UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ETSI Aeroespacial y Diseño Industrial

# UNIVERSITAT POLITÈCNICA DE VALÈNCIA

## Escuela Técnica Superior de Ingeniería Aeroespacial y Diseño Industrial

## Diseño, simulación e implementación de un robot para resolver el cubo de Rubik

Trabajo Fin de Máster

Máster Universitario en Ingeniería Mecatrónica

AUTOR/A: Martínez Olmos, Sergio

Tutor/a: Casanova Calvo, Vicente Fermín

CURSO ACADÉMICO: 2023/2024

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ETSI Aeroespacial y Diseño Industrial

# UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Escuela Técnica Superior de Ingeniería Aeroespacial y Diseño Industrial**

## DISEÑO, SIMULACIÓN E IMPLEMENTACIÓN DE UN ROBOT PARA RESOLVER EL CUBO DE RUBIK

## PROYECTO FINAL DE MÁSTER

*TRABAJO FINAL DEL*

Máster en Ingeniería Mecatrónica

*REALIZADO POR*

**Sergio Martínez Olmos**

*TUTORIZADO POR*

**Vicente Fermín Casanova Calvo**

**CURSO ACADÉMICO: 2023/2024**

# Resumen

En el presente trabajo se diseñará un robot capaz de resolver el cubo de Rubik desde cualquier configuración inicial. Para ello se utilizará el diseño 3D, para su posterior impresión y montaje; implementación de un circuito electrónico que alimente y controle los 6 motores que girarán cada una de las caras del cubo; así como una simulación del funcionamiento del robot y el diseño de una interfaz de usuario para manejarlo. El modelo de simulación del mecanismo se realizará empleando la librería Simscape Multibody de Matlab, a partir del conjunto de piezas diseñadas empleando SolidWorks. Para el control del prototipo real se empleará Arduino, que dará las ordenes apropiadas a las articulaciones a partir de la implementación de un algoritmo de resolución del cubo de Rubik.

**Palabras clave**: Simscape Multibody; Arduino; Cubo de Rubik

# Resum

Al present treball es dissenyarà un robot capaç de resoldre el cub de Rubik des de qualsevol configuració inicial. Per a això s'utilitzarà el disseny 3D, per a la seua posterior impressió i muntatge; implementació d'un circuit electrònic que alimente i controle els 6 motors que giraran cadascuna de les cares del cub; així com una simulació del funcionament del robot i el disseny d'una interfície d'usuari per a manejar-lo. El model de simulació del mecanisme es realitzarà emprant la llibreria Simscape Multibody de Matlab, a partir del conjunt de peces dissenyades emprant SolidWorks. Per al control del prototip real s'emprarà Arduino, que donarà les ordres apropiades a les articulacions a partir de la implementació d'un algorisme de resolució del cub de Rubik.

**Paraules clau**: Simscape Multibody; Arduino; Cub de Rubik

# Abstract

In this work we will design a robot capable of solving Rubik's cube from any initial configuration. For this purpose, the 3D design will be used, for its printing and assembly; implementation of an electronic circuit that feeds and controls the 6 motors that will rotate each of the faces of the cube; as well as a simulation of the movement of the robot and the design of a user interface to operate it. The simulation model of the mechanism will be made using the Simscape Multibody library of Matlab, from a set of parts designed using SolidWorks. Arduino will be used for the control of the real prototype, which will give the appropriate orders to the joints from the implementation of a Rubik's cube solving algorithm.

**Key words**: Simscape Multibody; Arduino; Rubik’s Cube

# Índice de contenido

# Índice de imágenes

# Índice de tablas



# Índice de gráficas



# Índice de planos

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ETSI Aeroespacial y Diseño Industrial

# UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Escuela Técnica Superior de Ingeniería Aeroespacial y Diseño Industrial**

## DISEÑO, SIMULACIÓN E IMPLEMENTACIÓN DE UN ROBOT PARA RESOLVER EL CUBO DE RUBIK

## DOCUMENTO I: MEMORIA TÉCNICA

*TRABAJO FINAL DEL*

**Máster en Ingeniería Mecatrónica**

*REALIZADO POR*

**Sergio Martínez Olmos**

*TUTORIZADO POR*

**Vicente Fermín Casanova Calvo**

**CURSO ACADÉMICO: 2023/2024**

# Índice de la memoria técnica

# 1 Introducción

El cubo de Rubik, inventado por el escultor y profesor de arquitectura Ernő Rubik en 1974, es uno de los rompecabezas mecánicos más emblemáticos de nuestra era. Desde su comercialización en 1980, ha fascinado a millones de personas, convirtiéndose en un fenómeno cultural y un desafío intelectual. En la actualidad, se realizan campeonatos que ponen a prueba la destreza y velocidad de los participantes en la resolución de este; el tiempo récord se sitúa en los 3.13 segundos, conseguido por el estadounidense Max Park en junio de 2023.

Con el objetivo de medir el potencial que puede lograr una máquina, se crean robots que son capaces de resolver el cubo en tiempos muy reducidos. El tiempo récord actual de resolución mediante un robot lo sostienen Jared Di Carlo y Ben Katz, con la cifra de 0.38 segundos, conseguido en 2018. También existe una versión comercial de un robot de resolución (GAN Robot), que promete resolver cualquier mezcla en menos de 15 segundos.

Además, uno de los puntos clave es el algoritmo de resolución que se utiliza. El cubo se puede analizar y resolver utilizando principios matemáticos, principalmente de teoría de grupos, combinatoria y algoritmos. Para hacerse una idea, el conjunto de todas las posibles configuraciones del cubo forma un grupo, este consta de 43 trillones de posibles combinaciones, aproximadamente, para un cubo de 3x3x3. La combinatoria se usa para calcular el número total de posibles configuraciones del cubo y diseñar algoritmos que reduzcan el número de movimientos necesarios para resolverlo. El algoritmo más eficaz es el denominado Algoritmo de Dios, que se refiere al número máximo de movimientos necesarios para resolver el cubo desde cualquier configuración posible; en este caso es el número 20, cualquier configuración del cubo podrá resolverse en 20 movimientos o menos.

Todas las máquinas de resolución tienen puntos en común:

- Consta de una estructura que soporta los motores que permitirán girar las caras del cubo.
- La unión entre el motor y el cubo se realiza mediante una pieza que haga de nexo entre el eje del motor y la pieza central de la cara del cubo.
- Se utiliza un controlador para coordinar el funcionamiento de todos los motores.
- Se hace uso de visión artificial para informar al controlador de la configuración inicial de la que parte el cubo.
- Se utiliza el algoritmo de Kociemba para la resolución.

Por lo tanto, el prototipo diseñado cumplirá en gran parte con todas estas características. Usaremos los materiales y técnicas más económicas que estén a nuestro alcance, así que deberemos tener en cuenta las limitaciones que esto supone. La velocidad de resolución no podrá acercarse a las máquinas más caras, pero el tiempo de resolución será menor al del ser humano medio. El control de motores también es limitado, ya que el precio de seis servomotores con la suficiente potencia y precisión, contando con sensores de posición para una mejor regulación, es bastante elevado.

En resumen, será un prototipo económico, que usará materiales reciclados en su estructura. El propósito de este será usarlo como ejercicio docente, ya que toca varios aspectos relacionados con la Mecatrónica, como pueden ser el diseño 3D, la simulación de un mecanismo, control mediante software, comunicación entre dispositivos, aspectos electrónicos y mecánicos.

## 1.1 Objetivos

El objetivo principal del presente trabajo es la implementación de una máquina capaz de resolver el cubo de Rubik de 3x3x3 de forma automática en el menor tiempo posible. Para ello se deberá conseguir un diseño 3D óptimo de una estructura rígida; una simulación del funcionamiento del robot lo más cercana a la realidad; controlar los motores del prototipo a través de una interfaz de usuario sencilla e intuitiva. Todo ello consiguiendo el presupuesto más económico posible.

## 1.2 Motivación o justificación

La construcción de este prototipo viene motivada en gran parte por desarrollar habilidades y conocimientos de algunas áreas de la Mecatrónica. En particular:

- Simular un sistema mecánico complejo, empleando un módulo de Simulink que no se ha estudiado a lo largo del máster.
- Diseñar toda una estructura que albergue los seis motores, que sea rígida y con medidas precisas, supone todo un reto y estimulación de poder trasladar un diseño teórico a la práctica.
- Tener la oportunidad de ampliar el conocimiento de Arduino y poder emplear otro lenguaje de programación tan útil como lo es Python, así como conseguir conectar ambos entornos y poder manejar el prototipo desde una interfaz de usuario.
- Atracción personal por el cubo de Rubik y los métodos de resolución de puzles/rompecabezas.

# 2 Marco teórico

## 2.1 Notación del cubo de Rubik

La notación es un sistema estándar para describir los movimientos que se deben realizar en el cubo para resolverlo o para describir secuencias de movimientos. La más común es la llamada “notación de Singmaster”, que se utiliza en la mayoría de los tutoriales y guías avanzadas de resolución.

En esta, cada cara del cubo se denota con una letra, que es la inicial de la palabra en inglés, que la define:

- **U** (Up): la cara superior, pieza central de color blanco.
- **D** (Down): la cara inferior, pieza central de color amarillo.
- **L** (Left): la cara izquierda, pieza central de color naranja.
- **R** (Right): la cara derecha, pieza central de color rojo.
- **F** (Front): la cara frontal, pieza central de color verde.
- **B** (Back): la cara trasera, pieza central de color azul.

En cuanto a los movimientos básicos, cuando se coloca una letra sola, implica girar una cara 90 grados en el sentido de las agujas del reloj, siendo la vista de la persona mirando a esa cara. Cuando la letra va acompañada por un apóstrofe, el giro será de 90 grados también, pero en sentido contrario. El movimiento doble implica girar la cara 180 grados y se denota con el número 2 seguido de la letra.

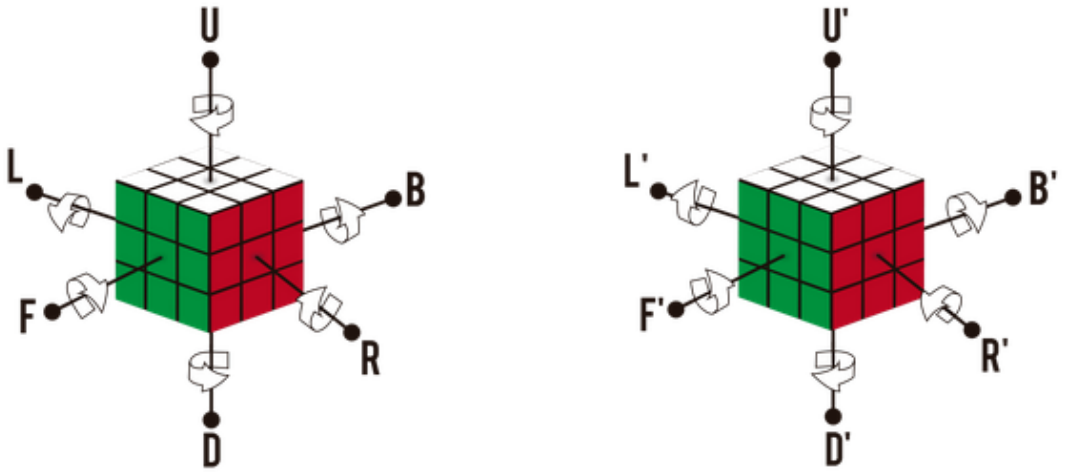



*Imagen 1. Notación del cubo de Rubik.*

Así, un ejemplo de una secuencia básica podría ser: R U R’ B2. Los giros a realizar serían: 90 grados en sentido horario de la cara derecha (centro rojo), 90 grados en sentido horario de la cara superior (centro blanco), 90 grados en sentido antihorario de la cara derecha y 180 grados de la cara trasera (centro azul).

## 2.2 Algoritmo de resolución

Como ya se ha comentado, el uso de un algoritmo de resolución eficiente es un punto clave. Los algoritmos descomponen el complejo problema del cubo en pasos manejables, proporcionando una metodología eficiente a los usuarios. Por ejemplo, muchos *speedcubers* emplean el método de Fridrich, diseñado para minimizar el número de movimientos necesarios para resolver el cubo.

Pero, pudiendo usar un ordenador, necesitamos encontrar el algoritmo más potente entre ellos. Desde la aparición del famoso rompecabezas, cantidad de investigadores se preguntaban cuál sería el máximo número de movimientos que se necesitan para resolver el cubo. Trataban de encontrar el algoritmo que resolviera el cubo en el menor número de pasos posibles y a este, se le denominó Algoritmo de Dios, llamando así al número máximo de movimientos necesarios Número de Dios. Se tardaron 35 años en demostrar que dicho número es exactamente 20, a manos de Tomas Rokicki, Herbert Kociemba, Morley Davidson y John Dethridge, en 2010. Estos resolvieron cada posición del Cubo de Rubik con ayuda de computadoras donadas por Google, demostrando que ninguna configuración inicial requiere más de veinte movimientos.

