Universidad Zaragoza
1542

Trabajo Fin de Grado

Programación de un brazo robótico para manipular piezas de ajedrez

Programming a robotic arm to manipulate chess pieces

Autor

Gustavo Chen Fernández

Director

Tomas Cortes Arcos

Escuela Universitaria Politécnica La Almunia

Noviembre 2025

Página intencionadamente en blanco.

Escuela Universitaria
Politécnica - La Almunia
Centro adscrito
Universidad Zaragoza

ESCUELA UNIVERSITARIA POLITÉCNICA
DE LA ALMUNIA DE DOÑA GODINA (ZARAGOZA)

MEMORIA

Programación de un brazo robótico para manipular piezas de ajedrez

Programming a robotic arm to manipulate chess pieces

608.25.113

Autor: Gustavo Chen Fernández

Director: Tomas Cortes Arcos

Fecha: 11 2025

Página intencionadamente en blanco.

INDICE DE CONTENIDO BREVE

INDICE DE CONTENIDO

INDICE DE ILUSTRACIONES

INDICE DE TABLAS

1. RESUMEN

Este Trabajo de Fin de Grado (TFG) busca la programación de un brazo robótico para jugar ajedrez. Para cumplir con el objetivo principal se plantea la integración de varias tecnologías como la de visión artificial para reconocer el tablero junto a sus piezas, la implementación de un motor de ajedrez para obtener la mejor jugada, y finalmente la calibración del brazo para lograr agarrar las piezas.

La primera parte correspondiente a la visión artificial será encargada de reconocer las piezas y determinar su posición exacta en el tablero, con coordenadas de casillas. Gracias a este sistema, el robot será capaz de identificar las jugadas realizadas por el humano y actualizar en tiempo real la disposición de las piezas, proporcionando de esta manera información necesaria que contribuya a planificar los movimientos del brazo de forma precisa

El cuanto a la implementación del motor de ajedrez, este analizara el tablero y determinará la jugada más conveniente en cada turno, dicho motor devolverá una jugada en formato FEN que será puesto en el prompt, que ejecutara la acción.

Para lograr que el brazo robótico sea capaz de mover las piezas, se aplicara un estudio de cinemática inversa. Este método será capaz de calcular los ángulos de cada servomotor que den como combinación que la pinza del brazo actúe sobre una casilla en concreto, para obtener dichos ángulos se realizará una simulación en Matlab, además de añadir una posición de descanso que permita al robot no interrumpir al jugador contrario mientras se estén ejecutando órdenes o no sea su turno.

1.1. PALABRAS CLAVE

- Ajedrez
- Visión Artificial
- Brazo robótico
- Cinemática Inversa
- Programación

2. ABSTRACT

This Final Degree Project (TFG) seeks to program a robotic arm to play chess. To achieve this main objective, several technologies are integrated, such as computer vision to recognize the board and its pieces, the implementation of a chess engine to obtain the best move, and finally, the calibration of the arm to grasp the pieces.

The first part, corresponding to computer vision, will be responsible for recognizing the pieces and determining their exact position on the board, using square coordinates. Thanks to this system, the robot will be able to identify the moves made by the human and update the arrangement of the pieces in real time, thus providing the necessary information to contribute to the precise planning of arm movements.

Regarding the implementation of the chess engine, it will analyze the board and determine the most convenient move for each turn. This engine will return a move in FEN format, which will be placed in the prompt, which will execute the action.

To enable the robotic arm to move the pieces, an inverse kinematics study will be applied. This method will be able to calculate the angles of each servomotor that will allow the arm's gripper to act on a specific square. To obtain these angles, a simulation will be carried out in Matlab, in addition to adding a rest position that allows the robot to not interrupt the opposing player while orders are being executed or it is not their turn.

2.1. KEY WORDS

- Chess
- Computer Vision
- Robotic Arm
- Inverse Kinematics
- Programming

3. INTRODUCCIÓN

3.1. MOTIVACIÓN

El ajedrez es un deporte que incentiva el desarrollo cognitivo de sus usuarios, sin embargo, es difícil que los jóvenes tengan iniciativa propia por sumergirse en dicho campo, mayormente surge por presión de padres que conocen sus beneficios. Po otro lado el uso de robots tiene un impacto mucho mayor y genera un gran interés debido a su versatilidad y a la fascinación que se encuentra al observar su uso, ante ello la combinación de ambos campos me resulta un punto de partida interesante para incentivar este deporte mental dentro de la población joven.

Existen muchos casos de implementaciones de sistemas informáticos y mecánicos en el ajedrez, como lo son los tableros que trasmiten imagen a tiempo real de manera digital de la posición actual, además de numerosos motores de ajedrez capaces de predecir las jugadas con mayor probabilidad de ganar. Sin embargo, los tableros son muy costosos con una media de 400€ y los motores de ajedrez que normalmente se encuentran en páginas web tienen un precio de suscripción para su uso ilimitado. Ante los problemas principalmente económicos que puede generar tener equipo de alta calidad para mejorar uno mismo, este TFG busca programar un brazo robótico capaz de manipular las piezas de ajedrez dando la mejor jugada posible, esto se lograra en 3 partes importantes, la identificación del tablero de ajedrez con la posición de las piezas, la implementación de un motor de ajedrez y la programación del brazo robótico.

3.2. CONTEXTO Y ANTECEDENTES

3.2.1. Una herramienta inteligente para detectar piezas de ajedrez [1]

YOLO es un algoritmo usado en la detección de objetos a tiempo real con visión artificial. Lo interesante de YOLO resulta en que si tiene una buena base de datos te puede indicar correctamente la clase de objeto que detecta y su cuadro delimitador, además de las probabilidades de que la predicción sea correcta.

El funcionamiento de YOLO usa una arquitectura CNN, que también es usada comúnmente en la clasificación de imágenes, lo que hace esta arquitectura es extraer características claves de la imagen dada, como

bordes, colores, texturas, formas, brillo, y luego los compara con una base de datos [1]. El algoritmo de YOLO tiene por defecto 20 clases diferentes de objetos, pero se pueden entrenar, y posteriormente predice de acuerdo a las características claves que objeto puede llegar a ser y con qué probabilidad, como resultados del análisis de YOLO en piezas de ajedrez se logró apreciar cómo llegó a un 90% de precisión la versión full de YOLO, mientras que para el YOLO tiny fue de 89%.

En cuanto a la utilidad que tiene YOLO para nuestro proyecto, debido a que en el ajedrez se encuentran muchas piezas, en total 32 de las cuales hay 6 clases diferentes de cada color, resulta muy útil entrenar la base de datos para poder detectar de manera eficaz cada una de las piezas, además, YOLO al dar coordenadas de la ubicación del cuadro delimitador nos permitirá convertirlas en coordenadas del tablero de ajedrez

Ilustración 1- Piezas de Ajedrez

3.2.2. Análisis robusto de ajedrez mediante visión artificial e interacción con un robot humanoide[2]

En este trabajo se realizó la detección del tablero de ajedrez mediante el algoritmo de Canny, si bien es cierto que podríamos haber usado YOLO, como fue en el caso anterior para la detección de las piezas, resulta más útil usar el algoritmo de Canny dado que traza líneas que cuadran de manera exacta con nuestro tablero, y no de forma aproximada como trabaja YOLO. El algoritmo se encarga de detectar líneas rectas dentro de nuestra imagen, dado que el tablero consta de una cuadrícula de 9X9, nos resulta tanto útil para detectar el exterior del tablero, como las casillas que lo comprenden.

El funcionamiento del algoritmo de Canny es:

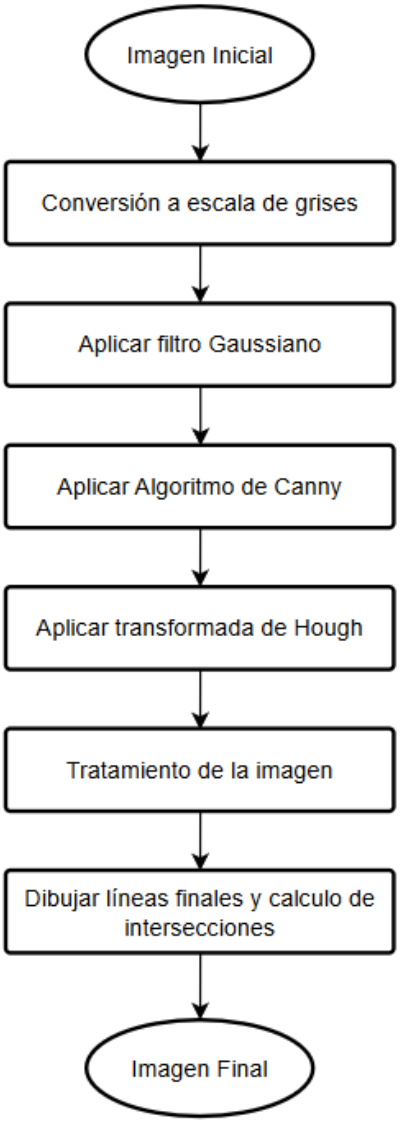


Ilustración 2-Diagrama de Flujo Canny[2]

Es importante señalar que, dependiendo de la intensidad de luz de la imagen, puede que los resultados, a pesar de usar los mismos parámetros de suavizado y filtros, la imagen no dé las mismas rectas. Además, como señalaron Tzer-Yeu y I-Kai [2], al aplicar el algoritmo de Canny sobre un tablero con piezas, se apreciaron errores en los que las líneas del tablero se combinaban con las líneas de las piezas. Por lo tanto, en este TFG lo que haremos será analizar el tablero vacío para obtener de manera más eficiente sus bordes y coordenadas de la cuadrícula interna, y luego, sin mover el tablero, colocar las piezas, dado que no es necesario realizar este paso de calcular coordenadas cada que se mueva una pieza, sino solo la inicial.

4. DESARROLLO

4.1. MARCO TEORICO

4.1.1. Algoritmo de Turing

No es de sorprender que el matemático, analista e ingeniero Alan Turing mostrara interés por el ajedrez, siendo una de las figuras más influyentes en los inicios de la automatización del juego. Aunque en su época no contó con una computadora lo suficientemente potente para ejecutar su propuesta, sentó las bases de lo que hoy conocemos como inteligencia artificial aplicada al ajedrez.

Durante las primeras décadas del siglo XX, existía un intenso debate sobre si las máquinas podrían llegar a razonar por sí mismas. Aunque ya se habían desarrollado dispositivos capaces de resolver problemas lógicos simples, el ajedrez se percibía como un desafío esencialmente "humano", debido a la enorme cantidad de estrategias, aperturas, variantes y finales posibles.

En 1947, Turing desarrolló un algoritmo al que llamó TuroChamp, diseñado para, a partir de una posición inicial como entrada (input), generar una jugada sugerida como salida (output). Este algoritmo se basaba en la asignación de valores numéricos a las piezas (peón = 1, alfil = 3, caballo = 3, torre = 5, dama = 9), junto con penalizaciones por dejar piezas vulnerables y bonificaciones por generar amenazas. De este modo, era capaz de evaluar la puntuación de una posición y optimizarla analizando posibles movimientos, de forma similar a los actuales motores de ajedrez, aunque limitado a una profundidad de tan solo dos jugadas.

Todo este sistema lo desarrolló únicamente con lápiz y papel, sin acceso a una máquina que pudiera ejecutarlo. Sin saberlo, Turing dio el primer paso hacia la creación de la inteligencia artificial.

No fue hasta el año 2012 cuando TuroChamp fue implementado y probado en una partida real por el campeón mundial Garry Kasparov. Aunque el algoritmo fue derrotado en tan solo 16 movimientos, logró jugar sin cometer errores reglamentarios, lo cual resulta extraordinario considerando que fue diseñado más de medio siglo antes, sin herramientas informáticas.

4.1.2. Notación FEN[4]

La importancia de la notación FEN radica en la facilidad de poder traducir cualquier posición de ajedrez en una línea de texto capaz de ser utilizada por programas informáticos para su representación gráfica o realizar cálculos con ella.

A diferencia de su contraparte PGN, que es una notación jugada por jugada hasta llegar a su posición final, la notación FEN únicamente traduce la posición actual, aportando información como:

- Posición de piezas.
- Color activo.
- Posibilidad de enroques.
- Posibilidad de peón al paso.
- Regla de las 50 jugadas.
- Jugadas completas.

Por ejemplo, la siguiente posición:

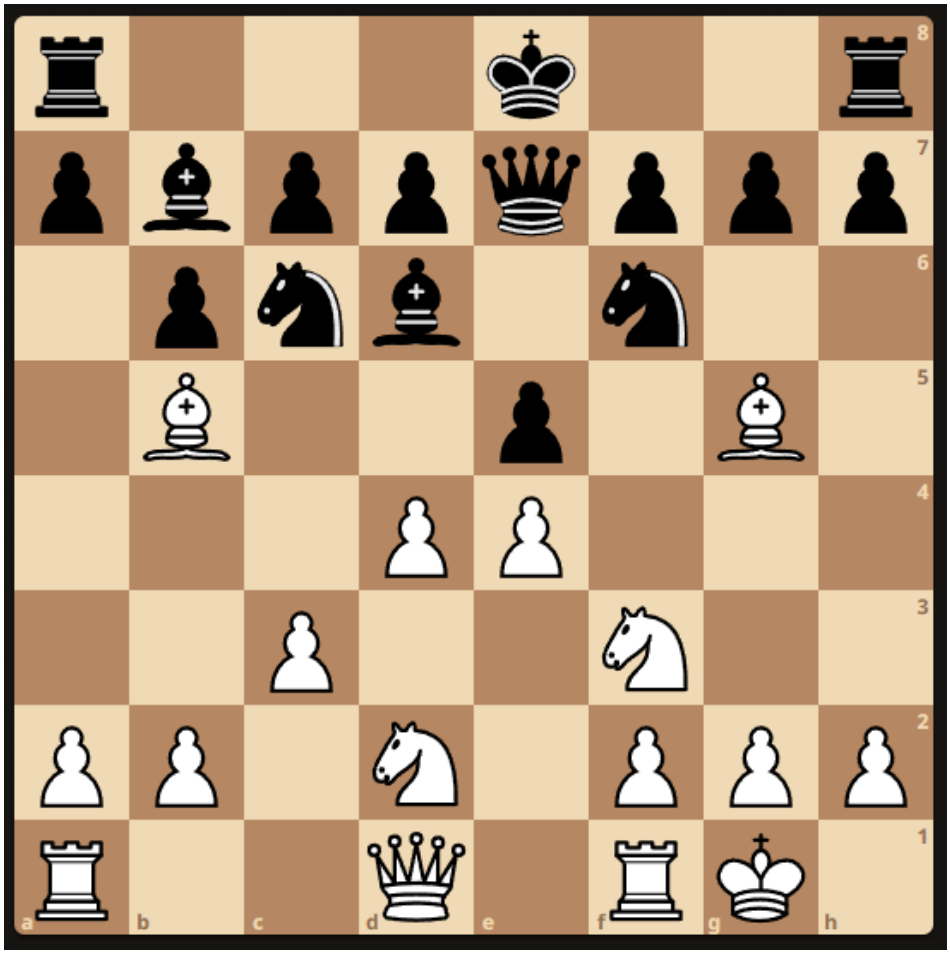


Ilustración 3 -Posición ejemplo para notación FEN

Tendría la siguiente notación FEN:

r3k2r/pbppqppp/1pnb1n2/1B2p1B1/3PP3/2P2N2/PP1N1PPP/R2Q1RK1 w kq - 0 16

La cual podemos dividir en las 6 partes mencionadas anteriormente, siendo la primera parte:

r3k2r/pbppqppp/1pnb1n2/1B2p1B1/3PP3/2P2N2/PP1N1PPP/R2Q1RK1

Esto representa el tablero fila por fila, partiendo desde la casilla a8, y cada letra representa una pieza:

- minúsculas = negras
- mayúsculas = blancas
- r = torre, n = caballo, b = alfil, q = dama, k = rey, p = peón
- Los números indican casillas vacías consecutivas

Por ejemplo, r3k2r significa: torre negra, 3 casillas vacías, rey negro, 2 casillas vacías y torre negra, correspondientes a la línea 8. De manera consiguiente, se logra formar el tablero en su plenitud.

La segunda parte del código FEN indica de quién es el turno; en este caso, al ser "w", indica que es el turno de las piezas blancas.

La tercera parte del código es "kq", que indica la posibilidad de enroque corto (k) y enroque largo (q) de las piezas negras al estar en minúsculas. Como las piezas blancas ya han realizado su enroque, no es posible que lo vuelvan a realizar, por lo que no hay mayúsculas.

La cuarta parte del código corresponde a la posibilidad de realizar el movimiento de peón al paso. Al ser representado por "-", indica que no es posible en esta jugada. En caso de ser posible, aparecerá indicada la casilla en la que terminaría el peón al realizar el movimiento, en lugar de "-".

La quinta parte del código indica el número de jugadas desde el último movimiento de peón o captura de pieza. Al llegar a 50, significa que se otorga un empate automático; en el caso actual es "0".

La última parte del código indica las jugadas completas realizadas hasta el momento, es decir, que aumenta su valor en una unidad cada vez que las negras realizan su movimiento.

4.1.3. Stockfish

Stockfish es el módulo de ajedrez más fuerte de la actualidad. Es capaz de analizar millones de jugadas por minuto y sus variantes con el fin de optar por la mejor, siguiendo el enfoque de asignación de valores de piezas, como en el caso del algoritmo de Turing, pero también añade factores posicionales respecto a la actividad de las piezas, seguridad del rey, control de casillas importantes, etc.

Lo bueno de Stockfish es que es de código abierto, por lo que su utilización en nuestro proyecto resultará crucial al tener que analizar cada posición, ya que jugaremos contra el ordenador prácticamente.

Si bien es cierto que Stockfish da la mejor jugada e incluso derrota sin problemas al campeón mundial, es posible también ajustar su dificultad para mantenernos en los objetivos del TFG y que sea algo que atraiga a nuevos jugadores.

4.1.4. Brazo robótico[5]

El brazo robótico utilizado es uno de 6 DoF que se puede encontrar fácilmente en AliExpress. Si bien es cierto que la utilización de un brazo con tantos grados de libertad nos puede dificultar los cálculos para la cinemática inversa, y lo hará, en realidad no son 6, dado que la rotación de la muñeca ni la apertura o cierre de la pinza afectan a la posición final del brazo; por lo tanto, únicamente contaremos con 4 grados de libertad.

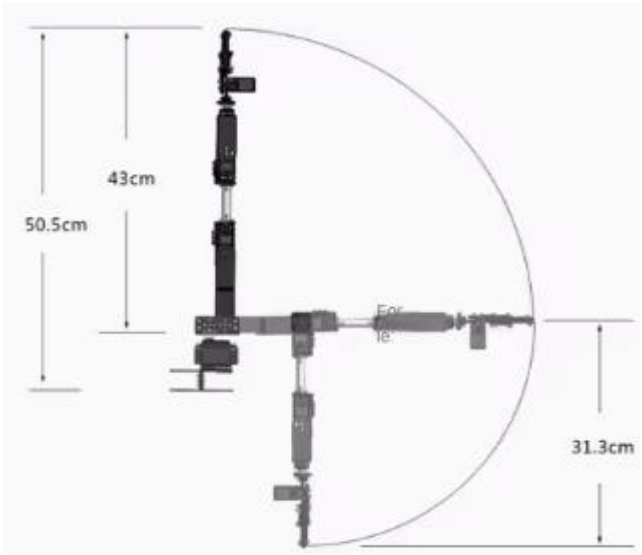


Ilustración 5- Dimensiones maximas

Ilustración 4-Brazo robotico 6 DoF

Se ha elegido este modelo frente a otros debido a su abundancia y, además, su precio no es tan alto como en otros modelos.

4.1.5. Servomotores 180°[6]

En cuanto a los servomotores utilizados, se usarán servomotores de 180°, más concretamente los MG996R, recomendados por el diseñador del brazo.

Ilustración 6- Servomotor MG996R

Sus parámetros característicos son:

- Dimensión: 40 mm x 19 mm x 43 mm
- Peso neto: 65 g
- Color negro
- Velocidad de funcionamiento: 0,17 segundos / 60 grados (4,8 V sin carga)
- Velocidad de funcionamiento: 0,13 segundos / 60 grados (6,0 V sin carga)
- Par de puestos: 13 kg-cm (180,5 oz-in) a 4,8 V
- Par de puesto: 15 kg-cm (208,3 oz-in) a 6V
- Voltaje de funcionamiento: 4,8 - 7,2 voltios

Con estos valores de par, son más que suficientes para soportar el peso propio de la estructura además del de la pieza.

