenase al robot moverse por los puntos, sin embargo, no fue posible enviar más de 15 datos de posición una sola vez sin que el controlador del robot dejara de responder, teniéndolo que reiniciar. Es por ello que se eligió enviar los datos de a uno y que cuando el robot lo leyera, el controlador mandaría un acuse de recibo con su posición actual y en qué punto está, para mandar el siguiente.

La comprobación de los modelos cinemáticos realizados se hizo mediante el software CIROS Studio que cuenta con modelos de cinemática inversa y directa. Con él se pudo entender el funcionamiento de los parámetros de orientación de Mitsubishi, en específico el parámetro A, del cual en ninguna referencia se explicaba bien como obtenerlo, por lo que mediante su teach box virtual y varios puntos se realizó un ajuste lineal para obtener su fórmula.

Sobre los errores de posición de las trayectorias, estos no representan fielmente los verdaderos resultados, esto es porque no es posible obtener una sincronía en el programa Melfa para que envíe los valores al llegar al punto, por un lado, porque los envía mientras el robot se va moviendo y porque el modo CNT está activado, esto para hacer un movimiento ligado entre los puntos. Al no activar el modo CNT el robot recorre los puntos, pero se observan cambios bruscos en la velocidad, lo que hace que el dibujo no quedara muy bien.

7.1. Trabajo futuro

Con el fin de seguir investigando la forma de comunicación con el robot, se podría estudiar si es posible implementar una comunicación en tiempo real usando el comando Melfa *MXT*, con el cual se podrían resolver sistemas de control, como por ejemplo el movimiento de una carga colgante en el EF reduciendo su movimiento pendular.

También se podría estudiar aspectos como detección y evasión automática de colisiones, donde la toolbox propia de Matlab (Robotics System Toolbox) cuenta con algunos algoritmos donde se usan modelos robóticos en formato URDF.

Aterrizar este trabajo a temas relacionados con la industria también podría ser viable, como los sistemas de picaroca en minería, los cuales podrían ser tele-operados mediante una aplicación o especificar

trayectorias a seguir mediante planos CAD industriales, los cuales son fácilmente convertibles a formato SVG.

El uso de visión por computador también es un tema muy relevante, por lo que, continuando con la misma idea de dibujos, se podría tomar una foto para que, al procesarla, el robot dibuje lo capturado. La técnica de Error Diffusion Dithering podrían ser usada para este caso[17].

También se podría expandir el uso de red para manejar el robot mediante la aplicación desde cualquier parte usando la red de internet.

A | Funciones Cinemática Inversa

A.1. Cinemática Inversa geométrica

```
function theta=ik(x,y,z,Rot,type,rightleft,abovebelow)
%se debe ingresar XYZ en milímetros, tipo de rotacion ('t5', 'AB', 'RPY') en grados
%y opcional la configuracion 'de Frente' o 'de Espalda'
%y 'Codo arriba' o 'Codo abajo' (con apostrofe)
%Esta funcion soporta vectores de posiciones y entrega las coordenadas
%articulares
%Entrega como salida theta=[theta1,theta2,theta3,theta4,theta5,ok_c,ok_t,ok_all];
%donde ok_c es una flag 0/1 si el punto esta dentro del workspace y es alcanzable
%ok_t es una flag 0/1 si los valores de las coordenadas articulares estan en su rango
%y ok_all si es posible obtener ese valor
if nargin==5
    RL='de Frente';
    AB='Codo arriba';
elseif nargin==7
    RL=rightleft;
    AB=abovebelow;
else
    error(sprintf('Error al ingresar argumentos\nIngrese ...
        (x,y,z,[I],type)\ntype puede ser\n "AB" => I=[A,B]\n"RPY"=> ...
        I=[R,P,Y]\n"t5"=> I=[t5,B]\nlas configuraciones pueden ser etc'))
end

theta5back=0;
if strcmp('t5',type) || strcmp('J6',type)
    theta5back=Rot(:,1);
    phi=Rot(:,2);
elseif strcmp('AB',type)
    phi=Rot(:,2);
elseif strcmp('RPY',type)
    T=rp2tr(Rot(:,3),Rot(:,2),Rot(:,1),'deg');
    phi=acosd(T(3,3,:));
    phi=phi(:);
    theta5back=atan2d(T(3,2,:),-T(3,1,:));
end
ok_c=zeros(numel(x),1);

%EL LARGO DEL ULTIMO ESLABON ES 72+GARRA, MODIFICAR SI SE CONSIDERA ESTA
a = 300; %Longitud del hombro
b = 250; %Longitud del brazo
c = 160; %Longitud del antebrazo
%      d = 72+210; %Longitud de la muñeca + GARRA
d = 72; %Longitud de la muñeca

EFxy=sqrt(x.^2+y.^2); %distancia origen y proyeccion EF en eje xy
px2=EFxy-d*sind(phi);
py2=(z-a)-d*cosd(phi);
act3=(px2.^2+py2.^2 - (b.^2 + c.^2))./(2*b*c);
%      cuando px2~py2 theta2 es 45 y theta3 0 por
```

```

% lo que puede ocurrir error numerico y hacerse lo del
% interior de acos un poco mayor a 1 lo que da complejo
% por lo que se ajusta entre -1 y 1,
% Si es complejo es porque no es posible accede a ese punto
% y se entrega la flag ok_c =0
% Al setearlo entre -1 y 1 el resultado final estara lo mas cerca a ese punto
% al cual no se puede acceder
for i=1:numel(x)
    if gt(act3(i),1)
        act3(i)=1;
        ok_c(i)=0;
    elseif lt(act3(i),-1)
        act3(i)=-1;
        ok_c(i)=0;
    else
        ok_c(i)=1;
    end
end
gamma=atan2d(py2,px2);
switch RL
    case 'de Frente'
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        theta5=ones(numel(x),1).*theta5back;
        thetal=atan2d(y,x); %angulo entre x e y,proyeccion eje xy

        switch AB
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        case 'Codo arriba'
            theta3=acosd(act3);
            beta=atan2d(c*sind(theta3),b+c*cosd(theta3));
            theta2=(90-gamma-beta);
            theta4=phi-(theta2+theta3);

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        case 'Codo abajo'
            theta3=-acosd(act3);
            beta=atan2d(c*sind(theta3),(b+c*cosd(theta3)));
            theta2=(90-gamma-beta);
            theta4=phi-(theta2+theta3);

        end
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        case 'de Espalda'
            if theta5back >= 0
                theta5=theta5back-180;
            else
                theta5=theta5back+180;
            end

            thetalback=atan2d(y,x); %angulo entre x e y,proyeccion eje xy
            if (30<=thetalback && thetalback<=150)
                thetal=thetalback-180;
            elseif (-150<=thetalback && thetalback<=-30)
                thetal=thetalback+180;
            elseif (-30<thetalback && thetalback<30)
                thetal=thetalback+180;
            warning('No puede tener forma inversa(rear),probar con FR=1')
            end

            switch AB
            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
            case 'Codo abajo'
                theta3=acosd(act3);
                beta=atan2d(c*sind(theta3),b+c*cosd(theta3));

```

```

        theta2=(90-gamma-beta);
        theta4=phi-(theta2+theta3);

        theta2=-theta2;
        theta3=-theta3;
        theta4=-theta4;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        case 'Codo arriba'
            theta3=-acosd(act3);
            beta=atan2d(c*sind(theta3),(b+c*cosd(theta3)));
            theta2=(90-gamma-beta);
            theta4=phi-(theta2+theta3);

            theta2=-theta2;
            theta3=-theta3;
            theta4=-theta4;

        end

    end

    ok_t=zeros(numel(x),1);
    ok_all=zeros(numel(x),1);

    % caso con AB se calcula theta5
    if strcmp('AB',type)
        A=Rot(:,1);
        theta5=A-theta1.*cosd(phi);
    end

    %Comprobacion que los valores estan en sus rangos
    for i=1:numel(x)
        if (-150≤theta1(i) && theta1(i)≤150)&&(-60≤theta2(i) && ...
            theta2(i)≤120)&&(-110≤theta3(i) && theta3(i)≤120)&&(-90≤theta4(i) && ...
            theta4(i)≤90)&&(-200≤theta5(i) && theta5(i)≤200)
            ok_t(i)=1;
        else
            ok_t(i)=0;
        end
        %si se puede llegar al punto (ok_c) y esta dentro del rango ok_all=1
        if (ok_t(i)==1 && ok_c(i)==1)
            ok_all(i)=1;
        else
            ok_all(i)=0;
        end
    end
    theta=[theta1,theta2,theta3,theta4,theta5,ok_c,ok_t,ok_all];

end

```

A.2. Cinemática inversa numérica con J_M

```

function [qm,i] = iksimple(q_in,P_in)
    %Calcula la cinematica inversa numerica con el jacobiano con los
    %parametros A y Bd
    %se ingresa como entrada q_in, el cual es la estimacion inicial
    % ej q_in=[0,20,40,60,0]
    %y la posicion de destino P_in que contiene X Y Z A y B
    %con XYZ en milimetros y AB en grados
    % ej iksimple([0 10 10 90 0],[275.81 48.63 637.76 50 90])
    % como salida entrega qm e i, donde qm son todos los valores de q que fueron
    %optimizandose hasta llegar al final, e i es el numero total de iteraciones
    %Para mayor precision hacer eta mas pequeno

    eta=10^(-3)

```

```

deg = pi/180;
d1=0.3;
% d5=0.072+0.210;
d5=0.072;
a2=0.25;
a3=0.16;
% af1=-pi/2;
% af2=pi/2;

%Cinematica directa
Px=@(q) cos(q(1))*(a2*sin(q(2)) +a3*sin(q(2) + q(3))+d5*sin(q(2) + q(3) + ...
q(4)));
Py=@(q) sin(q(1))*(a2*sin(q(2)) +a3*sin(q(2)+q(3)) + d5*sin(q(2) + q(3) + q(4)));
Pz=@(q) d1 + a2*cos(q(2)) + a3*cos(q(2)+q(3)) + d5*cos(q(2)+q(3)+q(4));
A=@(q) q(5)+q(1)*cos(q(2)+q(3)+q(4));
B=@(q) q(2)+q(3)+q(4);

%Jacobiano
Jac=@(q) [-sin(q(1))*(a2*sin(q(2))+a3*sin(q(2)+q(3))+d5*sin(q(2)+q(3)+q(4))), ...
cos(q(1))*(a2*cos(q(2))+a3*cos(q(2)+q(3))+d5*cos(q(2)+q(3)+q(4))), ...
cos(q(1))*(a3*cos(q(2)+q(3))+d5*cos(q(2)+q(3)+q(4))), ...
d5*cos(q(1))*cos(q(2)+q(3)+q(4)),0; ... %dx/q
cos(q(1))*(a2*sin(q(2))+a3*sin(q(2)+q(3))+d5*sin(q(2)+q(3)+q(4))), ...
(a2*cos(q(2))+a3*cos(q(2)+q(3))+d5*cos(q(2)+q(3)+q(4)))*sin(q(1)), ...
(a3*cos(q(2)+q(3))+d5*cos(q(2)+q(3)+q(4)))*sin(q(1)), ...
d5*cos(q(2)+q(3)+q(4))*sin(q(1)),0; ... %dy/q
0,...
-a2*sin(q(2))-a3*sin(q(2)+q(3))-d5*sin(q(2)+q(3)+q(4)), ...
-a3*sin(q(2)+q(3))-d5*sin(q(2)+q(3)+q(4)), ...
-d5*sin(q(2)+q(3)+q(4)),0; ... %dz/q
cos(q(2)+q(3)+q(4)), ...
-q(1)*sin(q(2)+q(3)+q(4)), ...
-q(1)*sin(q(2)+q(3)+q(4)), ...
-q(1)*sin(q(2)+q(3)+q(4)),1; ... %dA/q
0,1,1,1,0] ; %dB/q

i=1;

q{i}=( [q_in(1) q_in(2) q_in(3) q_in(4) q_in(5)] '*deg); %inicio INPUT

Pxd=P_in(1)/1000;
Pyd=P_in(2)/1000;
Pzd=P_in(3)/1000;

Ad=P_in(4)*deg;
Bd=P_in(5)*deg;
error=[Pxd;Pyd;Pzd;Ad;Bd]-[Px(q{i});Py(q{i});Pz(q{i});A(q{i});B(q{i})];
alpha=0.05;

while norm(error)>eta
J=Jac(q{i});
% j_pseudo=J'/(J*J'); %m=3<n=5
% j_pseudo=(J'*J)\J'; %m=6>n=5
j_pseudo=inv(J); %m=5=n=5
A_q=alpha*j_pseudo*error;
q{i+1}=q{i}+A_q;

error=[Pxd;Pyd;Pzd;Ad;Bd]-...
[Px(q{i+1});Py(q{i+1});Pz(q{i+1});A(q{i+1});B(q{i+1})];
i=i+1;
end
qm=cell2mat(q)/deg;

%opcional dibuja la evolucion de las estimaciones
% figure
% t=linspace(0,numel(qm(1,:)),numel(qm(1,:)));