Este complejo algoritmo se divide en dos fases principales:

- Fase 1. En esta, el objetivo es llevar el cubo a un estado llamado “subgrupo G1”. En este, deben cumplirse las siguientes condiciones: todos los bordes están orientados correctamente, las esquinas pueden estar desorientadas, pero no intercambiadas de posición y los bordes se encuentran en el subgrupo de posiciones restringidas.
- Fase 2. Una vez que el cubo se encuentre en el subgrupo G1, se resuelve completamente el cubo, usando algoritmos sistemáticos.

Este algoritmo se implemente en programas informáticos que pueden calcular la solución rápidamente, lo que lo hace ideal para aplicaciones de software y robots que resuelven el cubo. La aplicación de este algoritmo implica:

- Representación del estado del cubo. Este se representa en términos de sus caras y colores, convirtiéndose en una estructura de datos que el algoritmo puede manipular.
- Búsqueda en espacio de estados. Se utiliza una búsqueda en el espacio de estados del cubo, empleando estrategias para guiar la búsqueda hacia la solución.
- Uso de tablas de movimiento. Se usan tablas precomputadas que indican cómo mover el cubo desde una configuración dada a otra deseada, lo que acelera el proceso de búsqueda.

## 2.3 Impresión 3D

La impresión 3D es una tecnología de fabricación aditiva que crea objetos tridimensionales a partir de modelos digitales. Funciona añadiendo material capa por capa hasta que se forma el objeto completo. Esta tecnología se ha vuelto muy popular debido a su versatilidad y capacidad para crear formas complejas que serían difíciles o imposibles de fabricar con métodos tradicionales.


*Imagen 2. Impresora 3D.*

Existen distintos materiales usados en el mundo de la impresión 3D. Estos son manufacturados en forma de filamento, enrollados sobre una bobina. Las opciones más asequibles serían:

- PLA (Ácido Poliláctico). Un material biodegradable y fácil de imprimir. Además, es ecológico, es necesaria una baja temperatura de impresión y cuenta con un *warping* mínimo (cuando un objeto se retuerce, perdiendo su forma). El precio oscila entre los 15 y los 25€ el kg.
- ABS (Acrilonitrilo Butadieno Estireno). Plástico resistente y duradero. Utilizado comúnmente en productos de consumo. Este soporta altas temperaturas. El precio es similar al PLA, siendo un poco más caro.
- PETG (Polietileno Tereftalato Glicol). Material resistente y fácil de imprimir, combina propiedades del PLA y ABS. Buena resistencia mecánica y química, poco *warping* también. Precio similar a los anteriores.

## 2.4 Cortado láser

El cortado láser es una tecnología de fabricación que utiliza un rayo láser concentrado para cortar o grabar materiales con alta precisión. Este tiene una resolución de corte que puede ser de milésimas de milímetro, así, la focalización del láser permite cortes muy finos y detalles complejos. Además, el corte es generalmente rápido, especialmente para materiales delgados. En cuanto a la calidad de corte, esta tecnología produce bordes limpios y suaves con rebabas mínimas, reduciendo la necesidad de posprocesado.

*Imagen 3. Cortado láser.*

Entre los materiales que pueden usarse en el corte láser, destacan el acero inoxidable, el aluminio, el latón, varios tipos de madera y el acrílico.

## 2.5 Conexión serial

El puerto serie (también conocido como comunicación serial) es una interfaz de comunicación que permite la transferencia de datos entre un ordenador y un dispositivo externo. Es uno de los métodos más comunes para la comunicación entre dispositivos debido a su simplicidad y eficacia.

Se transmite la información de forma secuencial, es decir, bit a bit, se transmiten datos enviando un solo bit a la vez a través de un solo canal. Esta comunicación es asincrónica, lo que significa que no se utiliza una señal de reloj separada. En lugar de eso, ambos dispositivos acuerdan una velocidad de transmisión (la denominada *baud rate*), que es el número de bits transmitidos por segundo. Las tasas de baudios típicas son: 9600 bps para aparatos electrodomésticos, 19200 bps para dispositivos industriales, 38400 bps para militares, 115200 bps para productos sanitarios.

En el contexto de comunicación con Arduino, nuestro caso, suele referirse a la comunicación UART (*Universal Asynchronous Receiver/Transmitter*). El puerto de Arduino TX (*Transmitter*), se encarga de enviar los datos; mientras que el RX (*Receiver*), se encarga de recibirlos.

# 3 Producto

## 3.1 Diseño 3D

Para la correcta sujeción de los motores, se diseñará una estructura rígida mediante la creación y posterior unión de varias piezas. Para ello se opta por utilizar el software SolidWorks, que nos permite realizar el diseño 3D de las piezas por separado y además realizar ensamblajes de estas, así como crear planos acotados. Además, existen miles de archivos en la red de motores idénticos a los nuestros, importables al programa, que nos ayudarán en gran manera a realizar un diseño más preciso.

### *3.1.1 Estructura primer diseño*

Sabiendo las medidas exactas del motor, se procede a la creación de la base de la estructura (Imagen 4). Se realiza en forma poligonal con cuatro pilares que apoyarán la estructura superior y unas paredes que sujetan el cuerpo del motor, con un hueco para la salida de los cables.

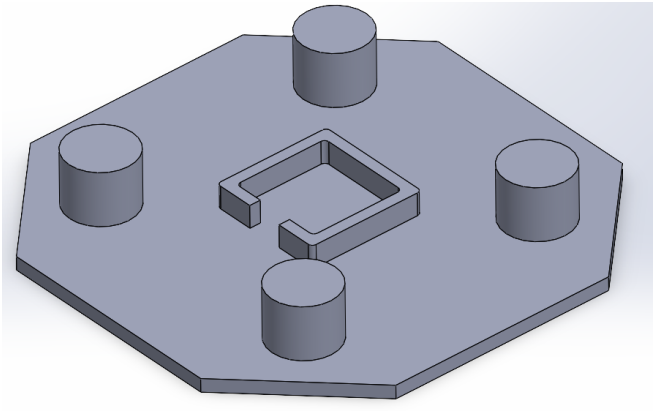


*Imagen 4. Base 1er diseño.*

La estructura central (Imagen 5) debe tener un espacio en su centro por el que pase el motor inferior. Además, cuenta con cuatro soportes con un pequeño pasaje saliente para encajar las piezas que soportarán los cuatro motores laterales.

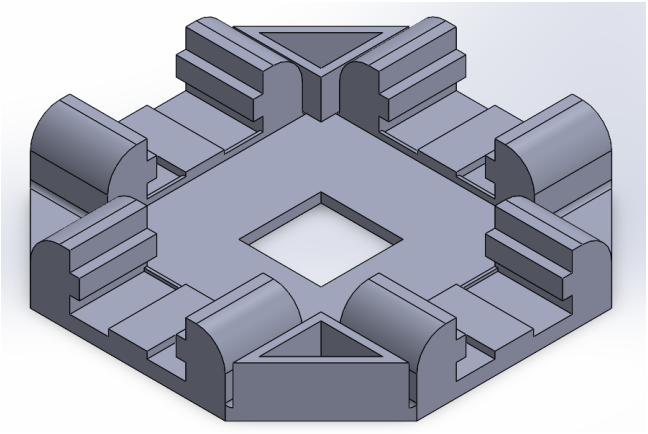


*Imagen 5. Estructura central 1er diseño.*

La pieza lateral (Imagen 6) podrá ajustarse gracias a las muescas en sus laterales inferiores, desplazándose hacia delante y atrás, sirviendo también para poder introducir y sacar el cubo.

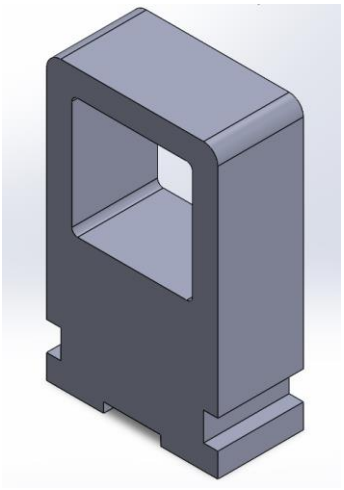


*Imagen 6. Soporte motor lateral 1er diseño.*

De igual manera, en las dos diagonales de la estructura central se diseña un hueco triangular que albergará la estructura superior (Imagen 7). Esta estructura sujetará el motor de la cara superior.

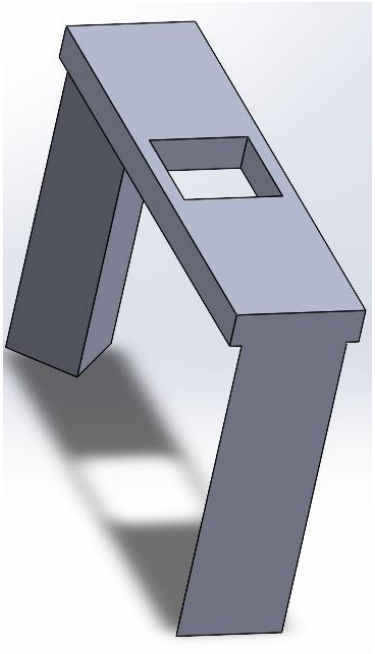


*Imagen 7. Estructura motor superior 1er diseño.*

Por último, es necesaria una pieza que sirva de unión entre el eje del motor y la cara del cubo. Para ello se diseña por un extremo un cilindro que encaje muy justo en el eje del motor, y por el otro extremo una estructura que encaje en la pieza central de la cara del cubo, de forma que quede encajada y giren las 3 partes solidarias.

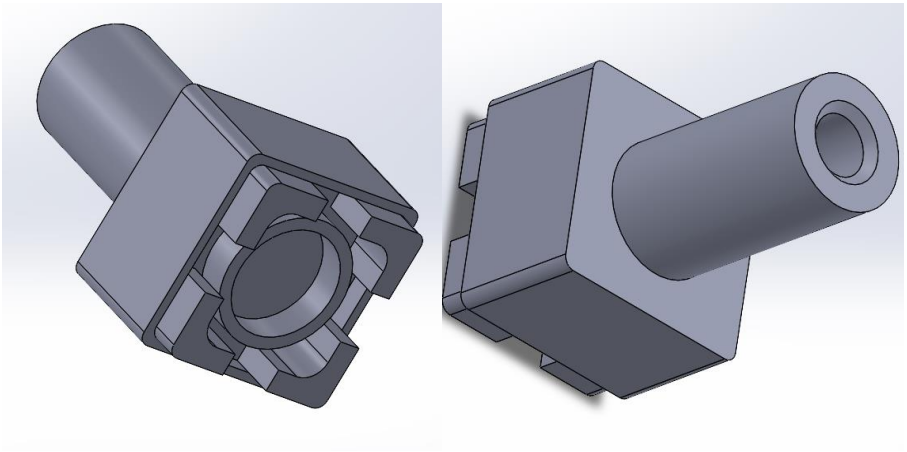


*Imagen 8. Vistas pieza de agarre 1er diseño.*

A falta del diseño de las piezas que fijen los motores laterales a la pieza de soporte lateral y una pieza que fije el motor superior con la pieza de soporte superior, se realiza el ensamblaje de las piezas, para cuadrar y revisar medidas en las piezas, quedando un resultado final ilustrado en la Imagen 9.

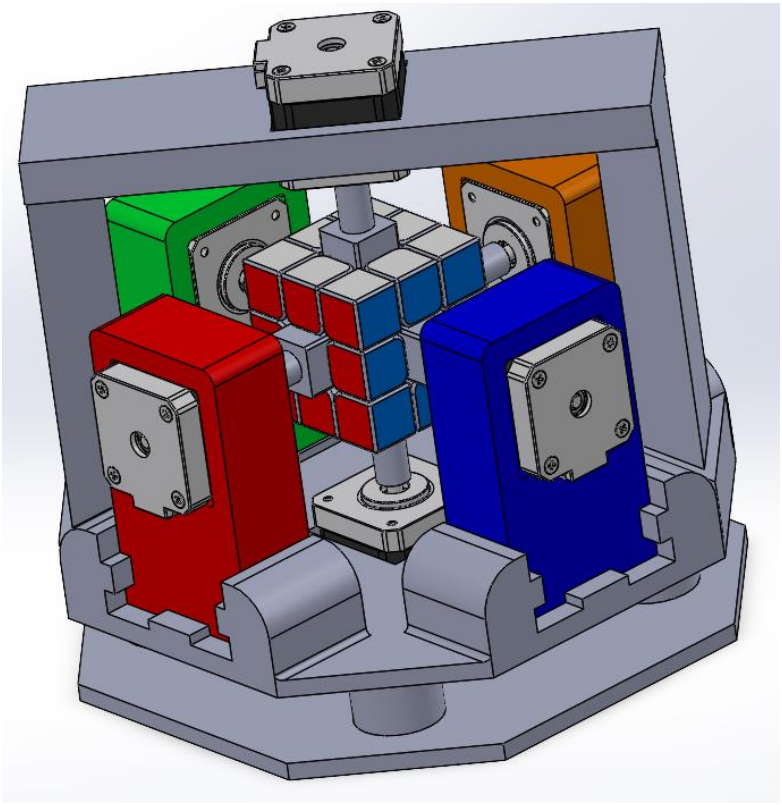


*Imagen 9. Ensamblaje 1er diseño.*

### *3.1.2 Discusión*

Para la impresión de nuestro primer diseño el material ideal sería el PLA, ya que necesitamos que los soportes sean resistentes y rígidos. El problema es el tamaño y

volumen de las piezas, sobre todo de las bases; estas tardarían un tiempo considerable en fabricarse.

