



Escuela
Politécnica
Superior

Interfaz hombre-máquina mediante estimación de pose



Grado en Ingeniería Robótica

Trabajo Fin de Grado

Autor:

Adrián Sanchis Reig

Tutor/es:

Francisco Antonio Pujol Lopez

Junio 2022



Universitat d'Alacant
Universidad de Alicante

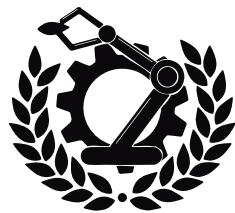
Interfaz hombre-máquina mediante estimación de pose

Autor

Adrián Sanchis Reig

Tutor

Francisco Antonio Pujol Lopez
Tecnología Informática y Computación



Grado en Ingeniería Robótica



Escuela
Politécnica
Superior



Universitat d'Alacant
Universidad de Alicante

ALICANTE, Junio 2022

Resumen

En este trabajo se pretende desarrollar una interfaz hombre-máquina que permita teleoperar en tiempo real robots manipuladores a partir los movimientos que realice el operario con su brazo. La interfaz obtendrá la posición de la mano procesando las imágenes de unas cámaras mediante Machine Learning y la utilizará para calcular la posición deseada a la que mover el robot manipulador. Esto permitirá, por ejemplo, poder controlar robots en entornos peligrosos sin necesidad de estar presente. El trabajo cuenta también con distintas interfaces según si se quiere utilizar Gazebo o se quiere teleoperar un brazo robot real.

Palabras clave: Estimación de Pose, Estéreo, Machine Learning, MediaPipe, Python, Gazebo, ROS Noetic.

Abstract

The aim of this work is to develop a human-machine interface that allows manipulators to teleoperate in real time based on the movements made by the operator with his arm. The interface will obtain the position of the hand by processing the images from cameras using Machine Learning and will use it to calculate the desired position to move the manipulator robot. This will allow, for example, to be able to control robots in dangerous environments without the need to be present. The work also has different interfaces depending on whether you want to use Gazebo or if you want to teleoperate a real robot arm.

Keywords: Pose Estimation, Stereo, Machine Learning, MediaPipe, Python, Gazebo, ROS Noetic.

Agradecimientos

Quisiera empezar agradeciendo a Paco Pujol, mi tutor, por su ayuda y apoyo a la hora de realizar este TFG. A su vez, por mostrar siempre interés en la robótica, promoviendo actividades y eventos que animasen a la gente a interesarse por este mundo. También quiero agradecer al grupo Huro, que me han permitido poder probar el proyecto en un robot real.

En segundo lugar, me gustaría darle las gracias a mi grupo de amigos de toda la vida, así como a aquellos que han ido y venido con el paso de los años. Soy lo que soy gracias a todos los momentos y experiencias que he compartido con vosotros.

También me gustaría agradecer a las locas y maravillosas personas que he conocido durante estos años de carrera, que han estado a mi lado para reír en los momentos felices y para arrimar el hombro en los difíciles. Ojalá que algún día aprendamos a cerrar hilos.

Por último, me gustaría terminar agradeciendo a mi familia. En especial a mi hermano, que siempre me ha querido, aunque me lo muestre como él sabe; a mi madre, por aguantarme y darme todo su apoyo siempre; y a mi padre, por estar siempre a mi lado y animarme a ser un inventor desde que era pequeño.

Es a todos ellos a quienes dedico este trabajo.

*«This is life, and I will not lie
by saying every day will be sunshine.
But there will be sunshine again,
and that is a very different thing to say.
That is truth.»*

Wit, *Rhythm of War*

Índice general

| | |
|---|-----------|
| 1. Introducción | 1 |
| 1.1. Objetivos | 2 |
| 1.2. Estructura | 3 |
| 2. Marco Teórico | 5 |
| 2.1. Estimación de pose humana | 5 |
| 2.1.1. MediaPipe BlazePose | 7 |
| 2.1.1.1. Topología de BlazePose | 7 |
| 2.1.1.2. Pipeline de Blazepose | 8 |
| 2.2. Estéreo | 9 |
| 2.2.1. Calibración de las cámaras | 9 |
| 2.2.2. Calibración del estéreo | 11 |
| 2.2.3. Triangulación de puntos | 12 |
| 3. Metodología | 15 |
| 3.1. Herramientas utilizadas | 18 |
| 3.1.1. Software | 18 |
| 3.1.1.1. Robotics Toolbox for Python (Versión 0.11.0) | 18 |
| 3.1.1.2. PyQt5 (Versión 5.15.4) | 18 |
| 3.1.1.3. MediaPipe (Versión 0.8.9) | 18 |
| 3.1.1.4. OpenCV-Python (Versión 4.5.4.60) | 18 |
| 3.1.1.5. Gazebo - ROS Noetic | 19 |
| 3.1.2. Hardware | 19 |
| 4. Desarrollo | 21 |
| 4.1. Estimación de la pose | 21 |
| 4.2. Simulación brazo robot | 25 |
| 4.3. Desarrollo de la interfaz gráfica | 28 |
| 4.3.1. Modificación librería Swift | 29 |
| 4.4. Comunicación entre el detector y la interfaz | 31 |
| 4.5. Mejoras en el sistema de detección | 32 |
| 4.5.1. Estéreo | 32 |
| 4.5.1.1. Calibración de cámaras | 33 |
| 4.5.1.2. Calibración del estéreo | 34 |
| 4.5.1.3. Modificación código detección manos | 36 |
| 4.5.2. Ajuste mano-hombro | 36 |
| 4.6. Incorporación Gazebo | 37 |
| 4.7. Demo real | 38 |
| 4.7.1. Entorno URsim | 39 |

| | |
|---|-----------|
| 4.7.2. Código URscript | 40 |
| 4.7.3. Robot real | 40 |
| 5. Resultados | 43 |
| 5.1. Interfaces desarrolladas | 43 |
| 5.2. Rendimiento de la interfaz | 43 |
| 5.3. Complejidad de instalación y coste | 44 |
| 5.4. Caso de uso | 44 |
| 6. Conclusiones | 47 |
| Bibliografía | 49 |
| Lista de Acrónimos y Abreviaturas | 51 |
| A. Guía de instalación y uso | 53 |
| A.1. Instalación | 53 |
| A.1.1. Dependencias Python | 53 |
| A.1.2. Simulador Gazebo | 53 |
| A.1.3. Descargar Proyecto | 54 |
| A.2. Cómo usar | 54 |

Índice de figuras

| | | |
|-------|--|----|
| 2.1. | Ambigüedad de pose debido a la falta de profundidad. Fuente: [1] | 6 |
| 2.2. | Modelos para la estimación de pose humana. Fuente: [2] | 7 |
| 2.3. | Topología de BlazePose (La parte verde es de Coco). Fuente: [3] | 8 |
| 2.4. | Pipeline de BlazePose. Detector + Rastreador. Fuente: [4] | 9 |
| 2.5. | Tablero 9x6 aristas / 10x7 cuadrados para calibrar las cámaras. | 11 |
| 3.1. | Planteamiento utilizando cinemática directa. | 16 |
| 3.2. | Planteamiento utilizando cinemática inversa. | 17 |
| 4.1. | Detección de las manos mediante MediaPipe. | 22 |
| 4.2. | Detección del cuerpo mediante MediaPipe. | 23 |
| 4.3. | Landmarks de MediaPipe Pose. Fuente: [3] | 24 |
| 4.4. | Detección de los brazos y coordenadas de las muñecas. | 25 |
| 4.5. | Simulación UR5 con Swift en el navegador. | 26 |
| 4.6. | Interfaz gráfica sin conexión con el simulador. | 28 |
| 4.7. | Interfaz gráfica intentando cargar la simulación. | 29 |
| 4.8. | Cambio en la librería de Swift. | 30 |
| 4.9. | Interfaz gráfica con simulación implementada. | 31 |
| 4.10. | Pipeline con una cámara. | 32 |
| 4.11. | Distintos puntos de vista del patrón. | 33 |
| 4.12. | Pipeline con dos cámaras y estéreo. | 36 |
| 4.13. | Interfaz generalizable a cualquier brazo robot. | 38 |
| 4.14. | Panel de programación de un UR5e en URSim. | 39 |
| 5.1. | Robot UR3 utilizado para la demo. | 45 |
| A.1. | Tablero 9x6 aristas / 10x7 cuadrados para calibrar las cámaras. | 55 |

Índice de Códigos

| | |
|--|----|
| 2.1. Código Python 3: Función OpenCV para calibrar la cámara. | 11 |
| 2.2. Código Python 3: Función OpenCV para calibrar el estéreo. | 12 |
| 2.3. Código Python 3: Función OpenCV para triangular un punto. | 13 |
| 4.1. Código Python 3: Detección de la mano con Mediapipe | 21 |
| 4.2. Código Python 3: Detección del cuerpo con MediaPipe | 22 |
| 4.3. Código Python 3: Obtención de solo los puntos deseados. | 24 |
| 4.4. Código Python 3: Código para simular un UR5 en Swift | 26 |
| 4.5. Código Python 3: Método run() de la clase Simulation. | 27 |
| 4.6. Código Python 3: Detección de esquinas y calibración de la camara. | 33 |
| 4.7. Código Python 3: Detección de esquinas en dos camaras y estéreo. | 34 |
| 4.8. Código Python 3: Obtención de las matrices de proyección. | 35 |
| 4.9. Código Python 3: Obtención de las coordenadas 3D mediante triangulación. . | 35 |
| 4.10. Código Python 3: Ajuste posición 3D con origen en el hombro. | 36 |
| 4.11. Código Python 3: Métodos de la clase gazebo_controller | 37 |
| 4.12. Código Python 3: Implementación de un UR3 a partir de su tabla DH. | 39 |

1. Introducción

En pleno auge de la industria 4.0, parecen lejanos aquellos días en los que la robótica no era más que un sueño futurista y no el elemento indispensable de la industria y sociedad en el que se ha acabado convirtiendo. Desde que el escritor checo Karel Capek acuñase por primera vez el término “robot” en el año 1921; se desarrollara el primer robot humanoide, Elektro; y se diese inicio a la Era de la Robótica en los años 80; la robótica ha ido ganando un peso cada vez mayor en la industria.

En la actualidad, la robótica sigue alcanzando hitos y mejorando a grandes pasos, como se puede ver con los últimos avances que ha logrado la empresa de Boston Dynamics en sus robots. Donde su robot humanoide Atlas [5], es capaz de caminar, correr, saltar y hasta recorrer circuitos de obstáculos, cosa que hasta hace unos años parecía imposible. También se puede destacar su robot con forma de perro Spot [6], el cual es un cuadrúpedo capaz de seguir a personas, patrullar o cargar con peso. Estos, aunque no hagan uso de Inteligencia Artificial (IA) han conseguido llevar a cabo tareas realmente complejas gracias a complejos algoritmos y a numerosas pruebas.

Entre las distintas disciplinas en las que se ha ido desarrollando la robótica, se puede destacar la telerrobótica o robótica teleoperada. La robótica teleoperada es el área encargada de todo aquello relacionado con el control de robots a distancia, principalmente mediante redes inalámbricas. La principal razón por la que la robótica teleoperada es un campo muy útil e interesante, es porque permite trasladar al operario de la zona de trabajo del robot. Gracias a esto, el operario se encuentra lejos de cualquier tipo de peligro o riesgo que pueda suponer estar en el entorno de trabajo, a la vez que puede supervisar y controlar el robot en caso de que sea necesario. La robótica teleoperada puede aplicarse hacia el entretenimiento, el rescate de personas, el transporte de materiales pesados o peligrosos, la manipulación de elementos a temperaturas peligrosas para el ser humano, la manipulación de materiales radiactivos, la manipulación de materiales explosivos...

Por otra parte, el campo del Machine Learning (ML) está cobrando cada vez más importancia y está consiguiendo superar en muchos ámbitos a los propios seres humanos. En los últimos años, se ha visto como este campo ha dado pasos agigantados gracias a las redes convolucionales, la aparición de los transformers, los encoders y las redes Generative adversarial networks (GANs). Es un campo que, a la vez que la robótica, es relativamente joven y va poco a poco innovando. Se puede ver casos en los que se ha juntado con éxito la robótica y el ML para dar lugar a sistemas capaces de llevar a cabo comportamientos complejos, donde es la IA la que se encarga de controlar al robot para que cumpla con su objetivo. Un caso muy destacado de esto serían los coches autónomos, los cuales gracias a la mejora que han tenido en los últimos años las redes neuronales en el campo de la visión, han podido desarrollarse

hasta el punto en el que a día de hoy son prácticamente funcionales. El ML puede, por tanto, ayudar a hacer avanzar aún más a la robótica al permitir aplicar un nuevo enfoque para solucionar problemas hasta ahora complejos o diseñar alternativas que simplifiquen otros.

De entre las distintas aplicaciones que se le han dado al ML, una de las más populares consiste en la obtención de la pose que realiza una persona en una imagen o vídeo mediante la detección de diversos puntos clave. Este tipo de redes abre la puerta a que se puedan desarrollar aplicaciones en las que sea posible medir y detectar los gestos, poses y movimientos que realice una persona para utilizarlos después como input en una aplicación que realice acción u otra en función del movimiento detectado.

Por todo lo mencionado anteriormente, surge la **motivación** para la realización del siguiente trabajo. La idea de juntar IA y robótica permite mejorar muchos apartados de la industria y la sociedad. Por lo que se pretende aprovechar ambos campos para crear una alternativa a los controladores estándar que permiten teleoperar brazos robots a distancia.

Utilizando elementos de visión, es posible desarrollar una alternativa a los controladores que necesitan de mandos, joysticks o la necesidad de instalar elementos como sensores sobre el cuerpo del operador. De esta forma, se trata de un controlador de bajo coste que no requiere de una instalación complicada y es sencillo de utilizar.

1.1. Objetivos

El objetivo principal de este proyecto consiste en el desarrollo de una Human-Machine Interface (HMI) o interfaz hombre-máquina, mediante técnicas de visión por computador e IA, la cual sea de bajo coste y requiera de una instalación mínima. Mediante esta, debe poder ser posible teleoperar cualquier o casi cualquier tipo de brazo robot haciendo uso de gestos o poses y, sin necesidad de utilizar equipo complejo o costoso.

Para llevar a cabo el proyecto cabe cumplir con una serie de objetivos específicos:

- **Detectar pose de una persona:** Se trabajará para conseguir extraer, a partir de la imagen de una persona y mediante visión por computador e IA, diferentes características que puedan ser útiles para realizar la teleoperación de un brazo robot.
- **Simular brazo robot:** Se buscará llevar a cabo una simulación de un brazo robot.
- **Controlar brazo robot mediante visión:** Se desarrollará un método con el que poder controlar el brazo robot simulado a partir de las características obtenidas de una persona.
- **Desarrollar interfaz gráfica:** Se desarrollará una interfaz gráfica desde la cual se pueda ver la simulación del brazo robot y proporcione la información necesaria para facilitar el control por parte del operario.

- **Mejorar software de detección:** Se buscarán formas para conseguir, corregir y mejorar los posibles problemas o dificultades que puedan surgir a la hora de detectar de forma precisa las características de una persona, como puede ser la detección de la profundidad.
- **Incorporar proyecto a otras plataformas:** Una vez se cuente con un modelo funcional, se buscará incorporar el proyecto a otras plataformas relacionadas con la robótica como ROS.
- **Realizar demo con robot real:** Finalmente, se llevará a cabo una demo del proyecto con un brazo robot real para probar su funcionamiento.

1.2. Estructura

Una vez mencionado cuáles son los objetivos del proyecto, se pasa a comentar cómo se encuentra estructurado el documento.

- **Marco Teórico:** Se dan a conocer las nociones básicas necesarias para entender el desarrollo del proyecto.
- **Metodología:** Se detallan los pasos seguidos durante el desarrollo del proyecto, así como el hardware y software que se ha utilizado.
- **Desarrollo:** Se expone en detalle cómo se ha llevado a cabo este trabajo.
- **Resultados:** Se valora el funcionamiento del proyecto y su rendimiento.
- **Conclusión:** Se visualizan los conocimientos finales alcanzados al finalizar este proyecto.