```

```

%           subplot(3,2,1);
%           plot(t,qm(1,:))
%           title(['q',num2str(1)])
%           subplot(3,2,2);
%           plot(t,qm(2,:))
%           title(['q',num2str(2)])
%           subplot(3,2,3);
%           plot(t,qm(3,:))
%           title(['q',num2str(3)])
%           subplot(3,2,4);
%           plot(t,qm(4,:))
%           title(['q',num2str(4)])
%           subplot(3,2,5);
%           plot(t,qm(5,:))
%           title(['q',num2str(5)])

end

```

A.3. Cinemática inversa numérica con J_a

```

function [qm,i] = iksimple_rpy_Ja(q_in,P_in)
%Calcula la cinematica inversa numerica con el jacobiano con roll
% pitch y yaw
%se ingresa como entrada q_in, el cual es la estimacion inicial
% ej q_in=[0,20,40,60,0]
%y la posicion de destino P_in que contiene X Y Z A y B
%con XYZ en milímetros y RPY en grados
% ej iksimple_rpy_Ja([5 40 -25 80 45],[275.81 48.63 637.7 100 40 90])
% como salida entrega qm e i, donde qm son todos los valores de q que fueron
%optimizandose hasta llegar al final, e i es el numero total de iteraciones
%Para mayor precision hacer eta mas pequeno

eta=10^(-3)
deg = pi/180;
d1=0.3;
% d5=0.072+0.210;
d5=0.072;
a2=0.25;
a3=0.16;

%Cinematica directa

Px=@(q) cos(q(1))*(a2*sin(q(2)) +a3*sin(q(2)+q(3))+d5*sin(q(2)+q(3)+...
q(4)));
Py=@(q) sin(q(1))*(a2*sin(q(2)) +a3*sin(q(2)+q(3)) + d5*sin(q(2)+q(3)+q(4)));
Pz=@(q) d1 + a2*cos(q(2)) + a3*cos(q(2)+q(3)) + d5*cos(q(2)+q(3)+q(4));

r11=@(q) cos(q(1))*cos(q(2)+q(3)+q(4))*cos(q(5)) - sin(q(1))*sin(q(5));
r21=@(q) sin(q(1))*cos(q(2)+q(3)+q(4))*cos(q(5)) + cos(q(1))*sin(q(5));
r31=@(q) -sin(q(2)+q(3)+q(4))*cos(q(5));
r12=@(q) -cos(q(1))*cos(q(2)+q(3)+q(4))*sin(q(5)) - sin(q(1))*cos(q(5));
r22=@(q) sin(q(1))*cos(q(2)+q(3)+q(4))*sin(q(5)) + cos(q(1))*cos(q(5));
r32=@(q) sin(q(2)+q(3)+q(4))*sin(q(5));
r13=@(q) cos(q(1))*sin(q(2)+q(3)+q(4));
r23=@(q) sin(q(1))*sin(q(2)+q(3)+q(4));
r33=@(q) cos(q(2)+q(3)+q(4));

roll=@(q) atan2(r21(q),r11(q));
pitch=@(q) atan2(-r31(q),sqrt(r11(q)^2 + r21(q)^2));
yaw=@(q) atan2(r32(q),r33(q));

A=@(q) q(5)+q(1)*cos(q(2)+q(3)+q(4));
B=@(q) q(2)+q(3)+q(4);

```

```

Jac=@(q)[-sin(q(1))*(a2*sin(q(2))+a3*sin(q(2)+q(3))+d5*sin(q(2)+q(3)+q(4))), ...
cos(q(1))*(a2*cos(q(2))+a3*cos(q(2)+q(3))+d5*cos(q(2)+q(3)+q(4))), ...
cos(q(1))*(a3*cos(q(2)+q(3))+d5*cos(q(2)+q(3)+q(4))), ...
d5*cos(q(1))*cos(q(2)+q(3)+q(4)),0; ... %dx/q
cos(q(1))*(a2*sin(q(2))+a3*sin(q(2)+q(3))+d5*sin(q(2)+q(3)+q(4))), ...
(a2*cos(q(2))+a3*cos(q(2)+q(3))+d5*cos(q(2)+q(3)+q(4)))*sin(q(1)), ...
(a3*cos(q(2)+q(3))+d5*cos(q(2)+q(3)+q(4)))*sin(q(1)), ...
d5*cos(q(2)+q(3)+q(4))*sin(q(1)),0; ... %dy/q
0,...
-a2*sin(q(2))-a3*sin(q(2)+q(3))-d5*sin(q(2)+q(3)+q(4)), ...
-a3*sin(q(2)+q(3))-d5*sin(q(2)+q(3)+q(4)), ...
-d5*sin(q(2)+q(3)+q(4)),0; ... %dz/q
1, (sin(roll(q)-q(1))*sin(pitch(q)))/cos(pitch(q)), ...
(sin(roll(q)-q(1))*sin(pitch(q)))/cos(pitch(q)), ...
(sin(roll(q)-q(1))*sin(pitch(q)))/cos(pitch(q)), ...
(cos(q(2)+q(3)+q(4))*cos(pitch(q))+ ...
sin(q(2)+q(3)+q(4))*cos(roll(q))*cos(q(1))*sin(pitch(q))...
+ sin(q(2)+q(3)+q(4))*sin(pitch(q))*...
sin(roll(q))*sin(q(1))/cos(pitch(q));...droll/q
0, cos(roll(q)-q(1)), cos(roll(q)-q(1)), ...
cos(roll(q)-q(1)), -sin(roll(q)-q(1))*sin(q(2)+q(3)+q(4));... ..
dpitch/q
0, sin(roll(q)-q(1))/cos(pitch(q)), ...
sin(roll(q)-q(1))/cos(pitch(q)), sin(roll(q)- ...
q(1))/cos(pitch(q)), ...
(cos(roll(q)-q(1))*sin(q(2)+q(3)+q(4))/cos(pitch(q))]; ...
%dyaw/q

i=1;

q{i}=[q_in(1) q_in(2) q_in(3) q_in(4) q_in(5)]*deg; %inicio INPUT

Pxd=P_in(1)/1000;
Pyd=P_in(2)/1000;
Pzd=P_in(3)/1000;

Rd=P_in(4)*deg;
Pd=P_in(5)*deg;
Yd=P_in(6)*deg;
error=[Pxd;Pyd;Pzd;Rd;Pd;Yd]-...
[Px(q{i});Py(q{i});Pz(q{i});roll(q{i});pitch(q{i});yaw(q{i})];
alpha=0.05;

while norm(error)>eta
    J=Jac(q{i});
    % j_pseudo=J'/(J*J'); %m=3<n=5
    j_pseudo=(J'*J)\J'; %m=6>n=5
    Δ_q=alpha*j_pseudo*error;
    q{i+1}=q{i}+Δ_q;

    error=[Pxd;Pyd;Pzd;Rd;Pd;Yd]-...
    [Px(q{i+1});Py(q{i+1});Pz(q{i+1});roll(q{i+1});pitch(q{i+1});yaw(q{i+1})];
    i=i+1;
    disp(i)
end
qm=cell2mat(q)/deg;
end

```

B | Configuración RT Toolbox2 para conexión

Teniendo un programa ya abierto, configurado y conectado en línea con el controlador del robot (IP robot=10.2.43.99), se debe configurar el puerto de enlace por el cual se comunica la aplicación de Matlab con el controlador del robot.

Para esto en la lista desplegable del “Workspace” se debe acceder a Online → Parameter → Communication Parameter → Ethernet. Luego se selecciona en la nueva ventana Device Line en donde se muestran las 8 posibles líneas de comunicación (COM1 a COM8) a las cuales se les vincula con un dispositivo de la lista central (OPT11-OPT19). En este proyecto se vincula el dispositivo OPT14 a la línea de comunicación COM4.

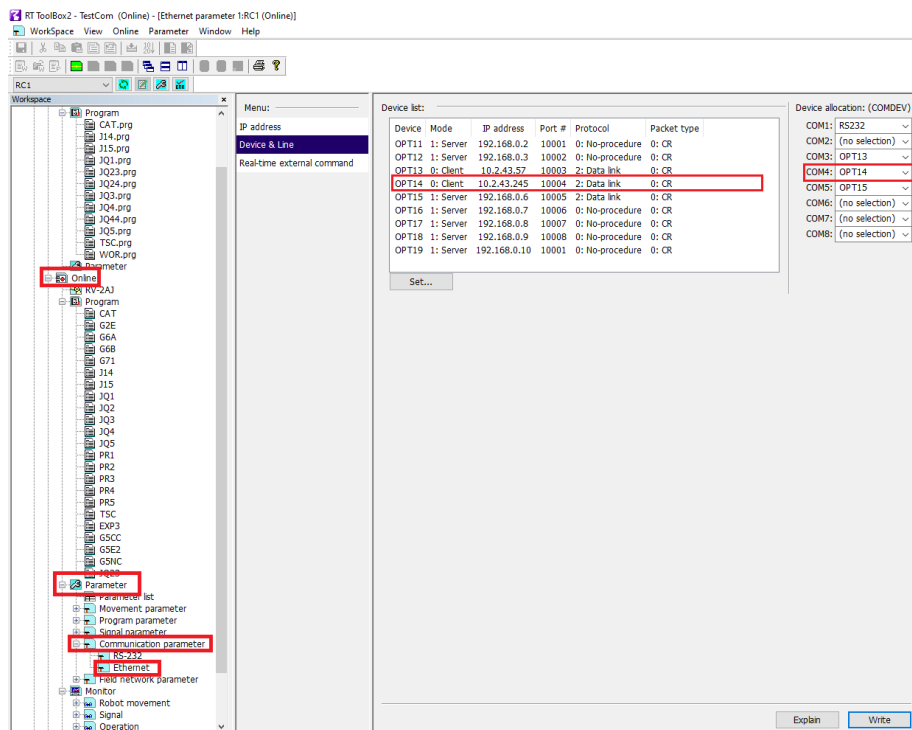


Figura B.1: Vista de RT Toolbox2 para configurar la línea de comunicación

Luego se deben configurar los parámetros de la línea de comunicación haciendo click en “Set”.

Device parameter setting

Device: OPT14

Autoconfiguration: None

Mode: (NETMODE(4)) 0: Client

IP address: (NETHSTIP(4)) 10 . 2 . 43 . 245

Port #: (NETPORT(5)) 10004

Protocol: (CPRCE14) 2: Data link

Packet type: (CTERME14) 0: CR

Allocation: (COMDEV) COM4

OK Cancel

Figura B.2: Configuración de parámetros

Debido a que la aplicación en Matlab genera el socket de servidor, el controlador se configura como cliente con el parámetro NETMODE(4). La dirección IP del parámetro NETHSTIP(4) debe ser la misma dirección IP con la que se crea el servidor, es decir con la IP del computador con la Aplicación en este caso la IP es '10.2.43.245'. El puerto se deja como está con NETPORT(5)=10004. Este puerto se especifica en la creación del servidor. El protocolo (CPRCE14) se especifica como 2:Data Link. El terminador CTERME14 se deja como 0: CR (Retorno de carro) y COMDEV es a que línea de comunicación está asociado dejándolo en COM4.

La configuración del protocolo TCP/IPV4 del computador con la aplicación de Matlab, conectado a la red de computadores del departamento de Electrónica es la siguiente.

Propiedades de Habilitar el protocolo de Internet versión 4 (TCP/IP... X

General

Puede hacer que la configuración IP se asigne automáticamente si la red admite esta funcionalidad. De lo contrario, deberá consultar con el administrador de red cuál es la configuración IP apropiada.