4.1.6. Arduino Mega 2560[7]

Arduino Mega 2560 es una placa de desarrollo con base en el microcontrolador ATmega2560, una de las ventajas de un Arduino Mega frente a otros, es su mayor capacidad de pines digitales y analógicos,

los cuales nos servirán a la hora de conectar los servos simultáneamente.

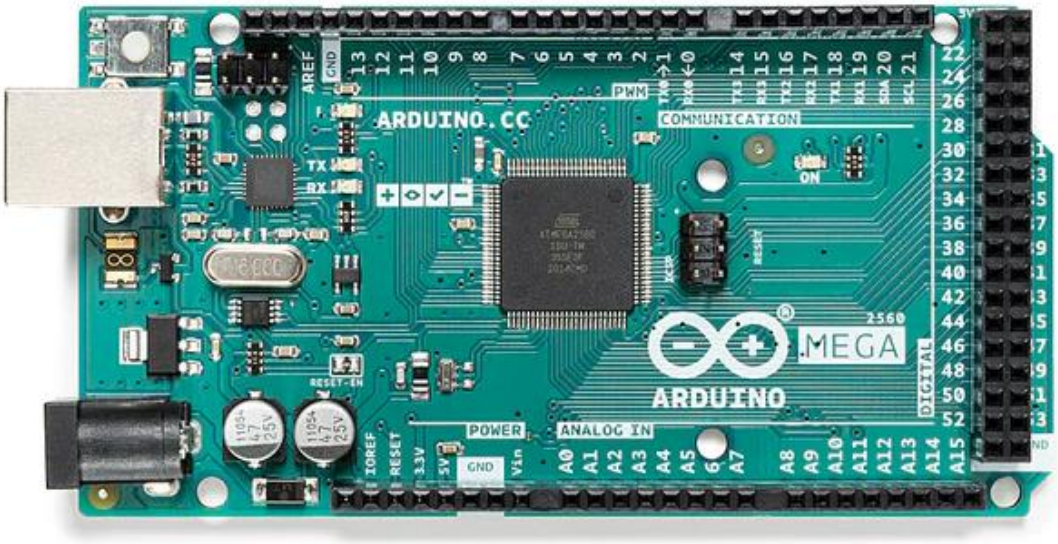


Ilustración 7- Arduino Mega 2560

Sus características son:

- Microcontrolador: ATmega2560
- Voltaje de operación: 5V
- Entradas analógicas: 16
- Pines digitales de entrada/salida: 54 (15 con salida PWM)
- Memoria SRAM: 8 KB
- Memoria EEPROM: 4 KB
- Frecuencia de reloj: 16 MHz
- Comunicación serial: 4 puertos UART, además de I2C y SPI

4.1.7. Cámara Aukey [8]

Dado que se necesita una cámara para captar la posición del tablero y las piezas, se optó por una webcam con las siguientes especificaciones:

- Modelo: PC-LM1E
- Definición: 1080p FullHD
- Cámara: 2 megapíxeles
- Sensor de imagen: CMOS de 1 / 2,9"
- Micrófonos estéreo: integrados

Ilustración 8-Cámara Aukey

Al ser una webcam conectada a la computadora, resulta más sencillo que tener que usar un celular y tener que pasar las fotos con cierta extensión, ya que OpenCV solo acepta JPEG, JPG y PNG. Además, se evita la necesidad de usar programas externos que puedan modificar las dimensiones de la foto al transferirla.

4.1.8. Python [9]

Python es un lenguaje de programación de alto nivel, interpretado, orientado a objetos y de propósito general, ampliamente utilizado en áreas como el desarrollo de software, automatización, análisis de datos, inteligencia artificial y control de hardware. Fue creado por Guido van Rossum y lanzado por primera vez en 1991. Su sintaxis clara y legible lo convierte en una opción ideal tanto para principiantes como para profesionales. Entre sus características más destacadas se encuentran:

- Lenguaje multiplataforma, disponible en Windows, Linux y macOS.
- Sintaxis clara, orientada a objetos y funcional.
- Amplia comunidad de desarrolladores y abundante documentación.
- Gran número de librerías útiles
- Capacidad de integración con otros lenguajes y herramientas de hardware, como Arduino a través de la librería pySerial.

4.1.9. OpenCV

OpenCV es una biblioteca de código abierto usada para el procesamiento de imágenes y visión artificial por computadora. Permite que el ordenador sea capaz de detectar imágenes y el entorno que las compone, sacando parámetros característicos clave, como número de objetos en la imagen, formas, colores, esquinas, caras, etc.

Es usada en diversos campos actualmente, como en el caso de la robótica para la navegación y reconocimiento de entornos. En el campo de la seguridad, es utilizada para detección facial y sistemas de videovigilancia, puesto que también permite el seguimiento de objetos. En la medicina, es útil para el diagnóstico con imágenes; a partir de una base de datos, es capaz de analizar qué tipo de enfermedad se tiene. Además, en automatización industrial, es capaz de detectar problemas de calidad y controlar procesos.

Ilustración 9 - Logo OpenCV

4.1.10. C++

C++ es un lenguaje de programación, también, al igual que Python, orientado a objetos. Permite a los desarrolladores crear aplicaciones organizadas y reutilizables mediante el uso de clases y objetos. Su capacidad para gestionar recursos de manera eficiente lo hace ideal para aplicaciones que requieren un alto rendimiento, como sistemas operativos, controladores de hardware y software de tiempo real.

Además, C++ incluye una biblioteca estándar (STL - Standard Template Library) que proporciona estructuras de datos y algoritmos

que facilitan el desarrollo de software. Su naturaleza multiplataforma permite que las aplicaciones se ejecuten en diferentes sistemas operativos, como Windows, macOS y Linux.

4.1.11. Label Studio

Label Studio es una página que permite el etiquetado de diferentes tipos de datos como lo son las imágenes, videos, audios, textos, etc., es una herramienta usada principalmente en proyectos de inteligencia artificial, machine learning y visión por ordenador.

El uso de label studio es muy sencillo:

- Primero debemos Crear etiquetas con todas las clases que existan
- Dibujar los bounding boxes
- Exportar el dataset y usarlo en el modelo para entrenar YOLO

Ilustración 10-Label Studio

4.2. DESARROLLO

En este apartado se realizarán todos los aspectos que lleven, finalmente, al movimiento del brazo robótico para jugar ajedrez. Ante ello, se ha planteado dividir este apartado en tres secciones: la primera consta del reconocimiento del tablero y las piezas; esto incluye la posición en coordenadas y su representación en una interfaz de ajedrez. La segunda parte consiste en el cálculo de la cinemática inversa para el movimiento del brazo; esta parte se realiza únicamente mediante código. Finalmente, la tercera parte consiste en la implementación de ambos sistemas físicamente, es decir, lo relacionado con la calibración y el cableado del brazo.

4.2.1. Reconocimiento mediante visión artificial

En este TFG estaremos usando una cámara colocada lateralmente. Esta decisión presenta pros y contras. Por un lado, entre las ventajas, es mucho más fácil el reconocimiento de las piezas, lo cual, personalmente, considero la parte más importante para el correcto desarrollo de este proyecto. Sin embargo, al ser una vista lateral, reconocer el tablero con sus casillas y coordenadas correctamente se dificulta mucho, ya que las piezas interfieren con las líneas interiores y los bordes.

4.2.1.1. Reconocimiento del Tablero

Una de las características principales de un tablero de ajedrez es la cantidad de líneas rectas que este posee, siendo de 9 filas y 9 columnas para la formación de un tablero 8x8.

Para lograr la correcta identificación del tablero, es necesario seguir el diagrama de flujo del apartado 3.2.2.

4.2.1.1.1. Captura de la imagen inicial

Partiremos de una imagen tomada desde una posición estándar de la que estaría nuestra cámara en todo momento.

Dado que tomaremos una foto con cada jugada, considero más apto dividir la orden de tomar fotos en un script separado que pueda ser usado tanto al capturar la foto del tablero vacío como la foto en cada movimiento. Ante ello, se implementó el siguiente script:

```
import cv2
import argparse
```

```
# Argumentos
parser = argparse.ArgumentParser(description="Captura una imagen desde la
webcam y la guarda con el nombre especificado")
parser.add_argument("--output", required=True, help="Nombre del archivo de
salida es importante que tenga terminacion para asignar ")
args = parser.parse_args()

# Inicializar la cámara
cap = cv2.VideoCapture(0) #Abrir la vision de la camara ubicada en el puerto
0

if not cap.isOpened():
    print("No se pudo abrir la cámara.") #En caso que la camara este en uso
de otra aplicacion no se abrira, por lo que saldra este mensaje
    exit()

# Capturar una imagen
ret, frame = cap.read() #Toma captura del frame actual de la camara

if ret:
    cv2.imwrite(args.output, frame)# para guardar el nombre de la imgane
    print(f"Imagen guardada como {args.output}")
else:
    print("No se pudo capturar imagenes")

# Para quitar la ventana de la imagen del frame de la camara
cap.release()
cv2.destroyAllWindows()
```

Para ejecutar correctamente el código, usaremos el prompt de Windows en lugar del de Visual Studio Code, que también podríamos usar. Sin embargo, como utilizaremos diversos lenguajes de programación en distintos compiladores, se me facilita más usar el prompt de Windows.

Usamos el siguiente comando en el prompt:

python tomar_foto.py --output tablero_vacio.jpg

Lo cual nos guardará la siguiente imagen como “tablero_vacio” en formato JPG.

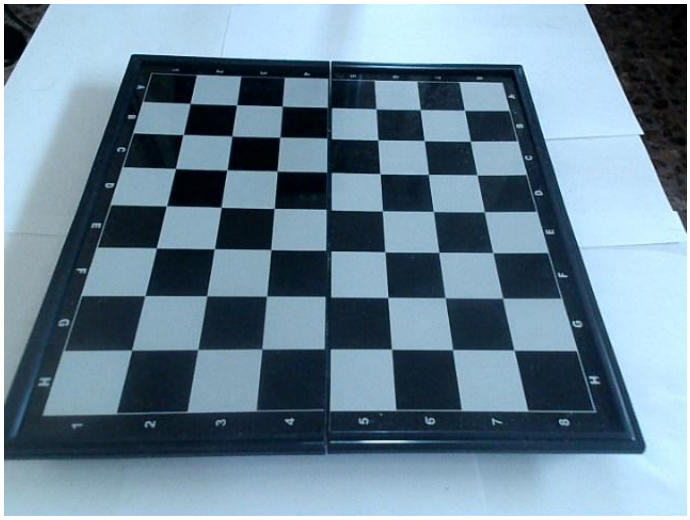


Ilustración 11-Imagen Inicial del tablero vacío

Como se puede ver, se decidió colocar un fondo blanco detrás del tablero para evitar interferencias con las líneas del entorno que puedan dificultar la aplicación del filtro de Hough y el uso del método de Canny.

4.2.1.1.2. Conversión a escala de grises

Teniendo ya la imagen inicial, procedemos a tratarla, primero realizando la conversión a escala de grises con el siguiente código:

```
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

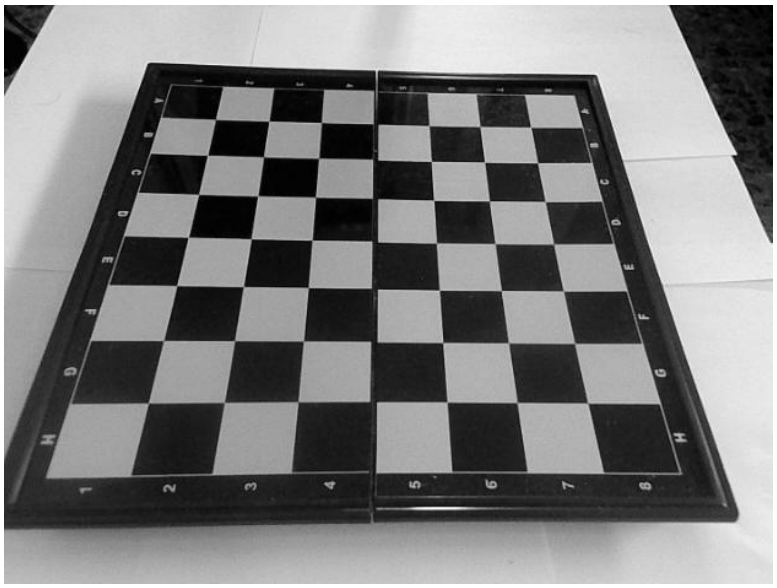


Ilustración 12-Imagen tablero escala de grises

4.2.1.1.3. Suavisado Gausiano

Es necesario aplicar el suavizado gaussiano para reducir el ruido que pueda existir por algunas líneas pequeñas y poder separar las líneas fuertes de las líneas débiles.

```
blurred = cv2.GaussianBlur(gray, (5, 5), 0)
```

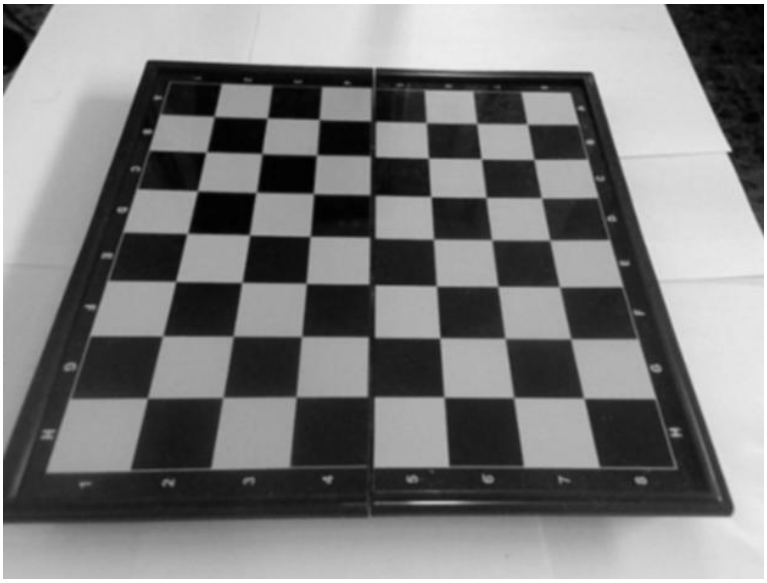


Ilustración 13-Imagen tablero suavizado

4.2.1.1.4. Aplicar Canny

Con el algoritmo de Canny se podrán separar las líneas fuertes de las débiles.

```
edges = cv2.Canny(blurred, 100, 200)
```

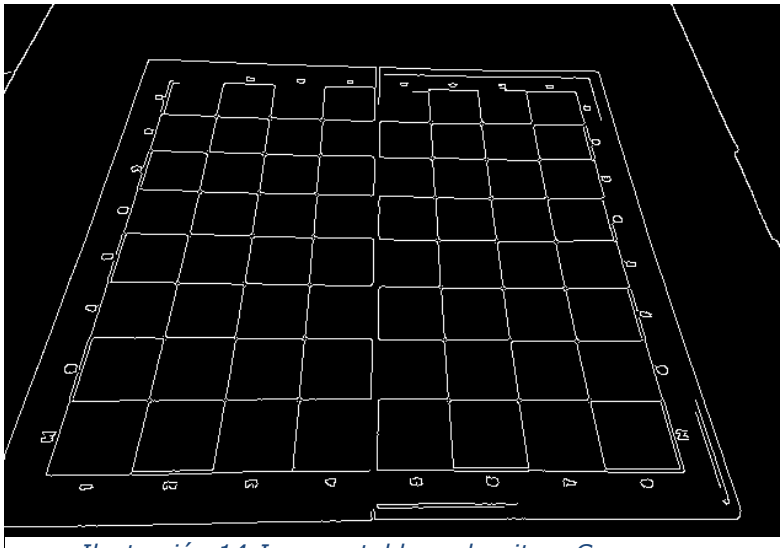


Ilustración 14-Imagen tablero algoritmo Canny

El filtro de Canny se utiliza para detectar bordes en una imagen, identificando cambios bruscos de intensidad. Como previamente se ha convertido la imagen original a escala de grises, resulta más sencillo detectar estos cambios, ya que se eliminan las variaciones de color y solo se trabaja con intensidades de luz. Por ejemplo, un paso repentino de blanco a negro indica la presencia de un borde definido.

El algoritmo analiza cada zona de la imagen y calcula un valor denominado gradiente, que representa la magnitud del cambio de intensidad. Este valor puede interpretarse como la "fuerza del borde": a mayor diferencia entre los tonos (por ejemplo, de blanco a negro), mayor será el valor del gradiente. Los valores de gradiente se miden en un rango de 0 a 255, donde los valores altos indican bordes más marcados.

Respecto a los parámetros 100 y 200, estos corresponden al ciclo de histéresis del filtro Canny. Dichos umbrales son claves a la hora de decidir qué tipo de píxeles deben ser dejados en la imagen final:

- Los píxeles con gradientes superiores a 200 se consideran bordes fuertes y se conservan.
- Los píxeles con gradientes entre 100 y 200 se consideran bordes débiles y solo se conservan si están conectados a un borde fuerte.
- Los píxeles con gradientes por debajo de 100 se descartan.

Como se puede apreciar en la imagen en escala de grises, tranquilamente se podrían haber tomado como bordes los causados por la sombra misma del tablero; sin embargo, al ser el paso entre gris y blanco "débil", no fueron conservados dichos píxeles.

En cuanto a los errores de la imagen, es interesante señalar que, en la última fila, al ser el borde del tablero negro y estar aplicado el filtro gaussiano, se pierde la continuidad de la línea. Lo mismo ocurre en el centro: al ser un tablero plegable, se distorsiona su forma.

Todos estos fallos en las líneas serán corregidos posteriormente mediante un tratamiento manual.

4.2.1.1.5. Aplicar transformada de Hough

Para convertir la imagen actual en un tablero es necesario trazar las líneas, esto se realizará con la transformada de Hough