2. Marco Teórico

Este apartado desarrolla distintos conceptos necesarios para poder entrar en detalle más adelante sobre el trabajo realizado. Se introduce al problema de la estimación de pose humana mediante métodos de ML, se desarrolla el funcionamiento que hay detrás de la red BlazePose y se introduce a los sistemas de visión estéreo mediante visión por computador.

2.1. Estimación de pose humana

La estimación de pose humana consiste en la tarea de predecir la localización de las articulaciones del cuerpo humano a partir de una imagen o una secuencia de imágenes de una persona [1]. Esencialmente, es una forma de capturar las coordenadas de las articulaciones mediante la obtención de puntos claves como: muñecas, hombros, rodillas, ojos, orejas, tobillos y brazos. Ya que estos permiten describir la pose de una persona.

Se pueden destacar dos tipos de modelos de estimación de poses humanas:

- **Estimación de pose 2D:** En este tipo de estimación de pose, solo se busca conseguir la localización de las articulaciones del cuerpo en el espacio 2D a partir de una imagen o vídeo. La posición de cada una de las articulaciones se representa como coordenadas en el plano XY de la imagen.
- **Estimación de pose 3D:** En este tipo de estimación de pose, se pretende transformar una imagen 2D a una 3D a través de estimar el valor de la dimensión Z. La estimación 3D permite predecir con mayor precisión la posición espacial de la persona u objeto representado.

En la actualidad, el problema de la estimación de pose 2D cuenta con variedad de soluciones que cuentan con un buen rendimiento. Sin embargo, la complejidad del problema de estimar una pose 3D es mucho mayor. Para solucionarlo, se puede hacer uso, por ejemplo, de cámaras de nubes de puntos, mediante las cuales es posible obtener los valores reales de profundidad del punto clave detectado.

Si se intenta resolver la estimación de pose 3D a partir de la detección de pose 2D, nos encontramos un problema mucho más complejo, puesto que la proyección 2D puede corresponderse con multitud de poses 3D 2.1 al haber una ambigüedad en el valor de la profundidad.

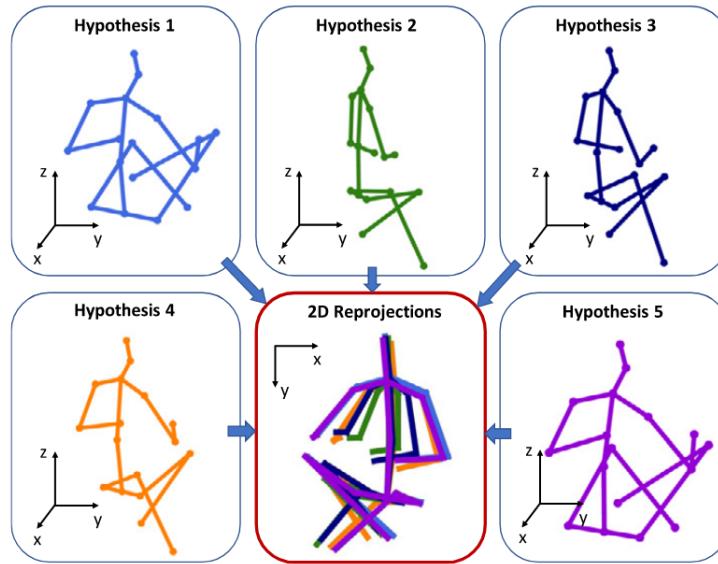


Figura 2.1: Ambigüedad de pose debido a la falta de profundidad.
Fuente: [1]

El modelo que se quiera utilizar para representar el cuerpo humano es muy importante para la estimación de pose humana. Este se utiliza para representar características y puntos clave extraídos de datos de entrada como imágenes. Por lo general, se utiliza un enfoque basado en modelos para describir e inferir las poses del cuerpo humano y representarlas en 2D o 3D.

La mayoría de los métodos utilizan un modelo cinemático rígido de N-articulaciones en el que el cuerpo humano se representa como un esqueleto con articulaciones y extremidades, que contiene información sobre la estructura cinemática y la forma del cuerpo.

Hay, normalmente, tres tipos de métodos para modelar el cuerpo humano [1] [7]:

- **Modelo cinemático:** También llamado modelo basado en esqueleto, es el más usado para la estimación de pose 2D. El modelo incluye un conjunto de posiciones de las articulaciones, y orientaciones de las extremidades, para representar la estructura del cuerpo humano. Por lo tanto, los modelos esqueléticos se utilizan para capturar las relaciones entre las diferentes partes del cuerpo y tiene la ventaja de ser una representación gráfica flexible. Sin embargo, este modelo está limitado a la hora de representar información de textura o forma.
- **Modelo planar:** También llamado modelo basado en contornos. Se usa para la estimación de pose 2D. Consiste en representar la forma y apariencia de un cuerpo humano haciendo uso de rectángulos, los cuales se usan para representar las partes del cuerpo aproximando su contorno. Un ejemplo de esto sería el Active Shape Model (ASM) [8], el cual se utiliza para capturar el grafo del cuerpo humano completo y las deformaciones de la silueta utilizando la técnica Principal Component Analysis (PCA) o análisis de componentes principales.

- **Modelo volumétrico:** Usados para estimar la pose en 3D. Destaca el modelo Skinned Multi-Person Linear (SMPL). En el cual el cuerpo humano se representa como una malla triangulada de 6890 vértices. Esta malla está parametrizada con parámetros de forma y pose.

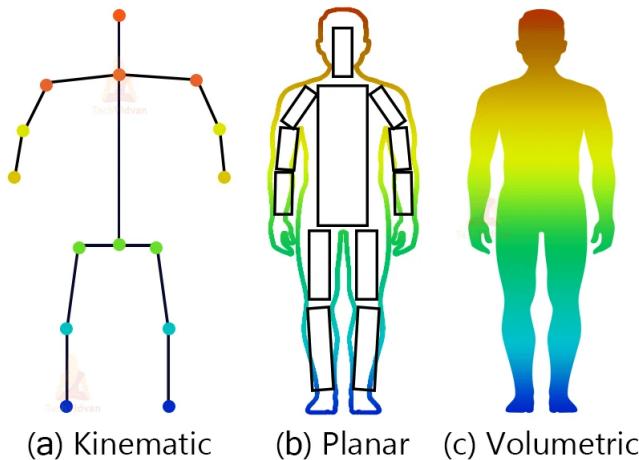


Figura 2.2: Modelos para la estimación de pose humana.

Fuente: [2]

2.1.1. MediaPipe BlazePose

MediaPipe [9] es un framework para desarrollar pipelines de ML que permitan procesar series temporales de datos como vídeos o audios. Mediante este framework, se han desarrollado distintas soluciones como detectores de pose, manos, o caras; detección de objetos o segmentación de personas, entre otros. De entre las distintas soluciones, para el proyecto se ha hecho uso de MediaPipe BlazePose, la cual es una red desarrollada utilizando este framework.

BlazePose [4], es una arquitectura de red neuronal convolucional ligera para la estimación de la pose humana que está diseñada para la inferencia en tiempo real en dispositivos móviles. Durante la inferencia, la red produce 33 puntos clave del cuerpo para una sola persona. El hecho de que sea una red ligera capaz de funcionar en dispositivos móviles la hace particularmente adecuada para casos de uso en tiempo real, como el seguimiento del estado físico y el reconocimiento del lenguaje de señas.

2.1.1.1. Topología de BlazePose

En el problema de la estimación de pose, la topología se refiere a la distribución de puntos clave utilizada para estimar la pose del sujeto. El estándar actual para la topología es la utilizada por el dataset Coco [10], la cual detecta 17 puntos clave del torso, brazos, piernas y cara. Esta topología tiene el problema de que no permite saber la orientación de pies y manos, ya que los puntos clave solo llegan hasta las muñecas y los tobillos.

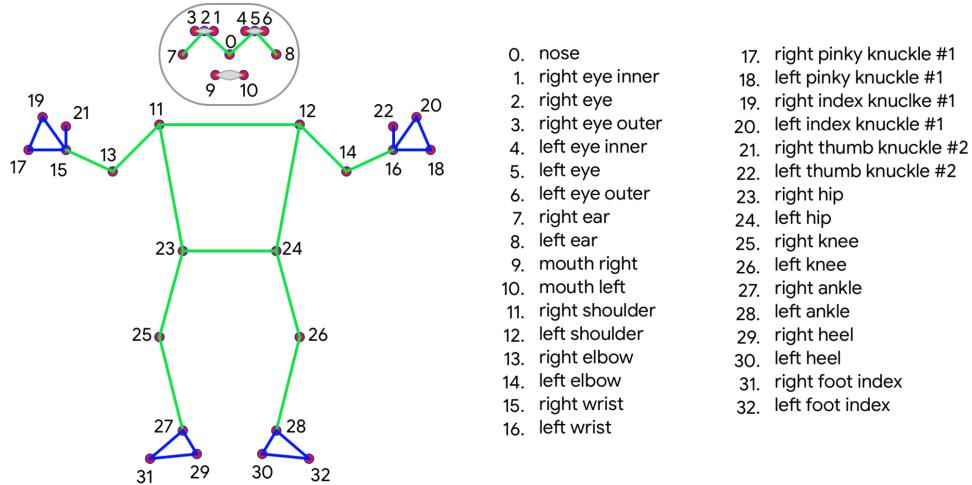


Figura 2.3: Topología de BlazePose (La parte verde es de Coco).

Fuente: [3]

BlazePose utiliza su propia topología compuesta de 33 puntos clave del cuerpo, que se destaca por usar una mínima cantidad de puntos clave sobre las manos, cara y pies para estimar el tamaño, rotación y posición de la zona de interés. Es un conjunto de la topología estándar de Coco junto con la usada en BlazePalm y BlazeFace, otras dos redes que utilizan MediaPipe.

2.1.1.2. Pipeline de Blazepose

El pipeline que usa la red se trata de un detector de personas ligero, seguido de una red de rastreamiento de la pose corporal. El rastreador predice las coordenadas de los puntos clave, la presencia de la persona en la imagen y la región de interés de la imagen actual. Puesto que el uso del detector para obtener la región de interés es costoso, en vídeos se utiliza solo para el primer frame y, una vez obtenida la primera región de interés, esta se va derivando en función de los anteriores puntos clave detectados. En caso de que el rastreador no sea capaz de obtener la posición de la persona, se vuelve a ejecutar el detector.

La arquitectura de la red consiste en un enfoque combinado de mapa de calor, compensación y regresión. Se usa el mapa de calor y la pérdida de compensación solo en la etapa de entrenamiento. Tras este, se eliminan las capas de salida correspondientes del modelo antes de ejecutar la inferencia, quedando solo la de regresión. El resultado es una red ligera que supervisa mediante mapas de calor el resultado de la regresión.

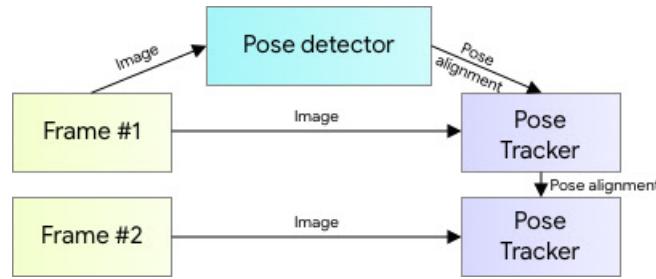


Figura 2.4: Pipeline de BlazePose. Detector + Rastreador.

Fuente: [4]

La primera parte del pipeline utiliza un detector de pose corporal ligero para obtener la región de interés de la imagen. Con el objetivo de obtener la posición del torso, se hace uso de un detector de caras, debido al alto contraste que ofrece una cara humana con respecto al resto del cuerpo, inspirado en el modelo BlazeFace [11]. A partir de este, se predice la posición del centro de las caderas, se obtiene el radio de la circunferencia que circunscribiría a la persona y el ángulo de inclinación de esta.

Tras esto, la segunda parte del pipeline hace uso de un rastreador que predice la posición de los 33 puntos clave a partir de la región de interés. Puesto que el uso del detector para obtener la región de interés es costoso, en vídeos se utiliza solo para el primer frame y, una vez obtenida la primera región de interés, esta se va derivando en función de los anteriores puntos clave detectados. En caso de que el rastreador no sea capaz de obtener la posición de la persona, se vuelve a ejecutar el detector. El rastreador funciona a partir de un enfoque de regresión, el cual está supervisado por la predicción combinada del mapa de calor de todos los puntos clave.

2.2. Estéreo

La visión estereoscópica consiste en la extracción de información tridimensional a partir de imágenes digitales. Al comparar la información de una escena capturada desde dos puntos de vista distintos, se puede extraer su información tridimensional al examinar las posiciones relativas de los objetos a las dos cámaras.

Para hacer una coincidencia estéreo, es importante que ambas imágenes tengan exactamente las mismas características, de forma que no tengan ninguna distorsión. En caso de que esto no sea posible, es necesario realizar la calibración de las cámaras antes de utilizarlas.

2.2.1. Calibración de las cámaras

Para entender el funcionamiento de una cámara, se hará uso del modelo Pinhole. El modelo Pinhole o estenopeico es la representación más simple de una cámara [12]. En este, la vista de una escena se obtiene proyectando cada punto del mundo tridimensional en el plano de la imagen utilizando una transformación de perspectiva que forma el píxel correspondiente. Debido a la sencillez del modelo Pinhole, algunas cámaras introducen una distorsión significa-

tiva en las imágenes. Dos tipos principales de distorsión son la distorsión radial y la distorsión tangencial. Por lo que para solucionar esto es necesario conocer los parámetros intrínsecos, extrínsecos y de distorsión de la cámara. Estos se pueden obtener realizando la calibración de la cámara.

La distorsión radial hace que las líneas rectas parezcan curvas. Esta se vuelve más grande cuanto más alejados están los puntos del centro de la imagen. Las ecuaciones que representan la distorsión radial de una imagen son las siguientes:

$$x_{distorted} = x(1 + k_1r^2 + k_2r^4 + k_3r^6) \quad (2.1a)$$

$$y_{distorted} = y(1 + k_1r^2 + k_2r^4 + k_3r^6) \quad (2.1b)$$

Por otra parte, la distorsión tangencial ocurre porque la lente que toma la imagen no está alineada perfectamente paralela al plano de la imagen. Por lo tanto, algunas áreas de la imagen pueden parecer más cercanas de lo esperado. Las ecuaciones que representan la distorsión tangencial de una imagen son las siguientes:

$$x_{distorted} = x + [2p_1xy + p_2(r^2 + 2x^2)] \quad (2.2a)$$

$$y_{distorted} = y + [p_1(r^2 + 2y^2) + 2p_2xy] \quad (2.2b)$$

Además de la distorsión de la imagen, también es necesario conocer los parámetros intrínsecos y extrínsecos de estas. Los parámetros intrínsecos de una cámara son la distancia focal y el centro óptico de la cámara. Mediante los parámetros intrínsecos, se puede definir la matriz de parámetros intrínsecos de la cámara. Esta puede usarse para eliminar la distorsión de una imagen.

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 0 \end{bmatrix} \quad (2.3)$$

donde: (f_x, f_y) → Es la distancia focal.

(c_x, c_y) → Es el centro óptico.

K → Es la matriz de parámetros intrínsecos de la cámara.

Por otra parte, los parámetros extrínsecos, son aquellos que definen el vector de translación T y rotación R que traducen la posición de un punto 3D a un sistema de coordenadas. La calibración de la cámara, devuelve todos estos parámetros. Para llevarla a cabo, normalmente se hace uso de un patrón bien definido como un tablero de ajedrez 2.5. A partir de distintos puntos de vista de este, se obtienen las coordenadas 3D de distintos puntos del tablero y sus correspondientes proyecciones 2D en la imagen. De esta forma, se obtienen los coeficientes de distorsión (k_1, k_2, p_1, p_2, k_3), la matriz de la cámara, los vectores de rotación y los vectores de translación.