☐ Obtener una dirección IP automáticamente

☒ Usar la siguiente dirección IP:

Dirección IP: 10 . 2 . 43 . 245

Máscara de subred: 255 . 255 . 255 . 0

Puerta de enlace predeterminada: 10 . 2 . 43 . 1

☐ Obtener la dirección del servidor DNS automáticamente

☒ Usar las siguientes direcciones de servidor DNS:

Servidor DNS preferido: 200 . 1 . 17 . 4

Servidor DNS alternativo: 200 . 1 . 21 . 80

☐ Validar configuración al salir

Opciones avanzadas...

Aceptar Cancelar

Figura B.3: Configuración de parámetros

C | Codigo Melfa Basic IV completo

```

1  OVRD 100          'Selecion de velocidad
2  '*****
3  '*****Inicializacion de Variables*****
4  '*****
5  DEF JNT J0        'Posicion de reposo
6  DEF JNT JN        'Coordenada articular dibujo
7  DEF POS PN        'Coordenada cartesiana dibujo
8  DEF JNT JT        'Coordenada articular teach
9  DEF POS PT        'Coordenada cartesiana teach
10 DEF INTE M        'tipo de modo
11 DEF INTE JUMPN    'Salto o no salto en dibujo
12 DEF INTE N        'numero de puntos del dibujo
13 DEF INTE X        'punto actual del dibujo
14 DEF INTE D        'distancia en Z de salto
15 CLOSE            'cierra lineas de comunicacion si estuvieran abiertas
16 D=-5
17 CNT 1            'Modo continuo
18 '*****
19 J0=(-90.00,+45.00,+45.00,+0.00,+90.00,+0.00)
20 MOV J0            'Inicia en punto inicial sobre la Hoja
21 '*****
22 *RETRY            'genera loop infinito hasta que se inicie el servidor
23 OPEN "COM4:" AS #1
24 IF M_OPEN(1) <> 1 THEN GOTO *RETRY
25 '*****
26 '*****
27 '*****
28 INPUT #1,M%       'Boton inciar programa manda modo inicial
29 *MODE             'seleccion de modo (JTCH,PTCH,DRW)
30 SELECT M%
31 CASE 1            'Modo=1 es Teach con coordenadas articulares
32 GOSUB *JTCH
33 BREAK
34 CASE 2            'Modo=2 es Teach con coordenadas cartesianas
35 GOSUB *PTCH
36 BREAK
37 CASE 3            'Modo=3 es dibujo con coordenadas articulares
38 GOSUB *DRW
39 BREAK
40 CASE 4            'Modo=4 es dibujo con coordenadas cartesianas
41 GOSUB *DRW
42 BREAK
43 END SELECT
44 GOTO *MODE
45 END
46 '*****
47 '*****SUBROUTINAS*****
48 '*****
49 *JTCH             'Joint Teach
50 WHILE 1
51 PRINT #1,J_CURR   'se mandan para ver error de J

```

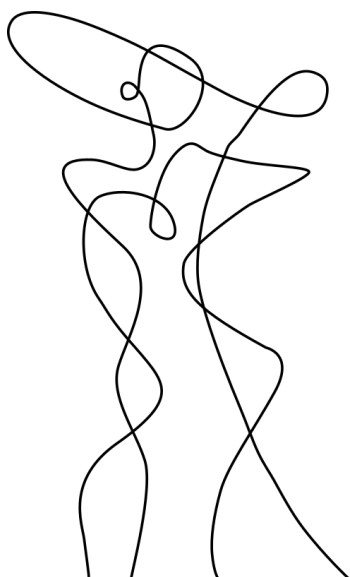
```

52 INPUT #1,JT,M          'recibo de coordenada y modo
53 IF M <> 1 THEN *OJT     'si el modo es distinto de 1 retorna a seleccion de modo
54 MOV JT
55 DLY 1
56 WEND
57 *OJT
58 RETURN
59 '*****
60 '*****
61 *PTCH                  'Position Teach
62 WHILE 1
63 PRINT #1,P_CURR       'se mandan para ver error de P
64 INPUT #1,PT,M          'recibo de coordenada y modo
65 IF M <> 2 THEN *OPT     'si el modo es distinto de 2 retorna a seleccion de modo
66 MOV PT
67 DLY 1
68 WEND
69 *OPT
70 RETURN
71 '*****
72 '*****
73 *DRW                   'Dibujo (Draw)
74 INPUT #1,M%            'Recibo de modo (3 o 4)
75 IF M<3 THEN RETURN     'si es modo 1 o 2 se devuelve a seleccion de modo
76 INPUT #1,N%            'Recibo de numero de puntos
77 SELECT M%              'Selecciona si es con coordenadas articulares o cartesianas
78 CASE 3
79 GOSUB *DRWJ
80 BREAK
81 CASE 4
82 GOSUB *DRWP
83 BREAK
84 END SELECT
85 MOV J0                 'Al terminar de dibujar se mueve a J0
86 RETURN                 'Vuelve a seleccion de Modo
87 '*****
88 *DRWJ                  'Modo articular
89 FOR X=1 TO N            'Itera entre la cantidad de puntos
90 INPUT #1,JN,M           'Recibe punto y modo
91 IF M <> 3 THEN *ODJ     'modo distinto de 3 retorna a select modo(Solo si se ...
    aprieta stop)
92 MOV JN
93 PRINT #1,J_CURR,X      'Envia posicion actual y punto actual
94 NEXT X                  'Siguiente X
95 *ODJ
96 RETURN                 'retorna a linea siguiendo donde se llamo a *DRWJ
97 '*****
98 *DRWP                  'Modo cartesiano
99 FOR X=1 TO N            'Itera entre la cantidad de puntos
100 INPUT #1,PN,JUMP,N,M   'Recibe punto ,salto y modo
101 IF M <> 4 THEN *ODP     'modo distinto de 4 retorna a select modo(Solo si se ...
    aprieta stop)
102 IF JUMP=0 THEN         'Si se debe hacer un salto o no
103 MOV PN
104 PRINT #1,P_CURR,X      'Envia posicion actual y punto actual
105 ELSE
106 CNT 0                  'Modo continuo desactivado
107 MVS ,D                 'Mueve lineal -5mm en direccion al eje Z de la garra
108 MOV PN,D               'Mueve sobre punto de destino
109 MVS PN                 'Mueve a punto de destino
110 CNT 1                  'Se reactiva modo continuo
111 PRINT #1,P_CURR,X      'Envia posicion actual y punto actual
112 ENDIF
113 NEXT X                  'Siguiente X
114 *ODP
115 RETURN                 'retorna a linea siguiendo donde se llamo a *DRWP
116 '*****

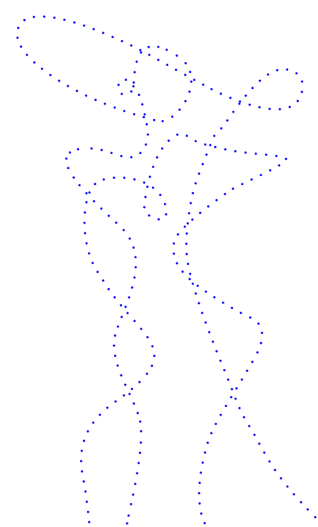
```

D | Dibujos

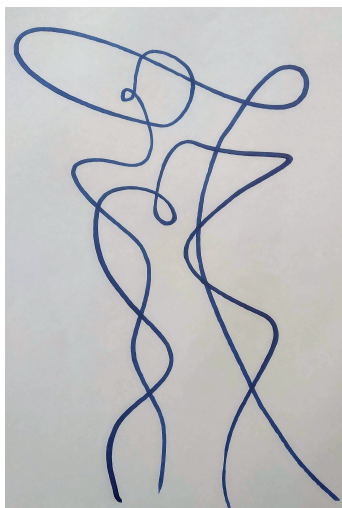
■ Mujer con sombrero



(a) Figura original



(b) Figura extraída de SVG



(c) Figura dibujada

Figura D.1: Comparación dibujos de Mujer con sombrero

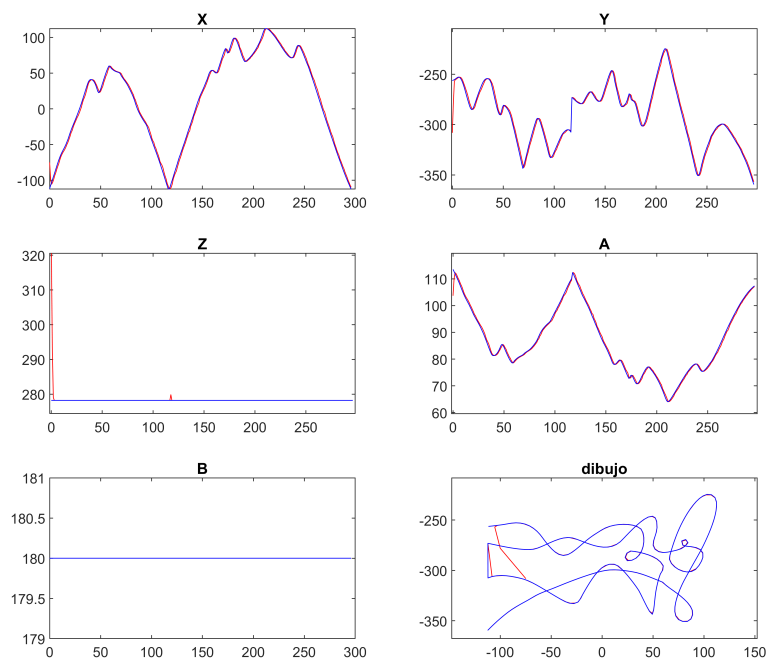


Figura D.2: Comparación de datos Cartesianos Mujer con sombrero

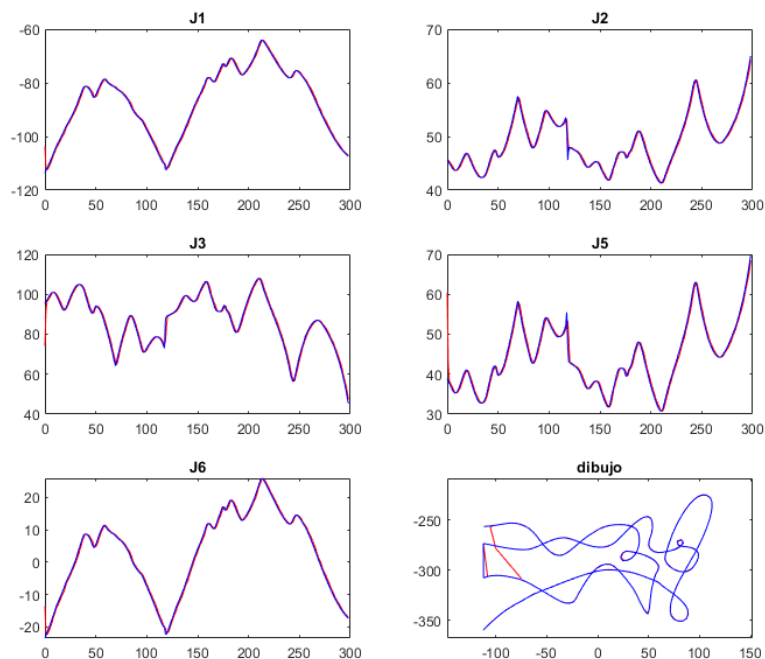


Figura D.3: Comparación de datos Articulares Mujer con sombrero

Coordenada Cartesiana	error
X	4.1958 mm
Y	4.346 mm
Z	2.8368 mm
A	0.9313 °
B	0 °

Tabla D.2: RMSE coordenadas cartesianas dibujo de Mujer con sombrero

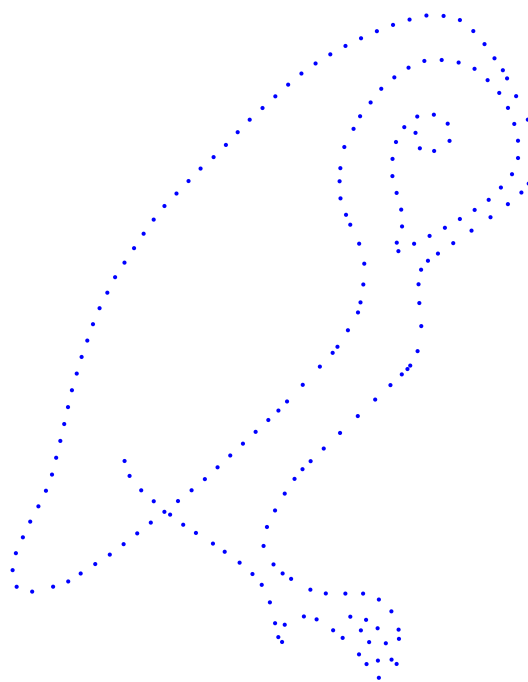
Coordenada articular	RMSE(°)
J1	0.9269 °
J2	0.5557 °
J3	1.9187 °
J5	1.5851 °
J6	0.9269 °