```
raw_lines = cv2.HoughLines(edges, 1, np.pi / 180, 120) #Ajustar para
modificar la sensiblidad del Hough
```

Es importante señalar que los parámetros π/180 y 120 son de sensibilidad y permiten seleccionar un tipo de línea u otro dependiendo de su ángulo de inclinación o de la fuerza que puedan tener. Es cuestión

de ir calibrando los parámetros para que, al menos, todas las líneas que nos interesen estén contempladas en la imagen.

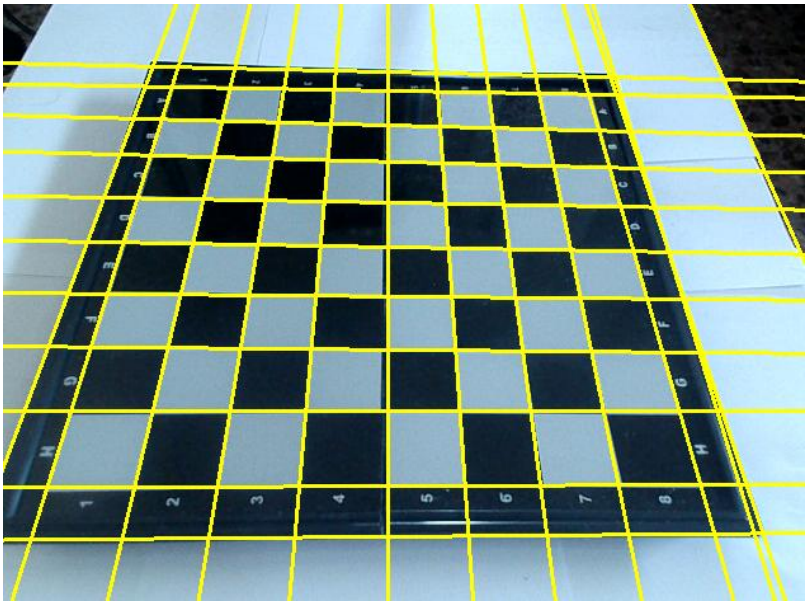


Ilustración 15-Tablero con líneas trazadas con la transformada de Hough

Como se puede apreciar, el filtro ayuda a identificar correctamente todas las líneas. Ahora debemos eliminar las líneas que nos sobran para quedarnos únicamente con un tablero 8x8, es decir, que debemos contar con 9 filas y 9 columnas de líneas.

4.2.1.1.6. Tratamiento de la Imagen

La eliminación de las líneas será un procedimiento manual. Para ello, debemos realizar una ordenación, asignar un índice a cada línea y eliminar las indeseadas.

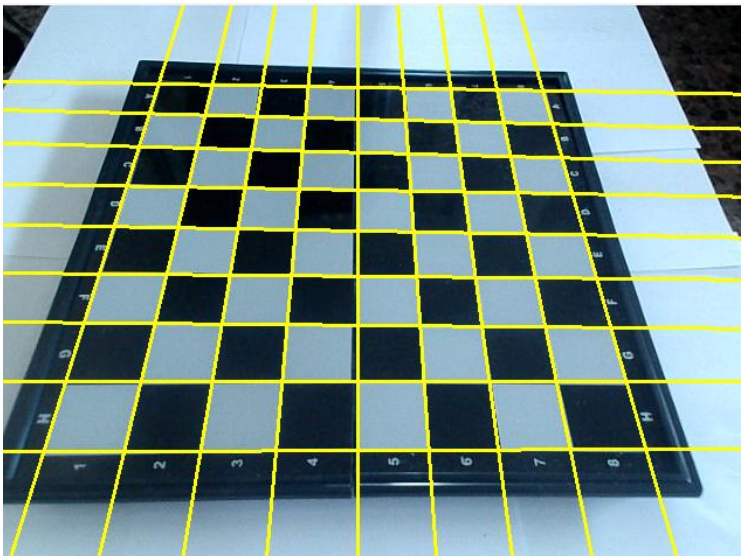


Ilustración 16-Tablero con eliminación manual de líneas

4.2.1.1.7. Dibujar rectas finales y cálculo de intersecciones

Dado que nos interesa dividir el tablero casillas para posteriormente calcular la ubicación de cada pieza, debemos realizar un cálculo de dichas intersecciones y las exportamos como un fichero json para facilitar su lectura.

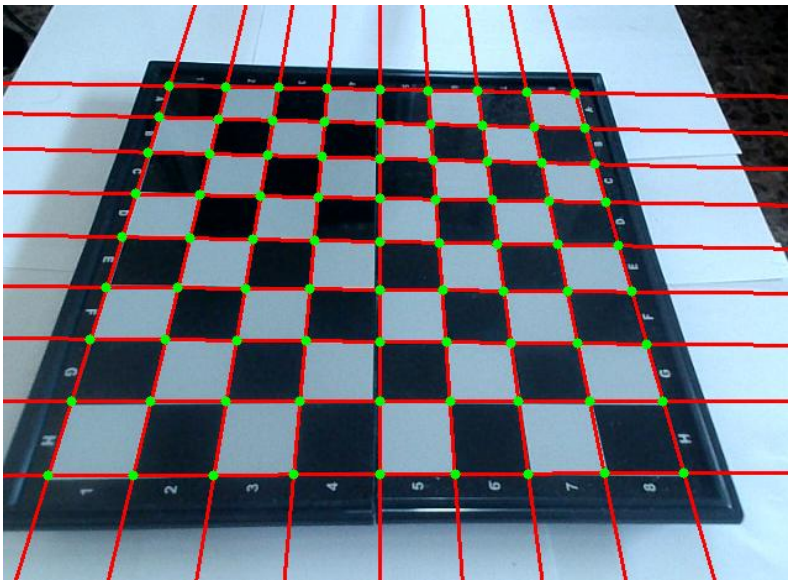


Ilustración 17-Tablero con rectas finales e intersecciones

4.2.1.2. Reconocimiento de las piezas

Teniendo ya solucionado, por el momento, el problema de la identificación del tablero con todas sus lineas e intersecciones procederemos a la segunda parte del TFG, el reconocimiento de las piezas, ante ello nos apoyaremos la inteligencia artificial, específicamente del algoritmo en YOLO, para ello tendremos que entrenar la base de datos por nuestra cuenta, dado que al contar con un conjunto de piezas de determinado modelo, resulta poco eficiente usar modelos de internet ya que las piezas difieren entre si y podrían ocasionar malas lecturas.

4.2.1.2.1. Tomar fotos

Para entrenar correctamente el algoritmo necesitamos una buena base de datos, en nuestro caso dicha base se comprenderá de una carpeta con 183 imágenes de diferentes posiciones de ajedrez tomadas de diferentes ángulos, si bien es cierto la cámara será estática, de igual forma resulta útil entrenar el algoritmo con las piezas vistas de diferente forma.

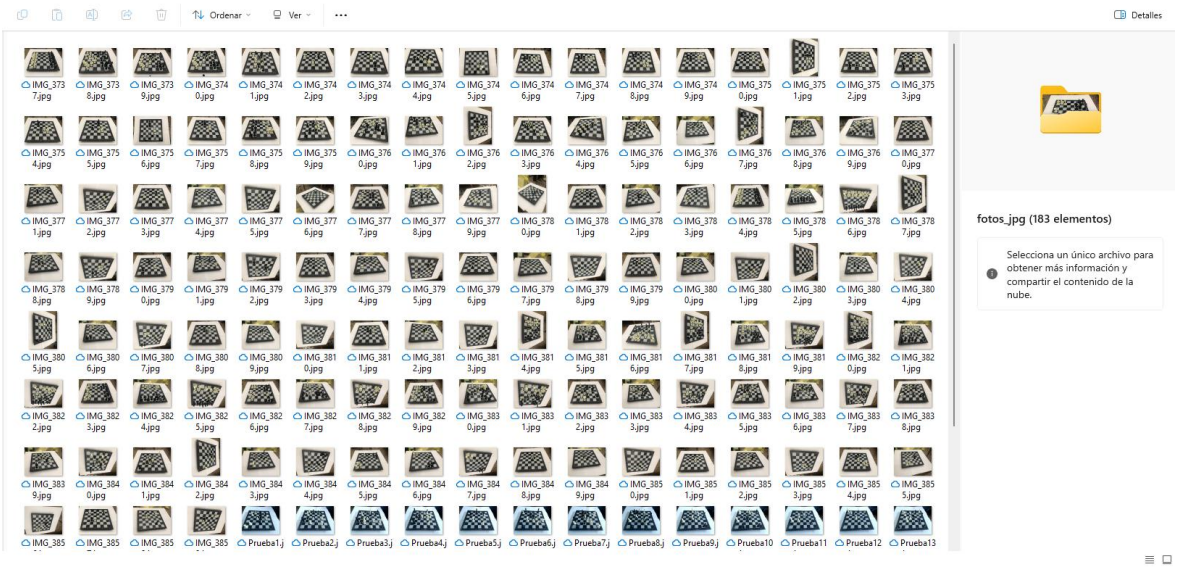


Ilustración 18-Carpeta con Imágenes de posiciones de tablero

4.2.1.2.2. Label Studio

Para que el algoritmo sea capaz de diferenciar tipos de piezas deberemos etiquetar cada imagen con el conjunto de piezas que la componen, para facilitar el tratamiento se usará Label studio que nos permitirá exportar las imágenes con sus bounding boxes en formato YOLO o json, según nos convenga.

4.2.1.2.2.1. *Importar imágenes*

Al abrir label studio nos aparecerá la opción de crear nuevo proyecto, le damos click a la ventana de "Data Import" y posteriormente cargaremos todas las imágenes de nuestra carpeta y le damos a guardar

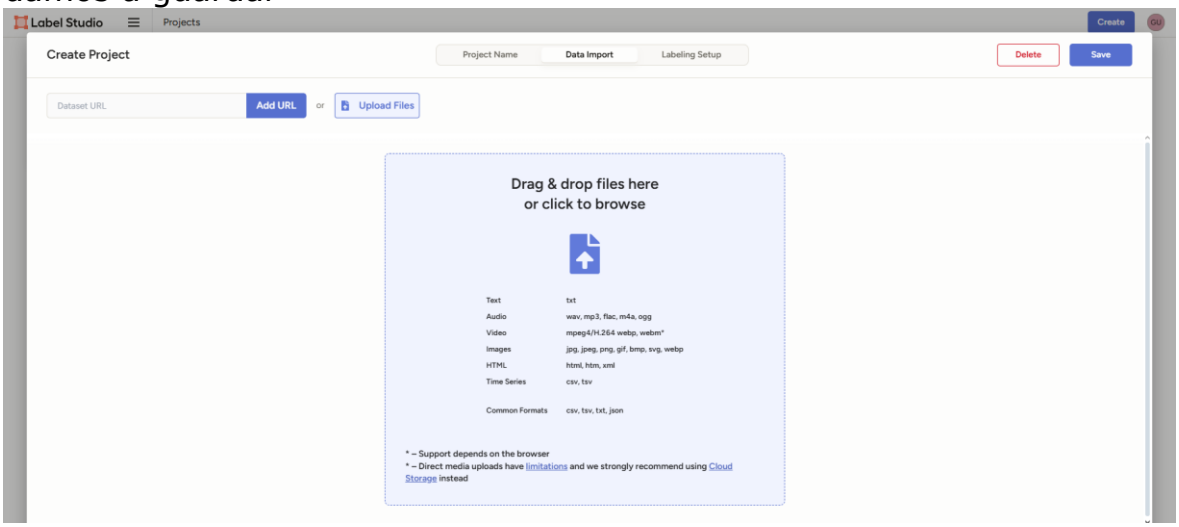


Ilustración 19- Importar imágenes

4.2.1.2.2.2. *Ajustes*

Para ir a los ajustes y configurar el tipo de bounding boxes que usaremos así como las etiquetas hacemos click en los tres puntos y "Settings"

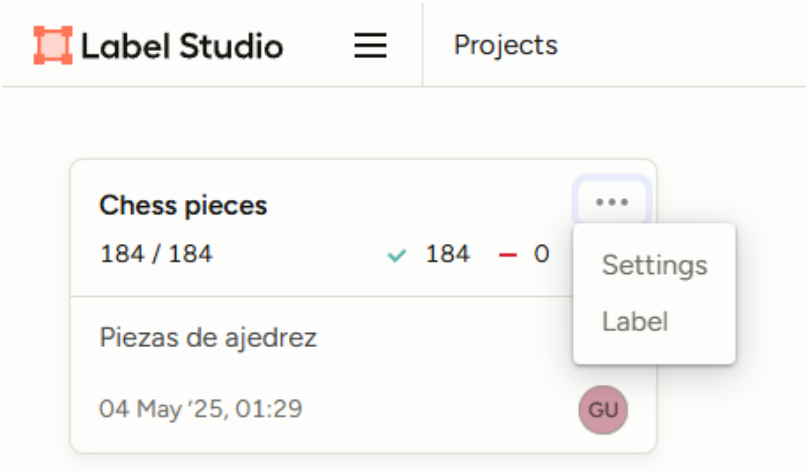


Ilustración 20 - Ajustes Label Studio

Luego vamos al apartado de "Labeling interface" y seleccionamos "Object detection with Bounding Boxes".

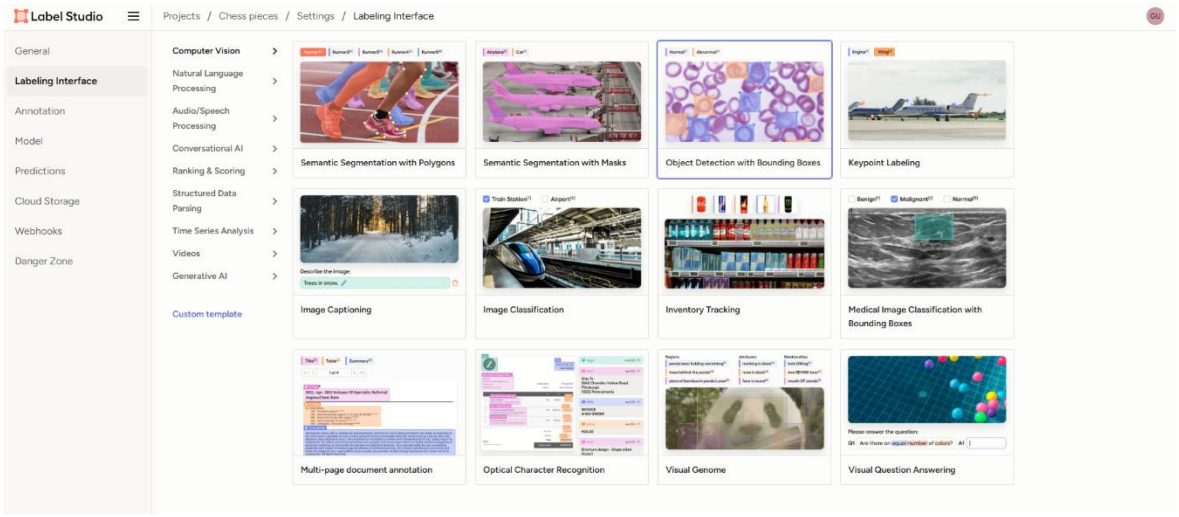


Ilustración 21 - Seleccionar la interfase

Luego, nombraremos cada una de las etiquetas que vamos a usar, en total 12, 6 siendo el tipo de tipo de pieza y el doble por el color, quedando:

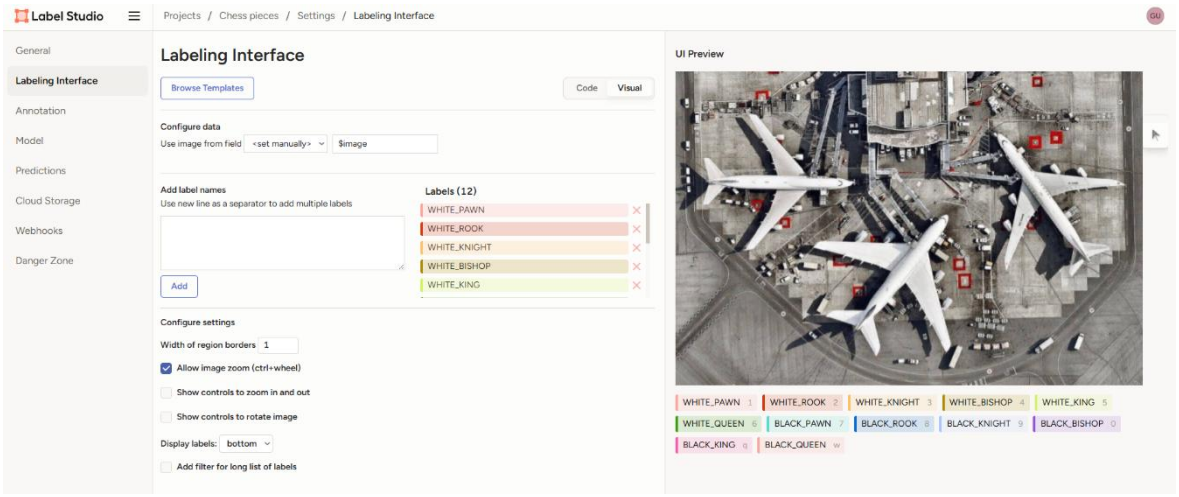


Ilustración 22 - Nombrar etiquetas

Finalmente guardamos y procederemos a crear los Bounding boxes

4.2.1.2.2.3. *Etiquetado de imágenes*

Al entrar a nuestro proyecto nos saldrá una lista de todas nuestras imágenes, hay que seleccionar una por una e ir creando los polígonos con las etiquetas según corresponda, mientras mayor sea la base de

datos habrá más certeza en nuestro modelo, cada una de las imágenes nos debería quedar de la siguiente forma:

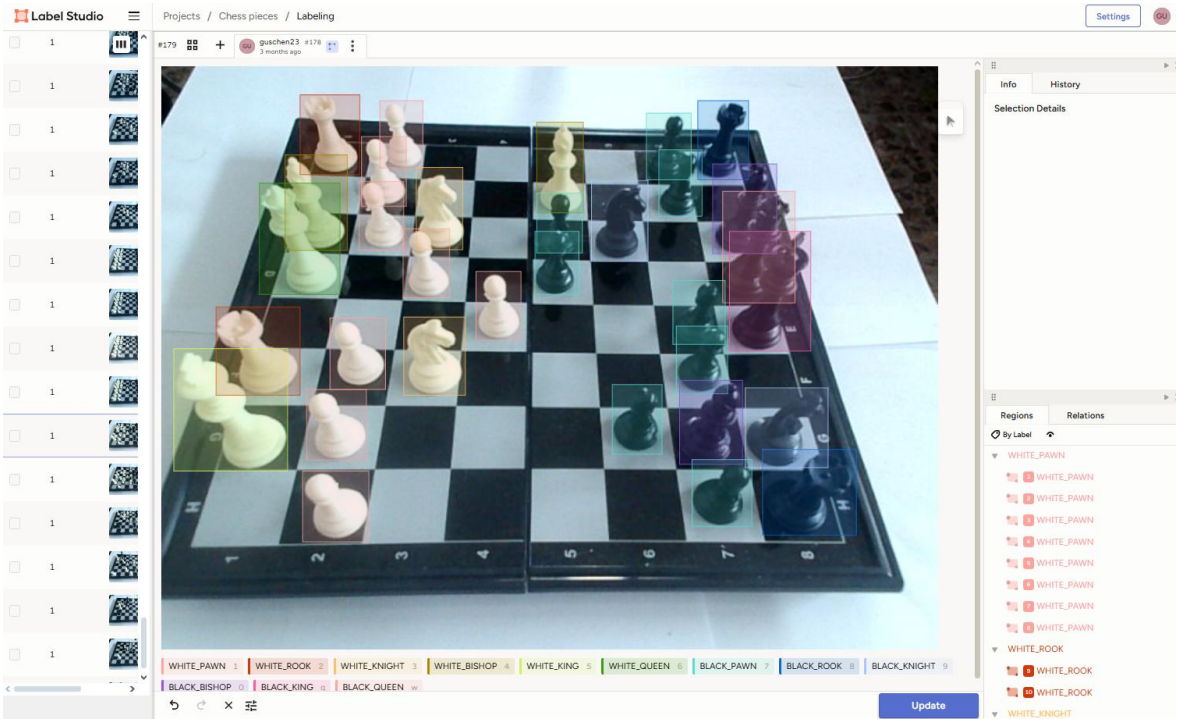


Ilustración 23 - Tablero con piezas etiquetadas

4.2.1.2.2.4. *Exportar Data*

Teniendo ya todas las imágenes etiquetadas con sus bounding boxes procederemos a exportar los datos, para ello nos vamos a la lista de imágenes y en la parte superior izquierda nos saldrá la opción de exportar

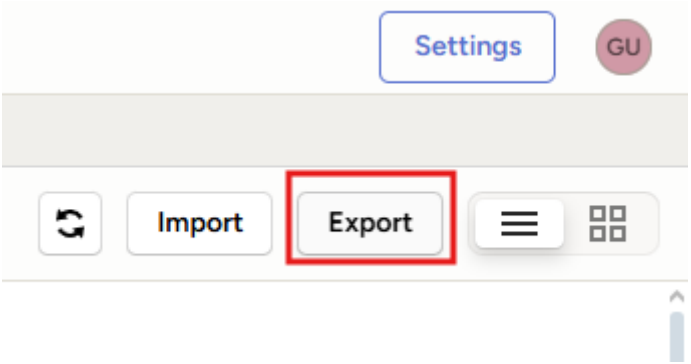


Ilustración 24-Exportar datos

Dentro seleccionaremos la opción de "YOLO with images" y le damos a exportar

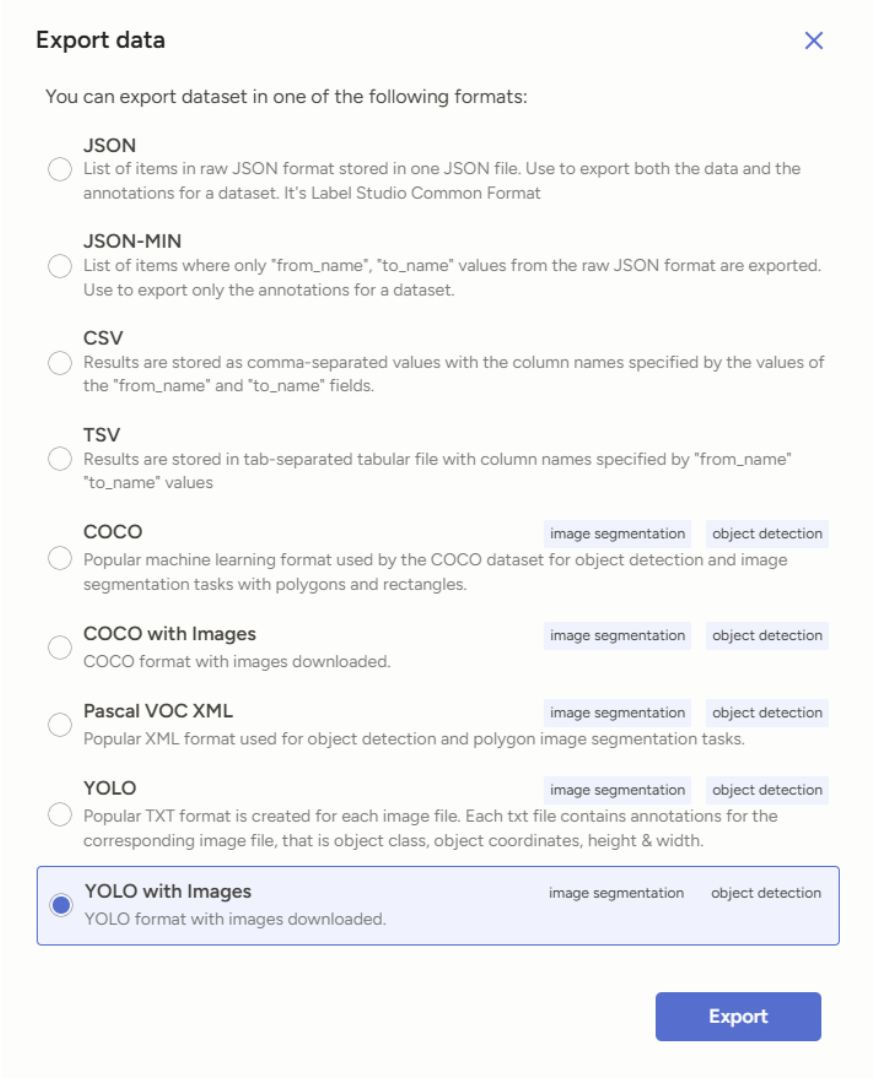


Ilustración 26 - Seleccionar formato de exportación

Al exportar se nos creará un archivo el cual estará compuesto de lo siguiente:

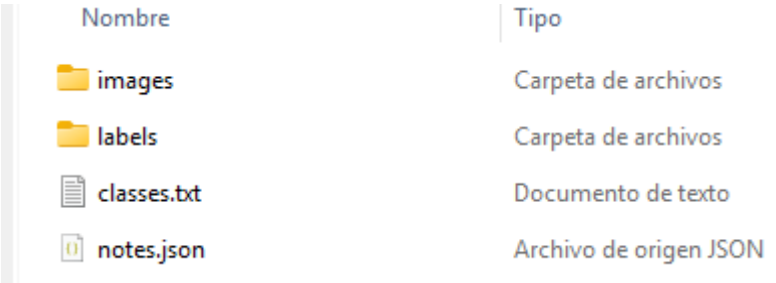


Ilustración 25 - Carpeta exportada

- Carpeta con imágenes: Esta carpeta será de las imágenes "crudas" es decir sin cuadros ni etiquetas

- Carpeta de etiquetas: Dentro habran ficheros en formato .txt para cada una de las imágenes, los ficheros contendran 5 columnas y tantas filas como bounding boxes hubieran en la imagen correspondiente

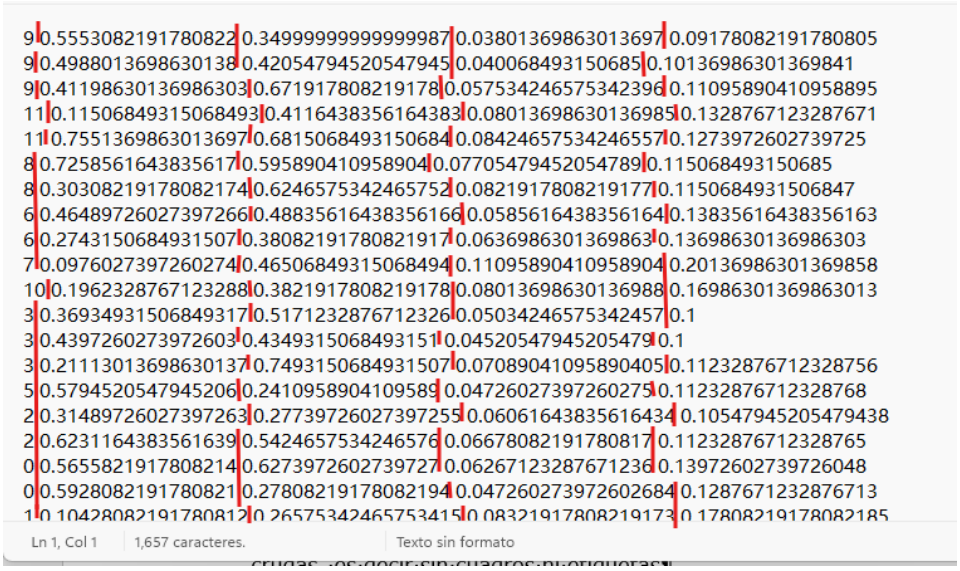


Ilustración 27 - Fichero en formato YOLO

  - La primera columna corresponde al la identificacion de clases
  - La segunda columna hace referencia a la coordenada X del centro del bounding box relativo de la imagen total
  - La tercera columna hace referencia a la coordenada Y del centro del bounding box relativo de la imagen total
  - La cuarta columna hace referencia al ancho relativo del bounding box
  - La quinta columna hace referencia a la altura relativa del bounding box

- Un archivo de las clases: Está en formato .txt y tiene unicamente el nombre de las etiquetas creadas
- Un archivo de notas: Está en formato .json es importante echar un ojo a este archivo ya que muestra todos los índices que han sido correspondidos con cada una de las clases

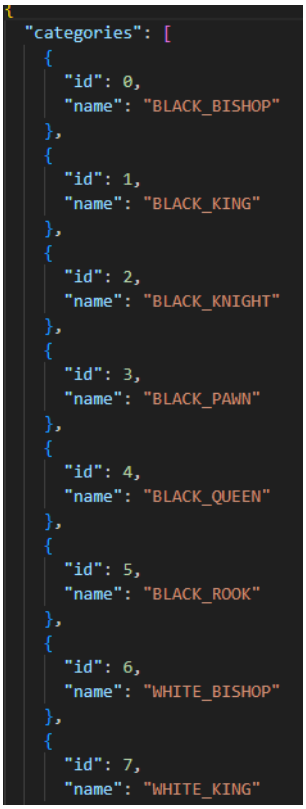


Ilustración 28 - Archivo .json con class ID

4.2.1.2.3. Modelo YOLO[9]

Para entrenar el modelo de entrenamiento de Yolo desarrollado por Evan Juras [9], dicho código es de libre acceso mediante la plataforma Google Colab, que nos permitirá usar Python en una GPU gratis de Google, cabe señalar que el acceso gratuito solo tiene un tiempo de 5 horas antes que se borre toda nuestra data, por lo que para archivos muy grandes no es recomendable su uso

4.2.1.2.3.1. *Paso 1: Entorno de Ejecución*

Al entrar al link de Google Colab proporcionado primero debemos ir a "Entorno de ejecución" y "Cambiar tipo de entorno de ejecución"

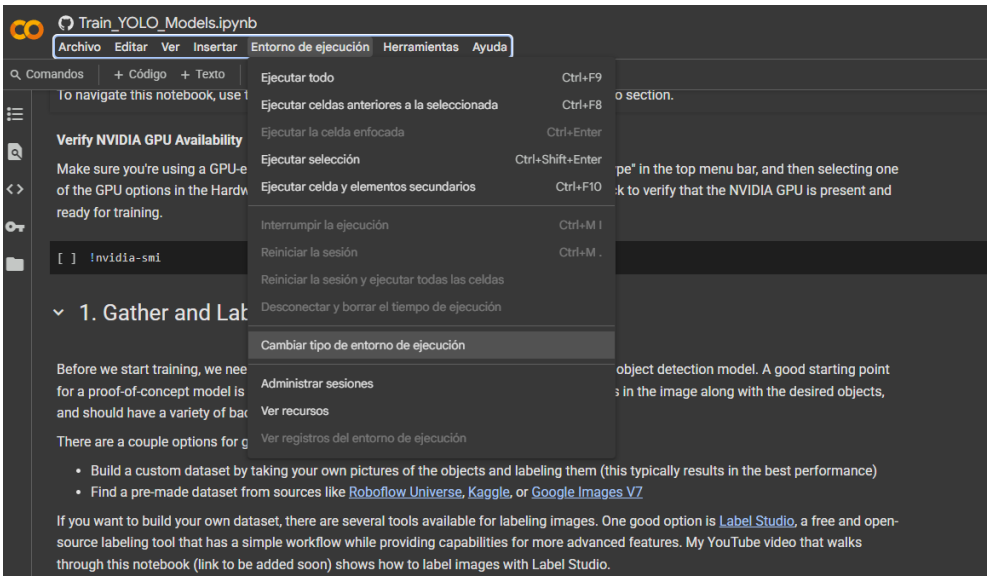


Ilustración 29 - Entorno de ejecución

Dentro nos aseguramos de seleccionar "GPU T4"

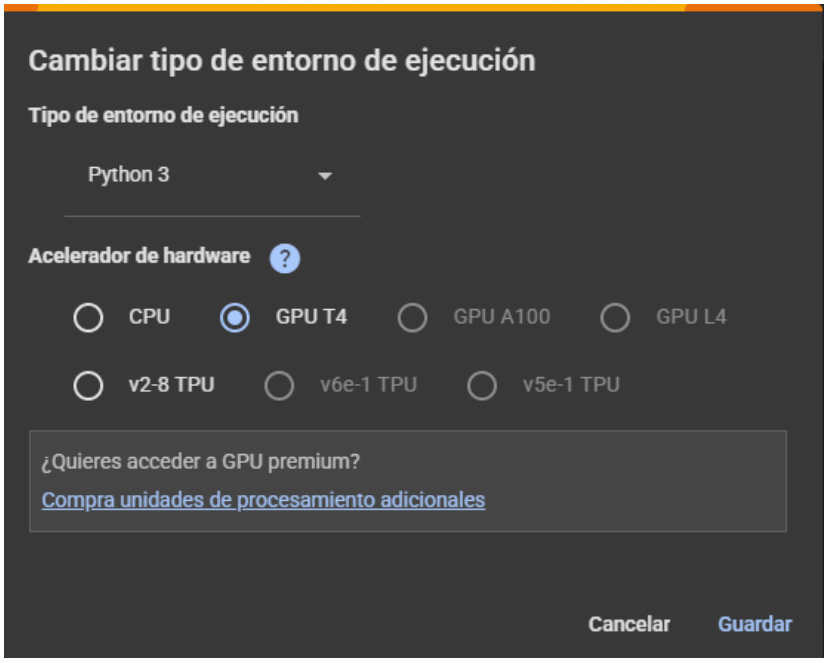


Ilustración 30 - Seleccionar hardware

Luego le damos click a "conectar" y ejecutamos el comando para asegurarnos que la GPU de nvidia está lista para entrenar

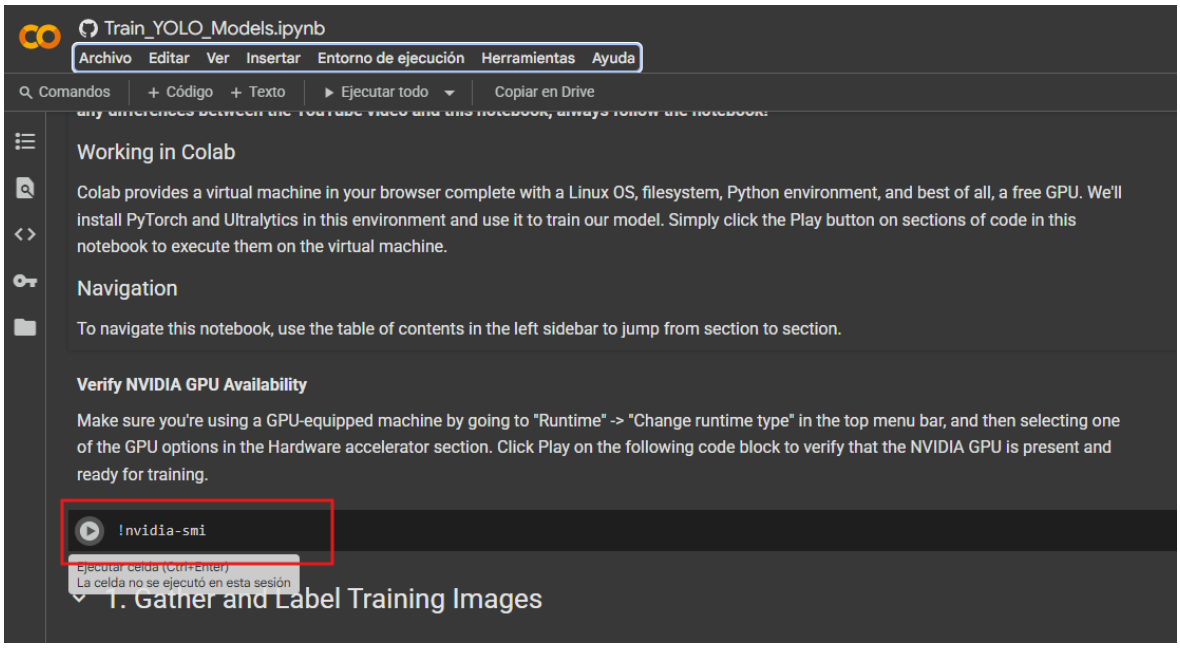


Ilustración 31 - Correr GPU

4.2.1.2.3.2. *Paso 2: Importar data*

Luego debemos importar los datos exportados desde Label Studio, es importante que sea la carpeta entera con las 2 carpetas y 2 ficheros en su interior, es importante renombrar la carpeta como "data.zip" ya que el código del script de Google Colab usa ese nombre de carpeta para llamarlo

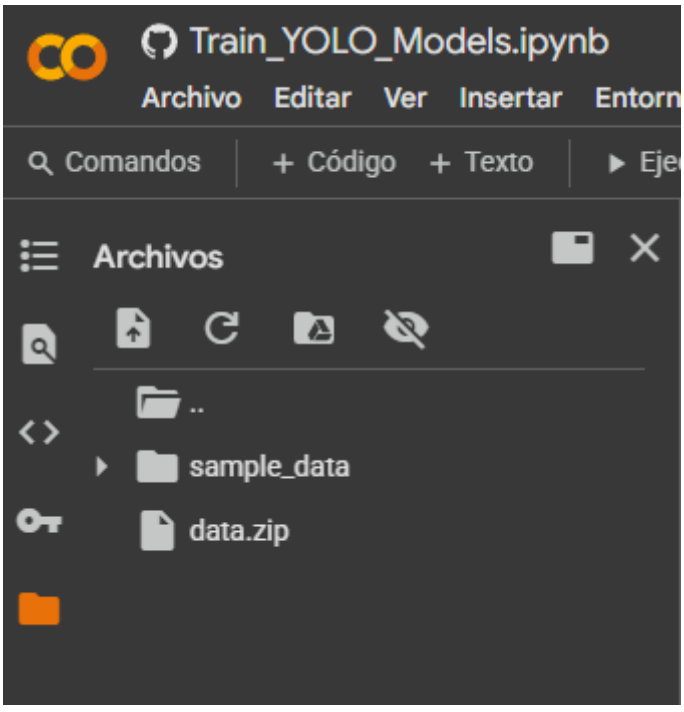


Ilustración 32 - Importar datos Google Colab

4.2.1.2.3.3. *Paso 3: Ejecutar comandos*

Teniendo ya importados los datos en formato YOLO, lo que debemos hacer es seguir el script ejecutando los bloques de comando.

El primer comando que nos encontraremos estará en el apartado 2.2, este será el encargado de descomprimir la carpeta de datos

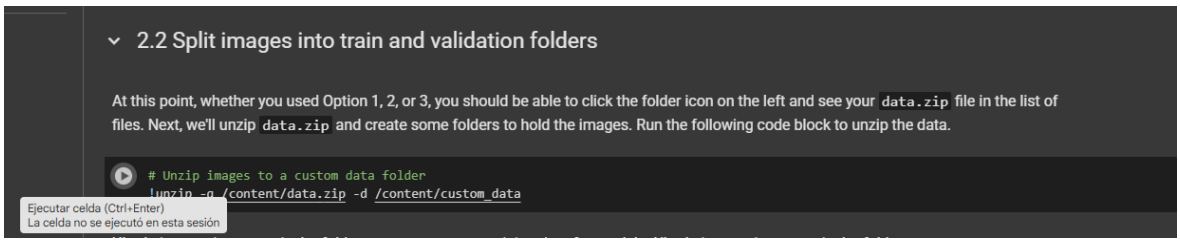


Ilustración 33 - Descomprimir folder de datos

Más abajo en el mismo punto tendremos el comando encargado de dividir los datos entre entrenamiento y validación, siendo de entrenamiento el 90% y validación un 10%

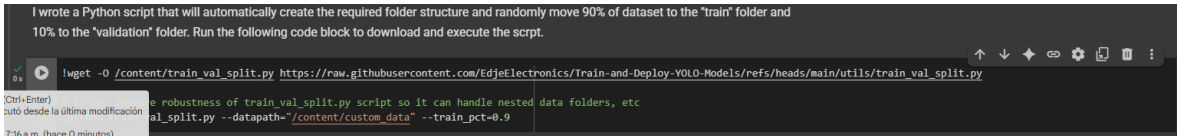


Ilustración 34 - Dividir datos

El siguiente comando que encontraremos estará en el punto 3 y consta de instalar Ultralytics, una librería de Python usada para entrenar el modelo de YOLO

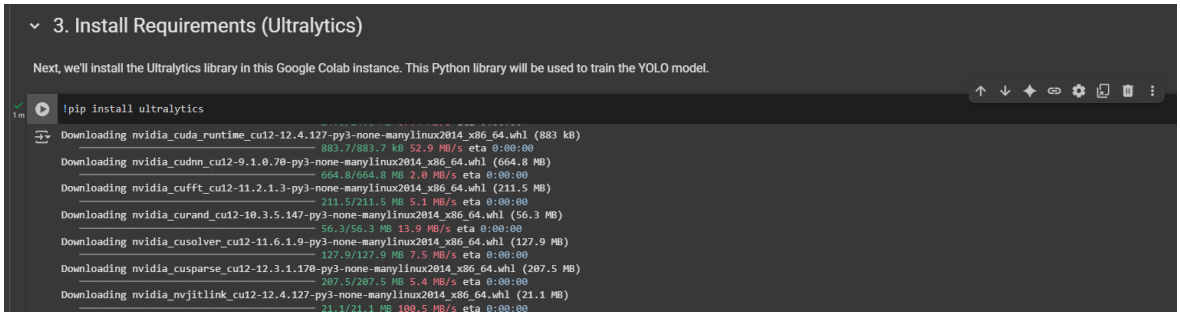


Ilustración 35 - Instalar Ultralytics

En el punto 4 encontraremos el siguiente comando para configurar el entrenamiento, lo que hace el comando es crear un diccionario de todas nuestras clases que posteriormente usaremos

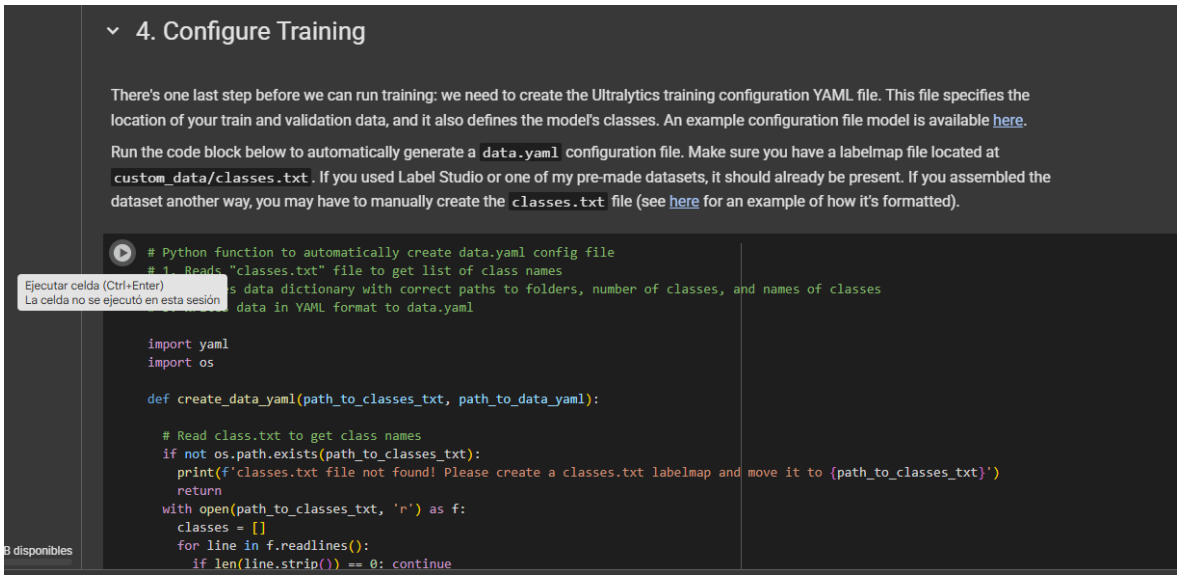


Ilustración 36 - Configurar entrenamiento

Continuamos con el comando de correr entrenamiento, los cuales por defecto vienen con los siguientes parámetros:

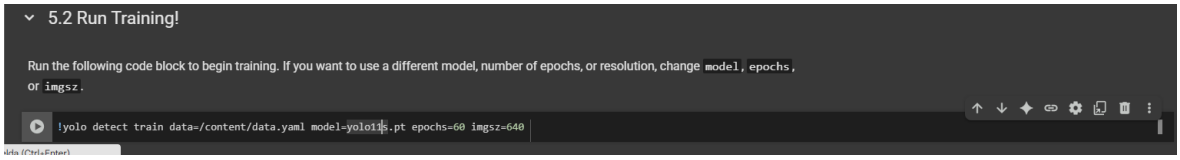


Ilustración 37 - Correr entrenamiento

Se usará el modelo de yolo11s debido a que es muy consistente, en cuanto a los epoch, al tener nuestra base de datos con relativamente pocos datos, 60 es un numero bastante recomendable para estos casos, finalmente en cuanto a la resolución, nuestra cámara cuenta con una resolución bastante buena, por lo que usar 640x640p es más que suficiente.

Después de correr el entrenamiento podemos al fin probar el modelo con el 10% de los datos correspondiente a los datos de validación.

Ilustración 38 - Probar modelo

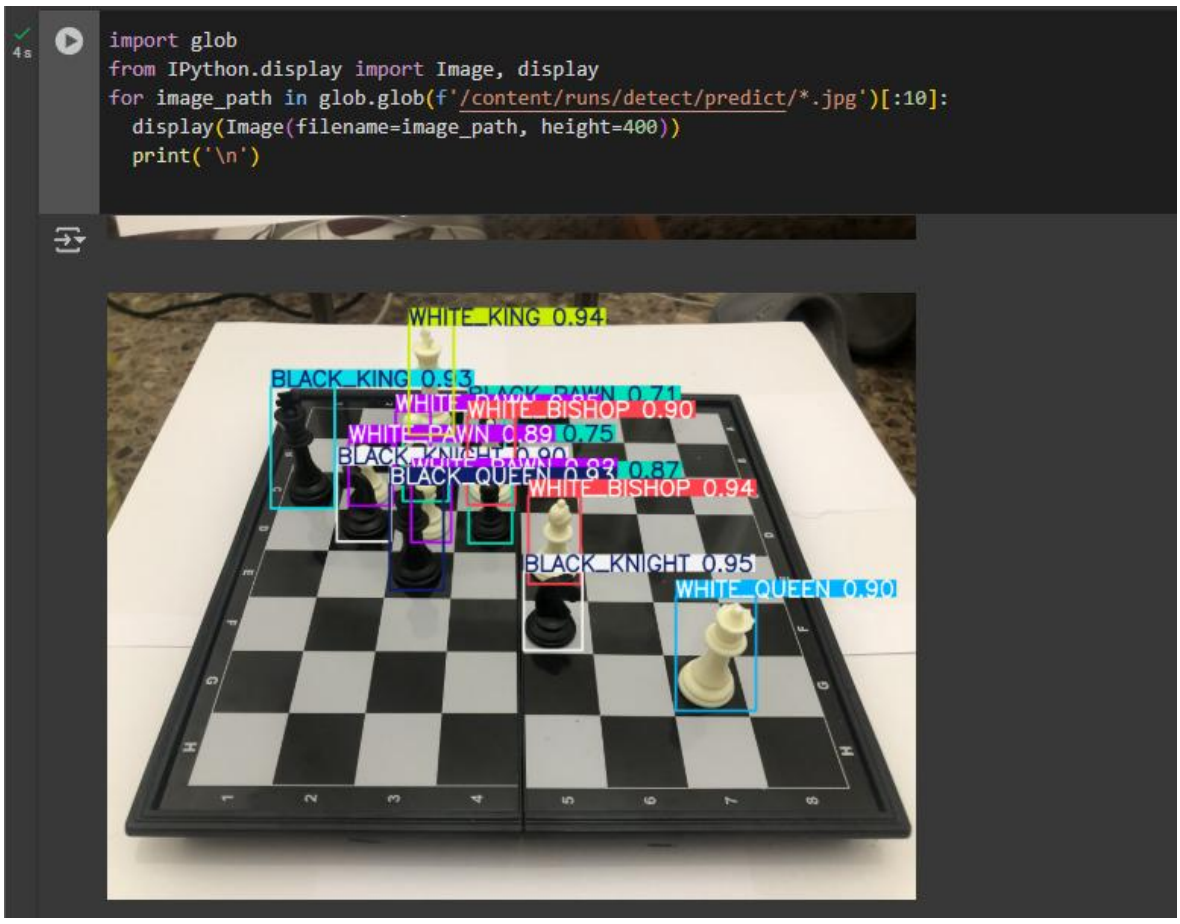


Ilustración 39 - Imagen con predicción de etiquetas

Como se puede ver en la imagen cada bounding box posee una etiqueta con la probabilidad de certeza de etiqueta que tiene el modelo sobre el objeto.

Finalmente debemos descargar el modelo

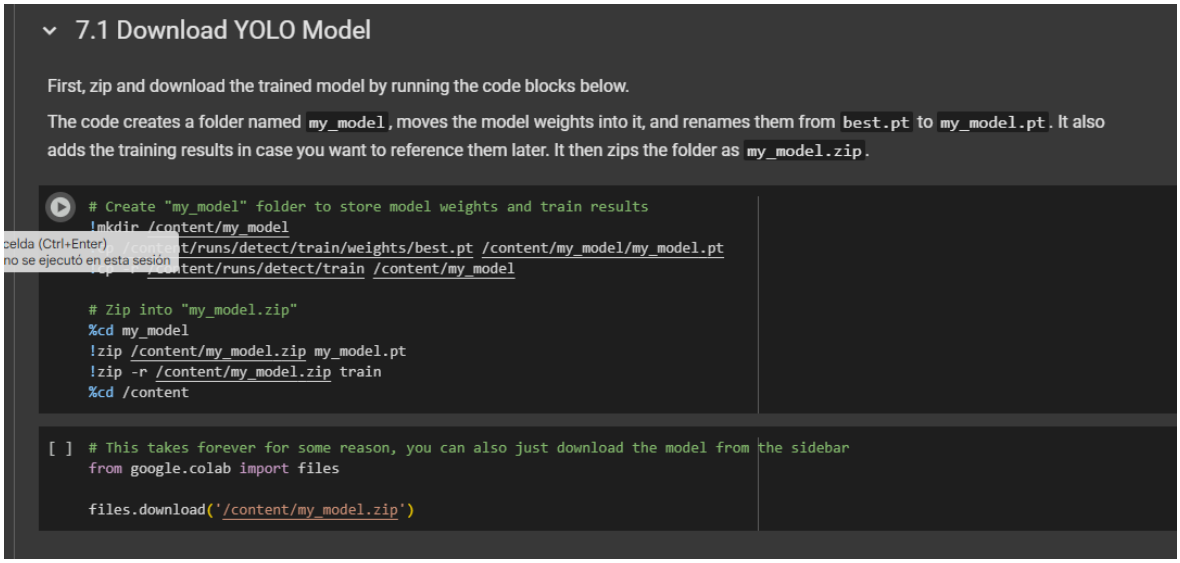


Ilustración 40 - Descargar modelo

4.2.1.2.4. Gráficas Interesantes