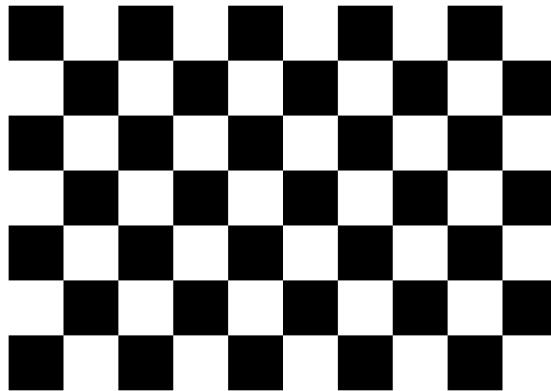


Figura 2.5: Tablero 9x6 aristas / 10x7 cuadrados para calibrar las cámaras.

Si se hace uso de la librería de OpenCV, se puede realizar la calibración con la siguiente función:

Código 2.1: Código Python 3: Función OpenCV para calibrar la cámara.

```
1 # Obtain camera calibration.
2 ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, frame_shape, None, None)
```

2.2.2. Calibración del estéreo

La calibración del estéreo [12], consiste en obtener la matriz de rotación R y el vector de traslación T . Estos, definen la relación entre los centros ópticos de ambas cámaras y su rotación relativa, relacionando el sistema de coordenadas de una cámara con la otra. Es decir, permiten traducir los puntos dados en el sistema de coordenadas de la primera cámara a puntos en el sistema de coordenadas de la segunda cámara. La tupla equivale a la posición de la primera cámara con respecto al sistema de coordenadas de la segunda, por lo que permiten realizar un cambio de base de un sistema a otro.

Si se calculan las poses de un objeto en relación con la primera cámara y la segunda cámara, (R_1, T_1) y (R_2, T_2) , respectivamente, para una cámara estéreo, donde la posición relativa y la orientación entre las dos cámaras son fijas, entonces puede establecerse una relación entre sí. Esto significa que, si se conoce la posición relativa y la orientación (R, T) de las dos cámaras, es posible calcular (R_2, T_2) cuando se proporciona (R_1, T_1) .

$$R_2 = RR_1 \quad (2.4a)$$

$$T_2 = RT_1 + T \quad (2.4b)$$

Por lo tanto, uno puede calcular la representación de las coordenadas de un punto 3D para el sistema de coordenadas de la segunda cámara en el sistema de coordenadas de la primera

cámara de la siguiente forma:

$$\begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \\ 1 \end{bmatrix} = \begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \\ 1 \end{bmatrix} \quad (2.5)$$

Para obtener la matriz de rotación y el vector de translación mediante OpenCV, se puede utilizar la siguiente función:

Código 2.2: Código Python 3: Función OpenCV para calibrar el estéreo.

```
1 ret, __, __, __, __, R, T, __, __ = cv2.stereoCalibrate(objpoints, imgpoints0, imgpoints1, mtx0, dist0,
2 mtx1, dist1, frame_shape, criteria = criteria, flags = cv2.CALIB_FIX_INTRINSIC)
```

2.2.3. Triangulación de puntos

Para obtener las coordenadas 3D de un punto en función de la proyección de este en dos cámaras distintas, es necesario contar con la matriz de rotación y el vector de traslación que relaciona los sistemas de coordenadas de las cámaras. Para hacer la triangulación, se trabaja con coordenadas homogéneas. Esto debido a que de esta forma se puede representar infinitas cantidades utilizando cantidades finitas. Las coordenadas homogéneas tienen la forma:

$$[x \ y \ z \ w] \quad (2.6)$$

Para realizar la triangulación, primero se obtienen las matrices de proyección de cada una de las cámaras. La matriz de proyección se define como $P = K[R|T]$, la matriz de parámetros intrínsecos por la de parámetros extrínsecos.

$$P = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} \quad (2.7)$$

Para hacer que el origen del sistema de coordenadas se encuentre en la primera, se puede definir la matriz de proyección P_1 de forma que la matriz de rotación sea la identidad y el vector de traslación sea nulo. Por otro lado, la matriz de proyección P_2 de la segunda cámara utilizará como matriz de rotación y vector de traslación, los obtenidos en la calibración del estéreo que relacionaban ambos sistemas.

$$P_1 = K_1 \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad (2.8a)$$

$$P_2 = K_2 \begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} \quad (2.8b)$$

donde: $K \rightarrow$ Es la matriz de parámetros intrínsecos de la cámara.

$R \rightarrow$ Es la matriz de rotación obtenida al calibrar el estéreo.

$T \rightarrow$ Es el vector de traslación obtenido al calibrar el estéreo.

Dadas las matrices de proyección P_1 y P_2 y dos pares de coordenadas de un punto dados en el sistema de referencia de cada una de las cámaras, se puede realizar la triangulación obteniendo la posición 3D de este punto. Mediante la librería de OpenCV, se puede hacer utilizando la siguiente función:

Código 2.3: Código Python 3: Función OpenCV para triangular un punto.

```
1 p4d = cv2.triangulatePoints(P1, P2, c1_point, c2_point) # Calculate (4D) homogeneous coordinates.  
2 p3d = (p4d[:3, :] / p4d[3, :]).T # Turn homogeneous coordinates to euclidean (3D) coordinates
```


3. Metodología

En el siguiente apartado, se pasará a hacer un repaso de la metodología y la planificación seguida a lo largo del proyecto, así como de las herramientas utilizadas para la implementación del mismo.

El primer paso fue establecer cuál método de control puede ser más óptimo para teleoperar un robot manipulador. Aunque desde un principio el proyecto estaba planteado teniendo en cuenta el uso de visión por computador para obtener la posición en el espacio de los brazos del operador, se investigaron distintos métodos que podían ser de gran utilidad para desarrollar el proyecto. Algunos de los métodos que se investigaron fueron:

- Control mediante joysticks o mandos.
- Control mediante un traje de captura en tiempo real.
- Control mediante un conjunto de sensores, como una Inertial Measurement Unit (IMU) o unidad de medición inercial, para detectar los movimientos del operador.
- Control mediante una cámara de nubes de puntos y visión por computador.
- Control mediante una cámara y visión por computador.

Puesto que el objetivo del proyecto es el desarrollo de un sistema de control de bajo coste, el cual haga uso de elementos de visión, se optó por hacer uso de una cámara para obtener la posición del operario a partir de las imágenes que esta proporcione haciendo uso de visión por computador e IA. La principal razón de la elección de este método, fue el hecho de que brinda la oportunidad de desarrollar una HMI que no necesite de aparatosos y costosos dispositivos colocados sobre el operario para realizar la teleoperación.

Hecho esto, se realizó una búsqueda sobre posibles métodos o herramientas utilizados en la actualidad para obtener la posición de una persona haciendo uso de una imagen de esta. Dentro del mundo del ML, este problema se conoce como *pose estimation* o estimación de pose. La estimación de pose cuenta con una gran variedad de soluciones que hacen uso tanto de solo visión por computador como de redes neuronales [13]. De entre las opciones, podemos encontrar distintas soluciones hechas en TensorFlow como PoseNet, pero se ha hecho uso de una red de Google llamada MediaPipe BlazePose [14]. Esto debido a que cuenta con una buena documentación y su uso es sencillo, preciso y robusto.

Haciendo uso de los ejemplos que proporciona la documentación de MediaPipe [15], se probaron en el equipo utilizado para el proyecto una serie de redes con el objetivo de comprobar qué tan bien podía funcionar la red ejecutándose en este. Se hicieron pruebas con los modelos

de Hand Estimation, Face estimation, Pose Estimation y Holistics, que es una mezcla de las redes anteriores.

Tras esto, se desarrolló una aplicación en Linux desde la cual se puede visualizar una simulación de un brazo robot y controlarlo. Esta aplicación se ha desarrollado haciendo uso de PyQt5, así como de la librería de Python 3 de Peter Corke [16], ya que se ha trabajado previamente con ella y cuenta con distintas herramientas para controlar y simular brazos robots. Fue necesario modificar el código del simulador Swift utilizado en esta librería para poder integrar la ventana de la simulación del robot dentro de la aplicación de PyQt5. También se le añadieron a la aplicación distintos elementos que muestran la información necesaria para que el operador pueda controlar al robot. Estos son los valores articulares del brazo, la posición cartesiana del extremo o su orientación.

Hecho esto, se realizó la comunicación entre el detector de pose de MediaPipe y la aplicación, la cual se lleva a cabo mediante comunicación TCP/IP. El objetivo inicial era realizar el control a partir de los ángulos de las articulaciones del brazo de la persona, los cuales se enviaban a la simulación para que esta replicase los movimientos mediante el cálculo de la cinemática directa. Tras este planteamiento inicial, se apreció que no era posible controlar todo tipo de brazos robots mediante este planteamiento, ya que en función de la cadena cinemática del robot que se pretenda controlar, harán falta más Grados de Libertad (GDL) o menos. Un ejemplo de este problema sería que el hombro de una persona es una articulación esférica, es decir, tiene 3 GDL mientras que la base de un UR5 cuenta con solo 2 articulaciones rotacionales. Debido a esto, la base del UR5 no llega a los 3 GDL necesarios para replicar la posición del hombro, haciendo inviable el control por cinemática directa.

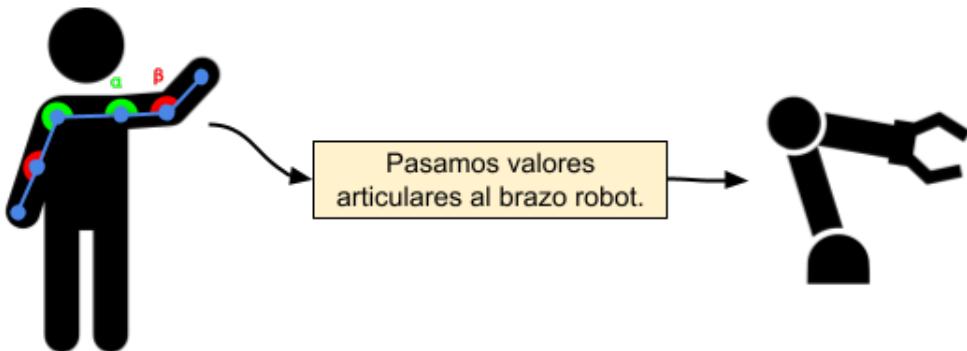


Figura 3.1: Planteamiento utilizando cinemática directa.

Resolver la posición del extremo mediante cinemática directa tiene la desventaja de que no es generalizable a cualquier tipo de brazo robot. Por ello, se cambió el planteamiento. En vez de obtener los ángulos de cada una de las articulaciones del brazo de una persona y mover el robot mediante el cálculo de la cinemática directa, se obtienen solo las coordenadas cartesianas de la muñeca de la persona. A partir de estas, se calcula la cinemática inversa del robot, obteniendo de esta forma los valores articulares necesarios para que el brazo robot alcance la posición deseada. Este nuevo planteamiento presenta una ventaja frente al anterior,

ya que ahora la HMI es generalizable a cualquier tipo de robot que tenga al menos 3 GDL. Por otra parte, es necesario calcular la cinemática inversa para cada uno de los puntos de la mano recibidos desde el detector de pose, lo que resulta a cambio en un aumento de la latencia del sistema.

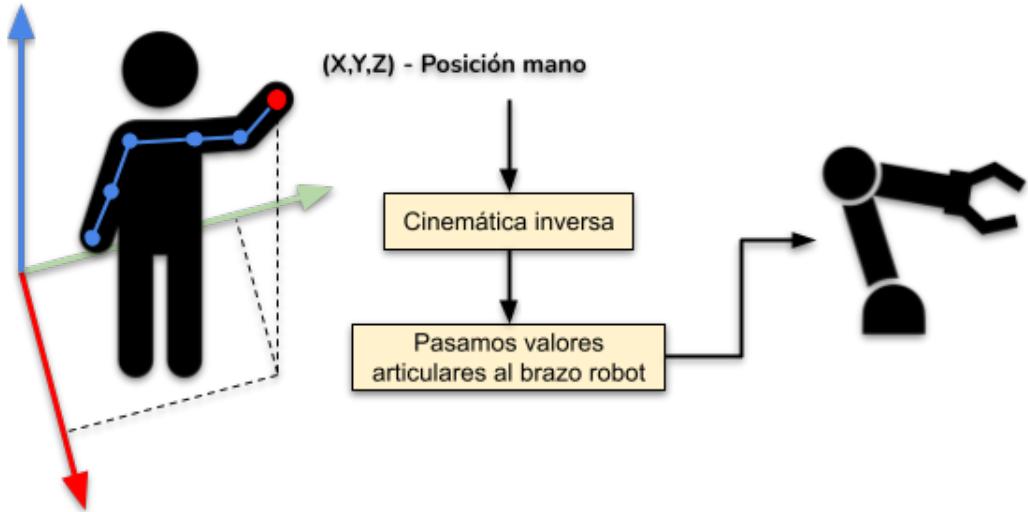


Figura 3.2: Planteamiento utilizando cinemática inversa.

Una vez implementado el nuevo pipeline en la aplicación, se realizaron pruebas para ver si la simulación del brazo robot era capaz de imitar los movimientos de un brazo humano. Aunque esta era capaz de imitar los movimientos en el plano XZ, no era capaz de moverse correctamente sobre el eje Y¹. Esto debido a que los valores de profundidad que devuelve MediaPipe para los puntos detectados son una estimación calculada por la red. Aunque la persona acerque o aleje la mano de la cámara, la simulación no representa fielmente el movimiento.

Para solucionar el problema, ha sido necesario replantear el pipeline con el objetivo de obtener un valor más preciso de la profundidad. Este consiste en hacer uso de una segunda cámara para realizar un estéreo. De esta forma, se pueden obtener de forma más precisa las coordenadas de la muñeca en el espacio 3D. Tras documentarse sobre cómo se puede montar un sistema estéreo con dos cámaras, se implementó al detector de poses para que se utilizase la red en ambas cámaras. Esto permite al detector obtener un valor de la profundidad más preciso, pero a cambio la latencia del detector se ha duplicado, ya que ahora la red ha de analizar dos imágenes cada vez en lugar de solo una.

Tras esto, se han hecho otras dos versiones de la aplicación. Una que funciona utilizando Gazebo, y otra que permite controlar un robot UR3 real. Para esta última, ha sido necesario cambiar la interfaz gráfica para poder mostrar la simulación de cualquier tipo de brazo robot y

¹En la simulación, el eje Z es el perpendicular al suelo y el eje Y el que recibe los valores de la profundidad. Mientras que para la cámara, el eje Z es el de la profundidad.

también se han acotado los posibles movimientos que puede tener el brazo robot por seguridad.

3.1. Herramientas utilizadas

A continuación, se detallan las distintas herramientas utilizadas en el proyecto para llevar cabo su desarrollo.

3.1.1. Software

3.1.1.1. Robotics Toolbox for Python (Versión 0.11.0)

La Robotics Toolbox de Python es una versión en Python de la librería desarrollada por Peter Corke para MATLAB. Esta cuenta con distintas funciones y clases que permiten definir robots en función de sus parámetros Denavit-Hartenberg (DH), calcular su cinemática directa o inversa, hacer cálculos de trayectorias o representar los robots con modelos 3D entre otros [16] [17].

3.1.1.2. PyQt5 (Versión 5.15.4)

Qt es un conjunto de bibliotecas C++ multiplataforma que implementan una API de alto nivel para acceder a muchos aspectos de los sistemas móviles y de escritorio modernos. Estos incluyen servicios de ubicación y posicionamiento, conectividad multimedia, NFC y Bluetooth, un navegador web basado en Chromium, así como el desarrollo tradicional de la interfaz de usuario.

PyQt5 es un conjunto de bindings de Python para Qt v5. Este, permite que Python se use como un lenguaje de desarrollo de aplicaciones alternativo a C++ en todas las plataformas compatibles, incluidos iOS y Android. [18].

3.1.1.3. MediaPipe (Versión 0.8.9)

MediaPipe [15] es un framework que permite desarrollar pipelines de ML. Proporciona múltiples soluciones como MediaPipe Pose [14], una solución de ML para el seguimiento de la postura del cuerpo de alta fidelidad, que infiere 33 puntos de referencia 3D y una máscara de segmentación de fondo en todo el cuerpo a partir de fotogramas de video RGB. Esta, logra un rendimiento en tiempo real en la mayoría de los teléfonos móviles modernos, computadoras de escritorio/portátiles, en Python e incluso en la web.