Otra solución sería hacer un diseño con piezas planas, que puedan unirse entre ellas, de manera que formen estructuras más complejas. De este modo podemos usar la técnica de cortado láser, usando una madera de contrachapado, ahorrando dinero en material y ganando una mayor velocidad de producción. Además, esta técnica realiza las piezas con mayor precisión, evitando los problemas de rediseño que implicaría la impresora 3D, ya que las tolerancias son mucho mayores.

Es por ello por lo que se decide efectuar un segundo diseño, teniendo en cuenta que todas las piezas creadas para la estructura de la máquina deben ser planas. No obstante, la pieza de agarre entre el motor y la cara del cubo sí debe ser impresa en 3D, ya que es necesario un diseño de formas más complejas, que con el corte láser no podemos conseguir. Además, estas son las piezas más pequeñas, por lo que el tiempo de producción es mínimo.

### *3.1.3 Estructura segundo diseño*

Tras recibir la información de los materiales disponibles en la facultad, se opta por realizar el diseño con madera de contrachapado de abedul, teniendo en cuenta que los grosores disponibles son de 3 y de 6 mm. Este segundo diseño se realiza con mayor soltura, ya que podemos utilizar las medidas que habíamos usado en el primero.

En primer lugar, las dos piezas que hacen de base se diseñan con pequeños huecos que servirán para acoplar otras piezas con pequeñas muescas que encajen. Para la unión entre ambas bases se diseñan unos soportes rectangulares a modo de columnas. Estas piezas se diseñan con un grosor de 6 mm, ya que son las encargadas de sostener toda la estructura, así como el peso del cubo y los motores; el resto de las piezas será de 3 mm.

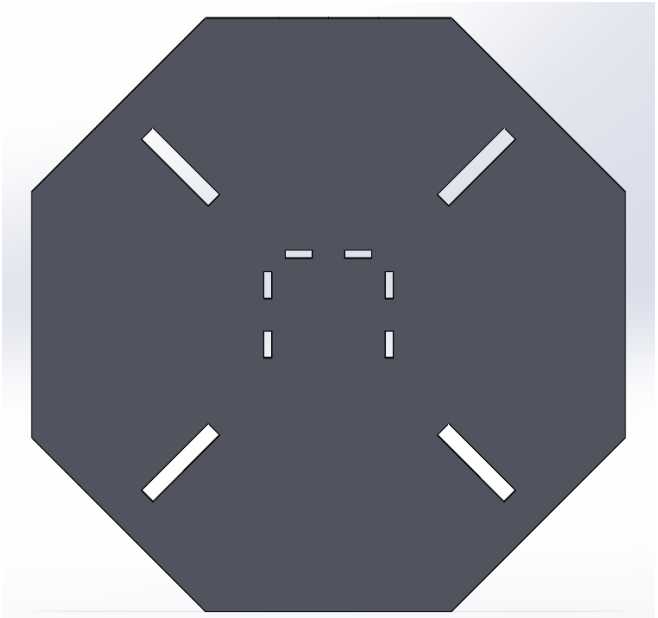


*Imagen 10. Base 2o diseño.*

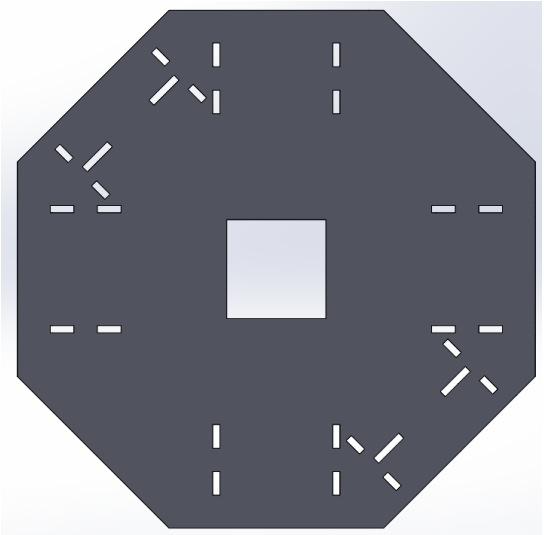


*Imagen 11. Estructura central 2o diseño.*

Se crean 3 paredes que encajarán en la base inferior y sujetarán al motor inferior, de modo similar al primer diseño.

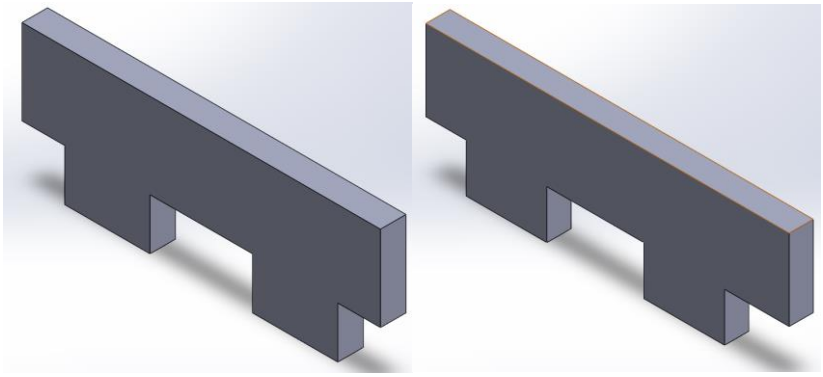


*Imagen 12. Paredes motor inferior 2o diseño.*

Para los cuatro motores laterales se diseñan cuatro piezas distintas. En primer lugar, creamos las piezas que se unen a la base mediante dos patas inferiores (Imagen 13). La siguiente pieza serán las paredes laterales (Imagen 14), que quedarán fijas con la pieza anterior mediante dos pasadores cilíndricos.

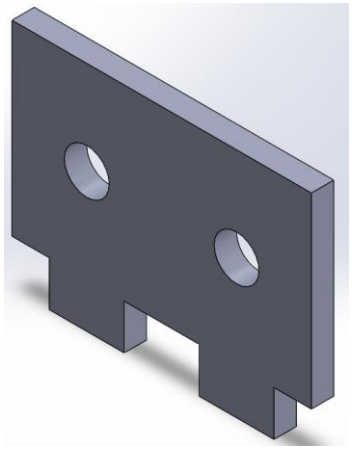


*Imagen 13. Sujeción pared lateral 2o diseño.*

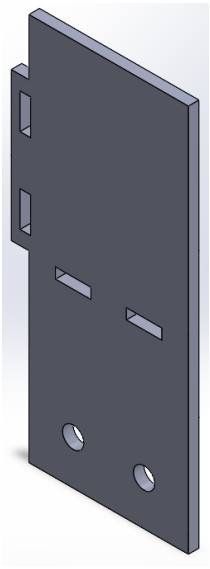


*Imagen 14. Pared motor lateral 2o diseño.*

A su vez, se diseña una pieza que se une con las dos paredes laterales mediante muescas (Imagen 15) y hace de apoyo para los motores laterales. Finalmente, se diseña una especie de "tapa" (Imagen 16), que se une también a las paredes laterales mediante muescas y cuenta con 4 agujeros para ser atornillado al motor, de la misma métrica que este.

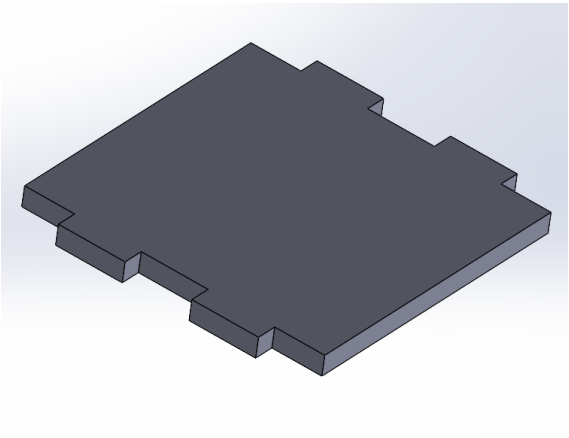


*Imagen 15. Soporte motor lateral 2o diseño.*

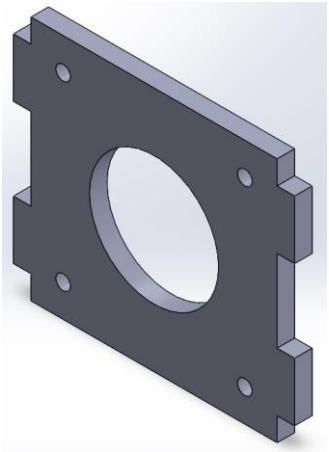


*Imagen 16. Tapa motor lateral 2o diseño.*

El montaje final, con uno de los conjuntos laterales montados, se muestra en la Imagen 17.

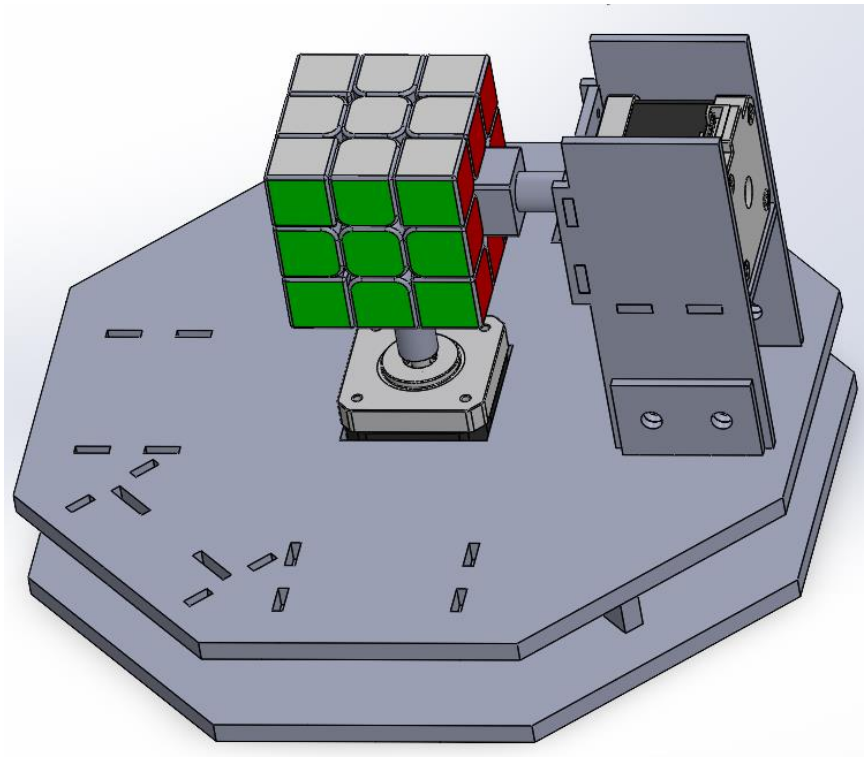


*Imagen 17. Ensamblaje estructura lateral 2o diseño.*

Por último, para sujetar el motor superior se diseña un total de cuatro piezas diferentes. La primera pieza (Imagen 18) se une a la base de forma diagonal mediante unas patas inferiores. Se colocarán dos unidades, que serán unidas mediante las piezas de la Imagen 19 y 20, mediante las muescas de sus costados. Además, la pieza de la Imagen 19 se unirá a su vez a la base mediante sus patas inferiores.

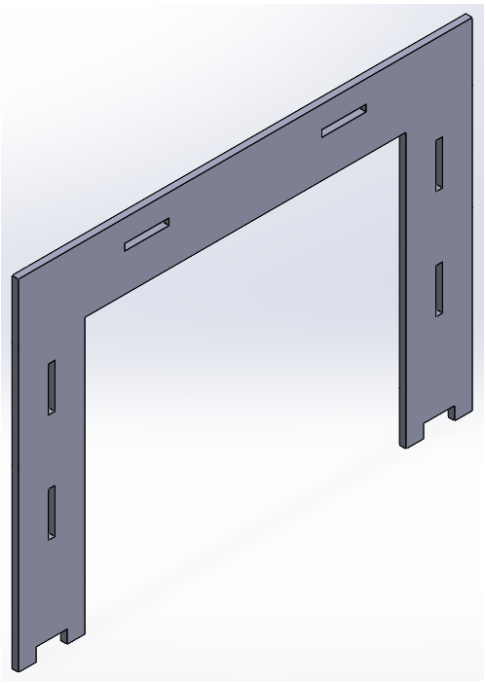


*Imagen 18. Pared motor superior 2o diseño.*

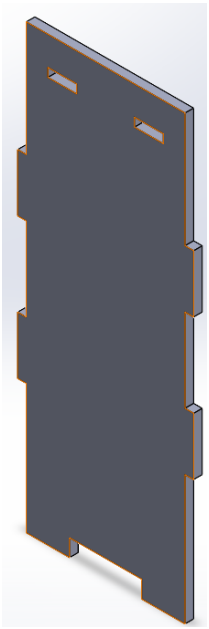


*Imagen 19. Pieza de unión paredes laterales 2o diseño.*

La tercera pieza (Imagen 20) cuenta con un hueco en su centro por el que pasará el motor superior. Esta se une a las dos piezas anteriores mediante sus muescas. Además, se diseña una tapa (Imagen 21) que se atornillará a la tercera pieza, sujetando el motor y dejando pasar el eje por el hueco circular en su centro.