Dentro de la carpeta descargada del modelo de YOLO tenemos graficas que nos indican la fiabilidad del modelo

4.2.1.2.4.1. *Matriz de confusión*

La matriz de confusión nos indica si las predicciones del conjunto de datos de validación han sido correctas.

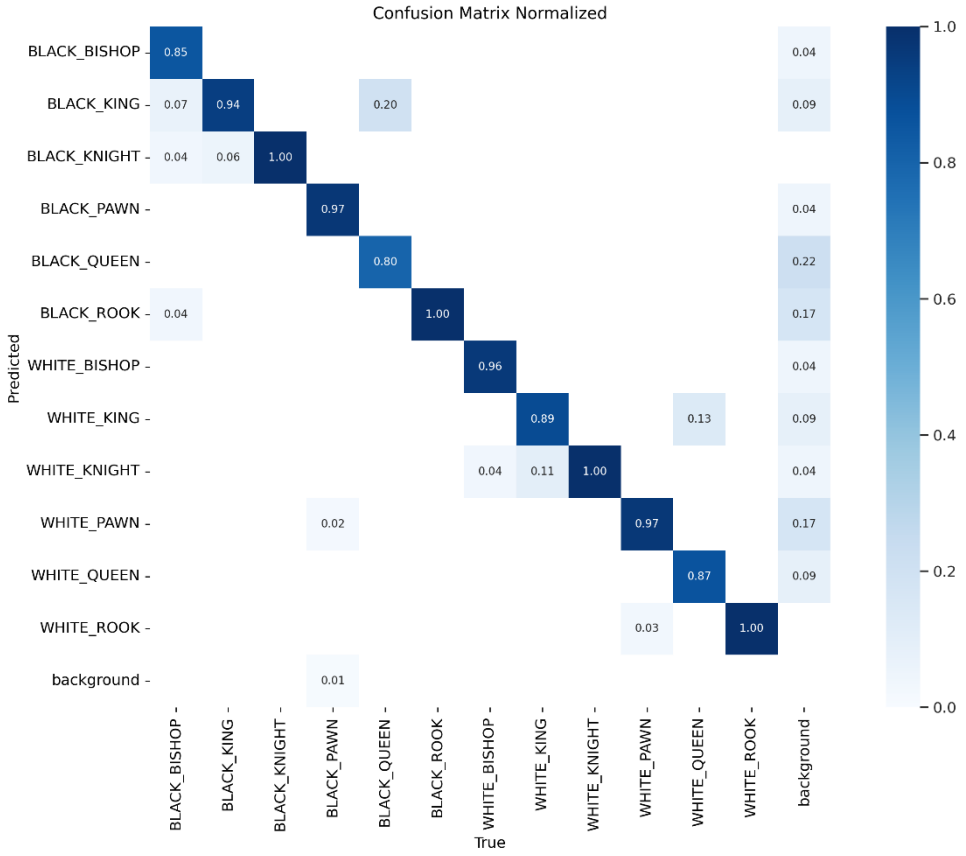


Ilustración 41 - Matriz de confusión normalizada

Al observar la diagonal principal vemos que todos los valores son mayores al 0.80, lo cual indica que hay como mínimo un 80% de certeza en todas las predicciones.

Podemos ver que, en el caso de las reinas, son las que cuentan con menor índice de certeza, siendo que son confundidas por sus respectivos reyes dada su imagen similar.

4.2.1.2.4.2. *Resultados de los epoch*

La grafica referida a los epoch y los resultados de los mismos como la precisión, el % de caja pérdidas, perdida focal de distribución(dfl), etc.

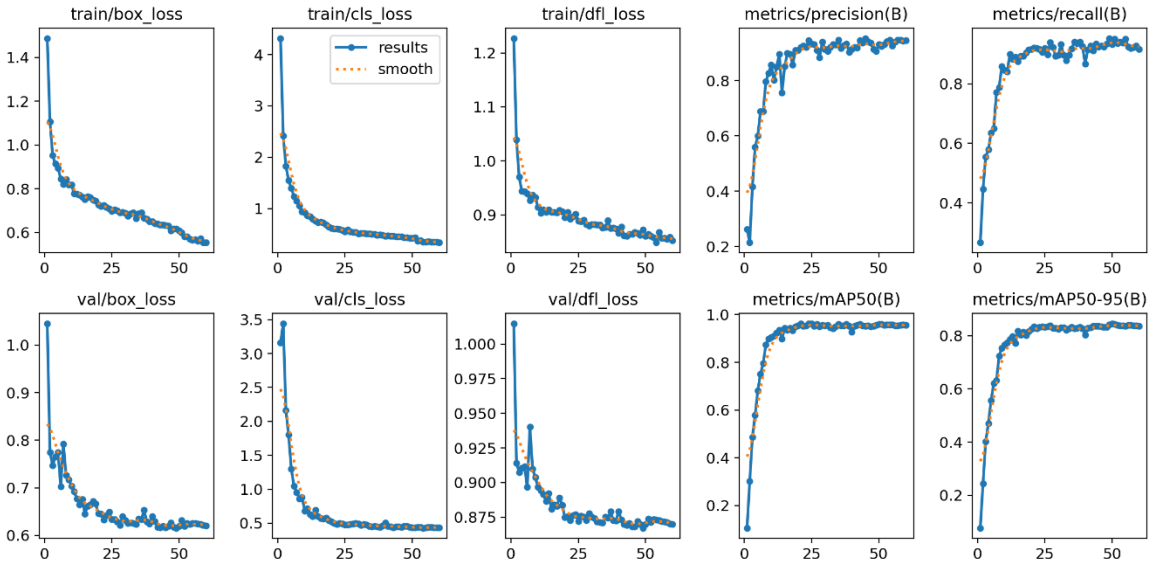


Ilustración 42 - Resultados de los epoch

Al ver el comportamiento de los resultados en los diferentes epoch, nos damos cuenta de que alrededor del epoch 25 ya se logra una buena precisión del modelo. En cuanto a la pérdida focal de distribución y la de caja son bajos y estables en epoch 25.

4.2.1.2.5. Ejecución del modelo

Para usar los modelos en nuevas imágenes debemos seguir una serie de pasos, que nos permita configurar el entorno.

4.2.1.2.5.1. *Comandos en el powershell de windows*

Primero debemos abrir el PowerShell de Windows y ejecutar la siguiente serie de comandos

- python -m venv yolo-env1

  Este comandando permite la creación de un entorno virtual

- .\yolo-env1\Scripts\Activate.ps1

  Este comando activa el entorno virtual

- pip install ultralytics

  Al igual que con la GPU virtual debemos instalar ultralytics en nuestro ordenador

- pip install --upgrade torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu124

  En caso de contar con una tarjeta gráfica nvidia es recomendable para una mayor fluidez el descargar pytorch

- cd "C:\Users\Gustavo Chen\OneDrive - unizar.es\Escritorio\TFG_PYTHON"

Nos dirigimos a la carpeta en la cual tenemos descomprimido del modelo descargado desde Google colab después de correr el script

- Invoke-WebRequest -Uri "https://raw.githubusercontent.com/EdjeElectronics/Train-and-Deploy-YOLO-Models/refs/heads/main/yolo_detect.py" -OutFile "yolo_detect.py"

Descargamos un script para correr el modelo en nuevas imágenes.

- python yolo_detect.py --model my_model.pt --source posicion.png --resolution 1280x720

Finalmente usamos este comando para correr el script descargado previamente en nuevas imágenes teniendo de base el modelo descargado desde Google colab con nuestras etiquetas.

4.2.1.2.6. Resultados gráficos

Al correr el modelo sobre una imagen nunca antes vista por la base de datos, nos damos cuenta como de igual forma es capaz de predecir en gran medida todas las piezas, así como los polígonos que lo delimitan, teniendo en su mayoría una presión mayor al 90%

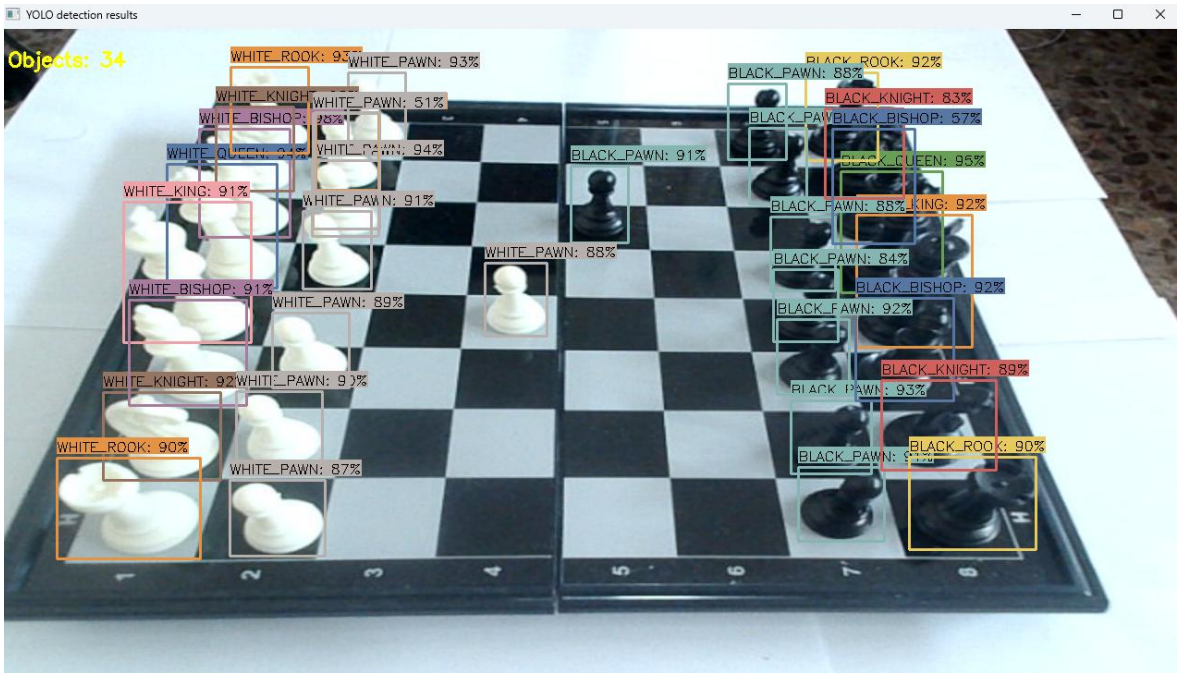


Ilustración 43 - Prueba del modelo en nueva imagen

4.2.1.3. Interfaz

Teniendo ya desarrolladas las partes de identificación de tablero e identificación de piezas, procederemos a crear una interfaz para representar de mejor manera la posición de las piezas, además esta interfaz nos ayudara a sacar la notación FEN que es necesaria si queremos usar Stockfish para el cálculo de la mejor jugada posible.

4.2.1.3.1. Asignar piezas a las casillas

Antes de crear un entorno virtual en el cual pueda ser reflejado nuestro tablero, es necesario la asignación de cada pieza a una sola casilla, para ello recurriremos al archivo .json que habíamos creado al final del punto 4.2.1.1.7 con las intersecciones del tablero.

Si recordamos la forma de los datos del fichero de datos YOLO en el punto 4.2.1.2.2.4, nos daremos cuenta que los parámetros de los bounding boxes están delimitados por su centro, por lo que pasaremos las coordenadas de las intersecciones de las casillas a centros, mediante la siguiente formula:

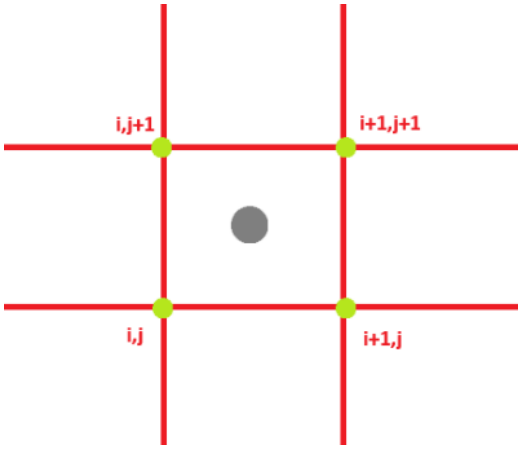


Ilustración 44 - Hallar punto medio

$$Punto\ medio = \left(\frac{X_i + X_{i+1}}{2}; \frac{Y_j + Y_{j+1}}{2}\right)$$

Ahora que tenemos los centros de las casillas podríamos a primera instancia compararlos con los centros relativos de cada bouding box, sin embargo si se analiza bien las imágenes dadas por la predicción del modelo(Ilustración 43), podemos ver como en piezas de gran altura su centro se desvía mucho de la casilla en la que está ubicada, a su vez las piezas que se ubican más a los extremos laterales tienden a tener su centro desplazado hacia la derecha en el caso de las piezas blancas y la izquierda en el caso de las piezas negras, por lo que es necesario realizar un tratamiento a sus centros.

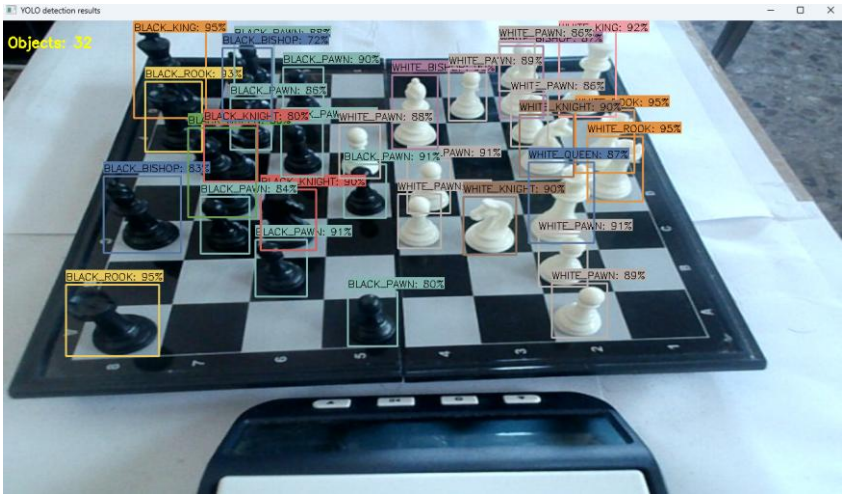


Ilustración 45 - Predicción del modelo con centro desviado

Mediante el siguiente código podremos colocar las piezas en las casillas que más se le adecuen