3.1.1.4. OpenCV-Python (Versión 4.5.4.60)

OpenCV es una biblioteca de software de aprendizaje automático y visión artificial de código abierto escrita en C/C++. Incluye más de 2500 algoritmos optimizados, que van desde la visión artificial clásica hasta los algoritmos de aprendizaje automático más avanzados. Todas estas características lo convierten en una herramienta fundamental para los proyectos, especialmente por los algoritmos de calibración de cámaras y reconstrucción 3D que proporciona.

OpenCV cuenta con una versión para Python, la cual consiste en una librería de bindings que permiten resolver problemas de visión por computador. Esto permite que el código sea tan rápido como el código C/C++ original (ya que es el código C++ real que funciona en segundo plano).

3.1.1.5. Gazebo - ROS Noetic

Gazebo [19] es un simulador 3D de robots formado por una colección de bibliotecas de software de código abierto, diseñadas para simplificar el desarrollo de aplicaciones de alto rendimiento. El simulador está estructurado para adaptarse a muchos casos de uso diferentes. Cada biblioteca dentro de Gazebo tiene dependencias mínimas, lo que les permite usarse en tareas que van desde la resolución de transformaciones matemáticas hasta la codificación de video y hasta la simulación y la gestión de procesos.

ROS cuenta además con un conjunto de paquetes que permiten la integración en su sistema del simulador Gazebo. Lo que permite combinar todas las herramientas que ROS ofrece para el desarrollo de robots junto con el simulador Gazebo.

3.1.2. Hardware

- Portátil con procesador Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz.
- Dos webcams Full HD 1080P.
- Brazo robótico de Universal Robots: UR3.

4. Desarrollo

En el siguiente apartado se hará una explicación en profundidad sobre el proceso seguido durante el desarrollo del proyecto, desde el funcionamiento de las tecnologías seleccionadas hasta las decisiones de diseño.

4.1. Estimación de la pose

Para la resolución de la estimación de la pose, se ha utilizado un modelo de ML para estimar la pose/postura de una persona a partir de una imagen o un video. Esto se lleva a cabo mediante la estimación de las ubicaciones espaciales de las articulaciones clave del cuerpo (puntos clave). Para obtener los puntos claves de las articulaciones del cuerpo, se ha hecho uso de la red de Google, MediaPipe BlazePose. Esta red permite aplicar modelos de ML a imágenes o videos para hacer detecciones de las poses que está haciendo una persona.

Mediante el siguiente código y una cámara, se puede visualizar en tiempo real el funcionamiento de la red para el problema de la detección de manos.

Código 4.1: Código Python 3: Detección de la mano con Mediapipe

```
1 import mediapipe as mp
2 import cv2
3
4 # Loading mediapipe objects for detection and drawing of the hands.
5 mp_drawing = mp.solutions.mediapipe.python.solutions.drawing_utils
6 mp_drawing_styles = mp.solutions.mediapipe.python.solutions.drawing_styles
7 mp_hands = mp.solutions.mediapipe.python.solutions.hands
8
9 cap = cv2.VideoCapture(2)
10
11 with mp_hands.Hands() as hands:
12
13     while cap.isOpened():
14         success, image = cap.read()
15
16         if not success:
17             continue
18
19         # Mark the image as not writeable to pass by reference.
20         image.flags.writeable = False
21         image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
22         results = hands.process(image) # Processing image for hand detection.
23         image.flags.writeable = True
24
25         image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
26
27         if results.multi_hand_landmarks:
28             for hand_landmarks in results.multi_hand_landmarks:
29                 mp_drawing.draw_landmarks(
```

```

30     image,
31     hand_landmarks,
32     mp_hands.HAND_CONNECTIONS,
33     mp_drawing_styles.get_default_hand_landmarks_style(),
34     mp_drawing_styles.get_default_hand_connections_style())
35
36 # Flip the image horizontally for a selfie-view display.
37 cv2.imshow('MediaPipe Hands', cv2.flip(image, 1))
38
39
40 if cv2.waitKey(5) & 0xFF == 27:
41     break
42
43cap.release()

```



Figura 4.1: Detección de las manos mediante MediaPipe.

Mediante la librería de MediaPipe y de OpenCV, es posible capturar imágenes de una cámara o cargar un archivo multimedia y procesarlo mediante la función *process()*. Esta función toma como argumento la imagen a procesar y devuelve como resultado las coordenadas tanto en píxeles como en el espacio 3D de los puntos clave detectados. Tras esto, se utiliza la función *draw_landmarks()* para mostrar sobre la imagen la detección.

MediaPipe tiene distintas redes, aparte de la detección de manos, cuenta con una detección más precisa de la cara y, la que más nos interesa: podemos obtener la detección de la postura del cuerpo completo mediante MediaPipe BlazePose. Un ejemplo de cómo se podría hacer esto sería el siguiente:

Código 4.2: Código Python 3: Detección del cuerpo con MediaPipe

```

1 import mediapipe as mp
2 import cv2
3
4 # Loading mediapipe objects for detection and drawing of the body.
5 mp_drawing = mp.solutions.mediapipe.python.solutions.drawing_utils
6 mp_drawing_styles = mp.solutions.mediapipe.python.solutions.drawing_styles
7 mp_pose = mp.solutions.mediapipe.python.solutions.pose
8

```

```

9
10 cap = cv2.VideoCapture(2)
11
12 with mp_pose.Pose() as pose:
13
14     while cap.isOpened():
15         success, image = cap.read()
16
17         if not success:
18             continue
19
20         # Mark the image as not writeable to pass by reference.
21         image.flags.writeable = False
22         image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
23         results = pose.process(image) # Processing image for hand detection.
24         image.flags.writeable = True
25
26         image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
27         mp_drawing.draw_landmarks(
28             image,
29             results.pose_landmarks,
30             mp_pose.POSE_CONNECTIONS,
31             landmark_drawing_spec=mp_drawing_styles.get_default_pose_landmarks_style())
32
33         # Flip the image horizontally for a selfie-view display.
34         cv2.imshow('MediaPipe Pose', cv2.flip(image, 1))
35
36         if cv2.waitKey(5) & 0xFF == 27:
37             break
38
39 cap.release()

```



Figura 4.2: Detección del cuerpo mediante MediaPipe.

Aunque para el proyecto se va a hacer uso de esta última red que detecta el cuerpo entero, devuelve muchos puntos que para el proyecto no son de interés. Como pueden ser los puntos de la cara, las manos o las piernas. En este caso, solo nos interesan los puntos que forman los hombros, el codo y la muñeca de cada uno de los brazos. Es decir, tan solo 6 de los 32 puntos que detecta la red. Para saber el identificador de los puntos que queremos obtener, se ha hecho uso de la siguiente imagen.

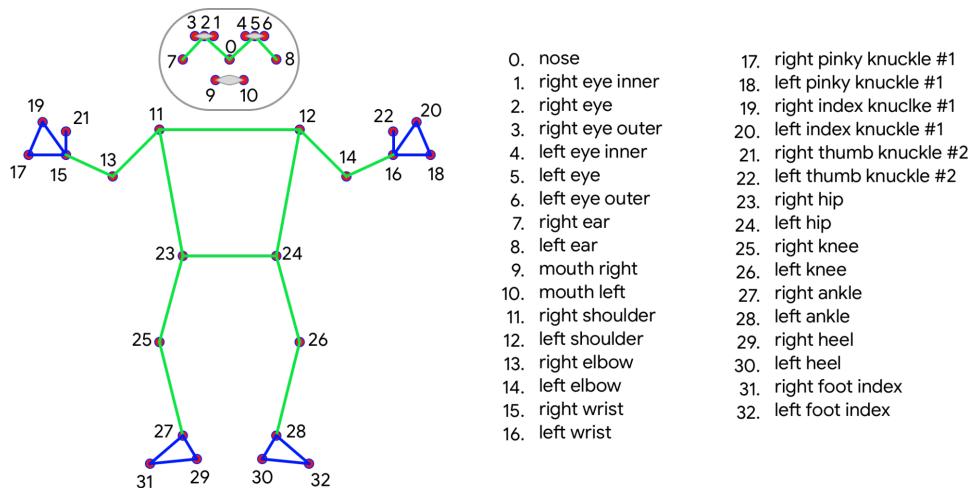


Figura 4.3: Landmarks de MediaPipe Pose.

Fuente: [3]

Una vez obtenidos los identificadores de los landmarks que queremos para la detección de los brazos, se ha modificado el código anterior para poder introducir a mano los puntos que queremos que se represente sobre la imagen. Los landmarks del hombro y el codo solo son necesarios para representar sobre la imagen lo que ha detectado la red y que de esta forma el usuario sepa si sus brazos están siendo bien detectados o no. Se ha hecho la función *DrawLines()* para que pinte solo los landmarks que seleccionemos.

Para la teleoperación de un brazo robot, solo nos interesan los landmarks de las muñecas. En el siguiente código se muestra como obtener sus coordenadas una vez procesada la imagen.

Código 4.3: Código Python 3: Obtención de solo los puntos deseados.

```

1 # Define desired landmarks.
2 pose_keypoints = {
3     "L_wrist": mp_pose.PoseLandmark.LEFT_WRIST,
4     "R_wrist": mp_pose.PoseLandmark.RIGHT_WRIST,
5 }
6
7 # We get keypoints coordinates.
8 frame_keypoints = []
9 if results.pose_landmarks:
10     for kpt in pose_keypoints:
11         x0 = int(results.pose_landmarks.landmark[pose_keypoints[kpt]].x * image.shape[1])
12         y0 = int(results.pose_landmarks.landmark[pose_keypoints[kpt]].y * image.shape[0])
13         frame_keypoints.append([x0,y0]) # save keypoint coordinates.

```

Finalmente, el resultado que obtenemos es el siguiente.

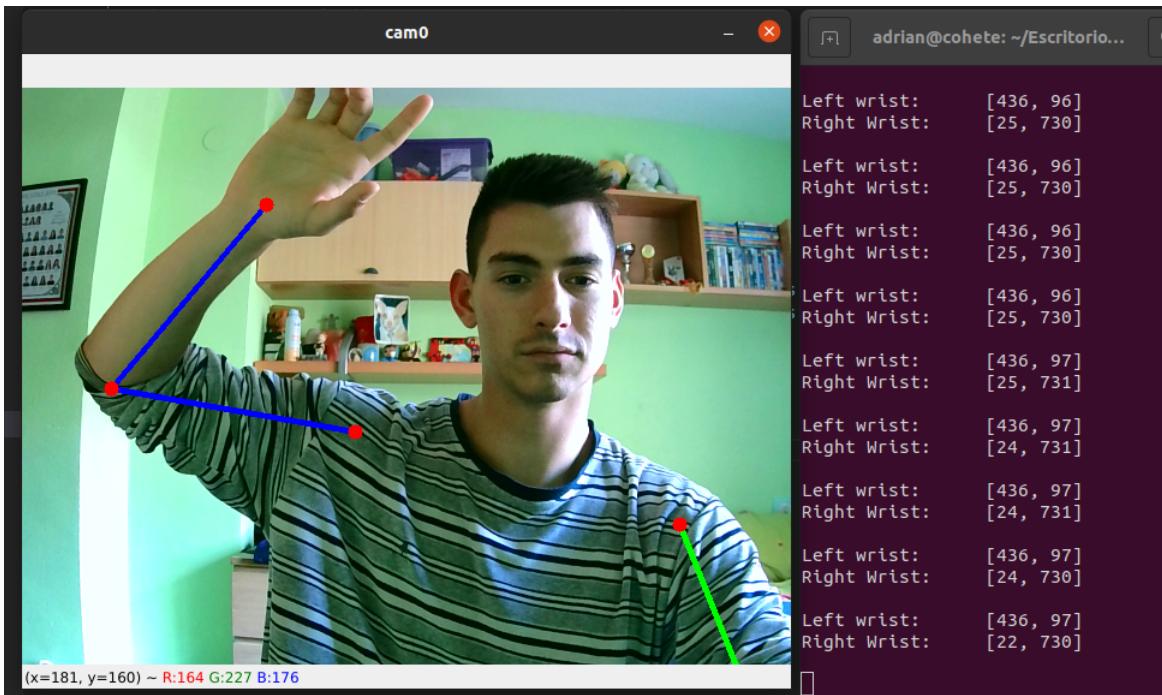


Figura 4.4: Detección de los brazos y coordenadas de las muñecas.

La razón por la que no se usa la coordenada Z de la profundidad para obtener las coordenadas 3D de las muñecas es debido a que la detección de la profundidad de la red no es del todo buena. Por esta razón, solo se obtienen las coordenadas 2D. Una vez se han obtenido las coordenadas de la muñeca, estas se mandan a la interfaz para que la simulación pueda procesarlas. Esto se hace mandando un mensaje al servidor TCP al que está conectado como cliente.

4.2. Simulación brazo robot

Existen muchas plataformas que proporcionan las herramientas necesarias para simular un brazo robot. A lo largo de la carrera, hemos trabajado en más de una ocasión con la Robotics Toolbox de Peter Corke, el cual recientemente creó una versión para Python [17]. Esta librería proporciona una multitud de herramientas muy útiles para modelar y controlar robots, como hacer cálculos de cinemática directa, inversa, dinámica o cálculo de trayectoria, entre otras. Además, permite simular brazos robots en Matplotlib o en Swift. Este último es un entorno de simulación que se ejecuta en el navegador y es la principal razón para crear la simulación con esta librería, ya que es posible crear una interfaz gráfica que contenga una ventana del navegador.

La librería permite simular cualquier configuración de brazo robot definiendo sus parámetros DH. Aunque esto no es necesario para la implementación de la simulación en Swift, ya que la librería tiene cargados por defecto modelos de robots reales. Estos cuentan además con sus propios modelos 3D, por lo que es posible visualizarlos sin necesidad de importar ninguno

modelo.

Código 4.4: Código Python 3: Código para simular un UR5 en Swift

```
1 import roboticstoolbox as rtb
2 from roboticstoolbox.backends.swift import Swift
3 from time import time
4
5
6 robot = rtb.roboticstoolbox.models.UR5()
7
8 backend = Swift()
9 backend.launch()
10 backend.add(robot)
11
12 start = time()
13
14 while time() - start < 5:
15     backend.step()
16
17 backend.close()
```

Mediante el anterior código, se pueden cargar los parámetros del robot UR5e, iniciar la simulación y abrir una ventana del navegador con el robot dentro de esta durante 5 segundos. Es importante utilizar la instrucción *step()*, ya que es la encargada de actualizar la posición del robot en la simulación.

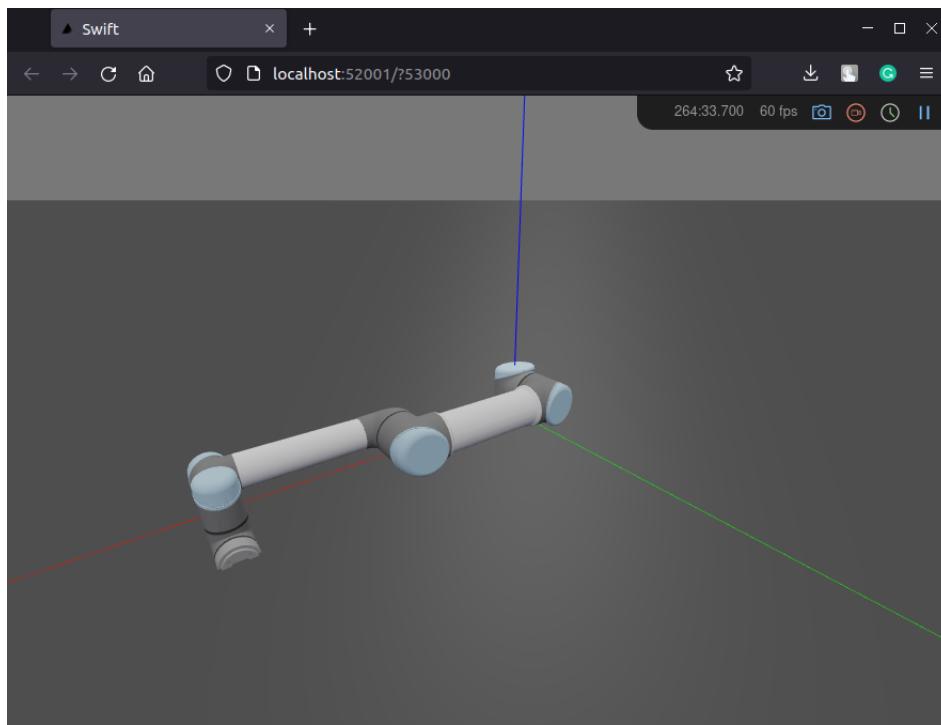


Figura 4.5: Simulación UR5 con Swift en el navegador.