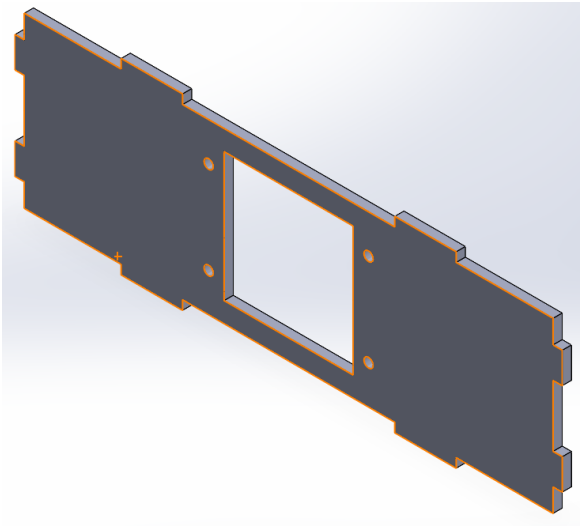


*Imagen 20. Soporte motor superior 2o diseño.*

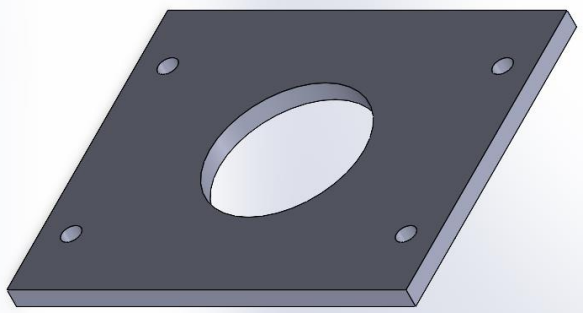


*Imagen 21. Tapa motor superior 2o diseño.*

El montaje final del conjunto del motor superior se muestra en la Imagen 22.

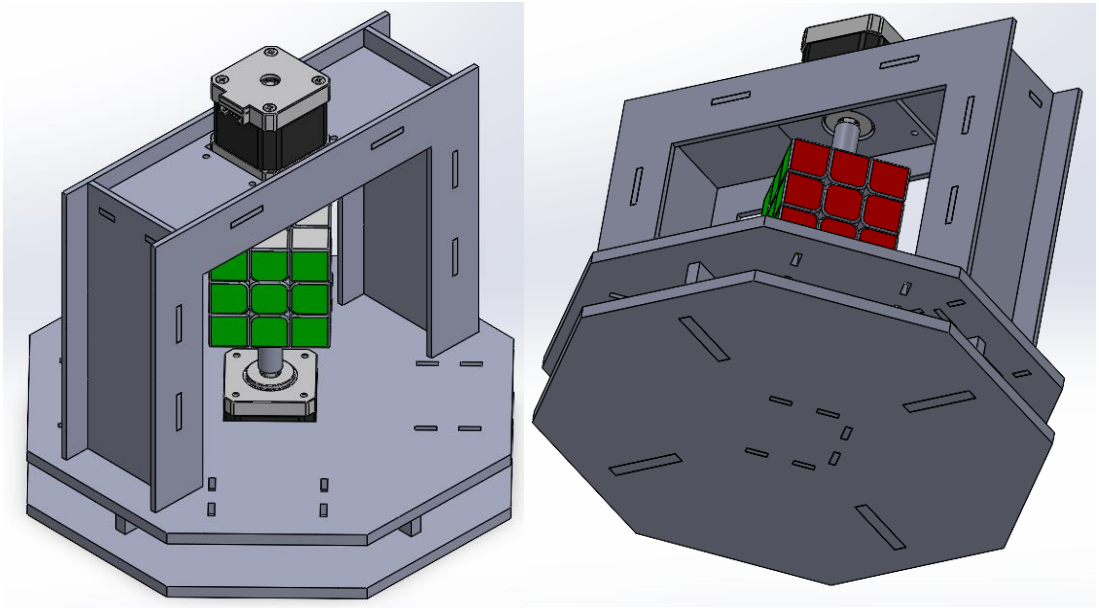


*Imagen 22. Vistas ensamblaje estructura superior 2o diseño.*

### *3.1.4 Diseño nuevo agarre*

Ya que las piezas laterales del segundo diseño no pueden desplazarse, con el objetivo de que estas queden fijas en el mecanismo se diseña un nuevo agarre. Este consta de dos piezas, la primera (Imagen 23) es un prisma rectangular recto, que cuenta con un entrante en su centro con el perfil del eje del motor, en el que irá metido, quedando ajustado a él.

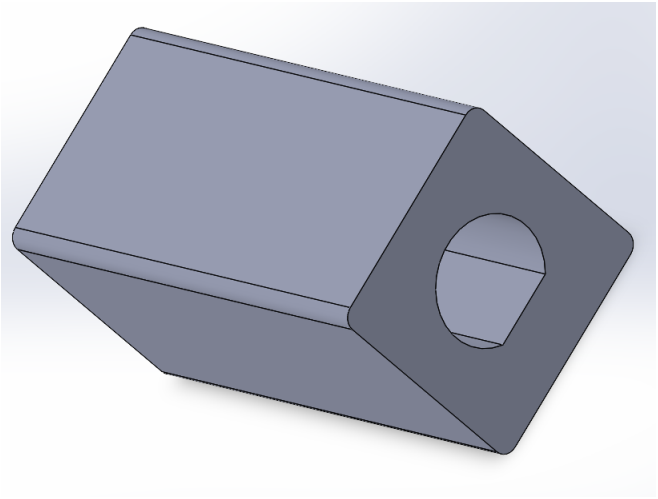


*Imagen 23. Agarre eje motor 2o diseño.*

La segunda pieza quedará ajustada al centro de la cara del cubo mediante una forma de cruz, cogiendo las cuatro paredes de la pieza central. Ambas piezas se conectarán mediante encaje. Se diseñan dos tipos distintos para esta pieza. La Imagen 24 muestra la pieza que estará presente en las caras laterales del cubo. Gracias a su forma de U por el extremo que engancha con la pieza del eje del motor, el cubo puede sacarse y meterse (Imagen 25) teniendo las paredes laterales con los motores fijos en la estructura.

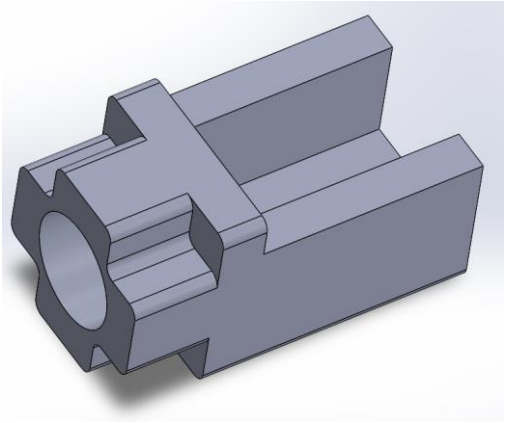


*Imagen 24. Agarre caras laterales 2o diseño.*

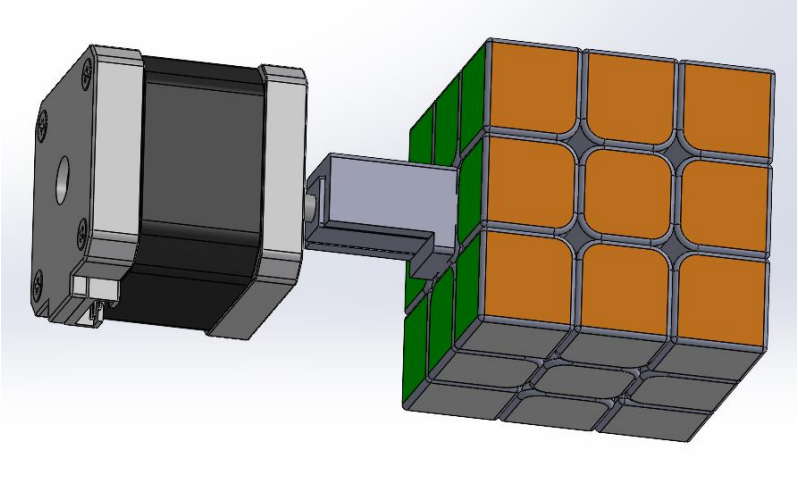


*Imagen 25. Ensamblaje motor lateral con agarre 2o diseño.*

Por otro lado, la Imagen 26 muestra la pieza que conectará las caras superior e inferior con el eje de los motores. Estas tendrán también forma de prisma rectangular recto, con un entrante con el perfil de la pieza que se conecta al eje, permitiendo así que entre y salga, quedando ajustada.

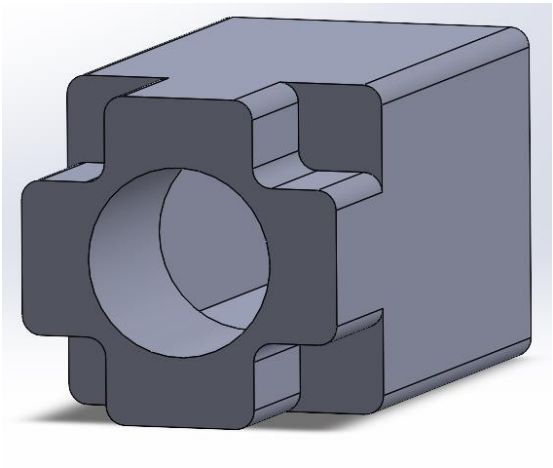
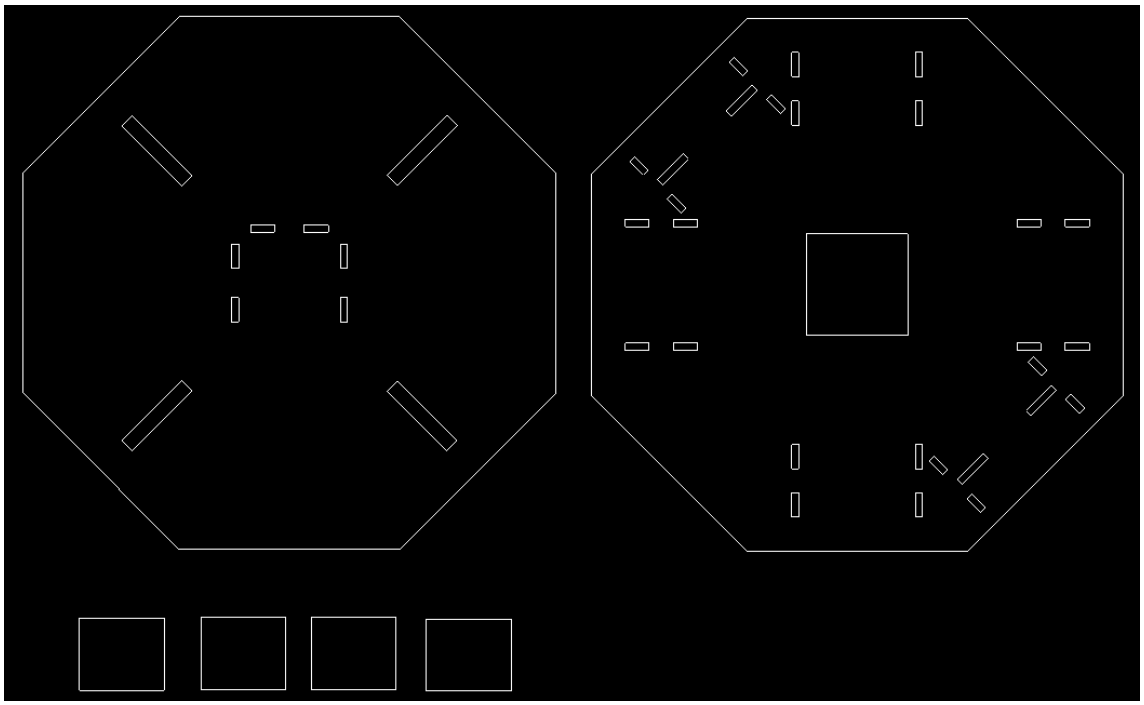


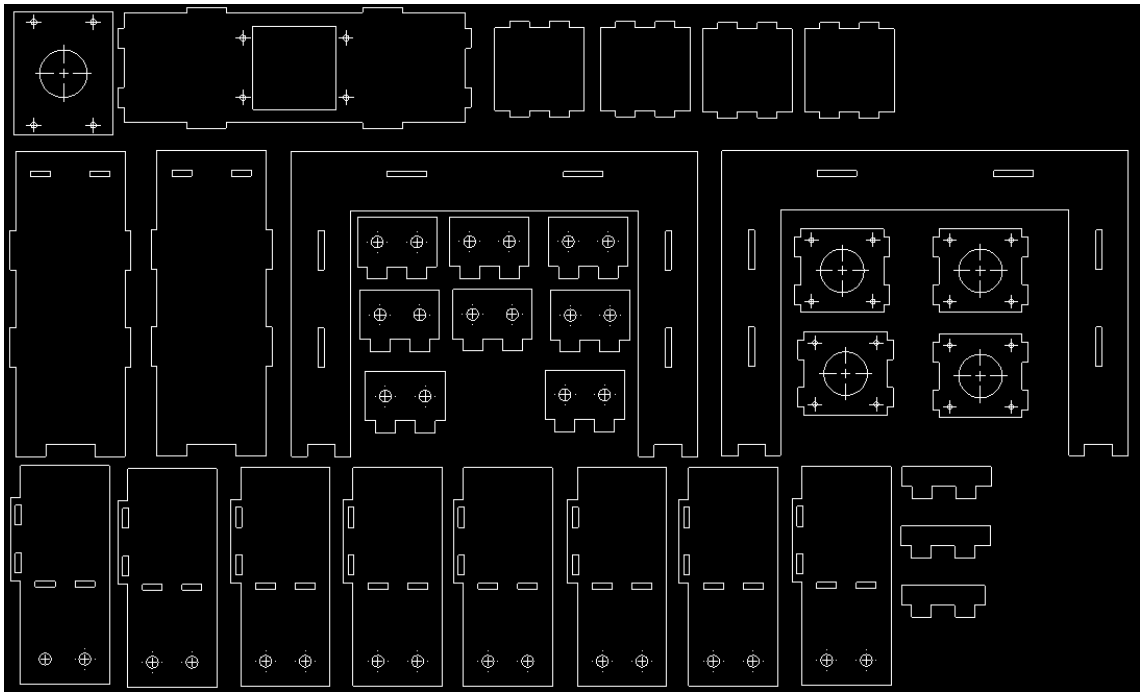
*Imagen 26. Agarre caras superior e inferior 2o diseño.*

### *3.1.5 Cortado piezas*

Para poder realizar el cortado láser de las piezas del segundo diseño, debemos agruparlas en un plano mediante SolidWorks. Una vez agrupadas, exportamos el archivo con la extensión .DXF. Se realizan dos archivos: uno con las piezas de grosor 6 mm y otro con las piezas de 3 mm.