Tabla D.1: RMSE coordenadas articulares dibujo de Mujer con sombrero

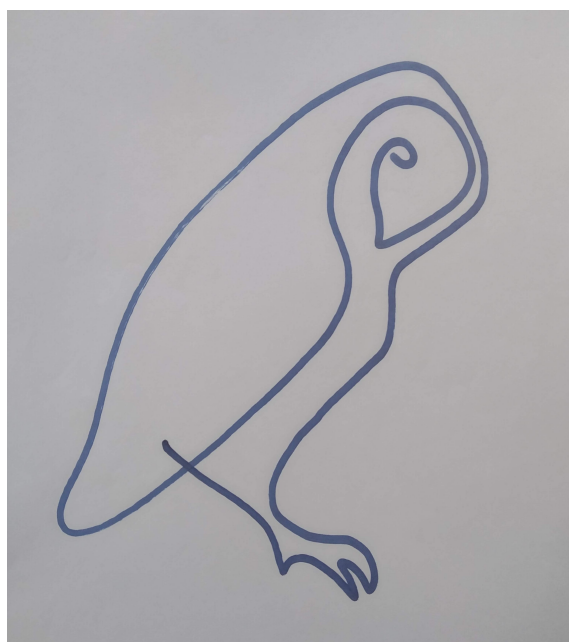
■ Búho



(a) Figura original



(b) Figura extraída de SVG



(c) Figura dibujada

Figura D.4: Comparación dibujos de Búho

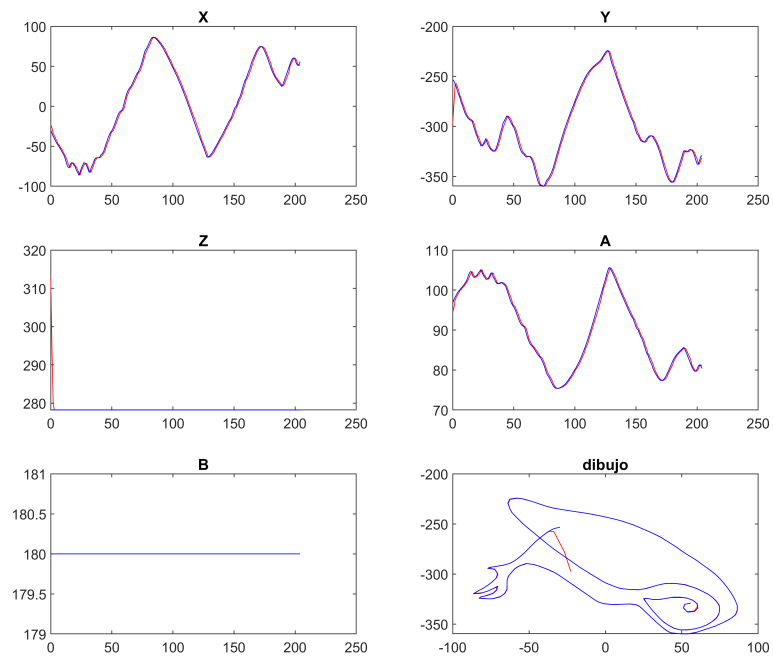


Figura D.5: Comparación de datos Cartesianos Búho

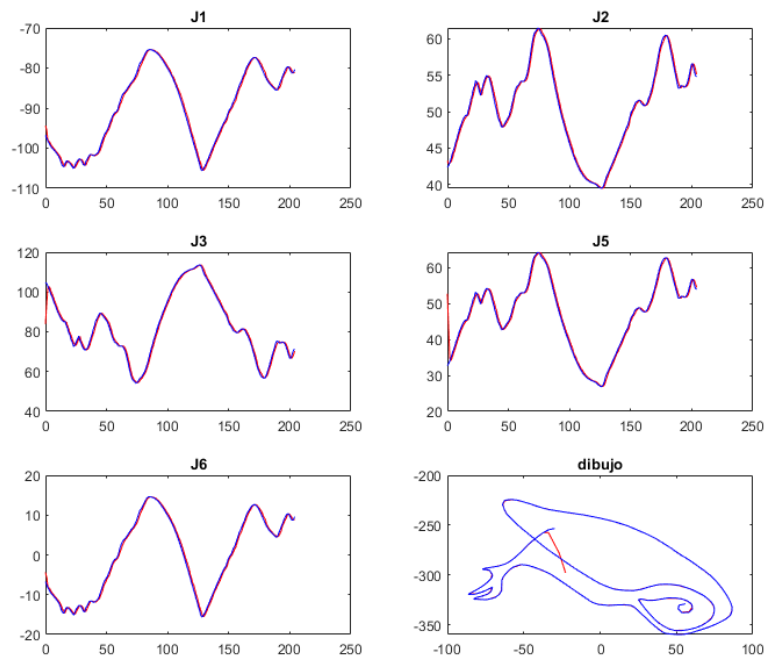


Figura D.6: Comparación de datos Articulares Búho

Coordenada articular	RMSE(°)
J1	0.7032 °
J2	0.5178 °
J3	2.117 °
J5	1.7744 °
J6	0.7032 °

Tabla D.3: RMSE coordenadas articulares dibujo de Búho

Coordenada Cartesiana	error
X	3.5672 mm
Y	4.532 mm
Z	2.6692 mm
A	0.7003 °
B	0 °

Tabla D.4: RMSE coordenadas cartesianas dibujo de Búho

■ Gato



(a) Figura original



(b) Figura extraída de SVG



(c) Figura dibujada

Figura D.7: Comparación dibujos de Gato

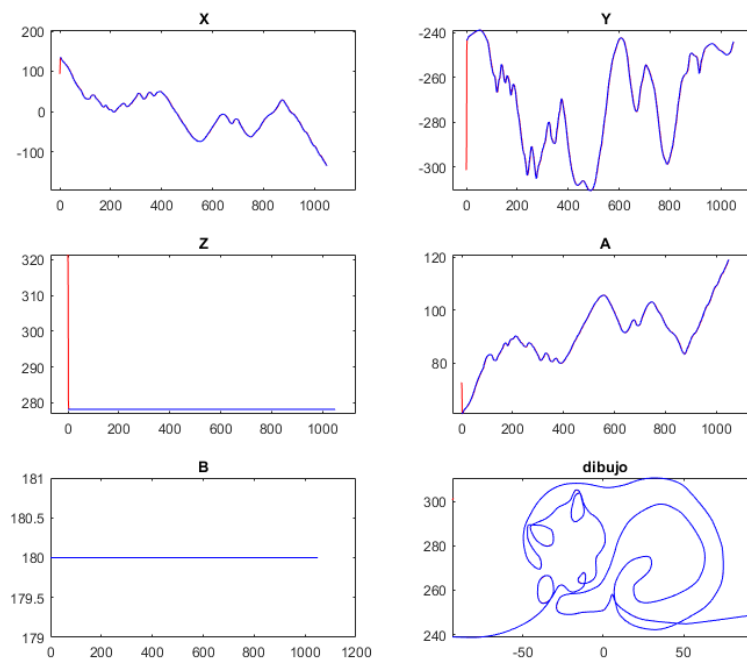


Figura D.8: Comparación de datos Cartesianos Gato

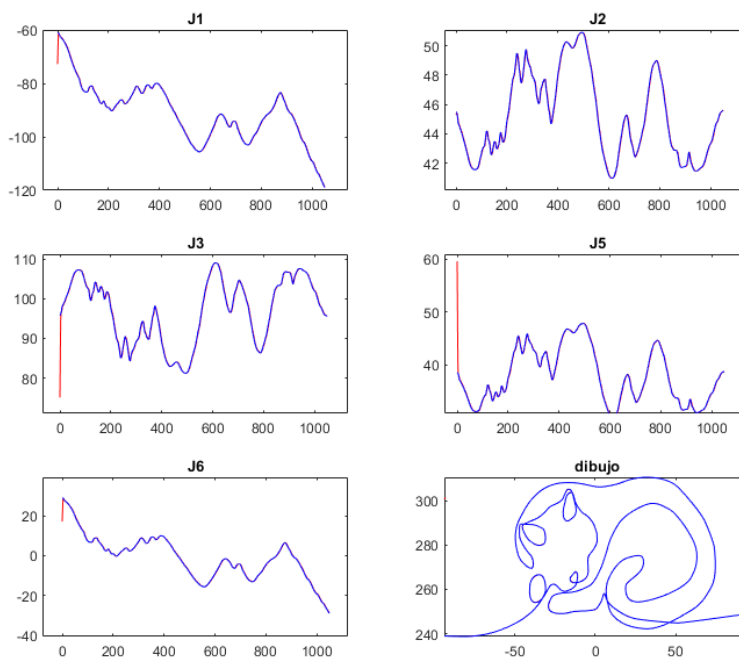


Figura D.9: Comparación de datos Articulares Gato

Coordenada articular	RMSE(°)
J1	0.512137 °
J2	0.100389 °
J3	0.857703 °
J5	0.838207 °
J6	0.512137 °

Tabla D.5: RMSE coordenadas articulares dibujo de Gato

Coordenada Cartesiana	error
X	1.883504 mm
Y	2.403819 mm
Z	1.713108 mm
A	0.508231 °
B	0 °

Tabla D.6: RMSE coordenadas cartesianas dibujo de Gato

■ Mapamundi



(a) Figura original



(b) Figura extraída de SVG



(c) Figura dibujada

Figura D.10: Comparación dibujos de Mapamundi

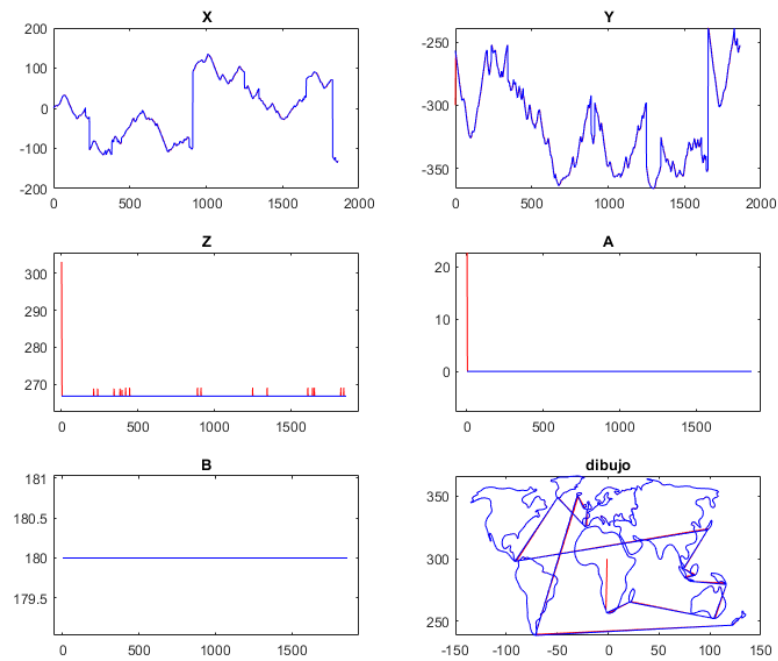


Figura D.11: Comparación de datos Cartesianos Mapamundi (A fijo)