Con el fin de estructurar mejor el código de la simulación para que sea más fácil de usar a

la hora de juntar tanto la interfaz como la simulación, se ha creado la clase *Simulation* la cual se lanza en un hilo aparte para no ralentizar a la interfaz mientras realiza cálculos o espera recibir una nueva posición. El constructor de esta clase carga automáticamente el modelo del robot, lanza la simulación de Swift 4.5 e incluye en esta al robot. Esta clase cuenta con la función *run()* que es la que contiene el bucle con la instrucción *step()* para ir actualizando el estado de la simulación en tiempo real.

Código 4.5: Código Python 3: Método run() de la clase Simulation.

```

1 def run(self):
2     self.com = server_tcp(port=12345, controller_ip="localhost")
3
4     # A la espera de poder conectarse con 2 clientes:
5     while self._RUN_ and self.com.connected < 1:
6         self.com.listen_for_connections(blocking=False)
7
8
9     self.desired_q = self.robot.q # Inicializamos la posicion deseada.
10    self.get_pos() # Obtenemos la posición actual del robot simulado.
11    self.signal_data.emit() # Emitimos señal para actualizar el panel de info.
12
13
14    print("[#]: Running...")
15    while self._RUN_:
16
17        # Recibimos datos de control.
18        data = self.com.recibir(self.com.omnibundle)
19
20        if data.any():
21
22            # Procesamos los datos.
23            data = np.array(self.home_p) + data # Coordenadas muñeca como offset.
24            self.desired_q = self.move_to(data)
25
26            # Actualizamos posición del robot y obtenemos la cinemática.
27            self.robot.q = self.desired_q
28            self.get_pos()
29
30            self.signal_data.emit() # Emitimos señal para actualizar el panel de info.
31
32
33        self.backend.step() # Send new frame.
34
35
36        print("[#]: FINISHED")
37        self._finnish_ = True

```

La función *run()* se encarga de abrir el servidor y esperar hasta que se conecte como cliente el script que usa MediaPipe para obtener las coordenadas de las muñecas. Tras conectarse, se iniciará la simulación, la cual tras recibir las coordenadas de la mano por TCP/IP procede a utilizarlas como un offset de la posición home del robot. Esto se hace para que el control sea más intuitivo y consista en desplazar al robot de la posición home mediante los movimientos de la mano. Tras sumar el offset, se ejecuta la función *move_to()* la cual calcula la cinemática inversa a partir del punto pasado como argumento para obtener los valores articulares del robot. En caso de que el punto al que se quiere mover el robot sea inalcanzable, la función devolverá la posición actual del robot, haciendo que este no se mueva hasta recibir una posición válida.

4.3. Desarrollo de la interfaz gráfica

Para realizar la interfaz gráfica encargada de mostrar la simulación del brazo robot y la información que puede ser necesaria para el operador, como las coordenadas articulares y cartesianas, se ha hecho uso de la librería de Python PyQt 5. Esta permite crear interfaces gráficas con Python de manera rápida y sencilla, esto debido a que la legibilidad del código de Python hace que sea una tarea sencilla realizar interfaces gráficas.

El layout de la aplicación que se ha diseñado se divide en 2 partes. El lado izquierdo de la aplicación, que muestra información de los valores en grados de las articulaciones, la posición cartesiana del extremo del robot, y por último, la orientación de este. Estos valores se actualizan en tiempo real cada vez que el brazo robot de la simulación se mueve. Por otra parte, se ha creado un objeto *QWebEngineView()* y se ha añadido al lado derecho de la aplicación. Este widget, permite cargar una ventana de navegador dentro de la aplicación 4.6 y realizar búsquedas en ella.

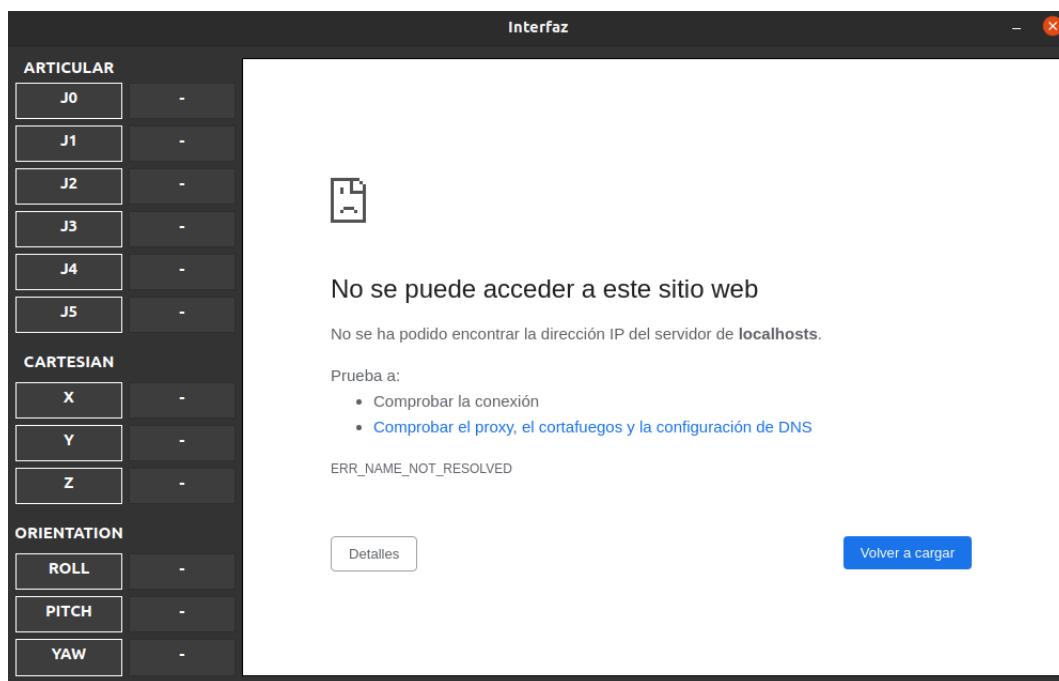


Figura 4.6: Interfaz gráfica sin conexión con el simulador.

El funcionamiento de la aplicación será, por tanto, que nada más abrir la interfaz, el navegador incorporado en esta se conecte a la dirección en la que se está hosteando la simulación de Swift, permitiendo que se muestre dentro de la ventana de la aplicación. Para llevar esto a cabo, cuando se inicia la aplicación, esta lanza en un hilo aparte la simulación hecha en Swift. Esto es para poder ejecutar independientemente la interfaz y los cálculos de la simulación en segundo plano. De esta forma, la interfaz no se ve ralentizada a la hora de hacer cálculos pesados como la cinemática inversa.

4.3.1. Modificación librería Swift

Como ya se ha explicado anteriormente, al ejecutar en otro hilo la simulación, esta se abre en una venta en el navegador en la dirección `http://localhost:PUERTO` siendo el puerto el primero que encuentre disponible. El problema que se encontró es que al lanzar la aplicación y hacer que la ventana del navegador que tiene integrada abra esa dirección, la simulación deja de funcionar y la ejecución de esta termina. Desde la interfaz gráfica, Lo único que se llega a ver es que carga el entorno 4.7, pero no se muestra el robot y además está congelada.

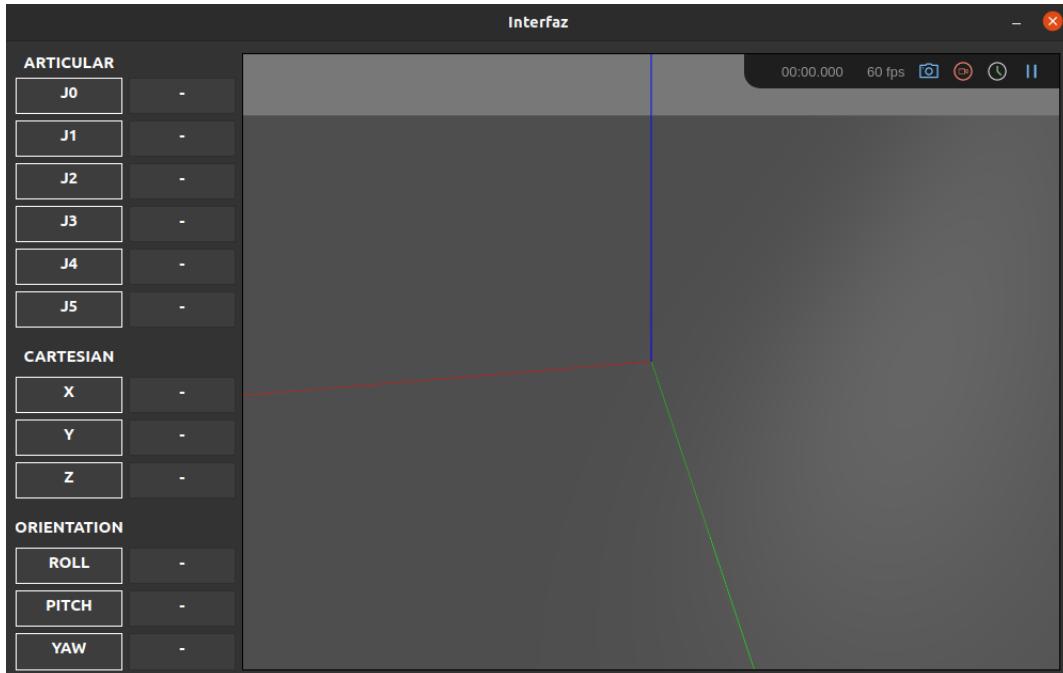


Figura 4.7: Interfaz gráfica intentando cargar la simulación.

Esto se debe a que la instrucción `launch()` de Swift, que inicia la simulación, abre automáticamente una ventana del navegador por defecto, con la dirección en la que se hostea, y muestra en esta la simulación. Por lo que al intentar la interfaz gráfica conectarse a la dirección en la que se hostea la simulación esta devuelve un error y termina, ya que el servidor detecta que hay dos clientes conectados cuando solo debería haber uno. Para poder solucionar este error, fue necesario impedir a la instrucción `launch()` que abriese por su cuenta un navegador, es decir, lanzar la simulación en modo "headless" para de esta forma el único cliente conectándose con la simulación sea el navegador de la aplicación.

Para ello, fue necesario revisar el código de la librería de Swift y crear una nueva versión modificada. La librería utiliza dos ficheros para hacer funcionar la simulación: `Swift.py` y `SwiftRoute.py`. La instrucción `launch()` cuenta con un argumento que permite lanzar la simulación en modo headless, el problema es que al poner este argumento a true, no solo no se inicia una ventana de navegador, sino que tampoco se inicia el servidor que se encarga de gestionar todo lo relacionado con hostear la simulación. Por lo que en su lugar, hay que mo-

dificar la función `start_servers()` que se encuentra en el fichero `SwiftRoute.py`. Esta se llama desde dentro de la función `launch()` y es la encargada de lanzar un hilo para el servidor, crear el servidor, crear la ventana del navegador y conectarla al servidor como cliente. De forma que para implementar la simulación en la aplicación, es necesario modificar esta función para que haga todo menos crear la ventana del navegador.

Se ha añadido a la función `start_servers()` el argumento "open_tab". Mediante este argumento, se puede activar o desactivar la parte del código encargada de crear la ventana. La función devuelve ahora además el puerto en el que se ha lanzado el servidor y el puerto en el que se ha lanzado el socket, para que la interfaz sepa a qué dirección debe conectarse.

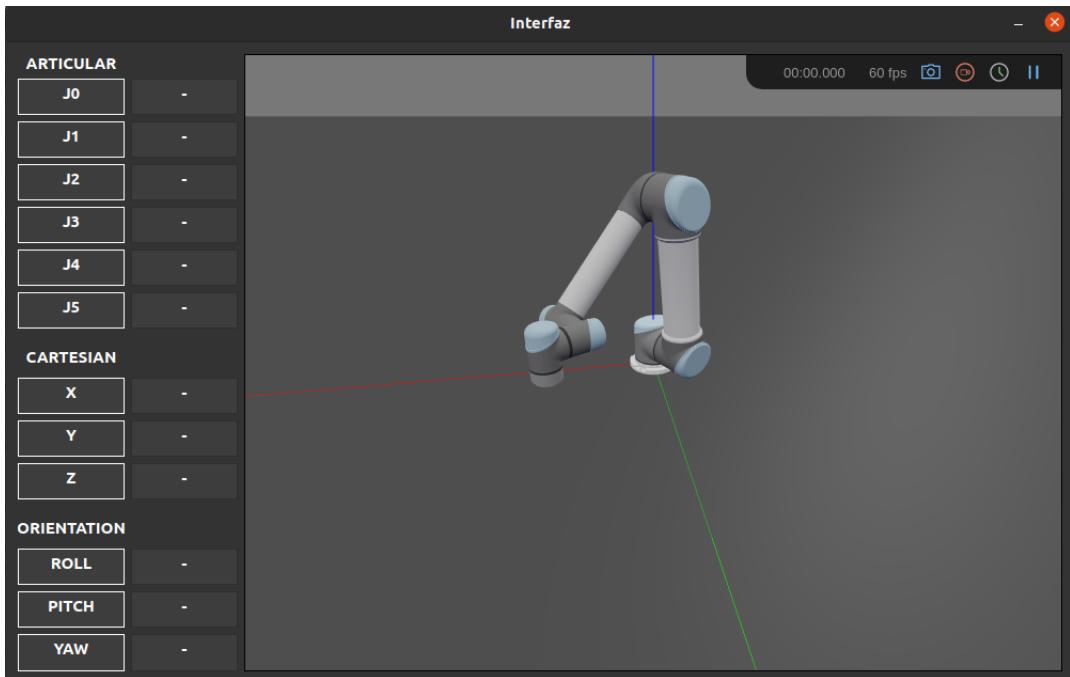
```

710     server.start()
711     server_port = inq.get()
712
713     if open_tab:
714
715         if browser is not None:
716             try:
717                 wb.get(browser).open_new_tab(
718                     "http://localhost:" + str(server_port) + "/" + str(socket_port)
719                 )
720             except wb.Error:
721                 print(
722                     "\nCould not open specified browser, " "using default instead\n"
723                 )
724                 wb.open_new_tab(
725                     "http://localhost:" + str(server_port) + "/" + str(socket_port)
726                 )
727             else:
728                 wb.open_new_tab(
729                     "http://localhost:" + str(server_port) + "/" + str(socket_port)
730                 )
731         else:
732             server = None
733
734         wb.get(browser).open_new_tab(
735             "http://localhost:" + str(3000) + "/" + str(socket_port)
736         )
737
738     # try:
739     #     inq.get(timeout=10)
740     # except Empty:
741     #     print("\nCould not connect to the Swift simulator \n")
742     #     raise
743
744     return socket, server, server_port, socket_port
745

```

Figura 4.8: Cambio en la librería de Swift.

Se ha creado el fichero `modified_swift.py`, el cual contiene todas las funciones necesarias para que funcione la librería de Swift, incluyendo los cambios para que no abra por él mismo una ventana del navegador. Tras comprobar que funcionaba correctamente, la interfaz queda de la siguiente forma:



(a) Aplicación gráfica.

```
adrian@cohete:~/Escritorio/TFG/V4 - Interfaz con Stereo$ python3 app.py
[#]: Esperando a detectar la aplicación!
[#]: Ventana detectada!
[#]: Socket a la espera...
```

(b) Salida del terminal.

Figura 4.9: Interfaz gráfica con simulación implementada.

Finalmente, podemos observar el resultado de la interfaz gráfica con la simulación incluida en esta. En la salida de la terminal se muestra como al ejecutar la aplicación, esta lanza el hilo en el que se ejecuta la simulación con las modificaciones en la librería y esta, una vez detecta que el navegador integrado en la aplicación intenta conectarse a la dirección en la que se está hosteando la simulación, procede a conectarse con la ventana y a publicar en esta la simulación. Se ha definido una posición inicial para la simulación, para que esta no comience con el brazo completamente estirado.