*Plano 1. Piezas de grosor 6 mm 2o diseño.*



*Plano 2. Piezas de grosor 3 mm 2o diseño.*

# 3.2 Simulación mecánica

Las herramientas de simulación en un proyecto en el que se involucran uniones y movimientos mecánicos son de gran importancia, ya que permiten anticipar y resolver problemas antes de que ocurran en la realidad. Algunos motivos clave pueden ser:

- Prueba de escenarios. Se pueden probar múltiples de ellos sin un riesgo real, evaluando el impacto de diferentes variables y decisiones.
- Prueba de innovaciones.
- Proporcionan datos y análisis cuantitativos.
- Reducción de retrabajo.
- Verificación y Validación.

Es por ello, que para el presente trabajo se opta por simular el movimiento del mecanismo, así como la unión entre las piezas.

## *3.2.1 Simscape*

Se trata de una aplicación dentro del entorno de MATLAB y Simulink, desarrollada por MathWorks. Simscape es una plataforma que facilita la simulación y análisis de sistemas físicos mediante la creación de modelos a partir de componentes predefinidos. Estos componentes pueden representarse gráficamente y conectarse entre sí para formar un sistema completo, pudiendo simular finalmente el movimiento de los elementos del sistema.

Se integra perfectamente con Simulink y MATLAB, lo que permite el uso de scripts y funciones de MATLAB para la automatización de simulaciones y análisis de resultados. Además, los usuarios pueden crear componentes personalizados (pueden importarse modelados 3D, diseñados con otro software) si los componentes predefinidos no cubren todas sus necesidades específicas.

Simscape incluye varias bibliotecas especializadas, como:

- **Simscape Electrical**. Para modelado y simulación de sistemas eléctricos y electrónicos.
- **Simscape Mechanical**. Para sistemas mecánicos, incluyendo cinemática y dinámica.
- **Simscape Fluids**. Para sistemas de fluidos hidráulicos y neumáticos.
- **Simscape Multibody**. Para modelado y simulación de sistemas de cuerpos rígidos y articulados.

En nuestro caso se hará uso de Simscape Multibody, que nos proporcionará las herramientas necesarias para unir la totalidad de piezas diseñadas y simular el movimiento de las caras del cubo solidarias a los respectivos ejes de los motores.

Cabe añadir que se ha utilizado el primer diseño en la simulación, ya que consta de menos piezas y el ordenador con el que se ha realizado el trabajo permite procesarlas, con el segundo diseño el programa se quedaba colgado. Aun así, el movimiento que se desea simular es el mismo en ambos casos y las distancias entre cubo y piezas son idénticas.

### 3.2.2 Elementos

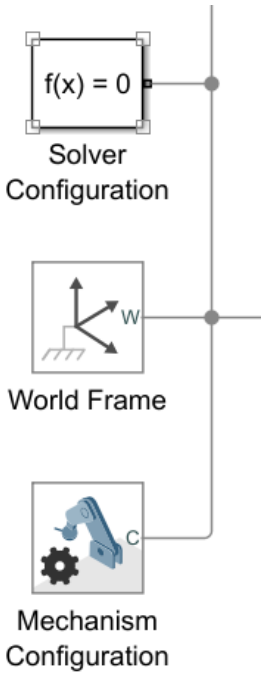



*Imagen 27. Bloques principales en Simscape.*

Los tres elementos principales son el ***Solver Configuration***, que permite ajustar la configuración del solucionador de ecuaciones diferenciales para la simulación; el ***Word Frame***, este es el sistema de referencia global, que usa un sistema de coordenadas ortogonal y diestro, predefinido en cualquier modelo mecánico; y ***Mechanism Configuration***, para que el usuario fije los parámetros mecánicos y de simulación que se aplicarán a la máquina, pudiendo especificar propiedades como el valor, sentido y dirección de la gravedad y el valor de la perturbación usada para calcular las derivadas parciales en la linealización.

Como podemos observar en la Imagen 27, los bloques pueden unirse a otros uniendo los nodos que salen de ellos. En este caso la "W" del segundo bloque constituye el nodo que representa el sistema de referencia global, así como la "C" del tercer bloque representa los valores de configuración mecánica que hemos introducido. Mediante la unión de los nodos presentes en los distintos bloques formaremos el mecanismo completo.

El siguiente bloque importante que usaremos es ***File Solid***. Este bloque representa un sólido cuya geometría, material y propiedades visuales son leídas desde un archivo externo. Es compatible con varias extensiones, pero en nuestro caso usamos .STP, una extensión que es directamente exportable desde SolidWorks. De esta forma, en cada uno de estos bloques importaremos las distintas figuras que hemos diseñado.

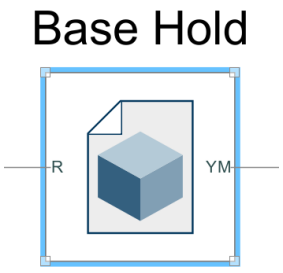



*Imagen 28. Bloque File Solid en Simscape.*

En la Imagen 29 se muestra la interfaz del interior del bloque. En *Geometry>File Name* importaremos el archivo deseado. En *Inertia>Density* especificamos la densidad del material de la pieza en cuestión, en nuestro caso es PLA (1240 kg/m$^3$) y pulsando el botón *Update*, el programa calculará el resto de los datos, mostrados en *Derived Values*.

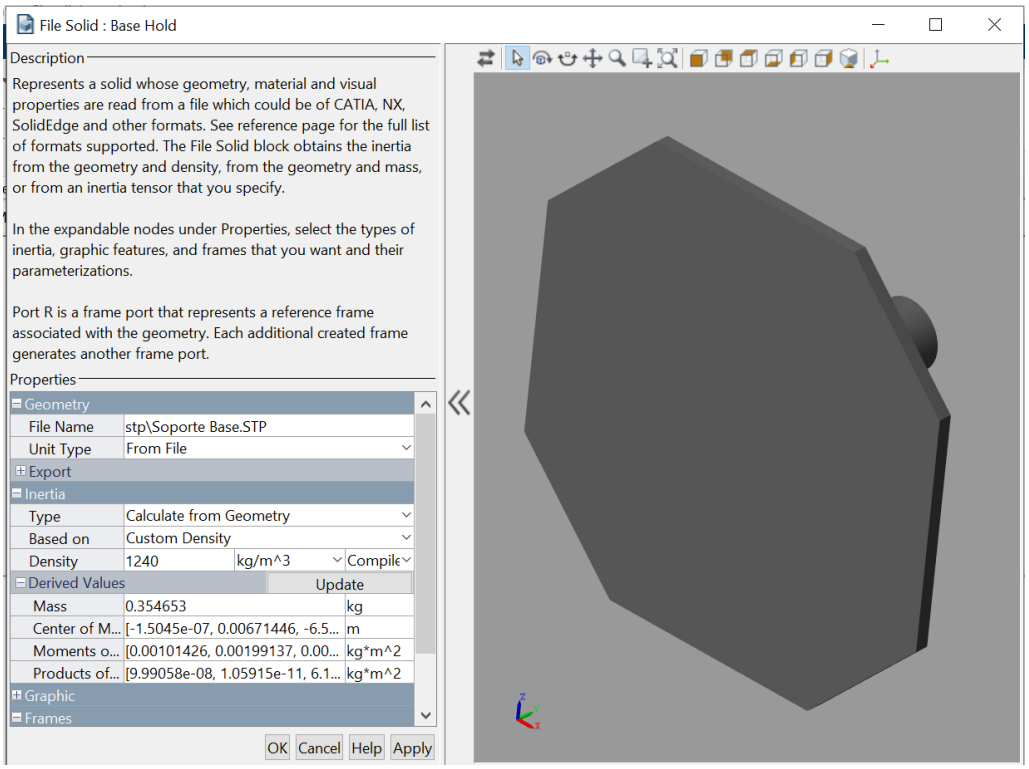



*Imagen 29. Interfaz principal del bloque File Solid en Simscape.*

En el apartado *Frames*, podemos crear tantos sistemas de coordenadas como deseemos, además de incluirse predeterminadamente el sistema de referencia R, que ya viene definido por la pieza importada. Estos sistemas creados pueden nombrarse y aparecerán en el bloque de Simscape, como ya observábamos en la Imagen 28. Para crear un sistema apretamos en el botón con una cruz al lado de *New Frame* (Imagen 30).

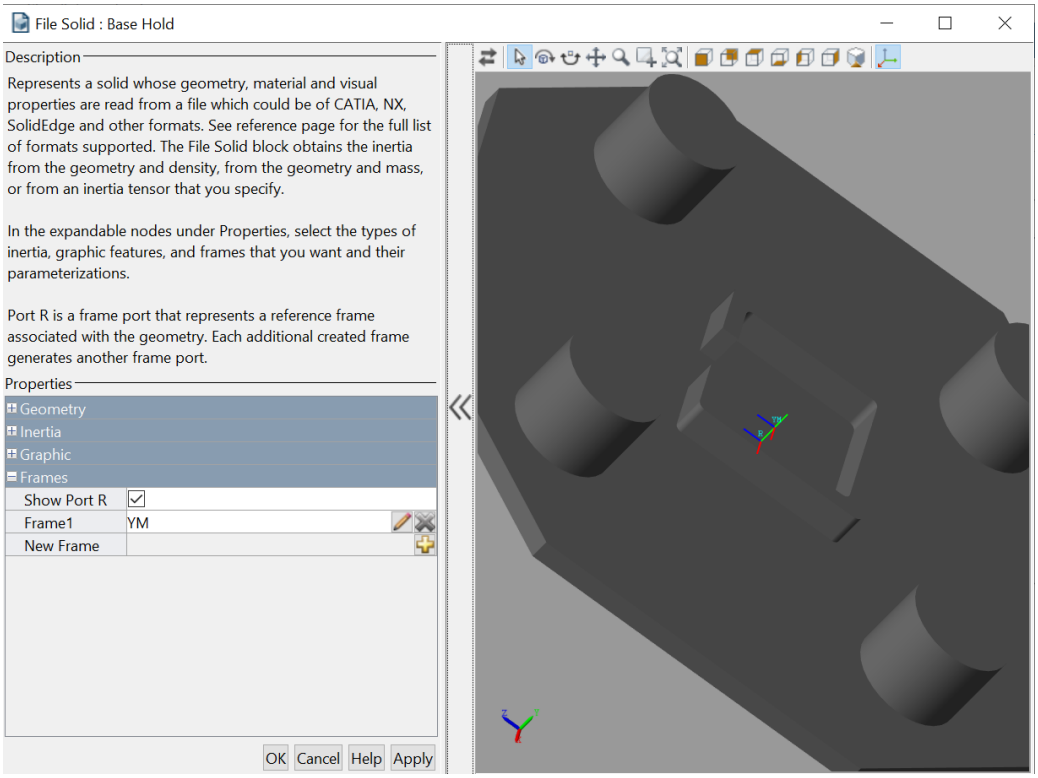



*Imagen 30. Apartado Frames en la interfaz del bloque File Solid en Simscape.*

Una vez creado el nuevo sistema se despliega un nuevo menú (Imagen 31). Seleccionando *Frame Origin>Based on Geometric Feature*, podemos seleccionar un punto en la figura, normalmente será el centro de una superficie o un vértice, que servirá de origen del sistema de coordenadas. Seguidamente, podemos modificar la dirección y sentido de los 3 ejes seleccionando las opciones disponibles en *Frame Axes*. Para poder ver el resultado de los sistemas de referencia podemos seleccionar la opción de los 3 ejes, situada justo debajo del botón de minimizar la ventana. Una vez configurado el sistema creado, damos a *Save* y volvemos al menú anterior.