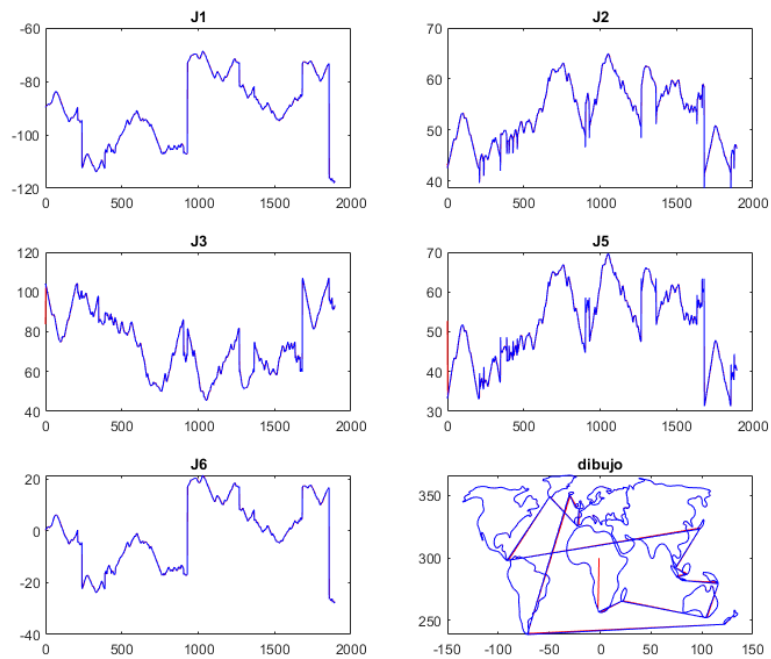


Figura D.12: Comparación de datos Articulares Mapamundi

Coordenada articular	RMSE(°)
J1	0.9634 °
J2	0.4381 °
J3	1.0298 °
J5	0.844 °
J6	0.9634 °

Tabla D.7: RMSE coordenadas articulares dibujo de Mapamundi

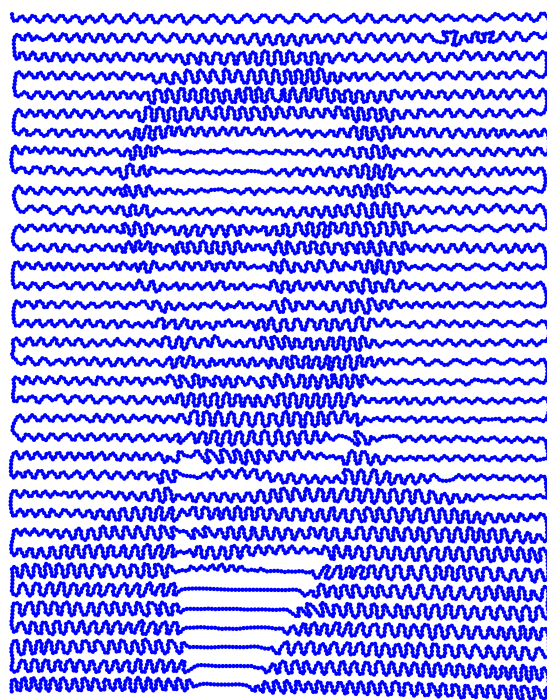
Coordenada Cartesiana	error
X	0.9663 mm
Y	1.4949 mm
Z	2.5473 mm
A	0 °
B	0 °

Tabla D.8: RMSE coordenadas cartesianas dibujo de Mapamundi

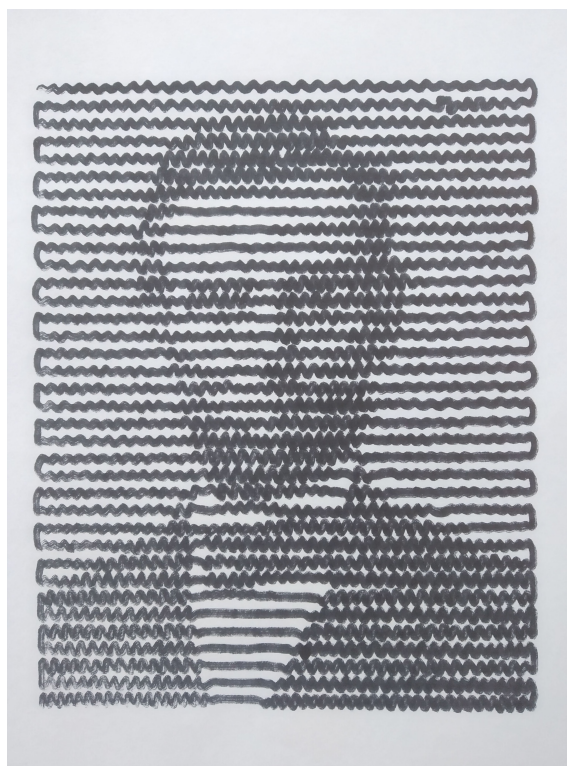
■ Abraham Lincoln



(a) Figura original



(b) Figura extraída de SVG



(c) Figura dibujada

Figura D.13: Comparación dibujos de Abraham Lincoln

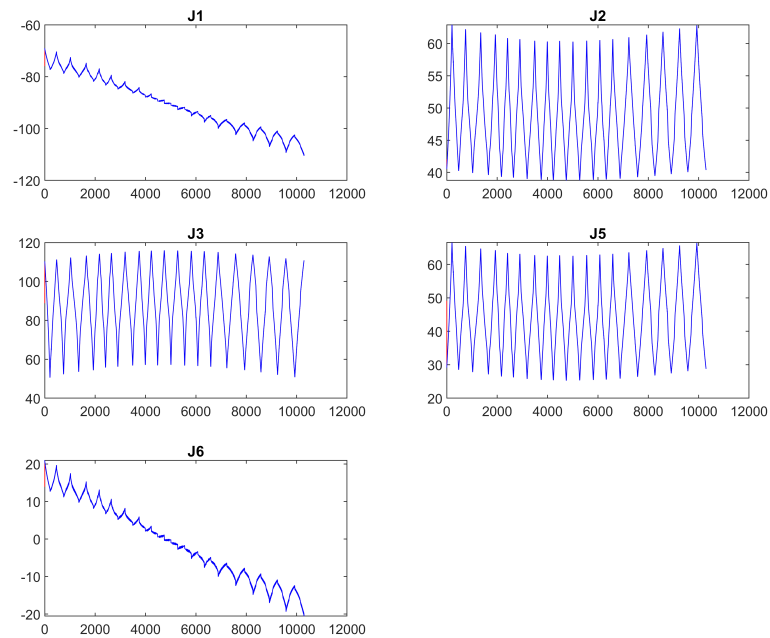
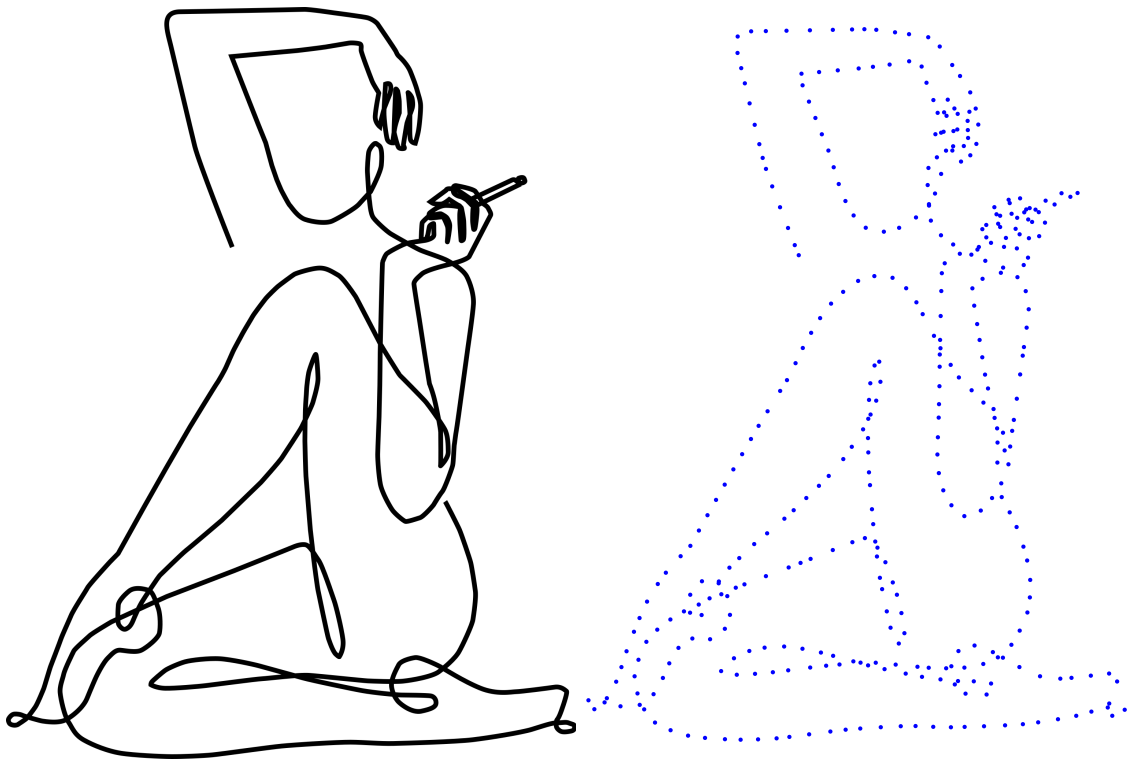


Figura D.14: Comparación de datos Articulares Abraham Lincoln

Coordenada articular	RMSE(°)
J1	0.1735 °
J2	0.1106 °
J3	0.3947 °
J5	0.3081 °
J6	0.1741 °

Tabla D.9: RMSE coordenadas articulares dibujo de Abraham Lincoln

■ Mujer con cigarro



(a) Figura original

(b) Figura extraída de SVG



(c) Figura dibujada

Figura D.15: Comparación dibujos de Mujer con Cigarro

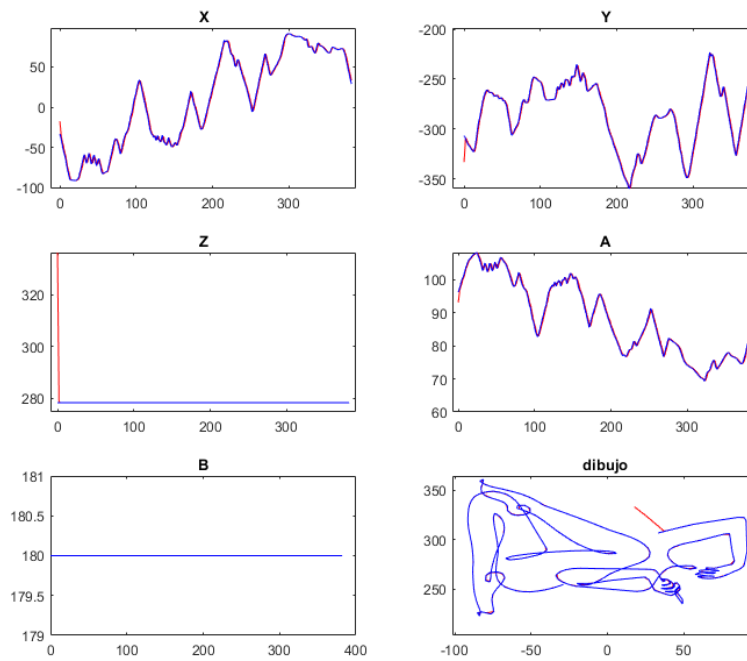


Figura D.16: Comparación de datos Cartesianos Mujer con Cigarro

Coordenada Cartesiana	RMSE
X	3.46443 mm
Y	3.278642 mm
Z	3.47472 mm
A	0.697659 °
B	0 °