4.4. Comunicación entre el detector y la interfaz

Para mover el brazo robot de la interfaz, es necesario que este reciba las posiciones a las que queremos que este se mueva. Para hacerlo, la interfaz inicia un servidor TCP/IP al cual se conecta el detector de la posición de las manos y le envía las coordenadas de estas.

Para hacer esto, se ha creado la librería *communication.py* en la cual se ha creado la clase *server_tcp* y la clase *client_tcp*. Estas cuentan con todos los métodos necesarios para configurar una conexión servidor-cliente usando el protocolo TCP/IP.

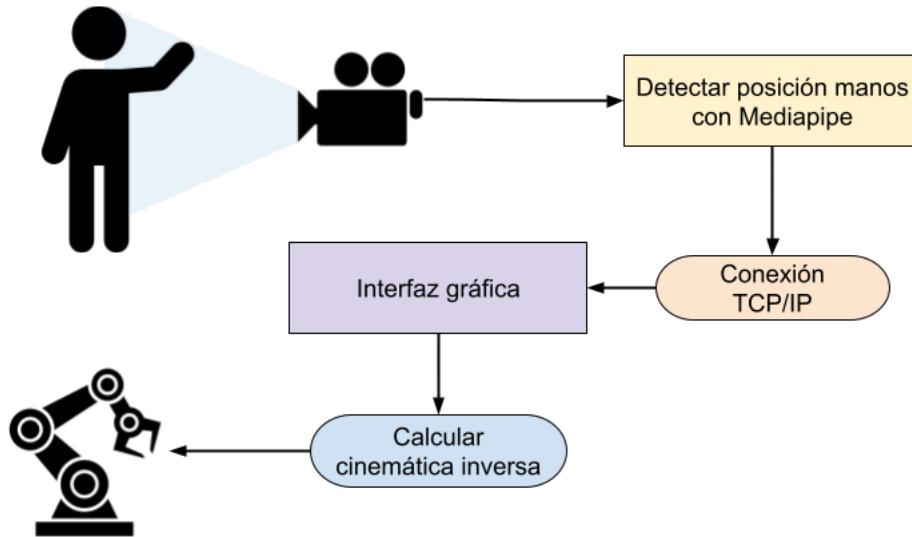


Figura 4.10: Pipeline con una cámara.

Una vez lanzado el servidor tras iniciar la simulación, este se queda a la espera de que alguien trate de conectarse, tal y como se puede ver en la figura 4.9b. Por otra parte, el detector de poses intenta conectarse como cliente al servidor de la aplicación. Una vez conectados, el detector de poses comienza a mandar las coordenadas de las manos por TCP/IP a la aplicación, y esta, a partir de cinemática inversa, procede a actualizar la posición del robot 4.10.

4.5. Mejoras en el sistema de detección

Tras finalizar la interfaz y el sistema de detección, se hicieron pruebas para comprobar distintos problemas que pudiese tener el proyecto.

4.5.1. Estéreo

En un comienzo, tras revisar la documentación de MediaPipe [14], se vio que la red es capaz de obtener la posición X e Y de la mano y aproximar la profundidad en el eje Z mediante IA. Es decir, la distancia a la que esta se encuentra la mano de la cámara. Pero una vez montada la simulación con el detector, se vio que la detección de la profundidad no era muy buena y el resultado era que el robot solo podía moverse correctamente en la simulación en el plano XZ.¹

¹Cabe mencionar que la Z en la simulación representa el eje perpendicular al plano del suelo, mientras que para MediaPipe la Z representa la profundidad. Por lo que el valor de la Z de la simulación corresponde

Puesto que mediante solo una cámara no es posible obtener la profundidad, se decidió cambiar el planteamiento inicial para utilizar dos cámaras y mediante las dos imágenes que estas generen, realizar un estéreo para obtener las coordenadas 3D de las manos.

4.5.1.1. Calibración de cámaras

Para poder realizar el estéreo, primero es necesario llevar a cabo la calibración de la cámara. Esto consiste en obtener los parámetros intrínsecos y extrínsecos de la cámara y los cinco coeficientes de distorsión. En concreto, solo será necesario obtener la matriz de parámetros intrínsecos de la cámara y el vector de coeficientes de distorsión.

Para obtener estos parámetros necesitamos varios puntos en el mundo 3D y sus coordenadas en imágenes 2D. Si conocemos estos pares de puntos 3D y 2D, podemos calcular los coeficientes que serían necesarios para obtener unos puntos 3D en función de sus puntos correspondientes en la imagen 2D. Para ello, se hace uso de un patrón bien definido, normalmente un tablero de ajedrez. A partir de este patrón y mediante la cámara, se obtienen fotos de distintos puntos de vista 4.11 de este para tener muchos pares de puntos 3D con sus respectivos puntos 2D.



Figura 4.11: Distintos puntos de vista del patrón.

Los puntos se detectan haciendo uso de la función de OpenCV *findChessboardCorners()*. Una vez detectados, se refinan las coordenadas de la detección haciendo uso de otra función llamada *cornerSubPix()* que mejora la detección de esquinas.

Código 4.6: Código Python 3: Detección de esquinas y calibración de la cámara.

```
1 # Find the chess board corners.
```

con el Y de MediaPipe, y la Y de la simulación con la Z MediaPipe.

```

2ret, corners = cv2.findChessboardCorners(frame_gray, boardSize, None)
3if ret:
4    # Refining image points.
5    corners = cv2.cornerSubPix(frame_gray, corners, (11,11), (-1,-1), criteria)
6    objpoints.append(objp)
7    imgpoints.append(corners) # Save corner point.
8
9# Obtain camera calibration.
10ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, frame_shape, None, None)

```

Tras obtener los puntos, se llama a la función *calibrateCamera()* y esta devuelve la matriz de parámetros intrínsecos de la cámara y el vector de coeficientes de distorsión. Puesto que estos valores no cambian y son únicos para cada cámara, se almacenan para ser usados después.

Todo este proceso lo lleva a cabo el fichero *camera_calibration.py*, el cual además cuenta con la función *find_devices()* que se encarga de buscar todas las cámaras funcionales y almacena su identificador para poder usar las cámaras automáticamente sin necesidad de saber cuáles son.

4.5.1.2. Calibración del estéreo

Una vez se ha obtenido la matriz de parámetros intrínsecos de la cámara y el vector de coeficientes de distorsión de cada una de las cámaras, estos se utilizan para calibrar el estéreo desde el fichero *stereo_calibration.py*. Desde este, se vuelve a repetir la obtención de pares de puntos 3D-2D, detectando el patrón y almacenando los puntos. Con la diferencia que esta vez se hace al mismo tiempo para ambas cámaras con el fin de poder obtener puntos que aparezcan en las dos cámaras a la vez. Tras esto, en vez de llamar a la función *calibrateCamera()* se llama a la función *stereoCalibrate()* que se encarga de calibrar el estéreo.

Código 4.7: Código Python 3: Detección de esquinas en dos camaras y estéreo.

```

1ret0, frame0_corners = cv2.findChessboardCorners(frame0_gray, boardSize, cv2.CALIB_CB_FAST_CHECK↔
      ↩)
2ret1, frame1_corners = cv2.findChessboardCorners(frame1_gray, boardSize, cv2.CALIB_CB_FAST_CHECK↔
      ↩)
3
4if ret0 and ret1:
5    image_number += 1
6    print("\t- Board found: ", image_number)
7
8    # Find the chess board corners.
9    _, corners0 = cv2.findChessboardCorners(frame0_gray, boardSize, None)
10   _, corners1 = cv2.findChessboardCorners(frame1_gray, boardSize, None)
11
12   # Refining image points.
13   corners0 = cv2.cornerSubPix(frame0_gray, corners0, (11,11), (-1,-1), criteria)
14   corners1 = cv2.cornerSubPix(frame1_gray, corners1, (11,11), (-1,-1), criteria)
15
16   # Add object points, image points (after refining them)
17   objpoints.append(objp)
18   imgpoints0.append(corners0)
19   imgpoints1.append(corners1)
20
21# Obtain Translation and Rotation matrix by stereo.
22ret, CM1, dist1, CM2, dist2, R, T, E, F = cv2.stereoCalibrate(objpoints, imgpoints0, imgpoints1, mtx0, dist0,
23mtx1, dist1, frame_shape, criteria = criteria, flags = cv2.CALIB_FIX_INTRINSIC)

```

La razón por la que la calibración de las cámaras no se hace a la vez que la del estéreo, aunque el código sea el mismo, es porque es más sencillo obtener capturas del patrón para cada una de las cámaras de forma independiente sin tener que preocuparse porque los puntos se vean en ambas cámaras a la vez.

Tras realizar la calibración del estéreo obtenemos, entre otras cosas, la matriz de rotación y la matriz de traslación que permiten cambiar de la base de la primera cámara a la de la segunda. Estas dos matrices se guardan también, ya que mientras no se muevan las cámaras, esta calibración será válida.

Se ha creado dentro del fichero *utils.py* la función *setup()* que es la que obtiene las matrices de proyección de cada una de las cámaras, donde la proyección de la primera cámara será su matriz de parámetros intrínsecos multiplicada por una matriz de transformación donde la rotación es la identidad y la traslación un vector nulo. Por otra parte, la matriz de proyección de la segunda cámara será su matriz de parámetros intrínsecos multiplicada por una matriz de transformación, donde la rotación y la traslación son los valores obtenidos en la calibración del estéreo. De esta forma, una vez triangulados los puntos en el espacio 3D, el origen de la posición de estos será la cámara 1.

Código 4.8: Código Python 3: Obtención de las matrices de proyección.

```

1 ##### Calculate projection matrix P1 and P2:
2 #RT matrix for Camera 1 is identity.
3 RT1 = np.concatenate([np.eye(3), [[0],[0],[0]]], axis = -1)
4 P1 = mtx[0] @ RT1 #projection matrix for C1
5
6 #RT matrix for Camera 2 is the R and T obtained from stereo calibration.
7 RT2 = np.concatenate([R, T], axis = -1)
8 P2 = mtx[1] @ RT2 #projection matrix for C2
9
10 return P1, P2

```

Las dos matrices de proyección que se obtienen, son las que se usarán para triangular la posición de los puntos en el espacio 3D a partir de las dos cámaras. Para ello, se hace uso de la función de OpenCV *triangulatePoints()*, la cual, a partir de las dos matrices de proyección y las coordenadas de un punto visto desde dos cámaras distintas, permite triangular la posición del punto en 3D en relación con el sistema de referencia de la cámara 1.

Código 4.9: Código Python 3: Obtención de las coordenadas 3D mediante triangulación.

```

1 for kpt, uv1, uv2 in zip(pose_keypoints, frame0_keypoints, frame1_keypoints):
2     p3d = np.array([-1, -1, -1])
3
4     if uv1[0] != -1 and uv2[0] != -1:
5         p4d = cv2.triangulatePoints(P1, P2, uv1, uv2)
6         p3d = (p4d[:3, :] / p4d[3, :]).T
7
8     points_3D[kpt] = p3d.copy()

```

4.5.1.3. Modificación código detección manos

Una vez realizado con éxito el estéreo de dos cámaras, para añadirlo al proyecto solo fue necesario modificar el código encargado de usar MediaPipe para detectar la posición de las manos. Ahora, el fichero *pose_stereo.py* se encarga de obtener las imágenes de dos cámaras a la vez, detectar las manos de una persona en cada uno de los frames por separado y pasar las coordenadas que ha devuelto la red para cada uno de los puntos 2D a la función *DLT()*. Esta función triangulará la posición en el espacio 3D tomando como origen la cámara 1 y devolverá las coordenadas 3D de este punto.

Una vez se tienen las coordenadas 3D del punto detectado, este se pasa igual que antes mediante TCP/IP a la interfaz para que actualice la posición del robot de la simulación. Debido a este cambio, el nuevo pipeline 4.12 utilizando dos cámaras y estéreo queda de la siguiente forma:

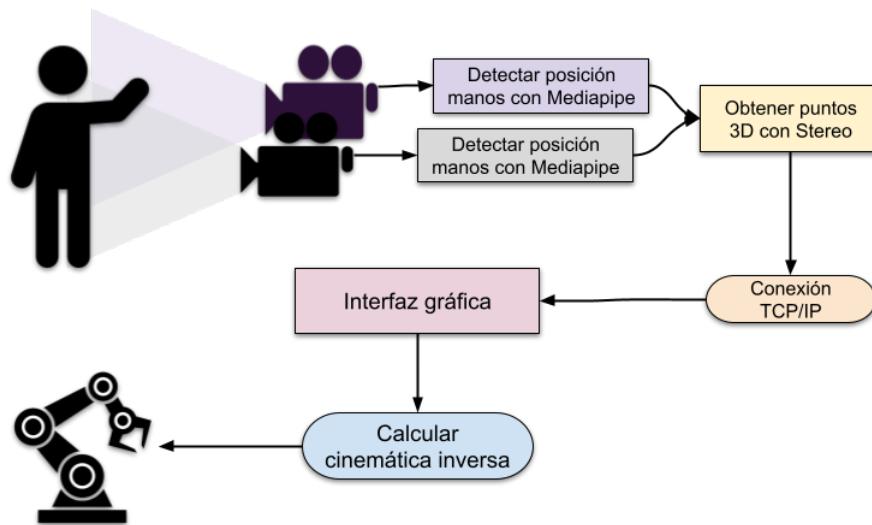


Figura 4.12: Pipeline con dos cámaras y estéreo.

4.5.2. Ajuste mano-hombro

Tras hacer pruebas con la detección usando estéreo, se vio que los valores de la posición podían adquirir valores demasiado grandes. Esto es debido a que las coordenadas del punto 3D son teniendo el origen en la cámara 1. Por ello, aunque la persona no mueva el brazo, si se acerca o aleja de la cámara, el brazo robot se mueve. Incluso puede darse el caso en que la persona se aleje tanto que el punto al que se intente mover al robot esté fuera de su alcance.

Para solucionar esto, en vez de usar tal cual la posición 3D de la mano obtenida tras realizar el estéreo, se obtiene también la posición 3D del hombro y se calcula la diferencia entre la posición de la mano y la del hombro. De esta forma, se obtiene la posición 3D de la mano que tiene como origen el hombro en vez de la cámara.

Código 4.10: Código Python 3: Ajuste posición 3D con origen en el hombro.

```

1 pose_keypoints = {
2     "L_wrist" : mp_pose.PoseLandmark.LEFT_WRIST,
3     "R_wrist" : mp_pose.PoseLandmark.RIGHT_WRIST,
4     "L_shoulder" : mp_pose.PoseLandmark.LEFT_SHOULDER,
5     "R_shoulder" : mp_pose.PoseLandmark.RIGHT_SHOULDER,
6 }
7
8 L_wrist = points_3D["L_wrist"] - points_3D["L_shoulder"] # Ponemos como origen el hombro izquierdo.
9 R_wrist = points_3D["R_wrist"] - points_3D["R_shoulder"] # Ponemos como origen el hombro derecho.

```

Tras este ajuste, la persona puede moverse sin preocuparse, acercándose o alejándose como desee de la cámara sin que el brazo robot varíe su posición. Mientras no mueva la mano del sitio, el robot seguirá sin moverse, ya que se tendrá en cuenta para el desplazamiento la distancia que hay de la mano al hombro en vez de de la mano a la cámara.

4.6. Incorporación Gazebo

Se ha implementado la HMI en ROS Noetic, de forma que es posible iniciar una simulación en Gazebo del brazo robot que se quiera controlar y utilizar la interfaz desarrollada para el caso base para que se haga cargo de controlar el brazo. Gazebo tiene la ventaja de que permite simular físicas y multitud de objetos, por lo que se pueden simular escenarios completos.

Para simular el robot UR5e en Gazebo, se ha hecho uso de un paquete de Universal Robots [20] el cual permite simular todos los modelos de UR en el entorno de Gazebo. Este paquete permite además realizar un control en tiempo real de un brazo robot real. Para iniciar la simulación, hay que lanzar el siguiente comando en un terminal: `roslaunch ur_gazebo ur<modelo>_bringup.launch`. Una vez hecho esto, podemos controlar el robot publicando en el topic `/pos_joint_traj_controller/command`.