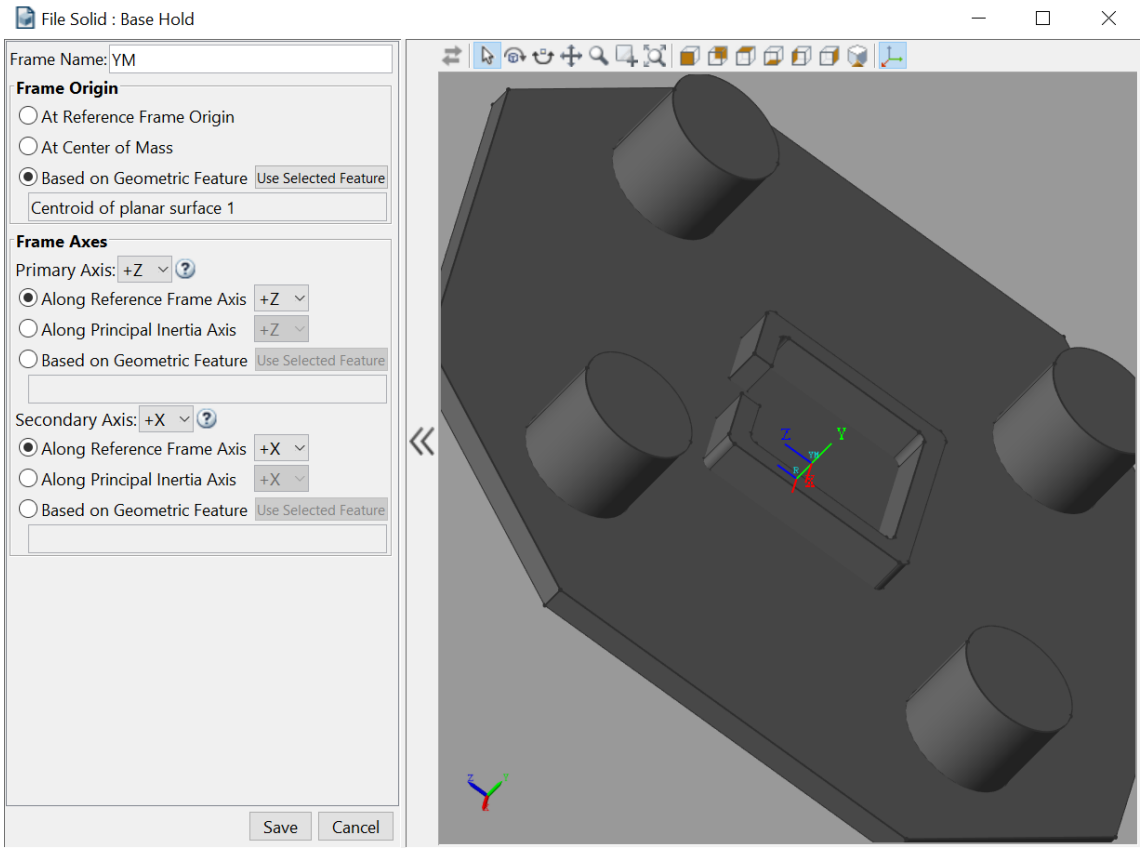



*Imagen 31. Interfaz de Frame del bloque File Solid en Simscape.*

El siguiente bloque utilizado es ***Rigid Transform***. Este se encarga de aplicar una transformación al sistema de coordenadas al que se le une a su nodo "B" (*Base Axes*) o a su nodo "F" (*Follower Axes*), como podemos ver en la Imagen 32.

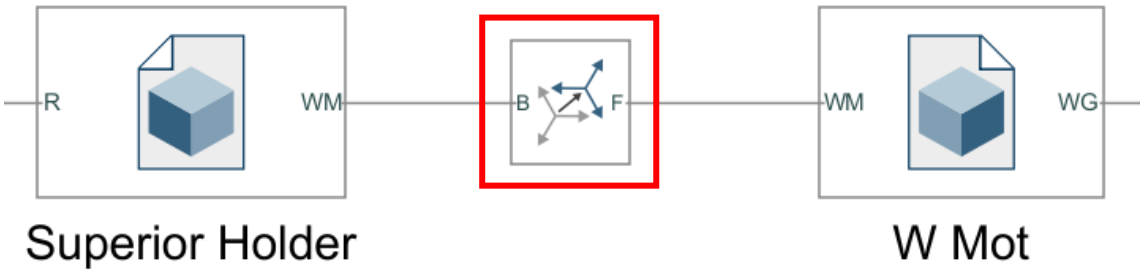



*Imagen 32. Bloque Rigid Transform en Simscape.*

Esta transformación puede ser de rotación, especificando el método que quiere aplicarse, así como al nodo que debe afectar la transformación y la secuencia de ejes a los que va a aplicar el vector; o de translación, especificando los mismos parámetros.

De este modo, este bloque nos será útil para orientar las piezas que deban estar unidas, aplicando rotaciones y translaciones cuando sea necesario.

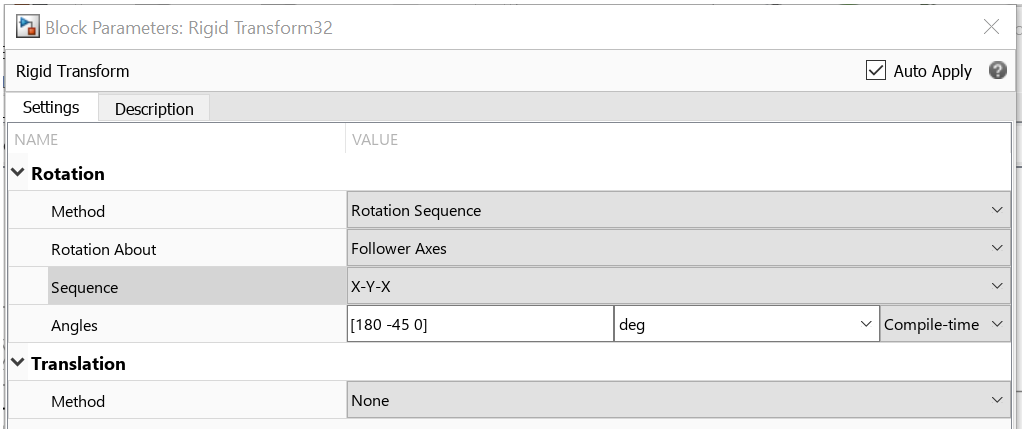



*Imagen 33. Interfaz del bloque Rigid Transform en Simscape.*

Otro de los bloques empleados es ***Revolute Joint*** (Imagen 34). Este representa un par de revolución entre dos sistemas de coordenadas. Este par tiene un grado de libertad, permitiendo solo el giro relativo alrededor de un eje. En nuestro proyecto usaremos este bloque para unir la pieza de agarre (que hemos denominado "W Grip" en Simscape) con el eje del motor.

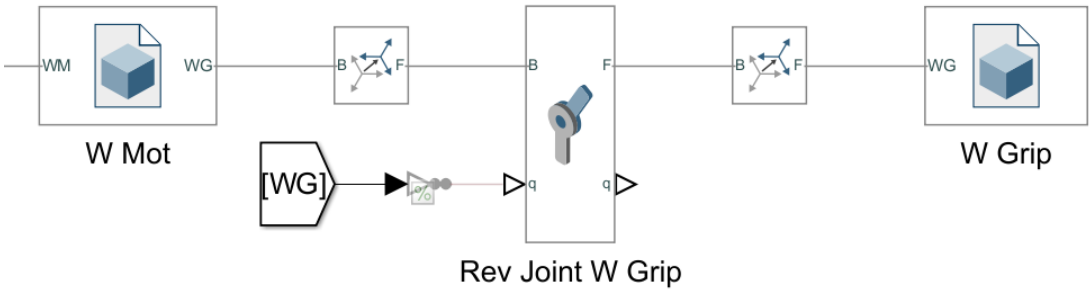



*Imagen 34. Bloque Revolute Joint en Simscape.*

Entre las opciones de configuración que nos ofrece el bloque, nos interesan *Internal Mechanics>Damping Coefficient* (el coeficiente de amortiguamiento, colocando un valor que simule un poco mejor la realidad) y *Actuation*, en la que tenemos la opción de actuar mediante fuerza o movimiento. Ya que nuestros motores son paso a paso, actuaremos por movimiento, introduciendo en el actuador una señal que indique los grados que debe girar el eje del motor. La fuerza debe estar en la opción de "Automatically Computed" si actuamos mediante movimiento.

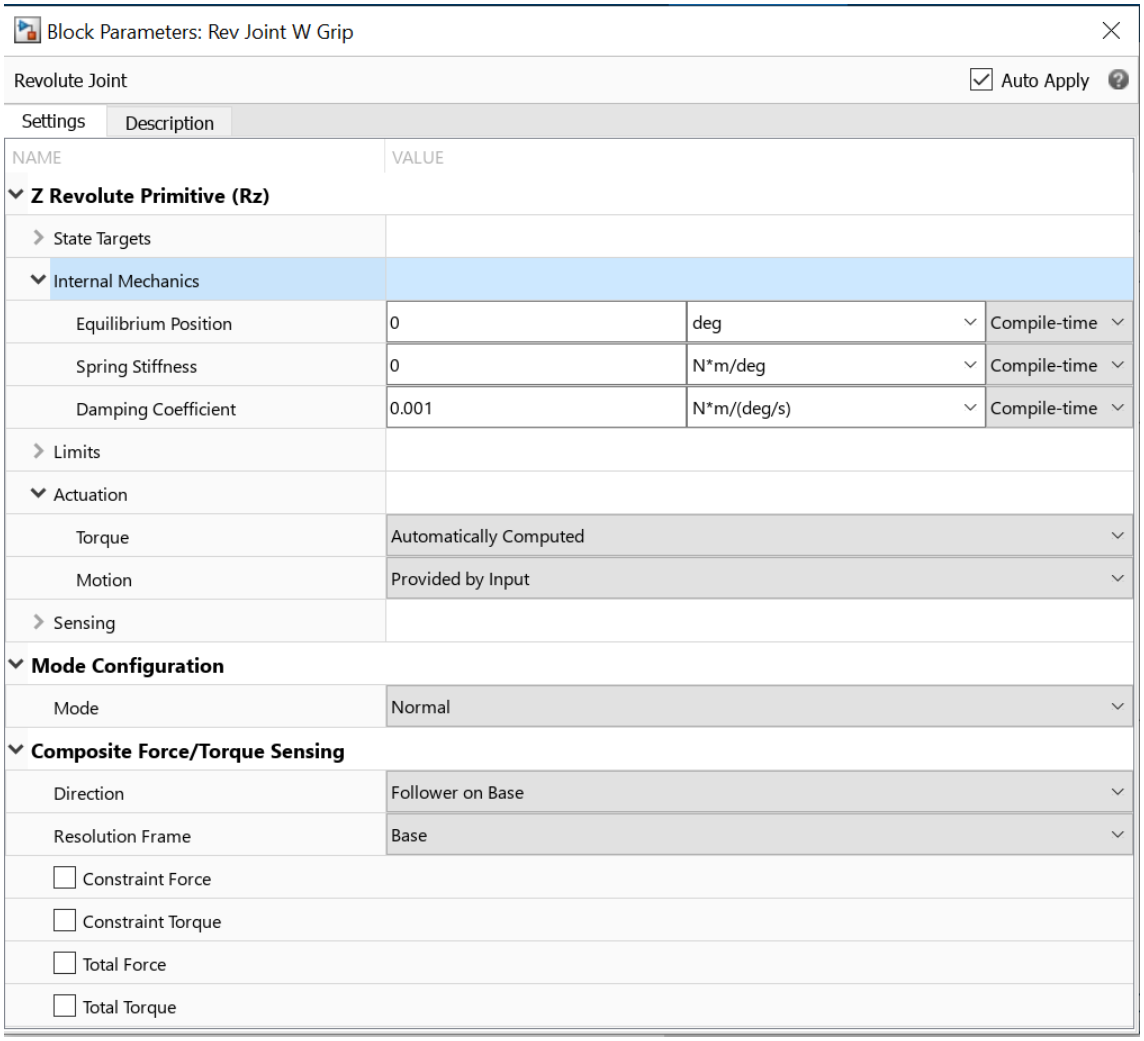



*Imagen 35. Interfaz del bloque Revolute Joint en Simscape.*

Para generar las señales utilizaremos varios bloques (Imagen 36). El primero de ellos es un *Step*, bloque que genera una señal de tipo escalón. Se debe especificar en qué segundo de la simulación va a aplicarse, así como el valor final al que se desea llegar, que en nuestro caso serán 90, -90 ó 180°.

El segundo es un sumador, encargado de sumar las señales escalón de entrada, ya que el mismo motor deberá girar varias veces para resolver el cubo.