Tabla D.10: RMSE coordenadas cartesianas dibujo de Mujer con Cigarro

E | SVG a Datos

E.1. Programa en Matlab

Este código invoca al código en python, procesa los puntos y los guarda como archivos “.mat”. Usa la función “W_circular_fnc” que obtiene los radios de los círculos que describen el Workspace dado un Z y B, la cual invoca a su vez a la función de la cinemática Inversa.

```
%%PARA GUARDAR SOLO LOS DATOS J,XY,salto
% como entrada el nombre del archivo SVG, mm_x_pixel, y si el dibujo
% es en vertical u horizontal
%      ej SVG_2_data('World map.svg',1,1)

function SVG_2_data(svgfile,mm_x_pixel,rot)

% hay que reiniciar las clases
% clear classes
% m = py.importlib.import_module('svgpath');
% py.importlib.reload(m);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% configuraciones para generar dibujos

% rot=0; %decide si se rota la imagen o no
% svgfile = 'World map.svg';mm_x_pixel=1; rot=1; %1103 rotate
% svgfile = 'owl2.svg';mm_x_pixel=5; %218
% svgfile='392.svg';mm_x_pixel=5;rot=0 %407
% svgfile = 'lady.svg';mm_x_pixel=5;
% svgfile = 'pepe.svg';mm_x_pixel=5;rot=1;
% svgfile = 'piano3.svg';mm_x_pixel=1;rot=1;
% svgfile = 'zom.svg';mm_x_pixel=5;
% svgfile = 'trollface.svg';mm_x_pixel=5;
% svgfile = 'one.svg';mm_x_pixel=1;
% svgfile = 'cat.svg';mm_x_pixel=1;rot=1; %hor
% svgfile = 'lincoln2.svg';mm_x_pixel=1;
% svgfile = 'USM3.svg';mm_x_pixel=1;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% hoja oficio 330x216 debe ser menor ya que tambien depende de en que punto
% de z esta el plano
w=270;
h=150;
if rot==0
    size_board=[w, h];
else
    size_board=[h, w];
end
% escala el dibujo en factor scale
scale=1;
hold on
% estimacion de z0, sirve para obtener los radios de los circulos
```

```

% concentricos con la funcion W_circular_fnc
% depende del valor que muestre el teachbox con el lapiz tocando la hoja
z0=278.21;
[r,R]=W_circular_fnc(z0);

%se ejecuta funcion svg_draw la cual retorna puntos de trayectoria sacados
%de imagen .svg
% como entrada esta el archivo, el tamaño de tablero y milímetros por pixel
% mientras mas pequeño sea el valor mas puntos se generan y mas preciso es
% el dibujo
data=py.svgpath.svg_draw(svgfile,size_board,mm_x_pixel);

% Salida
d_p=data{1};      %puntos x,y
np=data{3};       %numero de puntos

point_x=zeros(1,int64(np));
point_y=zeros(1,int64(np));

b=h*1.1;
% describe ecuacion despejada de  $R^2=(a/2)^2 + (r+b)^2$  que se forma con el
% triangulo
a=@(b) 2*sqrt(R^2 - (r+b)^2);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Escala los puntos generados a un rectangulo en z0 dentro del workspace
% con phi=180
% los puntos generados forman un rectangulo de size_board con la esquina
% inferior izquierda en (0,0) por lo que hay que centrarlo a la hoja en
% j1=-90
if rot==0
    for i = 1:length(d_p)
        point_x(i)=d_p{i}.x*scale;
        point_y(i)=d_p{i}.y*scale;
    end
    plot(point_x,point_y)
    point_x=point_x- max(point_x)/2;
    point_y=point_y+ r+(h*1.15-max(point_y))/2;
else
    for i = 1:length(d_p)
        point_x(i)=d_p{i}.y ;
        point_y(i)=-d_p{i}.x ;
    end
    plot(point_x,point_y)
    point_x=point_x- max(point_x)/2;
    point_y=point_y+ r*1.1 -min(point_y);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% se calcula distancia entre puntos consecutivos
dis=zeros(1,length(d_p)-1);
dis_ok=zeros(1,length(d_p)-1);

for k= 1:length(d_p)-1
    dis(k)= sqrt((point_x(k+1)-point_x(k))^2 + (point_y(k+1)-point_y(k))^2);
    if dis(k)<8
        dis_ok(k)=1;
    else
        dis_ok(k)=0;
        scatter(point_x(k),point_y(k), 'filled')
    end
end

```

```

end
dis_ok=[1,dis_ok];

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  círculos que describen workspace
viscircles([0 0],r);
viscircles([0 0],R);

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  rectángulo x y w h
rectangle("Position",[-a(b)/2 r a(b) b])
rectangle("Position",[-a(b)/2 -r-b a(b) b])
rectangle("Position",[-165 r-(216-157)/2 330 216],'EdgeColor','b')
rectangle("Position",[-165 -(r-(216-157)/2)-216 330 216],'EdgeColor','b')
% rectangle("Position",[0 0 270 150],'EdgeColor','r')
axis equal
plot(point_x,point_y)
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Los puntos generados hay que rotarlos 180 grados
theta = 180;
R = [cosd(theta) -sind(theta); sind(theta) cosd(theta)];
point = [point_x;point_y];
rotpoint = R*point;
plot(rotpoint(1,:),rotpoint(2,:))
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% guardado de archivo con puntos
z=zeros(length(d_p),1)+z0;
n_points=length(d_p);
i = (1:n_points);

name=extractBefore(svgfile, '.svg');

%           x           y           z
toW_P=[rotpoint(1,:);rotpoint(2,:);z';dis_ok]';
%           x           y           z           A
toW_PA=[rotpoint(1,:);rotpoint(2,:);z';-atan2d(rotpoint(2,:),rotpoint(1,:));dis_ok]';

% se calcula salto para coordenada articular
J=ik_full(rotpoint(1,:) ',rotpoint(2,:) ',z,[90,180], 'AB');
A_z=15;
JJ=0;
for i=2:n_points
    if dis_ok(i)==0
        E1=ik_full(rotpoint(1,i-1),rotpoint(2,i-1),z(1)+A_z,[90,180], 'AB');
        E2=ik_full(rotpoint(1,i),rotpoint(2,i),z(1)+A_z,[90,180], 'AB');
        J_a=linspace(1,i-1+JJ,i-1+JJ);
        J_b=linspace(i+JJ,n_points+JJ,n_points-(i-1));
        J=[J(J_a,:);E1;E2;J(J_b,:)];
        JJ=JJ+2;
    end
end
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% con coordenadas articulares y J6 variable con A=90
if all(J(:,8))
    toW_J=[J(:,1)';J(:,2)';J(:,3)';J(:,4)';J(:,5)']';
else
    disp('Algunos valores estan fuera del workspace o existe una singularidad')
end
save(append(name, '.mat'), "toW_P", "toW_PA", "toW_J")
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
end

```

E.2. Funcion Workspace circular

```
function [r,R] = W_circular_fnc(z0)

    % se toman valores de X, para ver los radios de los circulos en los que se
    % puede mover dado un phi y z0 definido
    phi=180;

    % valores extremos de xyz en workspace para generar puntos
    xml=0;
    xMl=589;
    % yml=0;
    % yMl=589;
    % zml=-164;
    % zMl=889;

    x=(xml:0.002:xMl)';
    y=zeros(numel(x),1);    %siempre sera 0, se sabe que puede girar con theta1
    % para plano inclindo phi=[90,180] (pendiente XZ) z(x)=mx+z0
    % se necesita z0 y m=tan(alpha), donde alpha=180-phi
    % z=z0+ones(numel(x),1);
    % se esta viendo cuales de todos los puntos a los que el robot puede
    % llegar con phi son
    z=z0+tand(180-phi)*x;

    t=ik(x,y,z,[0,phi],'AB');

    % verifica si la posicion es alcanzable
    % al recorrer creciente ,guarda el valor en donde deja de
    % estar inalcanzable, ese es el minimo
    % el segundo for guarda cuando deja de estar alcanzable
    ok=t(:,8);
    j=0;
    for i=1:numel(x)
        j=j+1;
        if ok(i)==1
            break
        end
    end
    min_index=j;
    for i=j:numel(x)
        j=j+1;
        if ok(i)==0
            break
        end
    end
    max_index=j;
    r=x(min_index);
    R=x(max_index-1);
end
```

E.3. Codigo Python

```
from __future__ import division
from xml.dom import minidom
from collections import MutableSequence
from math import sqrt, cos, sin, acos, degrees, radians, atan2

import sys
import os
```

```

import re

#parser
COMMANDS = set('MmZzLlHhVvCcSsQqTtAa')
UPPERCASE = set('MZLHVCSQTA')

COMMAND_RE = re.compile("([MmZzLlHhVvCcSsQqTtAa])")
FLOAT_RE = re.compile("[-+]?[0-9]*\.\?[0-9]+(?:[eE][-+]?[0-9]+)?")

#path
length_div = 100

#robotdraw
PIXELS_AS_OBJECTS = False    # Set to True to generate PDF or HTML simulations ...
                              # that include the drawn path
TCP_KEEP_TANGENCY = False    # Set to True to keep the tangency along the path

#desde svg
class Point(object):
    '''Creates a 2D point or vector with values x and y. Example: Point(10,-20)'''
    def __init__(self, x, y=0):
        '''Defines x and y variables'''
        if isinstance(x, complex):
            self.x = x.real
            self.y = x.imag
        else:
            self.x = x
            self.y = y

    def move(self, dx, dy):
        '''Translates the point'''
        self.x = self.x + dx
        self.y = self.y + dy

    def __str__(self):
        return "Point(%s, %s)"%(self.x, self.y)

    def __repr__(self):
        return self.__str__()

    def getX(self):
        '''Returns the X coordinate of the point'''
        return self.x

    def getY(self):
        '''Returns the Y coordinate of the point'''
        return self.y

    def distance(self, other):
        '''Calculates the distance between two points'''
        dx = self.x - other.x
        dy = self.y - other.y
        return math.sqrt(dx*dx + dy*dy)

    def switchXY(self):
        '''Switches the X coordinate for the Y coordinate. This is useful if we ...
        are switching from image coordinates (X points to the right, Y points ...
        down) to a different coordinate system (X points down, Y points to ...
        the right)'''
        newx = self.y
        self.y = self.x
        self.x = newx

    def angle(self):
        '''Returns the angle of a vector with respect to the X axis'''
        return atan2(self.y, self.x)

```

```

class Path_feature():
    '''Holds the svg information of a path feature. It also calculates the path ...
    polygons.'''
    def __init__(self, idname, path, color, width, fill):
        self.type = 'path'
        self.idname = idname
        self.path = path
        self.fill_color = fill
        self.line_color = color
        self.line_width = width
        self.poly_vertex = []
        self.poly_vector = []
        self.poly_scale = -1

    def __getitem__(self, index):
        p_i = self.poly_vertex[index]
        v_i = self.poly_vector[index]
        return Point(p_i.real, p_i.imag), [v_i.real, v_i.imag]

    def nPoints(self):
        '''Returns the number of points in the path polygon'''
        return len(self.poly_vertex)

    def getPoint(self, i):
        '''Returns the number of points in the path polygon'''
        #return Point(self.poly_vertex[i].real, self.poly_vertex[i].imag).switchXY()
        return Point(self.poly_vertex[i].imag, self.poly_vertex[i].real)

    def getVector(self, i):
        '''Returns the number of points in the path polygon'''
        return Point(self.poly_vector[i].imag, self.poly_vector[i].real)

    def calc_polygon(self, poly_div=100, scale=1):
        '''Calculates the polygon for the path with poly_div vertex and vector ...
        (path tangent)'''
        if len(self.poly_vertex) != poly_div or self.poly_scale != scale:
            self._calc_polygon(poly_div, scale)

    def _calc_polygon(self, poly_div=100, scale=1):
        '''Performs the polygon calculations (use calc_polygon instead)'''
        self.poly_vertex = []
        self.poly_vector = []
        self.poly_scale = scale
        #print('calculating polygon')
        factor = 1/(poly_div-1)
        for i in range(poly_div):
            p_i = self.path.point(i*factor)*scale
            self.poly_vertex.append(p_i)
        for i in range(poly_div-1):
            vi = self.poly_vertex[i+1] - self.poly_vertex[i]
            normvi = sqrt(vi.real*vi.real + vi.imag*vi.imag)
            if normvi < 1e-6:
                vi = complex(1,0)
            else:
                vi = vi/normvi
            self.poly_vector.append(vi)
        self.poly_vector.append(vi)

    def polygon_move(self, tx, ty):
        '''Translates the polygon by [tx,ty] coordinates'''
        poly_div = len(self.poly_vertex)
        for i in range(poly_div):
            p_i = self.poly_vertex[i]
            self.poly_vertex[i] = complex(p_i.real+tx, p_i.imag+ty)

    def calc_size_poly(self, poly_div=None):
        '''Calculates the size of the scaled polygon (returns minXY, maxXY)'''