```
with open(YOLO_TXT, "r") as f:
    for linea in f:
        partes = linea.strip().split()
        if len(partes) != 5:
            continue

        clase_id, x_norm, y_norm, w_norm, h_norm = map(float, partes)
        nombre_pieza = CLASES_YOLO.get(int(clase_id))
        if not nombre_pieza:
            continue
```

Esta parte del código revisa si el fichero de YOLO creado contiene las 5 columnas características de las bouding boxes que contiene la imagen, al ser como se señaló en el punto 4.2.1.2.2.4 centro relativo es necesario contar con todos los parámetros para convertirlos en coordenadas absolutas

```
        # Convertir a píxeles reales
        x_px = x_norm * ancho
        y_px = (y_norm + h_norm) * alto
```

Esta parte es la encargada de transformar los valores relativos en coordenadas de pixeles reales, luego ajustaremos los parámetros de ese centro para que este ajustado con la realidad

```
        # Aplicar homografía
```

```
        punto = np.array([[[x_px, y_px]]], dtype='float32')
        punto_transformado = cv2.perspectiveTransform(punto, matrix)[0][0]
        x_t, y_t = punto_transformado
```

Debido a que el tablero se distorsiona al estar inclinado y ser una vista ortogonal es necesario transformar los puntos si queremos compararlos con los previamente transformados de las intersecciones del tablero

```
        # Buscar casilla más cercana
        mejor_casilla = None
        min_dist = float("inf")
        for casilla, (cx, cy) in centros.items():
            dist = math.hypot(cx - x_t, cy - y_t)
            if dist < min_dist:
                min_dist = dist
                mejor_casilla = casilla

        if mejor_casilla:
            detecciones.append([nombre_pieza, mejor_casilla])
            for_fen.append([nombre_pieza, mejor_casilla])
```

Finalmente se busca la casilla más cercada y se le asigna el nombre de una pieza.

Si intentamos de todas formas buscar el centro más cercano sin realizar un tratamiento previo de los centros nos quedaría algo parecido a la siguiente imagen:

Ilustración 46 - Interfaz sin tratamiento de centros

Como se puede ver las piezas van yendo a la columna más a la izquierda ya que desde el punto de vista de la cámara es la fila más alejada, a medida que las piezas se alejan más se distorsiona su altura con respecto a las casillas.

Para solucionar los problemas de los centros se decidió multiplicar por un factor de 0.2 los centros en cuanto a su altura, para de esta manera asignar la pieza a la casilla que esté debajo de ella

```
        # Convertir a píxeles reales
        x_px = x_norm * ancho
        y_px = (y_norm + h_norm*0.2 ) * alto
```

Es importante señalar que se evitan factores más bajos dado que a veces las bounding boxes pueden tomar espacios inferiores. En cuanto a los problemas que puedan causar las desviaciones laterales se tomaron como insignificantes y se buscaron solucionar simplemente con un mejor posicionamiento manual de las piezas.

Ilustración 47 - Tablero con tratamiento de centros

Llegados a este punto se podría decir que se logró cumplir con el primer objetivo del TFG, siendo este de: Implementación de visión artificial para detección de piezas y tablero (al menos el 80% de precisión).

4.2.1.3.2. Estilo

En cuanto a los estilos de la interfaz se usó la librería de pychess para el tablero y los iconos son de la biblioteca libre de lichess [10]

4.2.1.3.3. Notación FEN

Dado que la notación FEN utiliza letras únicas y específicas para cada tipo de pieza como se señaló en el apartado 4.1.2 y estas difieren de las que usamos nosotros para las etiquetas, será necesario realizar un mapeo para asignarle a cada etiqueta que habíamos puesto su equivalente en formato FEN

Como ya habíamos asignado las piezas a su mejor casilla en el apartado anterior, se nos devolverá directamente un fichero en formato FEN de las casillas, sin embargo hay que recordar que el formato FEN no expresa únicamente la posición de las piezas en ese momento, sino su posición en el tiempo, al no partir de una posición inicial es imposible delimitar con certeza los demás parámetros como, posibilidades de enroques, numero de jugadas, numero de jugadas desde el ultimo movimiento de peón o captura, posibilidad de peón al paso, turno, etc. Por lo que estos parámetros comenzarán con su valor por defecto, siendo de:

- Turno: blancas "w"
- Enroques: "-"
- Peon al paso: "-"
- Medio movimiento "0"
- Movimiento completo "0"

Se decidió elegir sin posibles enroques como defecto dado que las posiciones en muchas ocasiones ya tienen el rey movido, en caso de partida nueva se cambiará eso por: "KkQq"

4.2.2. Cálculo de la mejor jugada

Aprovechando que en el apartado anterior logramos convertir una posición a formato FEN, simplemente procederíamos con insertar dicha nomenclatura en el motor de ajedrez StockFish, para ello hay varias opciones, siendo la más fácil de aplicar el copiar y pegar directamente lo que nos devuelva nuestro script en cualquier página de ajedrez que cuente con herramientas de análisis de posiciones.

Primero debemos ingresar a lichess (https://lichess.org/analysis)

Ilustración 48-Herramienta de analisis Lichess

En la Zona donde dice FEN, colocamos nuestro código FEN de la posición, siendo este de:

r1bq1rk1/ppn1n1bp/1np1p1p1/3pP3/2PP1B2/2N3P1/PPQ1NPBP/R3R1K1 w - - 0 1

Con lo cual nos quedaría dicha posición:

Ilustración 49- Calculo mejor jugada

Como se puede apreciar, nos indica la mejor jugada a realizar mediante una flecha e incluso cuanta ventaja poseemos.

Para agilizar este procedimiento se decidió realizar un script que nos abra de frente el link de la posición a analizar sin tener que copiar y

pegar, esto se logra dado a que el propio link de la posición cuenta con formato FEN:

Ilustración 50 - Link con formato FEN

Para el script usamos:

```
import webbrowser

# Leer FEN desde archivo
with open("fen_output.txt", "r") as f:
    fen = f.read().strip()

# Reemplazar espacios con %20
fen_url = fen.replace(" ", "%20")

# Construir URL
url = f"https://lichess.org/analysis/fromPosition/{fen_url}"

# Abrir en el navegador
webbrowser.open(url)
```

Con lo cual al ejecutarlo nos abre:

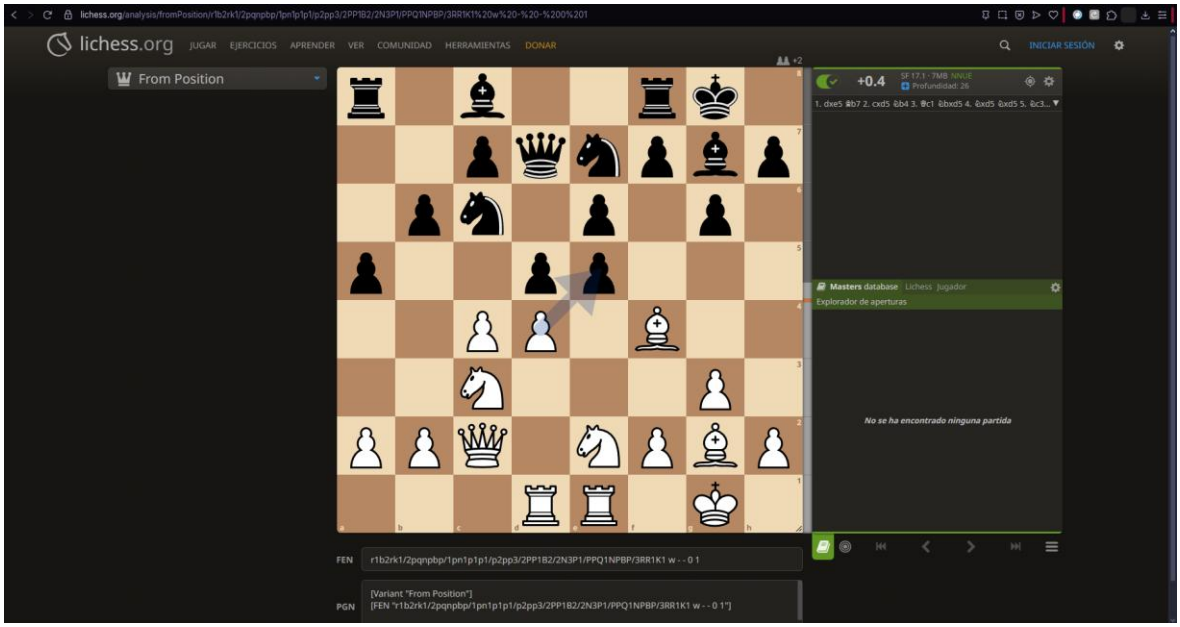


Ilustración 51 - Análisis abierto desde script

Nos damos cuenta que la posición es la misma, y que devuelve exactamente el mismo análisis de la posición con la misma jugada recomendada.

En cuanto al link:

Ilustración 52 - Link desde script

El link si bien es cierto cambió, lo único que se diferencia con el "original" es que se reemplazó los espacios por %20 que es su formato en código URL, por lo que todo lo demás está bien.

Con esto se daría por satisfecho el segundo objetivo de nuestro TFG, el cual consistía en la implementación de un motor de ajedrez para detectar las mejores jugadas posibles.

4.2.3. Movimiento del brazo

Para el movimiento del brazo es necesario aplicar algoritmos de cinemática inversa ante ello partiremos de modelos de cinemática directa mediante el método de Denavit-Hartenberg

4.2.3.1. Denavit-Hartenberg

4.2.3.1.1. Identificación de Articulaciones

Teniendo armado nuestro brazo resulta sencillo identificar las distintas articulaciones que este posee

Ilustración 53 - Brazo armado señalando articulaciones

Teniendo las articulaciones señaladas podemos analizar su movimiento, si bien es cierto es un brazo de 6 grados de libertad, las articulaciones 5 y 6 no representan cambio alguno en la ubicación del punto de agarre de la pinza, siendo el 5 un movimiento de rotación, mientras que el 6 la trivialidad entre abrir y cerrar la pinza, por lo que únicamente estaríamos trabajando con un brazo robótico de 4 grados de libertad, a lo que nos puede quedar el siguiente dibujo:

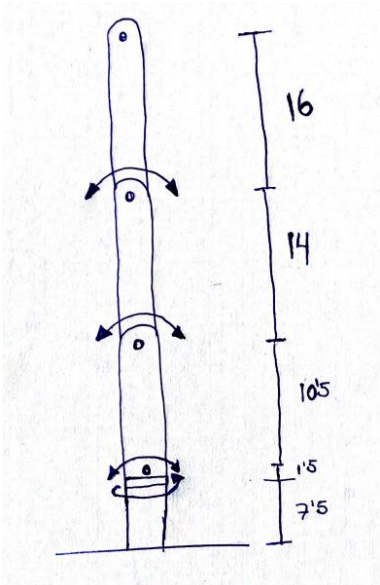
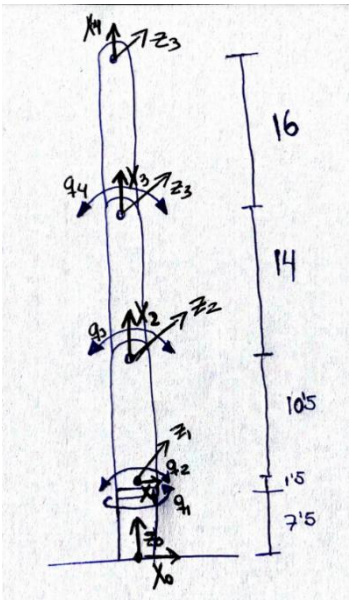


Ilustración 54 - Brazo robótico 4 DOF

4.2.3.1.2. Asignación de sistema de referencia



Teniendo los ejes ya dibujados procederemos con el diseño de la tabla para los parámetros θ, d, a, α para cada eslabón, quedándonos:

Tabla 1 - Parámetros DH

	$\boldsymbol{\theta}$	**d**	**a**	$\boldsymbol{\alpha}$
1	q_1	9	0	$-\pi/2$
2	q_2	0	10.5	0
3	q_3	0	14	0
4	q_4	0	16	0

4.2.3.1.3. Matrices de transformación de cada eslabón

Ilustración 55- Sistema de Referencia brazo 4 DOF

Cada eslabón permite definir una matriz de transformación, siendo esta de la forma:

$$A_i^{i-1} = \begin{matrix} cos\theta_i & -cos\alpha_i sin\theta_i & sin\alpha_i sin\theta_i & a_i cos\theta_i \\ sin\theta_i & cos\alpha_i cos\theta_i & -sin\alpha_i cos\theta_i & a_i sin\theta_i \\ 0 & sin\alpha_i & cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{matrix}$$

Teniendo la matriz de transformación de cada uno de los eslabones podremos continuar con el calculo de la transformada total, la cual sigue la siguiente forma:

$$T = A_1^0 * A_2^1 * A_3^2 * A_4^3$$

4.2.3.1.4. Transformadas en MATLAB

Teniendo la forma de la transformada de cada eslabón podemos definirla en Matlab para que la solucione automáticamente, siendo el código de esta:

```
function T = forwardTransfer(a, alpha, d, theta)
    T = [cos(theta), -sin(theta)*cos(alpha),  sin(theta)*sin(alpha), a*cos(theta);
         sin(theta),  cos(theta)*cos(alpha), -cos(theta)*sin(alpha), a*sin(theta);
         0,           sin(alpha),             cos(alpha),            d;
         0,           0,                      0,                     1];
    T = vpa(T, 3);
end
```

Una vez definida la función de transformada de cada eslabón, podremos también multiplicarlas entre sí para hallar la transformada total, con el siguiente código:

```
function [T0_i, Tj_i] = get_T0i(theta, a, d, alpha)

n = length(a); % Number of joints

T0_i = cell(1,n);
T0_j = cell(1,n);
Tj_i = cell(1,n);
T0_j{1} = eye(4); % Define as empty at first
for i = 1:n
    % Find single trans. matrix
    Tj_i{i} = forwardTransfer(a(i), alpha(i), d(i), theta(i));
    % Multiply that matrix with the multiplication of all previous ones
    T0_i{i} = T0_j{i}*Tj_i{i};
    % Save result to use in next loop
    T0_j{i+1} = T0_i{i};
end

assignin('base','Tj_i',Tj_i);
assignin('base','T0_j',T0_j);

end
```

4.2.3.1.5. Resolución simbólica con MATLAB

Teniendo las funciones de las transformadas, debemos crear una función que nos permita resolver el problema simbólicamente dando soluciones que permitan alcanzar la posición deseada del efector al final del brazo.

Primero recogeremos los parámetros claves de la matriz DH, para su facilidad de manejo en el futuro:

```
L_a = DH(:,1);
L_d = DH(:,2);
L_alpha = DH(:,3);
L_theta = DH(:,4);
n = length(L_a);
```

Dado que nuestro robot cuenta con límites físicos que no permiten cierto ángulo de rotación en los servomotores, debemos crear la posibilidad indicar el valor de dichos parámetros, de igual forma con los offsets de sistema, ambos parámetros tendrán rangos o valores predeterminados en caso de no ser señalados, además en caso de existir realmente offsets es necesario sumarle al rango de los ángulos del motor para evitar que no se hallen soluciones.

```
if nargin < 4
    angle_offset = [0 0 0 0];
end
if nargin < 3
    angle_range_motor = repmat([-180 180], n, 1) * pi/180;
end
% Rango para las variables theta con offset aplicado
angle_range_theta = [angle_range_motor(:,1)+angle_offset', ...
                     angle_range_motor(:,2)+angle_offset'];
```

Aplicar la cinemática directa simbólica para cada transformada de cada articulación, finalmente lo que devuelve es la posición simbólica del efector final en función de todos los ángulos θ de los motores

```
fprintf('Symbolic forward kinematics\n')
[T0_i, ~] = get_T0i(L_theta, L_a, L_d, L_alpha);
joint_pos_sym = cellfun(@(T) T(1:3,4), T0_i, 'UniformOutput', false);
```

Finalmente plantea un sistema de ecuaciones del efector final con la posición deseada

```
eq = joint_pos_sym{n} == target_pos(:);
```

Se declaran las variables en el orden correcto para usar vpasolve

```
% Declarar símbolos manualmente en orden
syms th0 th1 th2 th3
original_vars = [th0, th1, th2, th3];
original_var_names = {'th0','th1','th2','th3'};
```

Se revuelve el vpasolve, es importante señalar que vpa solve no funciona como un barrido en búsqueda de soluciones, sino que busca una solución, y si esta no se encuentra dentro de los rangos delimitados por los motores no da soluciones válidas, por lo que limitar mucho los servos resulta perjudicial para hallar una solución, lo mejor resulta tratar los datos y buscar ángulos que cumplan la misma función, pero en el rango deseado.

```
sol_angle = vpasolve(eq, original_vars, angle_range_theta);
```

Verifica si encontró alguna solución

```
if isempty(sol_angle) || any(structfun(@isempty, sol_angle))
    error('❌ No se encontró una solución válida para la cinemática inversa.');
end
```

4.2.3.1.6. Script principal

Ya hemos visto las funciones relacionadas con las transformadas, también vimos el procedimiento que sigue la cinemática inversa para hallar soluciones dentro de rangos delimitados, ahora aplicaremos las funciones para que nos den una solución.

Primero definimos los parámetros de DH, también asignamos una posición objetivo, en este caso las coordenadas [12.5 25 5] son las correspondientes a la casilla H8, es decir la más alejada del robot.