Se ha creado la clase `gazebo_controller`, la cual, por una parte, se encarga de publicar en el topic encargado de controlar al robot y por otra se suscribe al topic `/joint_states`. Este último topic permite obtener en tiempo real los valores articulares de cada una de las articulaciones del brazo robot en Gazebo. Los valores devueltos por este topic se utilizan para controlar la simulación del brazo implementada en la interfaz. De esta forma, esta no mostrará la posición deseada directamente, sino la posición que tiene en ese instante el brazo de Gazebo.

Código 4.11: Código Python 3: Métodos de la clase gazebo_controller

```

1 def callback(self, data):
2     feedback_q = list(data.position)
3     feedback_q[0], feedback_q[2] = feedback_q[2], feedback_q[0]
4     self.q = feedback_q
5
6 def move(self,q):
7     movement = JointTrajectory()
8     pts = JointTrajectoryPoint()
9
10    movement.joint_names = self.joint_names
11    pts.positions = q
12    pts.velocities = [0]*6
13    pts.accelerations = [0]*6

```

```

14     pts.effort = [0]
15     pts.time_from_start = rospy.Duration(0.01) # Tiempo en el que se ha de realizar el movimiento
16     movement.points = [pts]
17
18     self.pub.publish(movement)

```

Mediante estas dos funciones, se mueve al robot fijándole una posición destino a la que desplazarse y se obtiene su posición actual mediante un callback que se ejecuta cada vez que se actualiza el topic `/joint_states`.

4.7. Demo real

Para que la aplicación desarrollara funcione con el robot real, ha sido necesario rediseñar la simulación integrada en la aplicación. Esta se utiliza para que el operario tenga un feedback del estado del robot controlado en caso de no verlo. Hasta ahora se utilizaba el entorno gráfico de Swift, pero este solo es válido para representar aquellos robots que la librería de Peter Corke tenga con sus modelos 3D. En caso de que se requiera controlar un brazo robot que no se encuentre en la librería, es posible representarlo sin utilizar Swift. En su lugar se puede hacer uso de una representación en PyPlot, que funciona mediante la librería Matplotlib [21] 4.13.

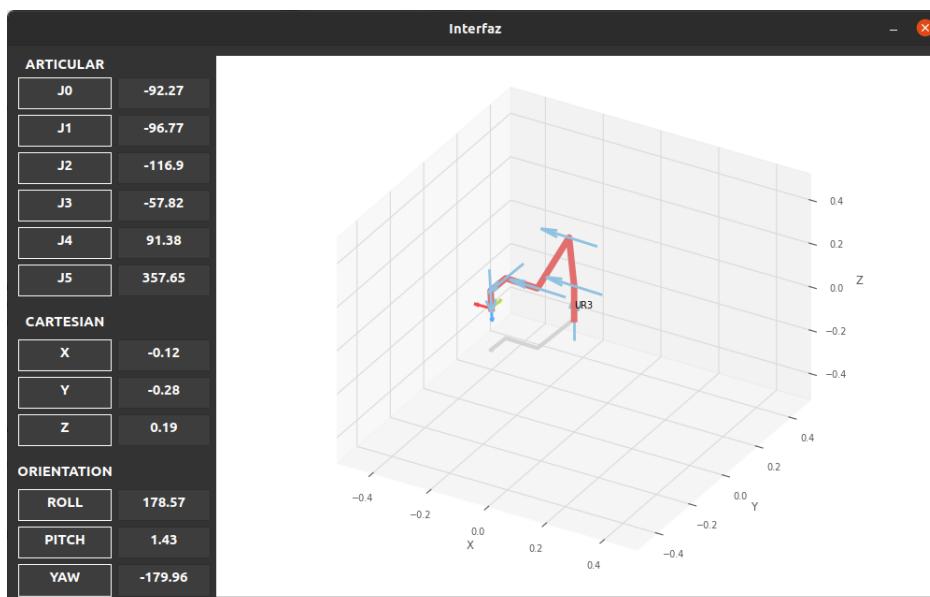


Figura 4.13: Interfaz generalizable a cualquier brazo robot.

Esta alternativa para representar los brazos robóticos en la interfaz tiene la ventaja de que es generalizable a cualquier tipo de brazo robot. Para poder representarlo, es necesario definir previamente su tabla DH mediante la Robotics Toolbox, que cuenta con distintas clases y funciones que permiten agilizar este proceso. En el caso de la demo real, se hace uso de un UR3, para el cual la librería no cuenta con un modelo predefinido. Por lo tanto, se ha definido a mano la tabla DH [22] del brazo robot haciendo uso de la clase *DHRobot*.

Código 4.12: Código Python 3: Implementación de un UR3 a partir de su tabla DH.

```

1 import roboticstoolbox as rtb
2 import numpy as np
3 from numpy import pi
4
5 robot_UR3 = rtb.DHRobot(
6     [
7         rtb.RevoluteDH(d=0.1519, alpha=pi/2, qlim=np.deg2rad([-363, 363])), # Base
8         rtb.RevoluteDH(a=-0.24365, qlim=np.deg2rad([-363, 363])), # Hombro (realmente de -364 a 363)
9         rtb.RevoluteDH(a=-0.21325, qlim=np.deg2rad([-363, 363])), # Codo
10        rtb.RevoluteDH(d=0.11235, alpha=pi/2, qlim=np.deg2rad([-363, 363])), # Muñeca 1
11        rtb.RevoluteDH(d=0.08535, alpha=-pi/2, qlim=np.deg2rad([-363, 363])), # Muñeca 2
12        rtb.RevoluteDH(d=0.0819, qlim=np.deg2rad([-1000, 1000])) # Muñeca 3 (sin límites articulares)
13    ], name="UR3")

```

Para poder integrar la nueva representación del robot en la aplicación, ha sido necesario modificar la librería de la Robotics Toolbox para permitir mostrar el robot en la figura creada desde PyQt5. Por lo que se ha creado una versión de la aplicación alternativa a la de Swift llamada *app_pyplot*. Esta, en lugar de integrar un navegador, integra la ventana de una figura.

4.7.1. Entorno URsim

Antes de realizar una prueba con el robot real, se han realizado varias pruebas con la HMI para asegurarse de que se diera ningún fallo como movimientos demasiado rápidos o colisiones. Para ello, se ha hecho uso de URSim.

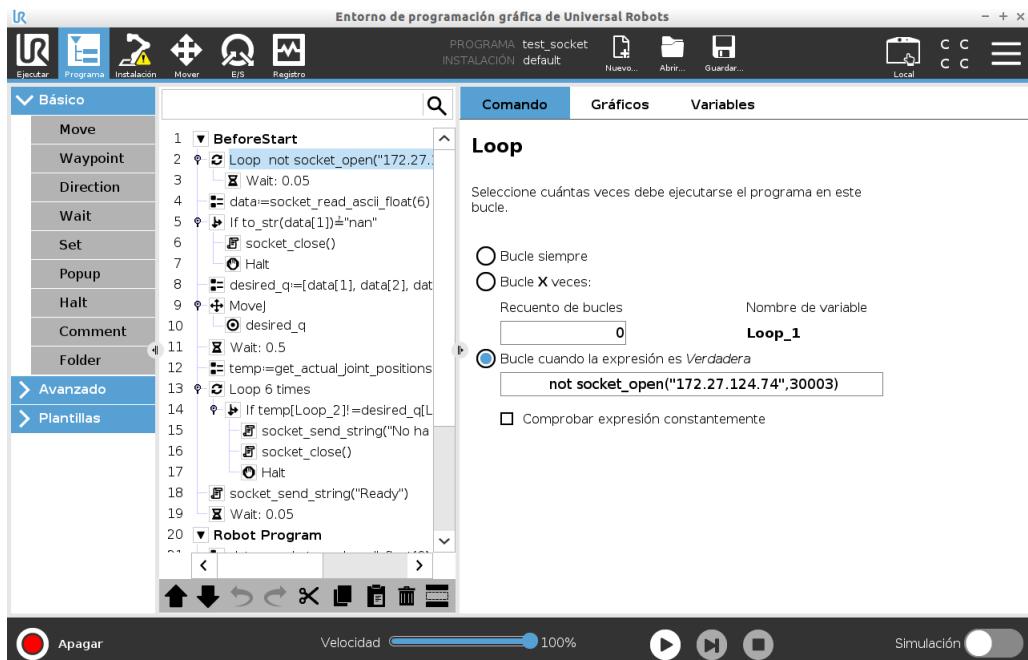


Figura 4.14: Panel de programación de un UR5e en URSim.

URsim [23] es un simulador de robots offline que se puede instalar de forma nativa en Linux o mediante una máquina virtual. Este, permite emular un robot UR real mostrando caracte-

rísticas como su panel de control, su configuración o un panel de programación 4.14, desde el cual es posible programar el comportamiento del robot mediante el lenguaje URScript.

La comunicación entre URSim y la aplicación es mediante TCP/IP. La aplicación inicia un servidor y se queda a la espera de que se conecten el cliente del detector de poses y el cliente de URSim. Una vez conectados, se envían mensajes con la posición deseada del robot al simulador.

Tras comprobar que era posible que el robot se moviera a posiciones en las que podía colisionar, se ha acotado el rango de valores que puede enviar el detector de pose para impedir que el robot colisione consigo mismo o realice movimientos demasiado amplios y choque con cualquier objeto de su entorno.

4.7.2. Código URscript

Los robots de Universal Robots cuentan con su propio lenguaje de programación llamado URscript. Mediante este, es posible programar una serie de instrucciones para que el robot lleve a cabo la tarea deseada. A partir de la documentación [24], se ha visto que es posible comunicarse mediante TCP/IP con el robot y controlarlo en tiempo real.

Inicialmente, el robot se queda a la espera intentando conectarse con el servidor, esto se puede observar en la figura 4.14, donde URSim intenta conectarse a la dirección y puerto en la que se ha creado el servidor. Tras conectarse, el robot recibe la posición home a la que debe moverse para iniciar el control. Mediante URscript, se puede mover a un robot utilizando la instrucción *movej()* y *servoj()* entre otras. La primera requiere que el robot esté inicialmente parado y sirve para mover al robot a una posición en función de unos valores articulares. La segunda, permite controlar las articulaciones en tiempo real. Por lo tanto, inicialmente se mueve el robot a la posición home utilizando la instrucción *movej()* para que vaya lentamente hacia esta, y tras confirmar que el robot ha alcanzado la posición home, se pasa a utilizar la instrucción *servoj()* para controlarlo en tiempo real.

Cada vez que el robot recibe un mensaje del servidor, este le envía de vuelta los ángulos de sus articulaciones. De esta forma, la representación del robot que se muestra en la interfaz refleja el movimiento que lleva a cabo el robot real, en vez de moverse directamente a la posición deseada. En caso de que el robot real dejase de moverse por cualquier razón, esto se vería reflejado en la interfaz, permitiendo al operario saber que el robot no se mueve incluso aunque este no lo vea.

Por seguridad, en caso de que el robot no reciba ningún mensaje desde el servidor pasados dos segundos, finalizará la ejecución del programa.

4.7.3. Robot real

Para utilizar el robot real, primero es necesario cargar el programa de URscript. Puesto que no se ha hecho ninguna instalación, el robot no dejará utilizarlo a no ser que se cargue el programa desde el panel de programación. Tras esto, solamente será necesario modificar la

IP a la que se debe conectar el robot y poner la ip local del ordenador en el que se inicie la aplicación y el servidor. Es necesario que tanto el robot como el ordenador estén en la misma subred para funcionar. Para la prueba en el robot real, se ha hecho uso del puerto 500002 y la IP del ordenador es 192.168.2.130, por lo tanto, dentro del código de Urscrip hay que pasar como argumentos a la función *socket_open()* la IP y el puerto.

Tras esto, el robot se moverá lentamente a la posición que se ha definido en la aplicación como home utilizando la instrucción *movej()* y, una vez llegue, mandará un mensaje al servidor avisando de que está listo para iniciar el control en tiempo real.

5. Resultados

En el siguiente apartado se va a hablar del resultado del proyecto, su funcionamiento y su rendimiento. Para el proyecto se ha creado una aplicación base que se encarga de gestionar y mostrar la simulación en una interfaz gráfica, y sobre esta se han creado más tarde una implementación en Gazebo y una demo real. De forma que el proyecto cuenta con tres interfaces distintas.

5.1. Interfaces desarrolladas

La interfaz base, se trata de una interfaz hombre máquina capaz de capturar los movimientos que realiza el operario con las manos y convertirlos en comandos para controlar un brazo robot simulado. Esta interfaz hace uso de la red neuronal de MediaPipe BlazePose que, detecta la pose de una persona y envía las coordenadas de los puntos clave a la interfaz. Además, para mejorar la precisión de la detección, se realiza sobre un estéreo utilizando dos cámaras, mediante el cual es posible obtener la profundidad de los puntos clave y detectar correctamente los movimientos en los tres ejes.

Por otra parte, la interfaz con Gazebo es una variación de la base, la cual permite controlar un brazo robot simulado en Gazebo haciendo uso del paquete de Universal Robots, publicando en los topics de este la posición deseada. La razón principal de realizar esta versión de la interfaz, es mostrar cómo es posible utilizarla para controlar brazos robots en simulaciones más complejas, como puede ser la de Gazebo en este caso.

Finalmente, la demo real es otra variación de la base, la cual en vez de publicar en los topics de ROS, se conecta mediante TCP/IP con un brazo robot real para enviarle a este las posiciones deseadas. La interfaz recibe de vuelta la posición actual del brazo real, por lo que la simulación incluida dentro de la aplicación replica los movimientos que lleva a cabo el brazo real.

Además, la interfaz permite cargar tanto los modelos predefinidos de la librería de Robotic Toolbox como modelos definidos a mano haciendo uso de los valores DH del robot. En función de cómo se haya definido al robot, la interfaz mostrará una representación del robot en el entorno Swift o en PyPlot.

5.2. Rendimiento de la interfaz

Tras realizar distintas pruebas, se ha visto que el retardo depende principalmente del uso del detector. Ya que este tarda en detectar la pose de media 0.05 segundos por cada frame. De forma que si solo se usa una cámara, la interfaz funciona de media a unos 18 FPS. Al

añadir la segunda cámara para realizar el estéreo, se utiliza al detector dos veces, por lo que el rendimiento cae a 10 FPS, tardando al rededor de 100 ms en capturar la posición de las manos.

Por otra parte, el cálculo de la cinemática inversa tarda también de media 0.05 segundos, por lo que en caso de que se utilizasen dos brazos robots en vez de solo uno, serían 100 ms por cada punto recibido del detector. Por lo tanto, si se quisiera controlar en tiempo real dos brazos robots, el control tendría un retardo de 200 ms.

Tanto para el caso de controlar dos brazos a la vez como para el de solo uno, el retardo es menor a 250 ms, por lo que se considera una cantidad de retardo aceptable para realizar una teleoperación directa del brazo robot.

Cabe mencionar que estos tiempos se han obtenido ejecutando el detector en la CPU. Aunque el detector BlazePose esté diseñado para ser rápido y ligero, tiene la opción de ejecutarse desde la GPU, lo que reduciría notablemente el retardo de la detección. Por otra parte, el proyecto se ha realizado en Python 3, un lenguaje interpretado y no compilado, que en muchos casos puede ser más lento que un lenguaje compilado. Por lo que sería posible reducir aún más el retardo mediante una versión del proyecto para C++.

5.3. Complejidad de instalación y coste

El objetivo del trabajo era la realización de una HMI de bajo coste que no requiriera de dispositivos complejos o aparatosos para llevar a cabo su cometido. En lugar de utilizar cámaras de nubes de puntos, sensores o colocar marcadores sobre el cuerpo del operario, solo es necesario un ordenador y dos cámaras, cuyo coste en total no llega a superar los 30 €, para poder utilizar el proyecto.

El proyecto se puede poner a punto en cualquier lugar, ya que no necesita de numerosos pasos, ni realizar una compleja instalación para ponerlo en funcionamiento. Solamente es necesario colocar las dos cámaras orientadas hacia el espacio en el que se encontrará el operario y calibrarlas haciendo uso de un patrón de ajedrez y ya se habrán realizado todos los pasos necesarios para poder utilizar el proyecto.