```

```

        if poly_div is None: poly_div = len(self.poly_vector)
        if poly_div == 0: poly_div = 100
        if self.poly_scale ≤ 0: self.poly_scale = 1
        self.calc_polygon(poly_div, self.poly_scale)
        #path_size = complex(0,0) #0+0*1j also works
        min_x = 1e6
        min_y = 1e6
        max_x = -1e6
        max_y = -1e6
        for i in range(len(self.poly_vertex)):
            min_x = min(min_x, self.poly_vertex[i].real)
            min_y = min(min_y, self.poly_vertex[i].imag)
            max_x = max(max_x, self.poly_vertex[i].real)
            max_y = max(max_y, self.poly_vertex[i].imag)

        return complex(min_x, min_y), complex(max_x, max_y)

def calc_size_path(self, poly_div=None):
    '''Calculates the size of the path in original coordinates (same ...
        coordinates as if we used Inkscape for example)'''
    pmin, pmax = self.calc_size_poly(poly_div)
    return pmin/self.poly_scale, pmax/self.poly_scale

def __repr__(self):
    string = self.type + ': ' + self.idname + '\n'
    string = string + 'line color: ' + self.line_color.__repr__() + '\n'
    string = string + 'line width: ' + str(self.line_width) + '\n'
    if len(self.poly_vertex) > 0:
        string = string + 'polygon vertex size: ' + ...
            str(len(self.poly_vertex)) + '\n'
        string = string + 'polygon scale: ' + str(self.poly_scale) + '\n'
        poly_sz = self.calc_size_poly()
        string = string + 'polygon scaled size: ' + str(poly_sz)
    else:
        string = string + 'polygons not calculated, run calc_polygon_fit()' + ...
            '\n'
    string = string + '\n'
    return string

class Svg(MutableSequence):
    '''Holds the data of an SVG image'''
    def __init__(self):
        self._features = []

    def __getitem__(self, index):
        return self._features[index]

    def __setitem__(self, index, value):
        self._features[index] = value

    def __delitem__(self, index):
        del self._features[index]

    def insert(self, index, value):
        self._features.insert(index, value)

    def __len__(self):
        return len(self._features)

    def __repr__(self):
        string = ''
        count = 0
        for x in self._features:
            count = count + 1
            string = string + '<feature %i>\n' % count
            string = string + '<width %f ; color %f, %f, %f>\n' % (x.line_width, ...
                x.line_color[0], x.line_color[1], x.line_color[2])
            string = string + repr(x.path)

```

```

        string = string + '\n'
    return string

def calc_size_path(self):
    '''Calculates the size of the path (minimum & maximum corner)'''
    img_minX = 1e6
    img_minY = 1e6
    img_maxX = -1e6
    img_maxY = -1e6
    for feat in self._features:
        pathmin, pathmax = feat.calc_size_path()
        img_minX = min(img_minX, pathmin.real)
        img_minY = min(img_minY, pathmin.imag)
        img_maxX = max(img_maxX, pathmax.real)
        img_maxY = max(img_maxY, pathmax.imag)
    return complex(img_minX, img_minY), complex(img_maxX, img_maxY)

def calc_size_poly(self):
    '''Calculates the size of the polygon, which can be scaled/replaced with ...
    respect to the path (minimum & maximum corner)'''
    img_minX = 1e6
    img_minY = 1e6
    img_maxX = -1e6
    img_maxY = -1e6
    for feat in self._features:
        pathmin, pathmax = feat.calc_size_poly()
        img_minX = min(img_minX, pathmin.real)
        img_minY = min(img_minY, pathmin.imag)
        img_maxX = max(img_maxX, pathmax.real)
        img_maxY = max(img_maxY, pathmax.imag)
    return complex(img_minX, img_minY), complex(img_maxX, img_maxY)

def size_poly(self):
    '''Calculates the size of the polygon.'''
    [corner_min, corner_max] = self.calc_size_poly()
    return Point(corner_max.imag, corner_max.real)

def calc_polygon_fit(self, fit_size=Point(500,500), arc_size=5):
    '''Calculates the path polygons and fits the svg image in the desired ...
    coordinates size'''
    img_min, img_max = self.calc_size_path()
    img_sz_x = img_max.real - img_min.real
    img_sz_y = img_max.imag - img_min.imag
    scale_x = fit_size.y / img_sz_x
    scale_y = fit_size.x / img_sz_y
    scale = min(scale_x, scale_y)
    #print(scale)

    for feat in self._features:
        poly_len_scaled = feat.path.length()*scale
        poly_div = round(poly_len_scaled / arc_size)
        poly_div = max(poly_div, 2) # make sure we have at least 2 vertex in ...
        the polygon
        MAX_POLY_SIZE = 20000
        if poly_div > MAX_POLY_SIZE:
            print('warning, polygon too large, max points set to = ' + ...
                  str(MAX_POLY_SIZE))
            poly_div = MAX_POLY_SIZE

        feat.calc_polygon(poly_div, scale)
        #feat.polygon_move(-img_min.real*scale, -img_min.imag*scale) #this is ...
        faster but less accurate

    polymin, polymax = self.calc_size_poly()
    for feat in self._features:
        feat.polygon_move(-polymin.real, -polymin.imag)

    return complex(img_sz_x, img_sz_y)*scale

```

```

#desde path
class Line(object):
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def __repr__(self):
        return '<Line start=%s end=%s>' % (self.start, self.end)

    def __eq__(self, other):
        if not isinstance(other, Line):
            return NotImplemented
        return self.start == other.start and self.end == other.end

    def __ne__(self, other):
        if not isinstance(other, Line):
            return NotImplemented
        return not self == other

    def point(self, pos):
        distance = self.end - self.start
        return self.start + distance * pos

    def length(self):
        distance = (self.end - self.start)
        return sqrt(distance.real**2+distance.imag**2)

class CubicBezier(object):
    def __init__(self, start, control1, control2, end):
        self.start = start
        self.control1 = control1
        self.control2 = control2
        self.end = end

    def __repr__(self):
        return '<CubicBezier start=%s control1=%s control2=%s end=%s>' % (
            self.start, self.control1, self.control2, self.end)

    def __eq__(self, other):
        if not isinstance(other, CubicBezier):
            return NotImplemented
        return self.start == other.start and self.end == other.end and \
            self.control1 == other.control1 and self.control2 == other.control2

    def __ne__(self, other):
        if not isinstance(other, CubicBezier):
            return NotImplemented
        return not self == other

    def point(self, pos):
        """Calculate the x,y position at a certain position of the path"""
        return ((1-pos) ** 3 * self.start) + \
            (3 * (1-pos) ** 2 * pos * self.control1) + \
            (3 * (1-pos) * pos ** 2 * self.control2) + \
            (pos ** 3 * self.end)

    def length(self):
        """Calculate the length of the path up to a certain position"""
        # Apparently it's impossible to integrate a Cubic Bezier, so
        # this is a geometric approximation instead.

        current_point = self.point(0)
        # I needed 100,000 subdivisions to satisfy assertAlmostEqual on the
        # Arc segment, so I go for the same here. Over 1,000,000 subdivisions
        # makes no difference in accuracy at all.
        subdivisions = length_div

```

```

    lenght = 0
    Δ = 1/subdivisions

    for x in range(1, subdivisions+1):
        next_point = self.point(Δ*x)
        distance = sqrt((next_point.real - current_point.real)**2 + ...
                        (next_point.imag - current_point.imag)**2)
        lenght += distance
        current_point = next_point

    return lenght

class QuadraticBezier(CubicBezier):
    # For Quadratic Bezier we simply subclass the Cubic. This is less efficient
    # and gives more complex calculations, but reuse means less bugs.
    # It is possible to calculate the length of a quadratic bezier so a TODO is to
    # replace the geometric approximation here.

    def __init__(self, start, control, end):
        self.start = start
        self.control1 = self.control2 = control
        self.end = end

    def __repr__(self):
        return '<QuadraticBezier start=%s control=%s end=%s>' % (
            self.start, self.control1, self.end)

class Arc(object):

    def __init__(self, start, radius, rotation, arc, sweep, end):
        """radius is complex, rotation is in degrees,
        large and sweep are 1 or 0 (True/False also work)"""

        self.start = start
        self.radius = radius
        self.rotation = rotation
        self.arc = bool(arc)
        self.sweep = bool(sweep)
        self.end = end

        self._parameterize()

    def __repr__(self):
        return '<Arc start=%s radius=%s rotation=%s arc=%s sweep=%s end=%s>' % (
            self.start, self.radius, self.rotation, self.arc, self.sweep, ...
            self.end)

    def __eq__(self, other):
        if not isinstance(other, Arc):
            return NotImplemented
        return self.start == other.start and self.end == other.end and \
            self.radius == other.radius and self.rotation == other.rotation and \
            self.arc == other.arc and self.sweep == other.sweep

    def __ne__(self, other):
        if not isinstance(other, Arc):
            return NotImplemented
        return not self == other

    def _parameterize(self):
        # Conversion from endpoint to center parameterization
        # http://www.w3.org/TR/SVG/implnote.html#ArcImplementationNotes

        cosr = cos(radians(self.rotation))
        sinr = sin(radians(self.rotation))
        dx = (self.start.real - self.end.real) / 2
        dy = (self.start.imag - self.end.imag) / 2

```

```

xlprim = cosr * dx + sinr * dy
xlprim_sq = xlprim * xlprim
ylprim = -sinr * dx + cosr * dy
ylprim_sq = ylprim * ylprim

rx = self.radius.real
rx_sq = rx * rx
ry = self.radius.imag
ry_sq = ry * ry

# Correct out of range radii
radius_check = (xlprim_sq / rx_sq) + (ylprim_sq / ry_sq)
if radius_check > 1:
    rx *= sqrt(radius_check)
    ry *= sqrt(radius_check)
    rx_sq = rx * rx
    ry_sq = ry * ry

t1 = rx_sq * ylprim_sq
t2 = ry_sq * xlprim_sq
c = sqrt(abs((rx_sq * ry_sq - t1 - t2) / (t1 + t2)))

if self.arc == self.sweep:
    c = -c
cxprim = c * rx * ylprim / ry
cyprim = -c * ry * xlprim / rx

self.center = complex((cosr * cxprim - sinr * cyprim) +
                      ((self.start.real + self.end.real) / 2),
                      (sinr * cxprim + cosr * cyprim) +
                      ((self.start.imag + self.end.imag) / 2))

ux = (xlprim - cxprim) / rx
uy = (ylprim - cyprim) / ry
vx = (-xlprim - cxprim) / rx
vy = (-ylprim - cyprim) / ry
n = sqrt(ux * ux + uy * uy)
p = ux
theta = degrees(acos(p / n))
if uy < 0:
    theta = -theta
self.theta = theta % 360

n = sqrt((ux * ux + uy * uy) * (vx * vx + vy * vy))
p = ux * vx + uy * vy
if p == 0:
    Δ = degrees(acos(0))
else:
    Δ = degrees(acos(p / n))
if (ux * vy - uy * vx) < 0:
    Δ = -Δ
self.Δ = Δ % 360
if not self.sweep:
    self.Δ -= 360

def point(self, pos):
    angle = radians(self.theta + (self.Δ * pos))
    cosr = cos(radians(self.rotation))
    sinr = sin(radians(self.rotation))

    x = cosr * cos(angle) * self.radius.real - sinr * sin(angle) * ...
        self.radius.imag + self.center.real
    y = sinr * cos(angle) * self.radius.real + cosr * sin(angle) * ...
        self.radius.imag + self.center.imag
    return complex(x, y)

def length(self):
    """The length of an elliptical arc segment requires numerical

```

```

integration, and in that case it's simpler to just do a geometric
approximation, as for cubic bezier curves.
"""

current_point = self.point(0)
# Here I need 100,000 subdivisions to satisfy assertAlmostEqual. It's
# a bit slow, but I'm not in a hurry. Over 1,000,000 subdivisions
# makes no difference in accuracy at all.
subdivisions = length_div
length = 0
Δ = 1/subdivisions

for x in range(1, subdivisions+1):
    next_point = self.point(Δ*x)
    distance = sqrt((next_point.real - current_point.real)**2 + ...
                    (next_point.imag - current_point.imag)**2)
    length += distance
    current_point = next_point

return length

class Path(MutableSequence):
    """A Path is a sequence of path segments"""

    def __init__(self, *segments):
        self._segments = list(segments)
        self._length = None
        self._lengths = None

    def __getitem__(self, index):
        return self._segments[index]

    def __setitem__(self, index, value):
        self._segments[index] = value

    def __delitem__(self, index):
        del self._segments[index]

    def insert(self, index, value):
        self._segments.insert(index, value)

    def __len__(self):
        return len(self._segments)

    def __repr__(self):
        return '<Path %s>' % ', '.join(repr(x) for x in self._segments)

    def __eq__(self, other):
        if not isinstance(other, Path):
            return NotImplemented
        if len(self) != len(other):
            return False
        for s, o in zip(self._segments, other._segments):
            if not s == o:
                return False
        return True

    def __ne__(self, other):
        if not isinstance(other, Path):
            return NotImplemented
        return not self == other

    def _calc_lengths(self):
        if self._length is not None:
            return

        lengths = [each.length() for each in self._segments]
        self._length = sum(lengths)