```
%% ---------------------------------------- Definición del modelo DH
syms th0 th1 th2 th3
DH = [0 10.5 14 16;           % a
      9 0    0  0 ;           % d
     pi/2 0 0  0;             % alpha
      th0' th1' th2' th3'];   % theta simbólicos (sin offset)

target_pos = [12.5 25  5];
```

Al igual que con la resolución simbólica del apartado anterior debemos definir los valores de inicialización, es importante señalar que los offset calculados en la tabla DH deben ir en la matriz de offset de ángulos, no en la DH. En cuanto a los rangos permitidos para cada motor he decidido no limitarlos tanto debido a que a veces no encuentra solución a pesar de que los valores que puedan ser soluciones se encuentren en dicho rango, posteriormente hará un tratamiento de los datos para ajustarlos a rangos reales.

```
%% ---------------------------------------- Inicializaciones
fprintf('Inicilizaciones\n')
L_a = DH(:,1);
L_d = DH(:,2);
L_alpha = DH(:,3);
L_theta = DH(:,4);
n = size(DH,1);

% Offsets de los motores (por montaje físico)
angle_offset = [0 90 0 0]*pi/180;

% Rango permitido para cada motor
angle_range_motor = [0 360; -120 120; -120 120; -120 120]*pi/180;
```

En cuanto a la cinemática inversa, simplemente llamamos a las funciones creadas anteriormente

```
%% ---------------------------------------- Cinemática inversa
L_motor_sol = ikine4DOF_v2(DH, target_pos, angle_range_motor, angle_offset);

% Guardamos solución previa (para comparar)
L_theta_sol_pre = L_motor_sol + angle_offset';
T0_i_pre = get_T0i(L_theta_sol_pre, DH(:,1), DH(:,2), DH(:,3));
final_pos_pre = double(T0_i_pre{end}(1:3,4));
```

Si nos quedamos con la solución que se nos de aquí pueden presentarse algunos problemas como codos indeseados.

4.2.3.1.7. Solucionar codos indeseados

Según la naturaleza de nuestro brazo robótico en el agarre de piezas de ajedrez, siguiendo el camino de las articulaciones, se eleva desde la base, se acerca a la posición deseada y finalmente baja el brazo, sin embargo, es posible llegar a la misma solución bajando el brazo y luego subirlo hacia la posición objetivo, dicho movimiento final resulta imposible en la realidad, dado que para agarrar una pieza con un movimiento ascendente desde abajo se debería atravesar el tablero.

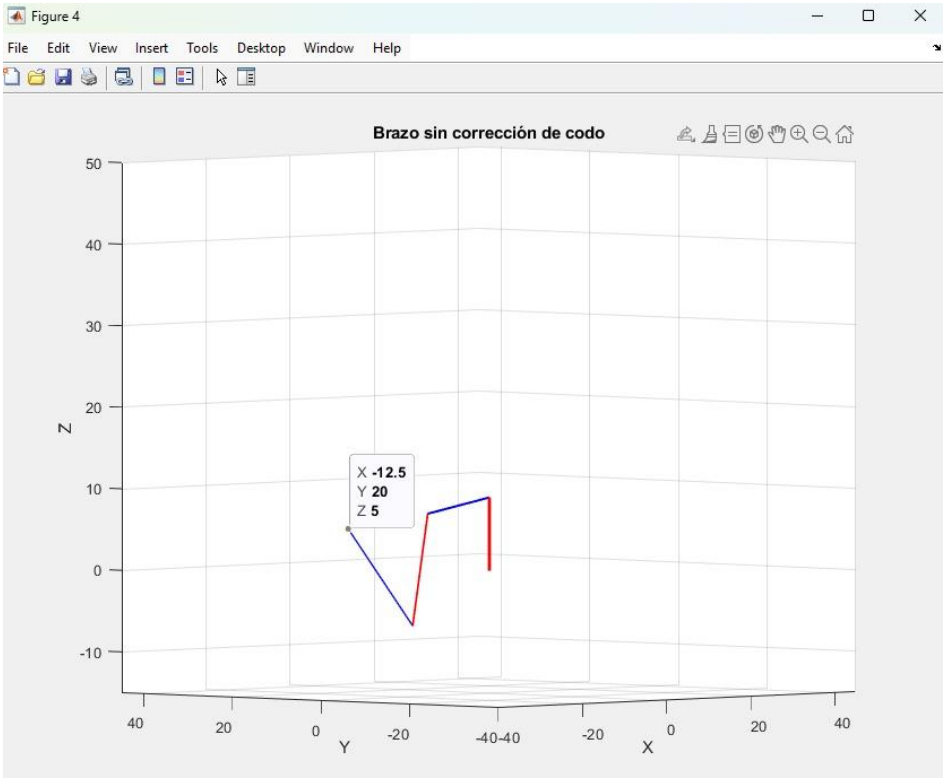


Ilustración 56 - Codo Indeseado

Como se puede ver en la imagen si bien es cierto que siguiendo los límites de los ángulos de los motores es posible llegar a dicha solución, en la realidad no es para nada factible dado la existencia de un plano que no se puede traspasar como lo es el tablero.

Para solucionar los codos se aprovechará de la propiedad de simetría axial

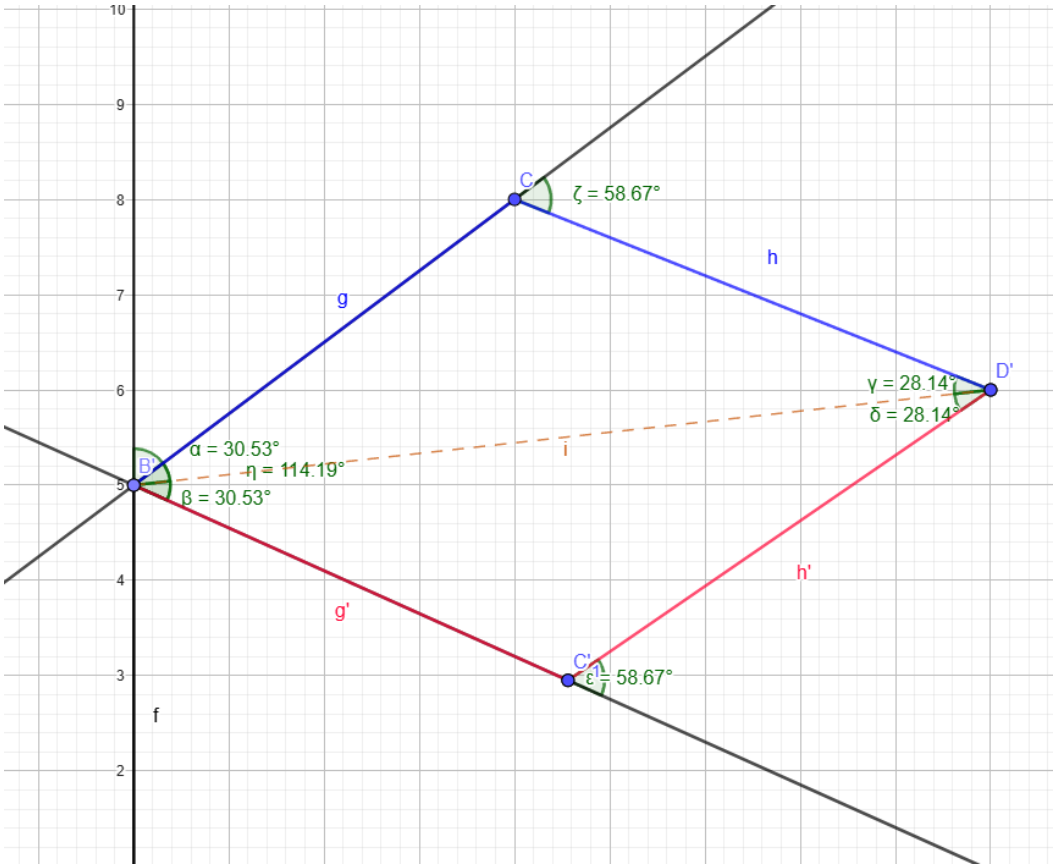


Ilustración 57 - Simetría axial para codo indeseado

Como podemos ver a simplevista es posible llegar al mismo punto simplemente realizando una simetría axial sobre una recta entre dos articulaciones separadas entre sí por al menos otra, si bien es cierto que únicamente no bastaría con cambiar de signo a los ángulos, dado que el ángulo de inclinación depende de la orientación de la articulación anterior, de igual forma es sencillo aplicar la simetría únicamente usando la ley del coseno dado que conocemos las longitudes de las articulaciones g y h y además el ángulo que forma dicha intersección, dado que es el suplementario del ángulo ε.

Los ángulos de los cuales tenemos conocimientos reales son los de η y ε, ahora, si los queremos cambiar para que el codo esté hacia abajo.

Primero calculamos la longitud del eje de simetría, siendo la fórmula:

$$i = g^2 + h^2 - 2 * g * h * \cos(180 - \varepsilon)$$

Resolviendo esa fórmula podremos calcular los ángulos internos del cuadrilátero y modificar los ángulos de los servomotores para que el codo se encuentre hacia abajo

$$\frac{i}{sen(180-\varepsilon)} = \frac{h}{sen(\alpha)} = \frac{g}{sen(\gamma)}$$

Teniendo todos los parámetros únicamente nos quedaría modificar los ángulos de los servos para que se cumpla la posición final.

Tabla 2 Tabla con conversión de codos

	Codo hacia abajo	Codo hacia arriba
Angulo servomotor i	η	$\eta + 2\alpha$
Angulo servomotor i+1	ε	$-\varepsilon$

En caso -el servomotor i+1 no sea el final del conjunto se deberá modificar el ángulo del servomotor i+2 para que la posición del efector final no varíe.

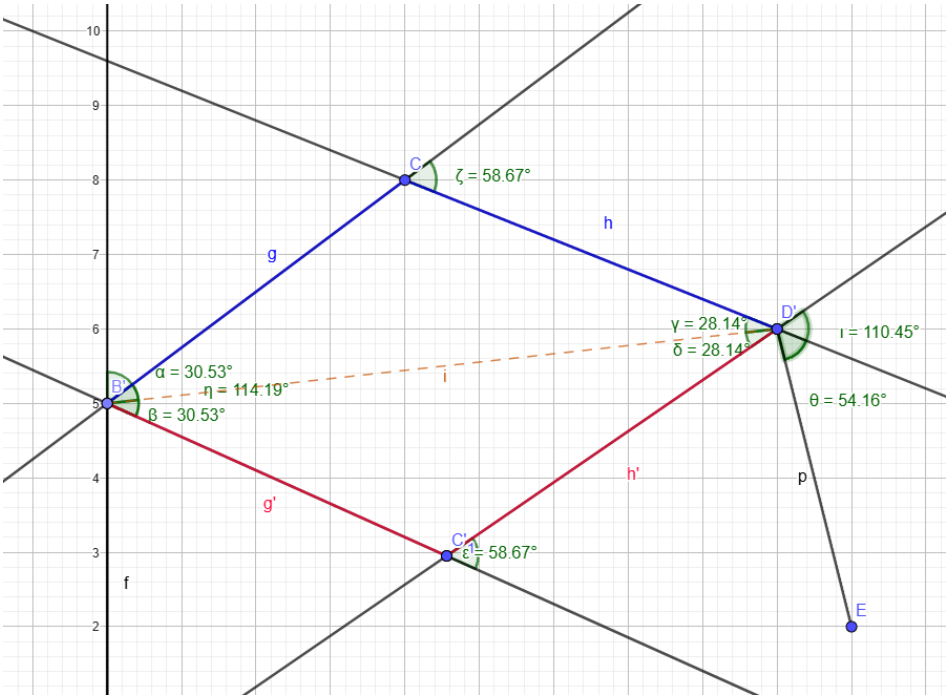


Ilustración 58 - Caso articulación extra

Podemos aprovechar el cálculo del ángulo γ en el caso anterior para modificar el angulo del servomotor i+2 para que la posición final se siga manteniendo, lo único que debemos hacer es:

Tabla 3 - Continuacion de tabla de conversion de codos

	Codo hacia abajo	Codo hacia arriba

Angulo servomotor i+2	ι	$\iota - 2\gamma$

Ahora que hemos realizado la parte matemática en cuanto al script de MATLAB, sería el siguiente:

Para el tercer ángulo en caso de ser mayor a 0, es decir que se forma un codo hacia arriba entre las articulaciones 2 y 3

```
%% ---------------------------------------- Corrección de codo hacia abajo (geométricamente)
L2 = DH(2,1);
L3 = DH(3,1);
L4 = DH(4,1);

% Extraer ángulos sin offset
theta1 = L_motor_sol(1); % por si lo usas después
theta2 = L_motor_sol(2);
theta3 = L_motor_sol(3);
theta4 = L_motor_sol(4);

if theta3 > 0
    fprintf('🔄 Corrigiendo codo hacia abajo usando reconstrucción vectorial\n');

    % Paso 1: calcular posición final del tercer eslabón (muñeca)
    x3 = L2*cos(theta2) + L3*cos(theta2 + theta3);
    y3 = L2*sin(theta2) + L3*sin(theta2 + theta3);

    % Paso 2: reflejar codo (cambia el signo de theta3)
    D = (x3^2 + y3^2 - L2^2 - L3^2) / (2*L2*L3);
    if abs(D) > 1
        error('⚠️ Punto fuera del alcance del brazo.');
    end
    theta3_new = -acos(D); % reflejo de codo (mismo valor absoluto)

    % Paso 3: calcular nuevo theta2 con atan2
    phi = atan2(y3, x3);
    theta2_new = phi - atan2(L3*sin(theta3_new), L2 + L3*cos(theta3_new));

    % Paso 4: mantener orientación del cuarto eslabón
    suma_antigua = theta2 + theta3 + theta4;
    suma_nueva = theta2_new + theta3_new;
    theta4_new = suma_antigua - suma_nueva;

    % Actualizar solución
    L_motor_sol(2) = theta2_new;
    L_motor_sol(3) = theta3_new;
    L_motor_sol(4) = theta4_new;
end
```

Para el cuarto ángulo en ser mayor a cero, se formaría un codo hacia arriba entre las articulaciones 3 y 4, al no existir una articulación 5, no es necesario aplicar lo que está en la Tabla 3.

```
if theta4 > 0
    fprintf('🔁 Corrigiendo codo hacia abajo usando reconstrucción vectorial\n');

    % Paso 1: calcular posición final del cuarto eslabón
    x4 = L3*cos(L_motor_sol(3)) + L4*cos(L_motor_sol(3) + L_motor_sol(4));
    y4 = L3*sin(L_motor_sol(3)) + L4*sin(L_motor_sol(3) + L_motor_sol(4));

    % Paso 2: reflejar codo (cambia el signo de theta4)
    D2 = (x4^2 + y4^2 - L3^2 - L4^2) / (2*L3*L4);
    if abs(D2) > 1
        error('⚠️ Punto fuera del alcance del brazo.');
    end
    theta4_new = -acos(D2); % reflejo

    % Paso 3: calcular nuevo theta3 con atan2
    phi = atan2(y4, x4);
    theta3_new = phi - atan2(L4*sin(theta4_new), L3 + L4*cos(theta4_new));

    % Actualizar solución
    L_motor_sol(3) = theta3_new;
    L_motor_sol(4) = theta4_new;
end
```

4.2.3.1.8. Cinemática directa y graficar

Teniendo ya los ángulos tratados procederemos a realizar la cinemática directa con dichos para posteriormente graficarlos

```
%% ---------------------------------------- Cinemática directa numérica
fprintf('Numeric forward kinematics\n')

L_theta_sol = L_motor_sol + angle_offset'; % aplicar offset solo al final
T0_i = get_T0i(L_theta_sol, DH(:,1), DH(:,2), DH(:,3));
for i = 1:n
    joint_pos_num{i} = T0_i{i}(1:3,4);
end

% Comparar posición final antes y después
final_pos_post = double(joint_pos_num{end});
fprintf('\nDiferencia de posición final (debe ser cercana a [0 0 0]):\n');
disp(final_pos_post - final_pos_pre);
```

Para graficar:

```
%% ------------------------------------------- Gráfica del brazo
fprintf('Protting\n')