El resultado del proyecto es una interfaz hombre máquina que permite controlar la gran mayoría de brazos robots, ya que es generalizable a cualquier cadena cinemática que estos tengan y solo necesita de la cinemática inversa de estos para funcionar.

5.4. Caso de uso

La versión de la interfaz de la demo permite llevar a cabo una prueba del proyecto con un robot real. Para la demostración real, se ha utilizado un robot UR3 al cual se le ha cargado el programa de urscript para que se conecte con el servidor.



Figura 5.1: Robot UR3 utilizado para la demo.

Puesto que no se cuenta con un modelo predefinido para este robot, se ha definido a mano 4.12 los parámetros DH del UR3 para poder representarlo y calcular su cinemática inversa. Tras esto, se inicia la aplicación, el robot se mueve la posición de home y se inicia el control de este en función de la posición en la que se encuentre la muñeca izquierda de la persona con respecto a su hombro izquierdo, la cual se usa como un offset de la posición home.

Puesto que es posible que los valores de la detección que se utilizan como offset sean demasiado grandes, estos están divididos por un factor de seguridad igual a 200, para evitar que el robot pueda hacer grandes movimientos. Tras probar a mover el robot real, se ha visto que este factor era demasiado grande y el robot a penas podía moverse, por lo que se ha reducido el factor y ahora se dividen las coordenadas de la posición de la mano por 80 en vez de 200.

Se ha observado que es posible que la detección falle si el operador está muy cerca de las cámaras o se sale del rango de visión de una de ellas. Puesto que la red de estimación de pose intenta estimar la posición de las manos incluso cuando están ocluidas, a menudo tiende a devolver valores aproximados de dónde estima que se encuentra esta incluso sin verla. Estos valores pueden llegar a tener mucho error con respecto a la posición real, haciendo que el robot intente moverse a una posición muy lejana en muy poco tiempo. Como medida de control, se han acotado los valores máximos y mínimos que puede devolver la función de estimación de pose, de forma que el robot no pueda chocarse consigo mismo o contra el suelo, ni tampoco llegue a estirarse demasiado (el rango está definido en ± 0.15).

Un problema que se ha podido observar, es que tras moverse el robot a la posición home, este se mueve mediante la instrucción *servoj()* a la posición deseada, lo cual puede hacer que el primer movimiento lo realice con mucha brusquedad. En caso de que la posición deseada

inicial esté muy alejada de la posición home, el robot iniciará un movimiento con mucha aceleración. La instrucción cuenta con un controlador proporcional, por lo que para reducir la velocidad de reacción del robot, se ha reducido el valor de la ganancia hasta el mínimo (100), de forma que ahora el robot se mueve de forma suave de un punto a otro. Para reducir la brusquedad, se ha hecho que el robot compruebe la distancia entre su posición actual y la deseada, y en caso de que sea mayor a un umbral, se llama a la instrucción *servoj()* con un valor de tiempo más grande para que tarde más en realizar el desplazamiento.

Por último, se ha observado que la detección funciona mejor a un metro de distancia de las cámaras, ya que esta tiende a estimar mejor la posición de la muñeca.

6. Conclusiones

Como conclusión de este TFG, una vez finalizado el desarrollo del proyecto se va a hacer un repaso de lo conseguido y se van a tratar de proponer algunas conclusiones, reflexiones y mejoras sobre lo aprendido a lo largo del proyecto.

Durante el desarrollo del proyecto, se han adquirido nuevos conocimientos en el campo de ML y en el de visión por computador. En primer lugar, se ha aprendido acerca del problema de la estimación de la pose de una persona y cómo se puede solucionar a día de hoy. Mediante esto, se ha profundizado más en las nociones aprendidas en la asignatura de Sistemas Inteligentes, donde se nos introdujo al mundo de la IA.

En segundo lugar, se reforzaron y ampliaron los conocimientos sobre técnicas de visión por computador, en este caso profundizando en la calibración de la cámara y del estéreo y en su desarrollo matemático. Esto ha permitido implementar con éxito un estéreo haciendo uso de dos webcams.

En tercer lugar, se han adquirido nuevos conocimientos para llevar a cabo la aplicación de la interfaz gráfica junto con la simulación. Aprendiendo la utilidad de paralelizar procesos lanzando hilos y creando servidores TCP/IP para la transmisión de datos entre el detector de pose y la interfaz.

Gracias a todo lo aprendido y experimentado, se ha conseguido alcanzar el objetivo final del proyecto; la creación de una HMI de bajo coste que permita controlar mediante visión un brazo robot. Como se ha mencionado anteriormente a lo largo del proyecto, la aplicación desarrollada se puede implementar en varios robots. Esto permite que independientemente de la cadena cinemática que forme el brazo robot, se pueda aplicar el mismo planteamiento para controlarlo haciendo cambios mínimos.

Como futuras mejoras, se puede investigar el uso de filtros y realimentación del error para estabilizar los valores de posición de los puntos clave en vez de que oscilen en torno a un valor, como por ejemplo el uso del filtro Butterworth [25]. Esto permitiría que el movimiento reflejado tanto en la simulación como en el robot real fuese más suave. Además, para optimizar la detección, se puede lanzar la red desde la GPU y pasar el código a C++ para acelerar los cálculos.

Bibliografía

- [1] Jinbao Wang, Shujie Tan, Xiantong Zhen, Shuo Xu, Feng Zheng, Zhenyu He, and Ling Shao. Deep 3d human pose estimation: A review. *Computer Vision and Image Understanding*, 210:103225, 2021.
- [2] Elisha Odemakinde. Human pose estimation with deep learning – ultimate overview in 2022. <https://viso.ai/deep-learning/pose-estimation-ultimate-overview/>. Última visita abril del 2022.
- [3] On-device, real-time body pose tracking with mediapipe blazepose. <https://ai.googleblog.com/2020/08/on-device-real-time-body-pose-tracking.html>. Última visita abril del 2022.
- [4] Valentin Bazarevsky, Ivan Grishchenko, Karthik Raveendran, Tyler Zhu, Fan Zhang, and Matthias Grundmann. Blazepose: On-device real-time body pose tracking, 2020.
- [5] Boston Dynamics. Robot atlas. <https://www.bostondynamics.com/atlas>. Última visita mayo del 2022.
- [6] Boston Dynamics. Robot spot. <https://www.bostondynamics.com/spot>. Última visita mayo del 2022.
- [7] Ce Zheng, Wenhan Wu, Chen Chen, Taojiannan Yang, Sijie Zhu, Ju Shen, Nasser Keh-tarnavaz, and Mubarak Shah. Deep learning-based human pose estimation: A survey, 2020.
- [8] T.F. Cootes, C.J. Taylor, D.H. Cooper, and J. Graham. Active shape models-their training and application. *Computer Vision and Image Understanding*, 61(1):38–59, 1995.
- [9] Camillo Lugaressi, Jiuqiang Tang, Hadon Nash, Chris McClanahan, Esha Ubweja, Michael Hays, Fan Zhang, Chuo-Ling Chang, Ming Guang Yong, Juhyun Lee, Wan-Teh Chang, Wei Hua, Manfred Georg, and Matthias Grundmann. Mediapipe: A framework for building perception pipelines, 2019.
- [10] Coco - common objects in context. <https://cocodataset.org/#keypoints-2020>. Última visita mayo del 2022.
- [11] Valentin Bazarevsky, Yury Kartynnik, Andrey Vakunov, Karthik Raveendran, and Matthias Grundmann. Blazeface: Sub-millisecond neural face detection on mobile gpus, 2019.
- [12] OpenCV. Camera calibration and 3d reconstruction. https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html. Última visita mayo del 2022.

- [13] Pose estimation. <https://paperswithcode.com/task/pose-estimation>. Última visita abril del 2022.
- [14] Documentación mediapipe - pose. <https://google.github.io/mediapipe/solutions/pose.html>. Última visita mayo del 2022.
- [15] Web mediapipe. <https://mediapipe.dev/>. Última visita mayo del 2022.
- [16] Peter Corke. Github robotics toolbox para python. <https://github.com/petercorke/robotics-toolbox-python>. Última visita mayo del 2022.
- [17] Peter Corke and Jesse Haviland. Not your grandmother's toolbox – the robotics toolbox reinvented for python. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11357–11363. IEEE, 2021. Accedido en mayo del 2022.
- [18] Documentación de pyqt5. <https://pypi.org/project/PyQt5/>. Última visita mayo del 2022.
- [19] Gazebo. <https://gazebosim.org/home>. Última visita abril del 2022.
- [20] Universal robots ros driver. https://github.com/UniversalRobots/Universal_Robots_ROS_Driver. Última visita mayo del 2022.
- [21] Matplotlib: Visualization with python. <https://matplotlib.org/>. Última visita mayo del 2022.
- [22] Dh parameters for calculations of kinematics and dynamics. <https://www.universal-robots.com/articles/ur/application-installation/dh-parameters-for-calculations-of-kinematics-and-dynamics/>, Enero 2022. Última visita mayo del 2022.
- [23] Universal Robots. Offline simulator - cb-series - non linux - ursim 3.14.3. <https://www.universal-robots.com/download/software-cb-series/simulator-non-linux/offline-simulator-cb-series-non-linux-ursim-3143/>. Última visita mayo del 2022.
- [24] Universal Robots. The urscript programming language. <https://s3-eu-west-1.amazonaws.com/ur-support-site/32554/scriptManual-3.5.4.pdf>, Abril 2018. Última visita mayo del 2022.
- [25] Emima Ioana Jiva. Desarrollo de la teleoperación de robots industriales y colaborativos mediante técnicas avanzadas de visión artificial, 2019.

Lista de Acrónimos y Abreviaturas

| | |
|-------------|----------------------------------|
| ASM | Active Shape Model. |
| DH | Denavit-Hartenberg. |
| GANs | Generative adversarial networks. |
| GDL | Grados de Libertad. |
| HMI | Human-Machine Interface. |
| IA | Inteligencia Artificial. |
| IMU | Inertial Measurement Unit. |
| ML | Machine Learning. |
| PCA | Principal Component Analysis. |
| SMPL | Skinned Multi-Person Linear. |

A. Guía de instalación y uso

En el siguiente apartado, se va a explicar cómo se puede instalar el proyecto en un equipo con Ubuntu 20.04.4 LTS y cómo utilizarlo.

- Se ha hecho un Github del proyecto: Enlace Github
- Se ha hecho un vídeo de demostración del proyecto: Enlace Vídeo

A.1. Instalación

Para usar el proyecto, es necesario realizar la instalación de distintas dependencias y paquetes. Estas pueden instalarse mediante el fichero *requirements.txt* que hay en el Github. Para evitar problemas con la instalación, es recomendable utilizar un entorno virtual. Además, si se quiere utilizar la simulación en Gazebo, también es necesario descargar el paquete de Universal Robots.

A.1.1. Dependencias Python

El proyecto se ha desarrollado utilizando Python 3.8.10, y las dependencias necesarias se pueden instalar fácilmente mediante el fichero requirements.txt del proyecto.

```
1 $ pip3 install -r requirements.txt
```

A.1.2. Simulador Gazebo

Por otra parte, si se quiere utilizar la simulación en Gazebo, también es necesario crear un workspace en el que instalar los paquetes de Universal Robots [20] para simular y controlar sus robots. El paquete necesita de un sistema que tenga instalado ROS. En este caso, se ha utilizado ROS noetic, aunque el paquete es también compatible con ROS melodic.

```
1 # source global ros
2 $ source /opt/ros/<your_ros_version>/setup.bash
3
4 # create a catkin workspace
5 $ mkdir -p catkin_ws/src && cd catkin_ws
6
```

```

7   # clone the driver
8   $ git clone https://github.com/UniversalRobots/Universal_Robots_ROS_Driver.←
     ↪ git src/Universal_Robots_ROS_Driver
9
10  # clone fork of the description. This is currently necessary, until the ←
     ↪ changes are merged upstream.
11  $ git clone -b calibration-devel https://github.com/fmauch/universal_robot.←
     ↪ git src/fmauch_universal_robot
12
13  # install dependencies
14  $ sudo apt update -qq
15  $ rosdep update
16  $ rosdep install --from-paths src --ignore-src -y
17
18  # build the workspace
19  $ catkin_make
20
21  # activate the workspace (ie: source it)
22  $ source devel/setup.bash

```

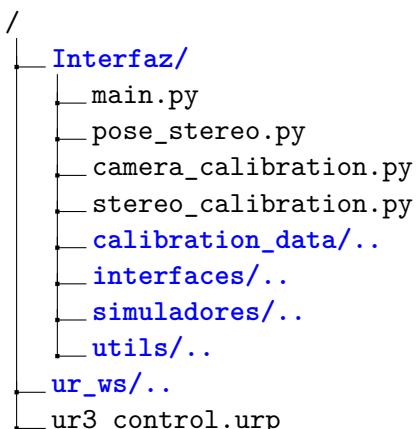
A.1.3. Descargar Proyecto

Finalmente, podemos descargar el proyecto subido a GitHub mediante el siguiente comando:

```
1  $ git clone https://github.com/Adriagent/Proyecto-TFG.git
```

A.2. Cómo usar

El conjunto de carpetas y ficheros que componen la versión final del proyecto se puede resumir en los siguientes:



Para iniciar el proyecto, es necesario haber realizado la calibración de las cámaras y del estéreo en primer lugar. Para ello hace falta utilizar un tablero de 10x7 cuadrados A.1. En la siguiente página se puede configurar a gusto el tablero que queramos: Generate Your Own Checkerboards.

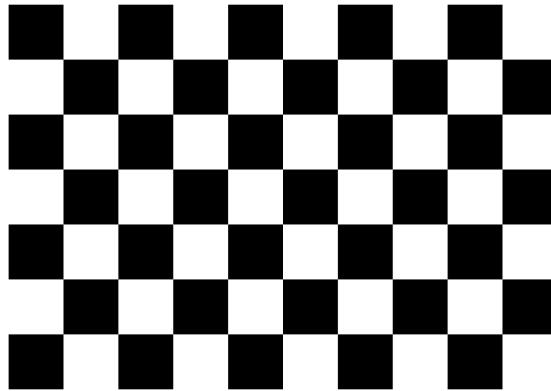


Figura A.1: Tablero 9x6 aristas / 10x7 cuadrados para calibrar las cámaras.

La calibración de las cámaras se lleva a cabo ejecutando el código *camera_calibration.py*, tras realizarla, se puede calibrar el estéreo mediante el código *stereo_calibration.py*. Tras realizar estos dos pasos, ya es posible comenzar a utilizar el proyecto.

Para iniciar la interfaz, hay que ejecutar el fichero *main.py*, el cual permite seleccionar el modo en el que se quiere utilizar la interfaz. En el modo base solo se puede controlar la representación del robot integrada en la interfaz. El modo Gazebo, que necesita que previamente se haya lanzado el launch del controlador, permite controlar tanto la representación del robot como una simulación de este en Gazebo. Por último, el modo demo permite controlar un brazo robot en tiempo real iniciando un servidor.

Si se quiere iniciar el modo Gazebo, previamente hay que lanzar el launch del controlador del robot que queremos controlar. El comando para hacerlo es el siguiente.

```
1 $ roslaunch ur_gazebo ur<modelo>_bringup.launch
```

Por otra parte, si se quiere controlar el robot real, la demo está preparada para comunicarse con un UR3. Dentro de la carpeta del proyecto se encuentra el fichero *ur3_control.urt*. Este es el programa que debe ejecutarse desde el robot real para que pueda comunicarse con la aplicación y moverse.

En caso de que no se disponga de dos cámaras, el proyecto cuenta con dos vídeos y la calibración de las cámaras y el estéreo que se utilizó para obtenerlos. Por lo que se puede ejecutar la interfaz sin necesidad de disponer de ninguna cámara ni de realizar la calibración. Para lanzar el proyecto en este modo, es necesario ejecutar el main con el argumento "-vid":

`python3 pose_stereo.py -vid.`

Por último, dentro del fichero *main.py* está definido el modelo que se va a usar del robot, así como su posición home. En caso de querer añadir un robot distinto a los UR3 y UR5e que se han utilizado para el proyecto, se pueden añadir definiendo sus parámetros DH tal y como se ha hecho para el UR3 4.12.