```

```

        self._lengths = [each/self._length for each in lengths]

    def point(self, pos):
        self._calc_lengths()
        # Find which segment the point we search for is located on:
        segment_start = 0
        for index, segment in enumerate(self._segments):
            segment_end = segment_start + self._lengths[index]
            if segment_end >= pos:
                # This is the segment! How far in on the segment is the point?
                segment_pos = (pos - segment_start) / (segment_end - segment_start)
                break
            segment_start = segment_end
        else:
            # This happens when pos is 1.0, and accumulated errors
            # mean that segment_end of the last segment is not quite 1.0.
            segment_pos = 1.0

        return segment.point(segment_pos)

    def length(self):
        self._calc_lengths()
        return self._length

#desde svg
def hex_2_rgb(colorstring):
    """ convert #RRGGBB to an (R, G, B) tuple """
    r, g, b = colorstring[:2], colorstring[2:4], colorstring[4:]
    r, g, b = [int(n, 16)/255 for n in (r, g, b)]
    return [r, g, b]

def svg_load(svgfile):
    '''Loads an SVG file into a Svg() class'''
    svg = Svg()
    xmldoc = minidom.parse(svgfile)
    itemlist = xmldoc.getElementsByTagName('path')
    #print(len(itemlist))
    #print(itemlist[0].attributes['d'].value)
    i=0
    path_data = []
    for s in itemlist:
        if s.parentNode.tagName != 'marker':
            i = i + 1
            data = s.attributes['d'].value
            path = parse_path(data)
            #print('Path: ',i)
            try:
                idname = s.attributes['id'].value
            except:
                idname = 'noname'

            haslinecolor = False
            try:
                style = s.attributes['style'].value + ';'
                lcolor = hex_2_rgb(re.search('stroke:(.*?);', style).group(1))
                haslinecolor = True
                strwidth = re.search('stroke-width:(.*?);', style).group(1)
                if strwidth[-2:] == 'px': strwidth = strwidth[:-2]
                lwidth = float(strwidth)
            except:
                #print('no stroke data for path: '+idname)
                lcolor = [0,0,0] # black as default color
                lwidth = 1 # 1 pixel width as default color

```

```

        try:
            fill = hex_2_rgb(re.search('fill:#[.]*?;', style).group(1))
            if not haslinecolor:
                lcolor = fill
        except:
            fill = lcolor

        svg.append(Path_feature(idname, path, lcolor, lwidth, fill))
    return svg

#desde parser

def _tokenize_path(pathdef):
    for x in COMMAND_RE.split(pathdef):
        if x in COMMANDS:
            yield x
        for token in FLOAT_RE.findall(x):
            yield token

def parse_path(pathdef, current_pos=0j):
    # In the SVG specs, initial movetos are absolute, even if
    # specified as 'm'. This is the default behavior here as well.
    # But if you pass in a current_pos variable, the initial moveto
    # will be relative to that current_pos. This is useful.
    elements = list(_tokenize_path(pathdef))
    # Reverse for easy use of .pop()
    elements.reverse()

    segments = Path()
    start_pos = None
    command = None

    while elements:

        if elements[-1] in COMMANDS:
            # New command.
            last_command = command # Used by S and T
            command = elements.pop()
            absolute = command in UPPERCASE
            command = command.upper()
        else:
            # If this element starts with numbers, it is an implicit command
            # and we don't change the command. Check that it's allowed:
            if command is None:
                raise ValueError("Unallowed implicit command in %s, position %s" ...
                                % (
                                    pathdef, len(pathdef.split()) - len(elements)))

            if command == 'M':
                # Moveto command.
                x = elements.pop()
                y = elements.pop()
                pos = float(x) + float(y) * 1j
                if absolute:
                    current_pos = pos
                else:
                    current_pos += pos

                if start_pos is None:
                    start_pos = current_pos

                # Implicit moveto commands are treated as lineto commands.
                # So we set command to lineto here, in case there are
                # further implicit commands after this moveto.
                command = 'L'

            elif command == 'Z':
                # Close path

```

```

        segments.append(Line(current_pos, start_pos))
        current_pos = start_pos
        start_pos = None

    elif command == 'L':
        x = elements.pop()
        y = elements.pop()
        pos = float(x) + float(y) * 1j
        if not absolute:
            pos += current_pos
        segments.append(Line(current_pos, pos))
        current_pos = pos

    elif command == 'H':
        x = elements.pop()
        pos = float(x) + current_pos.imag * 1j
        if not absolute:
            pos += current_pos.real
        segments.append(Line(current_pos, pos))
        current_pos = pos

    elif command == 'V':
        y = elements.pop()
        pos = current_pos.real + float(y) * 1j
        if not absolute:
            pos += current_pos.imag * 1j
        segments.append(Line(current_pos, pos))
        current_pos = pos

    elif command == 'C':
        control1 = float(elements.pop()) + float(elements.pop()) * 1j
        control2 = float(elements.pop()) + float(elements.pop()) * 1j
        end = float(elements.pop()) + float(elements.pop()) * 1j

        if not absolute:
            control1 += current_pos
            control2 += current_pos
            end += current_pos

        segments.append(CubicBezier(current_pos, control1, control2, end))
        current_pos = end

    elif command == 'S':
        # Smooth curve. First control point is the "reflection" of
        # the second control point in the previous path.

        if last_command not in 'CS':
            # If there is no previous command or if the previous command
            # was not an C, c, S or s, assume the first control point is
            # coincident with the current point.
            control1 = current_pos
        else:
            # The first control point is assumed to be the reflection of
            # the second control point on the previous command relative
            # to the current point.
            control1 = current_pos + current_pos - segments[-1].control2

        control2 = float(elements.pop()) + float(elements.pop()) * 1j
        end = float(elements.pop()) + float(elements.pop()) * 1j

        if not absolute:
            control2 += current_pos
            end += current_pos

        segments.append(CubicBezier(current_pos, control1, control2, end))
        current_pos = end

    elif command == 'Q':

```

```

        control = float(elements.pop()) + float(elements.pop()) * 1j
        end = float(elements.pop()) + float(elements.pop()) * 1j

        if not absolute:
            control += current_pos
            end += current_pos

        segments.append(QuadraticBezier(current_pos, control, end))
        current_pos = end

    elif command == 'T':
        # Smooth curve. Control point is the "reflection" of
        # the second control point in the previous path.

        if last_command not in 'QT':
            # If there is no previous command or if the previous command
            # was not an Q, q, T or t, assume the first control point is
            # coincident with the current point.
            control = current_pos
        else:
            # The control point is assumed to be the reflection of
            # the control point on the previous command relative
            # to the current point.
            control = current_pos + current_pos - segments[-1].control2

        end = float(elements.pop()) + float(elements.pop()) * 1j

        if not absolute:
            control += current_pos
            end += current_pos

        segments.append(QuadraticBezier(current_pos, control, end))
        current_pos = end

    elif command == 'A':
        radius = float(elements.pop()) + float(elements.pop()) * 1j
        rotation = float(elements.pop())
        arc = float(elements.pop())
        sweep = float(elements.pop())
        end = float(elements.pop()) + float(elements.pop()) * 1j

        if not absolute:
            end += current_pos

        segments.append(Arc(current_pos, radius, rotation, arc, sweep, end))
        current_pos = end

    return segments

#def svg_draw(svg_file):
def svg_draw(svg_file, size_b, pix):
    """Quickly shows the image result without checking the robot movements."""
    svgdata = svg_load(svg_file)

    SIZE_BOARD = size_b      # Size of the image. The image will be scaled keeping ...
                             # its aspect ratio
    MM_X_PIXEL = pix         # in mm. The path will be cut depending on the ...
                             # pixel size. If this value is changed it is recommended to scale the pixel ...
                             # object

    print('Board size %ix%i' % (SIZE_BOARD[0], SIZE_BOARD[1]))
    print('MM x Pixel %i' % (MM_X_PIXEL))
    IMAGE_SIZE = Point(SIZE_BOARD[0], SIZE_BOARD[1])    # size of the image in MM
    svgdata.calc_polygon_fit(IMAGE_SIZE, MM_X_PIXEL)
    size_img = svgdata.size_poly()    # returns the size of the current polygon

```

```
count = 0
np_c=0
lst_p=[]
lst_v=[]
for path in svgdata:
    count = count + 1
    np = path.nPoints()
    print('drawing path %i/%i' % (count, len(svgdata)))
    np_c=np+np_c

    for i in range(np):
        p_i = path.getPoint(i)
        v_i = path.getVector(i)
        lst_p.append(p_i)
        lst_v.append(v_i)
print('Points number %i' % (np_c))
return lst_p, lst_v, np_c
```

Bibliografía

- [1] P.I. Corke, “Robotics, Vision & Control”, Springer 2017, ISBN 978-3-319-54413-7. 1.2
- [2] “Robotics toolbox,” Peter Corke, 27-Apr-2020. [Online]. Available: <https://petercorke.com/toolboxes/robotics-toolbox/>. 5.1
- [3] “CIROS studio for 3D factory simulation,” VEROSIM Solutions, <https://www.verosim-solutions.com/en/industry/ciros-studio/>. [Accessed: 23-Mar-2022]. 1.2
- [4] RV-1A/RV-2AJ Series „Standard Specifications Manual (CR1-571 Controller) [http://suport.siriustrading.ro/02.DocArh/07.RI/03.Seria %20RV %20\(Vertical\)/03.RV-A/02.Manuale/RV-1A,2AJ %20- %20Standard %20Specifications %20Manual %20BFP-A8050-K %20\(09.09\).pdf](http://suport.siriustrading.ro/02.DocArh/07.RI/03.Seria%20RV%20(Vertical)/03.RV-A/02.Manuale/RV-1A,2AJ%20-%20Standard%20Specifications%20Manual%20BFP-A8050-K%20(09.09).pdf) 1.2, 2.4
- [5] Lennart Regebro, “Svg.path,” PyPI. [Online]. Available: <https://pypi.org/project/svg.path/>. 6.1
- [6] M. W. Spong, S. Hutchinson, and M. Vidyasagar, *Robot Modeling and Control*. second ed., 2020. 2.4
- [7] J. J. Craig, *Introduction to Robotics: Mechanics and Control*. third ed., 2005 2.2
- [8] B. Siciliano, *Robotics: Modelling, planning and control*. London: Springer, 2010.
- [9] K. Lynch and F. C. Park, *Modern Robotics: Mechanics, planning, and Control*. Cambridge, United Kingdom: Cambridge University Press, 2019. 2.2
- [10] S. B. Niku, *Introduction to robotics: Analysis, control, applications*. Hoboken (N.J.): John Wiley amp; Sons, 2020. 2.2
- [11] CR750/CR751 Series Controller, CR800 Series Controller *Ethernet Function Instruction Manual*,
- [12] M. A. Ayob, W. Wan Zakaria, J. Jalani, and R. Tomari, *Inverse kinematics analysis of a 5-axis rv-2aj robot manipulator*, vol. 10, pp. 8388–8394, 01 2015.
- [13] M. Coman, S. Stan, M. Manic, and R. Balan, *Design, simulation and control in virtual reality of a rv-2aj robot* pp. 2026 – 2031, 12 2009. 1.3
- [14] M. Coman, R. Balan, R. Donca, and D. Verdes, *Optimization of the control for the rv-2aj serial robot*, pp. 149–152, 01 2011. 1.3
- [15] M. Crainic and S. Preitl, *Development of a special setup for path tracking of rv-2aj robot*, in 2016 IEEE 11th International Symposium on Applied Computational Intelligence and Informatics (SACI), pp. 217–220, 2016. 1.3
- [16] M. Haklilar and I. Tasdelen, *Modeling, simulation and fuzzy control of an anthropomorphic robot arm by using dymola*, Journal of Intelligent Manufacturing, vol. 20, pp. 177–186, 04 2009. 1.3
- [17] Thomas Strothotte and Stefan Schlechtweg, *Non-Photorealistic Computer Graphics , PIXEL MANIPULATION OF IMAGES* 2002, Pages 31-81, ISBN 9781558607873 7.1