figure(4)
arm(1) = plot3([0 joint_pos_num{1}(1)], [0 joint_pos_num{1}(2)], [0 joint_pos_num{1}(3)], 'r', ...
    'LineWidth', 1+n/4);
hold on
for i = 2:n
    arm(i) = plot3([joint_pos_num{i-1}(1) joint_pos_num{i}(1)], ...
                   [joint_pos_num{i-1}(2) joint_pos_num{i}(2)], ...
                   [joint_pos_num{i-1}(3) joint_pos_num{i}(3)], ...
                   'Color', [mod(i,2) 0 mod(i+1,2)], ...
                   'LineWidth', 1+(n-i)/4);
end
xlim([-40 45])
ylim([-40 45])
zlim([-15 50])
grid on
xlabel('X'); ylabel('Y'); zlabel('Z'); title('Brazo con codo corregido (abajo)');
```

4.2.3.1.9. Gráficos

Al corregir los ángulos indeseados nos queda una posición más "natural" que tendría una mano al jugar ajedrez

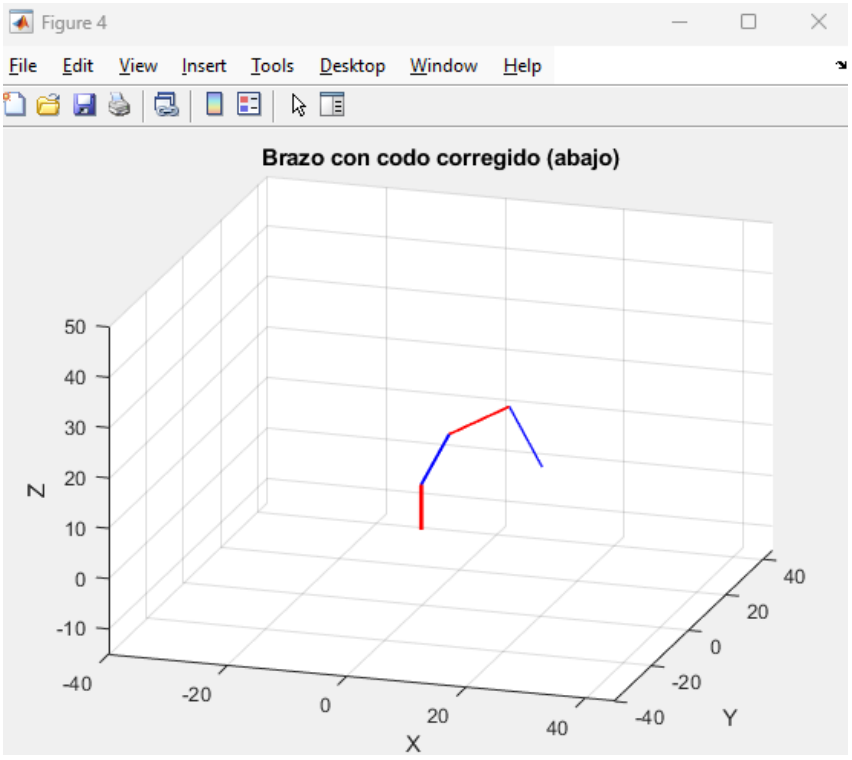


Ilustración 59 - Grafico de la cinemática inversa con codo hacia abajo

Continuaremos con otras posiciones de piezas para observar el comportamiento del robot

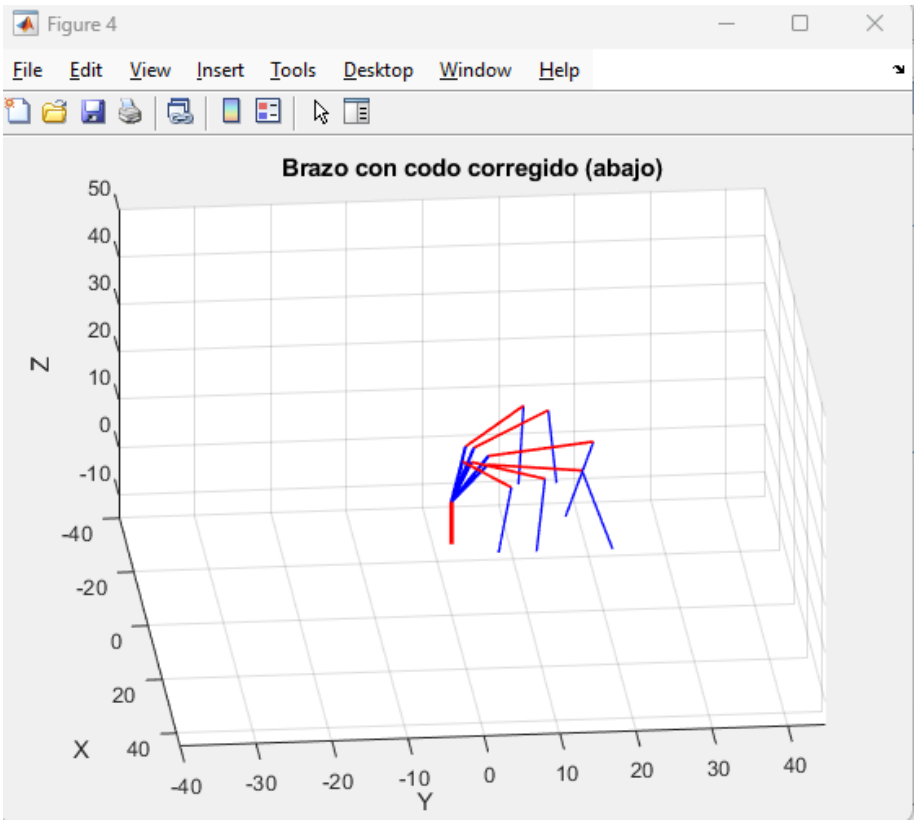


Ilustración 60 - Cinemática Inversa en diferentes posiciones

Como se puede apreciar, el programa es capaz de resolver ubicaciones objetivo mediante la cinemática inversa, por lo que, teóricamente, estaría completa la parte relacionada con el movimiento del brazo y la localización de las piezas dentro del tablero. Debido a la complejidad del sistema, se ha optado por simular cada ubicación de las casillas y guardarlas junto con las combinaciones de los ángulos de los servomotores que dan lugar a dicha posición. De esta manera, se obtiene un script de 64 líneas que contiene los ángulos de cada uno de los cuatro servomotores. Así, se aprovecha una de las ventajas de la nomenclatura FEN: los movimientos se definen únicamente señalando la ubicación inicial y final de la pieza, sin necesidad de recurrir a la nomenclatura tradicional, en la que se especifica el nombre de la pieza, su casilla de origen (en caso de que existan dos o más piezas iguales que puedan ir al mismo objetivo) y la casilla final.

4.2.3.1.10. Script de casillas:

Para saber la ubicación de cada casilla debemos medir apropiadamente la distancia entre el brazo robótico y el tablero, partiendo de la siguiente disposición:

Ilustración 61 - Disposición Final

Teniendo la posición final en la que se encontrara nuestro tablero respecto al brazo, simplemente tendremos que obtener la ubicación del tablero en las coordenadas x,y,z partiendo de (0;0;0) como la base de nuestro robot, pudiendo obtener los siguientes datos:

- Centro de la casilla A8: (-12.5;5)
- Centro de la casilla A1: (12.5;5)
- Centro de la casilla H8: (-12.5;30)
- Centro de la casilla H1: (12.5;30)

Contando con estos valores como puntos de partida, mediante interpolación podremos obtener el resto de casillas faltantes, sin embargo, falta un detalle importante, el cual es el eje z, si bien es cierto que hay alturas de la garra en la cual beneficiaria más a unas piezas que a otras, se optó por una altura intermedia de 5, quedando para las casillas anteriormente mencionadas las siguientes coordenadas:

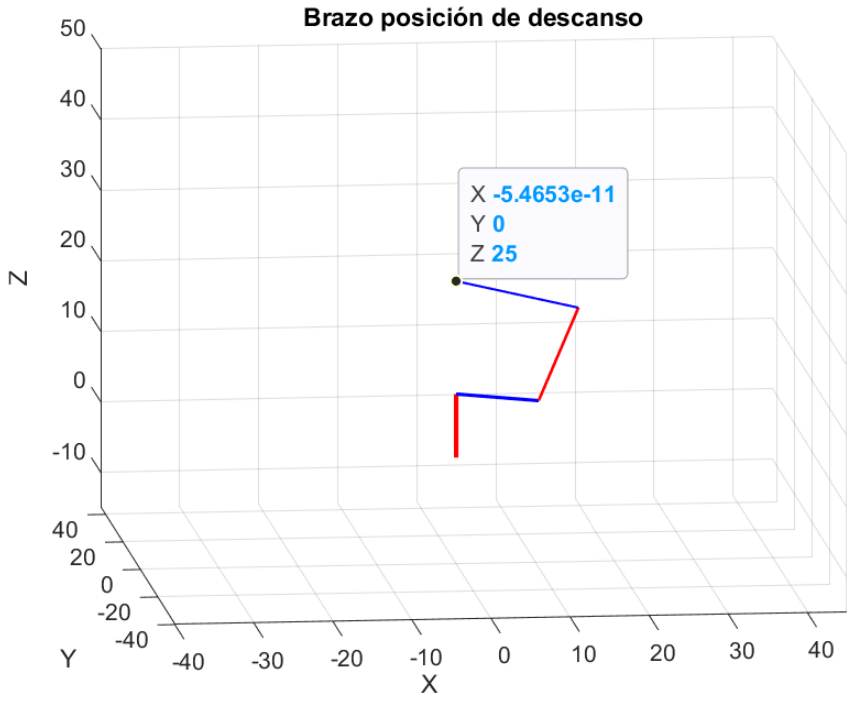
- Casilla A8: (-12.5;5;5)
- Casilla A1: (12.5;5;5)

- Casilla H8: (-12.5;30;5)
- Casilla H1: (12.5;30;5)

Contando con todas las coordenadas de las casillas calculadas, solo debemos simular una por una y guardar las combinaciones de los servos

4.2.3.1.11. Posición de descanso

Teniendo ya la ubicación en coordenadas de todas las casillas, procederemos a buscar una posición de descanso en la cual se quedará el robot mientras la contraparte sigue jugando, o mientras se está ejecutando las líneas de código para calcular la mejor jugada o analizar el tablero, se optó por la posición (0;0;25) dado que no afectará a la cámara ni se interpondrá en el movimiento de brazo de su contraparte.



4.2.4. Calibración de servomotores

4.2.4.1. Servomotores de grados de libertad

Una vez determinados los ángulos teóricos de los servomotores, se procederá con la calibración de los servomotores físicos. Tal como se indicó en el apartado 4.1.5 del marco teórico, se emplearán servomotores del modelo MG996R.

El objetivo de esta etapa es establecer la correspondencia entre los ángulos deseados y los ciclos de trabajo de la señal PWM aplicada a cada motor, con el fin de garantizar que los movimientos reales coincidan con los resultados obtenidos en el modelo de cinemática inversa.

Para lograr esta equivalencia, será necesario obtener y analizar ciertos parámetros característicos del servo:

- Frecuencia de trabajo: 50 Hz
- Ciclo de trabajo: 20ms
- Voltaje de operación: 4.8V -7.2V

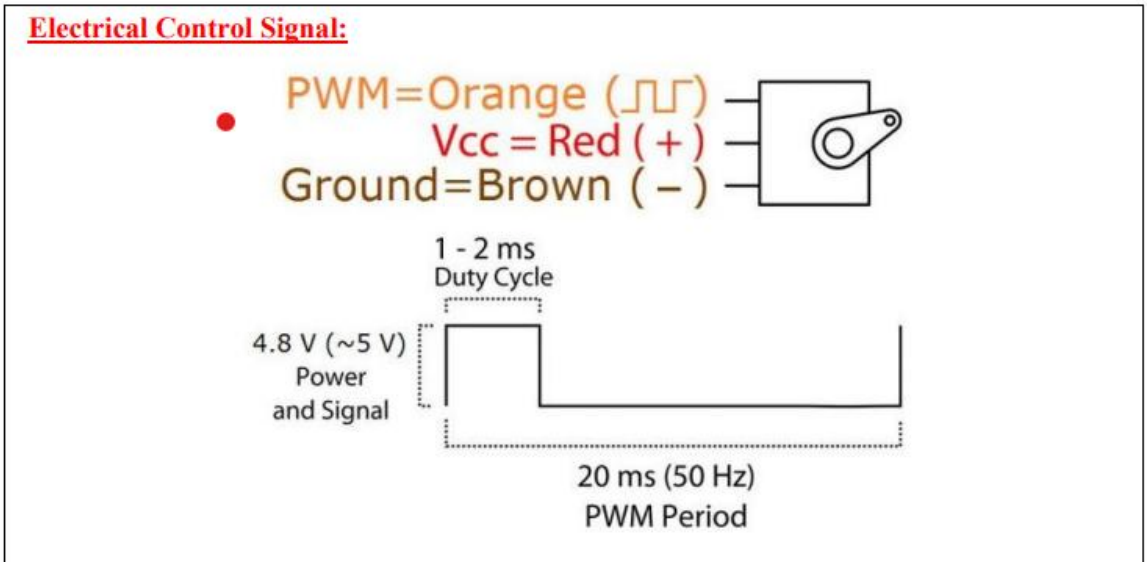


Ilustración 62- Ciclo de trabajo servo 996R[11]

Los servomotores reciben un pulso de control cada 20 ms (frecuencia de 50 Hz). De fábrica, estos dispositivos se calibran para que un pulso de 1.5 ms (1500 µs) corresponda a la posición central o 90° del recorrido. Del mismo modo, se establece que un pulso de 1.0 ms (1000 µs) representa el límite inferior, mientras que un pulso de 2.0 ms (2000 µs) corresponde al límite superior del movimiento.

Si se considera un desplazamiento total de 180°, puede asumirse una relación lineal entre el ancho del pulso y el ángulo de giro, expresada mediante la siguiente ecuación:

$$Ángulo_{deseado}(°) = \frac{Pulso_{deseado}(\mu s) - Pulso_{0°}(\mu s)}{Pulso_{180°}(\mu s)}$$

$$Ángulo_{deseado}(°) = \frac{Pulso_{deseado}(\mu s) - 1000}{1500}$$

Antes de convertir los ángulos deseados a pulsos, probaremos con ver si realmente los datos de los topes son reales, para ello usaremos la librería de arduino servo.h que nos ayudara a controlar mejor varios servos.

```
#include <Servo.h>

Servo servo1

void setup() {
  Serial.begin(9600);

  // Adjuntar cada servo a su respectivo pin
servo1.attach(8);   // si solo ponemos el puerto toma como defecto los valores 1000 , 1500 y 2000
}

void loop() {
  myservo.write(0);  // probamos con 0 y esperamos
  delay(200);

  myservo.write(90); // probamos con 90 y esperamos
  delay(2000);

  myservo.write(180); // probamos con 180 y esperamos
  delay(2000);
}
```

Al probar cada uno de los topes con los valores por defecto que toma la librería, nos podemos dar cuenta que no cuadran exactamente con lo señalado, con los ángulos de 0 y 180, el servomotor no llega a cumplir con dicho ángulo, sino que parece tener un rango más acotado, por lo que habría que intentar lo mismo con valores que nosotros mismos calculemos como topes:

Valores reales:

- 0°:600 µs
- 90°: 1450 µs
- 180°: 2300 µs

Con estos valores hay que cambiar la línea:

```
servo1.attach(8);   // si solo ponemos el puerto toma como defecto los valores 1000 , 1500 y 2000
```

por la línea:

```
servo1.attach(8,600,2300);   // valores ajustados
```

Ahora si probamos de nuevo los ángulos de 0,90,180 podemos ver que se adecuan más a los ángulos deseados

Teniendo el servomotor ya calibrado, despejaremos el pulso deseado de la ecuación, debido a que mediante la cinemática inversa ya contamos con el ángulo deseado:

$$Ángulo_{deseado}(°) = \frac{Pulso_{deseado}(\mu s) - 1000}{1500}$$

$$Pulso_{deseado}(\mu s) = \left(Ángulo_{deseado} * 1500\right) + 1000$$

4.2.4.2. Servomotor de Pinza

Para el caso de la pinza, dado que esta contaría únicamente con 2 posibles estados, abierto o cerrado, no fue usado para el cálculo de la posición final del brazo, así que su calibración consistirá en ir aumentando el ángulo hasta lograr un agarre sólido, siendo que 0° es la posición totalmente abierta de la pinza, dando como resultado final 86° como un ángulo de rotación del servomotor que otorga suficiente firmeza en el agarre de las piezas

4.2.5. Script para mover el brazo

Ya contando con las posiciones de todas las casillas, los ángulos necesarios para la apertura y cierre de la pinza y la posición de descanso con sus respectivos ángulos, lo único que queda por hacer es crear el script que nos permita mover el brazo al escribir por el promp el movimiento en código FEN, que, recordando, consta de dos partes, casilla origen y casilla destino.

Para ello usaremos arduino, y considerando que el puerto serie de arduino es incapaz de leer otros archivos de nuestro ordenador sin tener que usar tarjetas SD, el archivo .txt con los ángulos para cada casilla será puesto como un arreglo, Para solucionar el problema del arreglo y permitir únicamente números en lugar de letras, se crea una función para transformar una casilla en su equivalente, siendo que el arreglo estaría ordenado como a1,a2,a3,…a8,b1,…b8,c1…

```
inline int equivalente(char caracterfila, char caractercolumna) {
  int fila = tolower(caracterfila) - 'a'; / 0..7
```

```
  int columna = (caractercolumna - '1'); // 0..7
  return fila * 8 + columna;// 0..63
}
```

Y para asignar a cada motor su correspondiente ángulo tendremos el siguiente código:

```
void moveToIndex(int pt) {
  if (pt < 0 || pt > 64) return;
  servo1.write(ANGLES[pt][0]);
  servo2.write(ANGLES[pt][1]);
  servo3.write(ANGLES[pt][2]);
  servo4.write(ANGLES[pt][3]);
  delay(1000);
```

En el loop leeremos el puerto serie y dividiremos el texto en 2 partes, de pt1 y pt2 para posteriormente cada uno ser una parte del movimiento de la jugada.

```
  if (Serial.available()) {
    String s = Serial.readStringUntil('\n');
    s.trim();
    if (s.length() < 4) return; // sin validaciones, si es corto no hace
nada

    char f1 = s.charAt(0);
    char r1 = s.charAt(1);
    char f2 = s.charAt(2);
    char r2 = s.charAt(3);

    int pt1 = equivalente(f1, r1);
    int pt2 = equivalente(f2, r2);
```

5. RESULTADOS

Teniendo implementada la visión artificial, la creación de una interfaz para visualizar el tablero, el cálculo de la mejor jugada posible, la simulación de los ángulos para llegar a las casillas correspondientes, la calibración de los servomotores y su funcionamiento teórico, de igual forma en la realidad al intentar ejecutar el programa el brazo es incapaz de dirigirse correctamente a las casillas de origen y/o destino, siendo que varía considerablemente por 3 casillas de distancia de la deseada, si bien es cierto que se podría solucionar este problema al ajustar los ángulos, la realidad es que por el propio peso del brazo este varia su posición de una manera diferente para cada casilla, por lo que cada ajuste debería hacerse de manera individual para cada casilla, además hay zonas del brazo que no pueden ser correctamente ajustadas por deficiencias en el material que no permite un ajuste idóneo, además el cálculo de las distancias resulta complicado, ya que a pesar de colocarle paso a la base del brazo igualmente se movía haciendo que con cada jugada sea mayor la desviación.

6. CONCLUSIONES

En cuanto a las conclusiones, de los 4 objetivos principales planteados, se ha logrado cumplir satisfactoriamente los 2 primeros, siendo estos de:

- Implementación de visión artificial para detección de piezas y tablero (al menos el 80% de precisión).
- Implementación de un motor de ajedrez para detectar las mejores jugadas posibles.

Con respecto a los otros dos objetivos, uno ha sido parcialmente completado:

- Tiempo de respuesta máximo por jugada de 15 segundos

Dado que el tiempo entre el cálculo de la mejor jugada y el movimiento del brazo es menor a los 15 segundos, nuestro tiempo de respuesta está dentro del rango, sin embargo, aquí viene el último punto:

- Precisión en la colocación de las piezas de, al menos, el 80%.

Al ser la precisión de colocación de las piezas muy baja, dado que en muy pocas ocasiones las piezas terminan en las posiciones deseadas, es correcto señalar el fallo en este objetivo.

En cuanto a las limitaciones encontradas, principalmente en la parte mecánica, se podrían enumerar algunos factores que hicieron difícil la realización del movimiento del brazo físicamente, si bien es cierto en las simulaciones el brazo podía llegar a las zonas deseadas sin problema alguno, la simulación no toma en cuenta factores tales como la esbeltez del brazo que aunado con el peso del propio brazo desvían centímetros en cada articulación, que al ser un brazo de 4 grados de libertad van sumando desviaciones que resultan imposibles de predecir, además tomando en consideración que el tamaño de una casilla no superaba los 2cm de lado, tendríamos que operar con una motricidad extrafina para lograr el correcto movimiento de las piezas.

Un posible futuro trabajo de fin de grado tomando como base lo realizado en este trabajo, y el cual me resultaría interesante de observar, seria implementar el sistema de visión artificial y la implementación del motor de ajedrez, en un brazo de diseño propio, ya que cuestiones como los juegos y ajustes podrían ser tratados correctamente, evitando de esta manera oscilaciones, o en todo caso disminuir los grados de libertad del robot a simplemente 2 grados de libertad de los cuales dependa la posición del actuador, simplificando significativamente la dificultad de cálculos,

7. OBJETIVOS DE DESARROLLO SOSTENIBLE

Los objetivos de este Trabajo Fin de Grado están alineados con los siguientes Objetivos de Desarrollo Sostenible (ODS) y metas, de la Agenda 2030:

- Objetivo 4 - Garantizar una educación inclusiva y equitativa de calidad y promover oportunidades de aprendizaje permanente para todos
- Meta 4.a - Construir y adecuar instalaciones educativas que tengan en cuenta las necesidades de los niños y las personas con discapacidad y las diferencias de género, y que ofrezcan entornos de aprendizaje seguros, no violentos, inclusivos y eficaces para todos

- Objetivo 9 - Construir infraestructuras resilientes, promover la industrialización sostenible y fomentar la innovación
- Meta 9.c - Aumentar significativamente el acceso a la tecnología de la información y las comunicaciones y esforzarse por proporcionar acceso universal y asequible a Internet en los países menos adelantados de aquí a 2020

8. BIBLIOGRAFÍA

[1] Meneces, R. & Maia, H (2023)*Vista do an intelligent chess piece detection tool*. Department of Computer Engineering and Automation UFRN, Natal-RN, Brazil https://sol.sbc.org.br/index.php/semish/article/view/25062/24883

[2] Chen, A. T.-Y., & Wang, K. I.-K. (2019). Robust Computer Vision Chess Analysis and Interaction with a Humanoid Robot. Computers, 8(1), 14. https://doi.org/10.3390/computers8010014

[3] T.D. Phuc, B.C. Son (2025)Development of an autonomous chess robot system using computer vision and deep learning Results Eng., 25 , https://www.sciencedirect.com/science/article/pii/S2590123025001793

[4] FEN (Forsyth-Edwards notation) - Chess Terms. Chess.com. https://www.chess.com/terms/fen-chess

[5] Brazo Robótico https://es.aliexpress.com/item/1005005352898104.html

[6] Servomotor https://es.aliexpress.com/item/1005009351850813.html

[7] Arduino Mega https://store.arduino.cc/products/arduino-mega-2560-rev3

[8] Cámara https://www.pccomponentes.com/aukey-webcam-fullhd-usb?srsltid=AfmBOoqSNmxxuh9ydpxU7ZdbawJkqst1WexSdrdzlql6hfCbS4m15_lJ

[9] Vieira, D. (2024, abril 15). Python: ¿Qué es y cómo se usa este lenguaje de programación? Hostgator.mx. https://www.hostgator.mx/blog/python-como-usa-lenguaje-programacion/

[9] Juras, E (2025 Technology Consultants. Train YOLO Models in Google Colab https://colab.research.google.com/github/EdjeElectronics/Train-and-Deploy-YOLO-Models/blob/main/Train_YOLO_Models.ipynb

[10] Github Lichess https://github.com/lichess-org/lila

[11] Handson Technology, MG996R Metal Gear Servo Motor https://www.handsontec.com/dataspecs/motor_fan/MG996R.pdf

Relación de documentos

(X) Memoria 71 páginas

(_) Anexos 8 páginas

La Almunia, a 10 de noviembre de 2025

Firmado: Gustavo Chen Fernández