El tercero es un *Rate Limiter*. Este se encarga de limitar la tasa de cambio de una señal. Ya que los motores no giran instantáneamente, como sería el caso de la señal de escalón, es necesario añadir este bloque, que limita la primera derivada de la señal que pase por él, convirtiendo nuestro escalón en una rampa. Con el objetivo de simular mejor la velocidad de giro del eje, seleccionamos un límite superior e inferior adecuado, cuanto mayor sea este límite, menor será la velocidad de cambio de la salida.

El último bloque es un *GoTo*, que lleva la señal resultante hacia este punto, que podrá ser usado en cualquier otro lugar de la interfaz de Simulink.

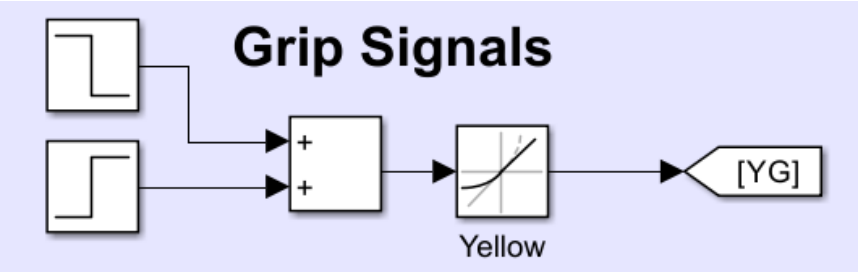



*Imagen 36. Bloques para generar señales en Simscape.*

Para cada uno de los seis motores se creará un grupo de señales, con una salida resultante. Esta salida se aplicará, con un bloque *From*, que recogerá la señal del bloque *GoTo*, a la entrada "q" de los pares de revolución (Imagen 34).

### *3.2.3 Simulación y resultados*

Con el objetivo de mostrar el movimiento de los ejes, es decir, la rotación de las seis caras del cubo durante el tiempo de resolución, juntamos las seis señales producidas en un mismo osciloscopio.

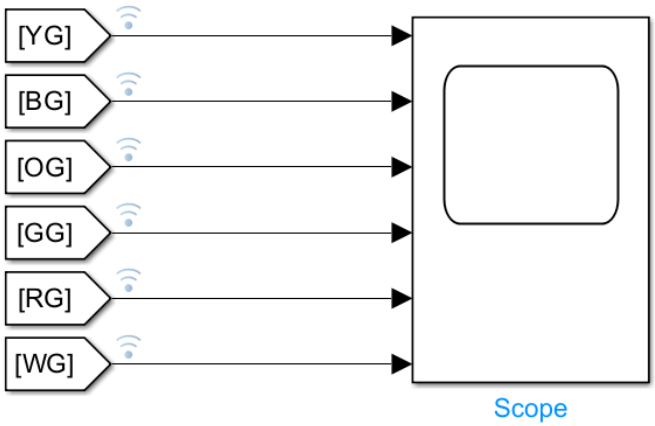
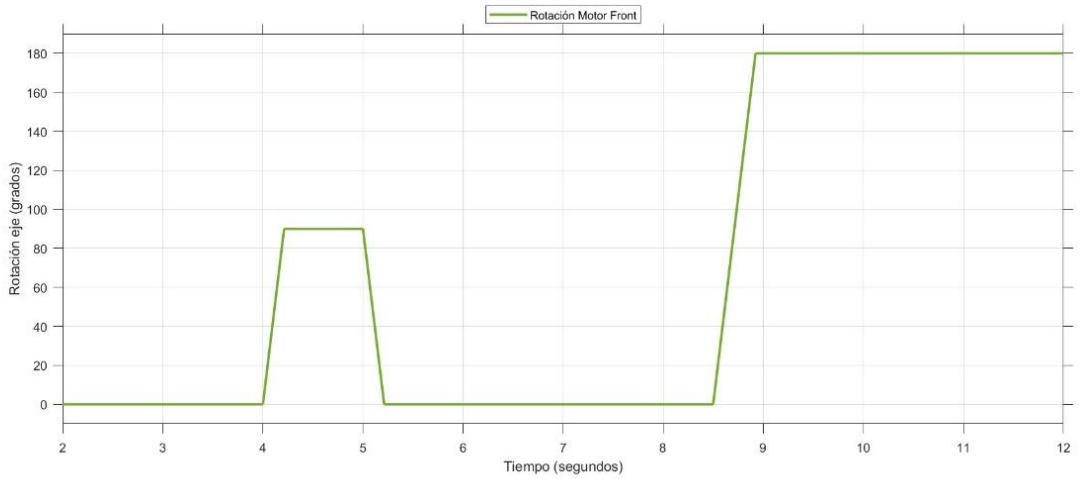


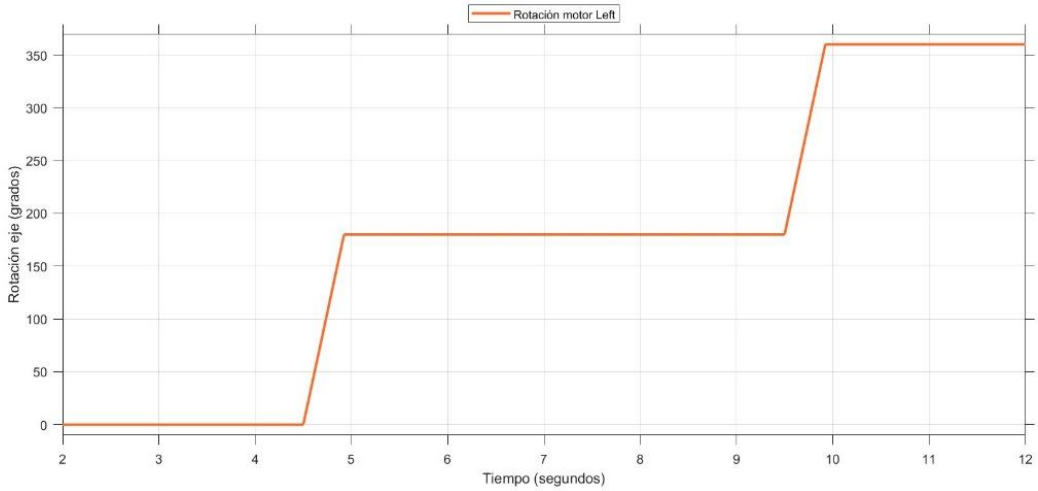

*Imagen 37. Señales entrando al osciloscopio en Simscape.*

Simulamos la secuencia de giros que realizarían los ejes para una configuración inicial determinada de nuestro cubo. Ajustamos el tiempo de simulación a 12 segundos, ya que comenzamos con el primer giro en el segundo dos, dejando 0.5 s entre giro y giro, sabiendo que el máximo de giros usando nuestro algoritmo es de 20.

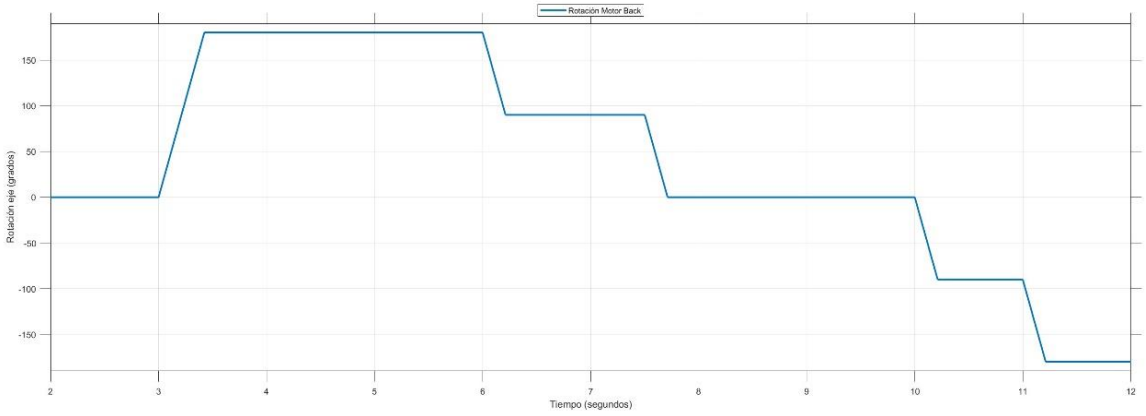
Las gráficas resultantes de rotación del eje de cada motor por separado:



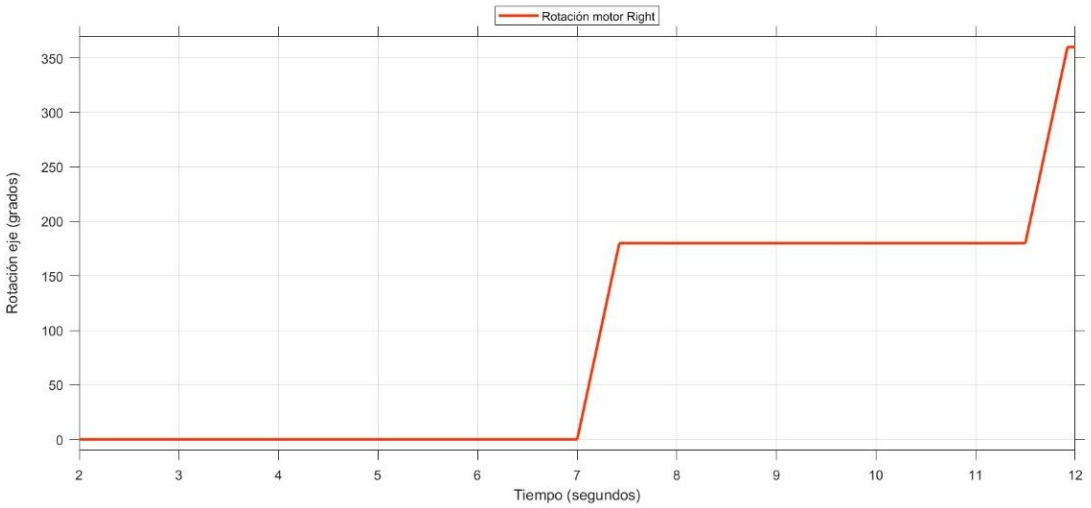

*Gráfica 1. Rotación del motor frontal en el tiempo.*



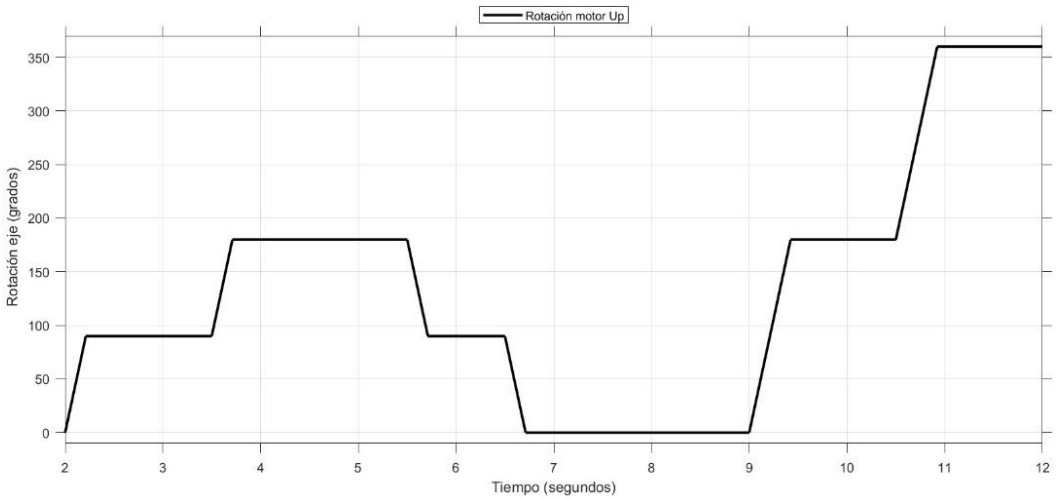

*Gráfica 2. Rotación del motor izquierdo en el tiempo.*



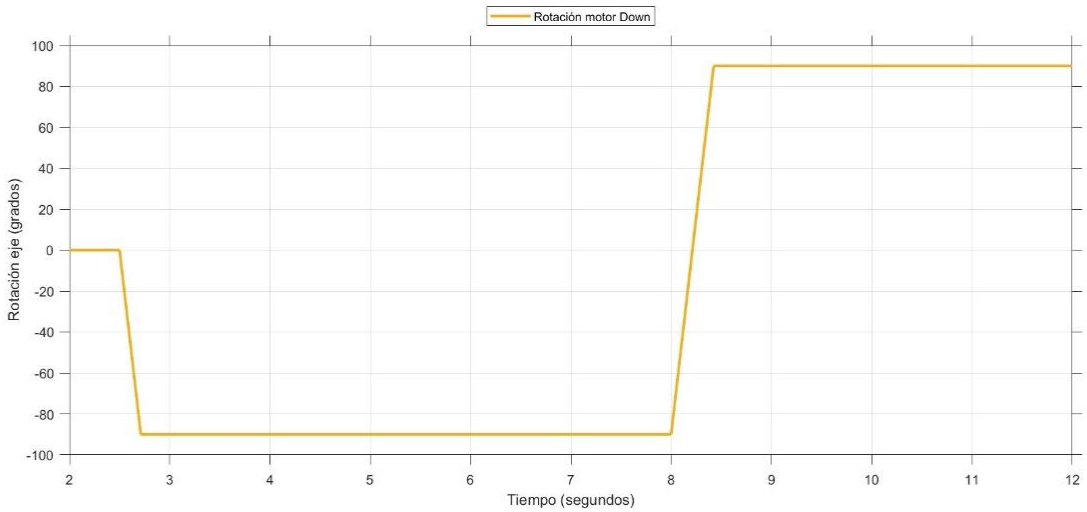

*Gráfica 3. Rotación del motor trasero en el tiempo.*




*Gráfica 4. Rotación del motor derecho en el tiempo.*

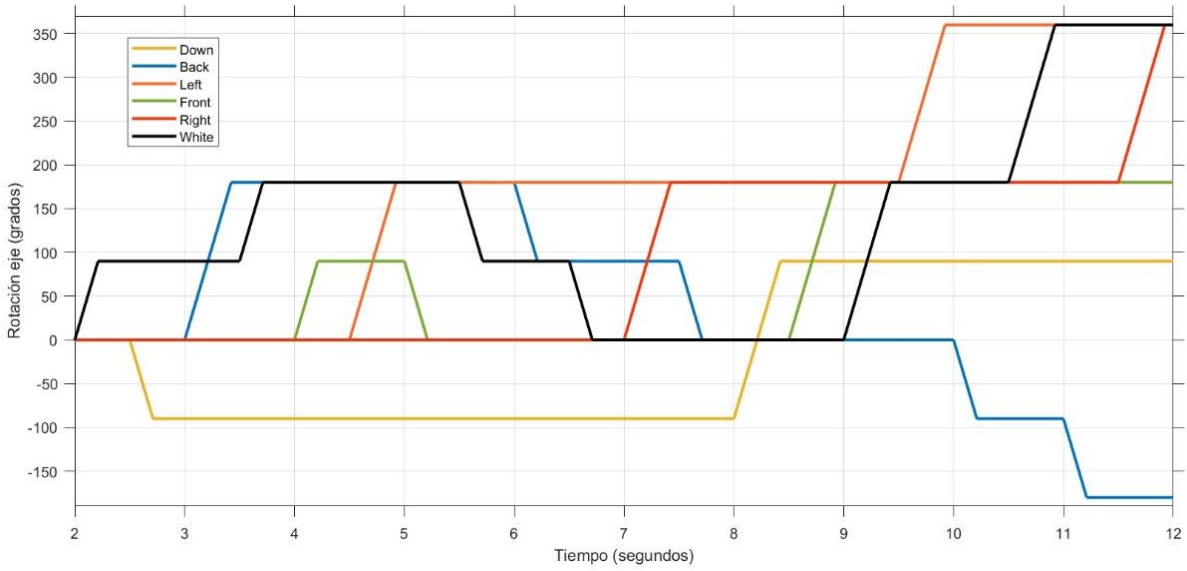



*Gráfica 5. Rotación del motor superior en el tiempo.*




*Gráfica 6. Rotación del motor inferior en el tiempo.*

La gráfica de rotación del eje de los seis motores juntos:




*Gráfica 7. Rotación de los motores en el tiempo.*

Como podemos apreciar, con el ajuste del limitador de la primera derivada y la aplicación de los escalones cada 0.5 segundos, las distintas rampas a lo largo del tiempo no llegan a solapar, por lo que la cara girará del todo antes de que la siguiente de la secuencia lo haga.

# 3.3 Electrónica

En el siguiente apartado se definirán los componentes electrónicos necesarios para conseguir aplicar el algoritmo con un control del movimiento de los motores, que permitirán girar las caras del cubo y finalmente, resolverlo. También se definirá el diagrama de conexiones empleado, así como comentar matices importantes para el montaje de este.

## *3.3.1 Componentes utilizados*

- 6 motores paso a paso Nema 17 Bipolar (42 Ncm, 1.5 A, 42x42x39 mm).
- 6 controladores A4988 para motor paso a paso.
- CNC Shield v3 para 4 controladores A4988.
- Tarjeta con microcontrolador ELEGOO Mega 2560 R3.
- Condensador de 100 µF.
- Cables macho-macho y hembra-hembra.
- Placa protoboard pequeña.
- Cargador ordenador portátil marca ASUS (19 V, 2.37 A)

### 3.3.2 *Montaje*

En primer lugar, para poder controlar los motores paso a paso correctamente y que estos no pierdan información ni se sobrecalienten, ajustaremos la tensión de referencia de los controladores A4988. Para ello usaremos la fórmula que nos proporciona el fabricante, extraída del *datasheet* del A4988:

$$I_{TripMAX} = V_{REF}/(8 \times R_S) \quad (1)$$

Donde,

$I_{TripMAX}$ es la corriente nominal de nuestro motor

$V_{REF}$ es la tensión de referencia del controlador

$R_S$ es la resistencia *shunt*, para medir la corriente del controlador

En este caso, conocemos la intensidad nominal del motor, 1.5 A, así como el valor de las resistencias *shunt*, que podemos observarlas directamente sobre el controlador, obteniendo un valor de R100 (0.1 Ω). Despejando, obtenemos un valor de tensión de referencia de 1.2 V.

Para ajustar este valor, realizaremos las conexiones mostradas en la Imagen 38, conectando con cables macho-macho los puertos VDD y GND del A4988 (Imagen 39) a los puertos 5V y GND de la tarjeta Elegoo, respectivamente. Después, conectaremos nuestra tarjeta al ordenador mediante USB, para que el controlador reciba la alimentación. Finalmente, conectaremos el terminal COM del multímetro al canal GND de la *protoboard* y el terminal VΩ al potenciómetro del A4988. Deberemos girar el potenciómetro hasta mostrar por pantalla los 1.2 V de referencia calculados. Realizaremos la tarea con los 5 controladores restantes.

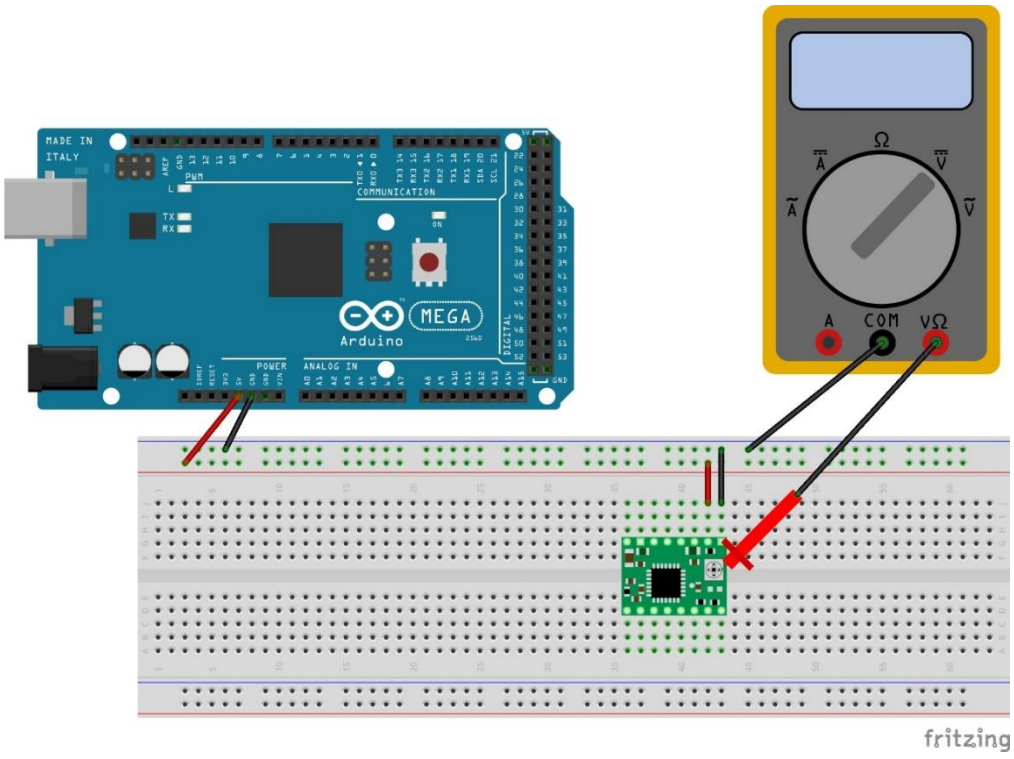



*Imagen 38. Esquema de conexiones para ajuste del driver A4988.*

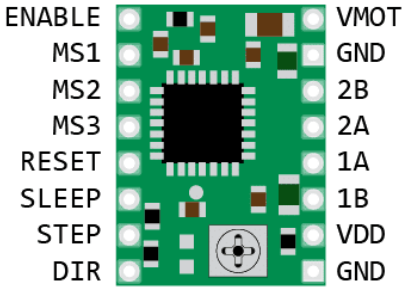



*Imagen 39. Esquema de pines del driver A4988.*

El *CNC Shield* es un módulo de expansión que se conecta directamente a nuestra placa de ELEGOO. Principalmente se usa en impresoras 3D, permitiendo desarrollar una máquina CNC, aunque en este caso nos servirá para alojar cuatro de los seis controladores A4988 y utilizar menos cables.

En primer lugar, colocaremos los controladores, cada uno en su posición (ejes X, Y, Z y A marcados en el CNC), comprobando que quedan colocados como en la Imagen 40, con el potenciómetro mirando hacia abajo. Después colocaremos el módulo sobre nuestra placa, colocando los pines superior e inferior derecho en el interior de los pines RX0 y A5, respectivamente, de la placa ELEGOO. En la Imagen 40 podemos observar las marcas 1 y 2, que sirven de guía.

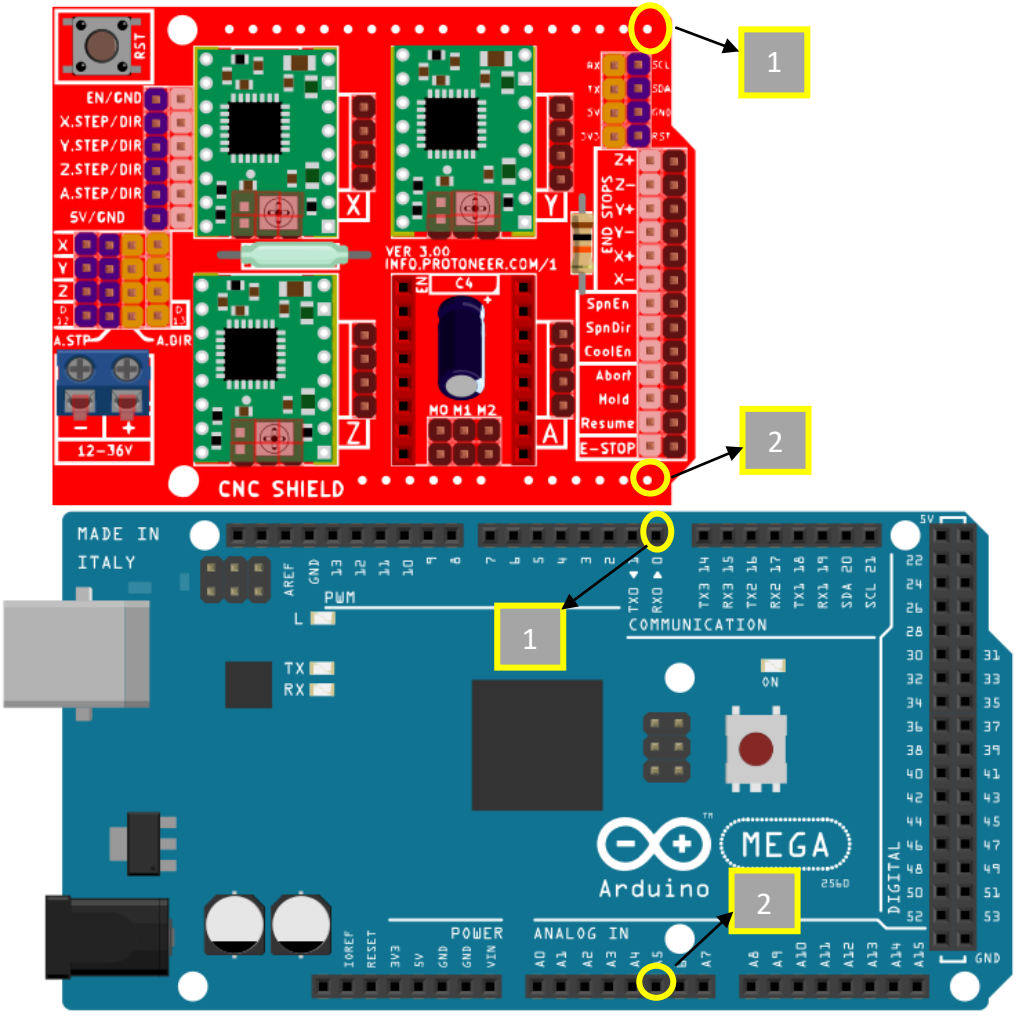



*Imagen 40. Esquema de conexión entre el módulo CNC y la controladora ELEGOO.*

La fuente de tensión se conectará por los terminales + y – del *CNC Shield*. Esta indica que soporta entre 12 y 24 V de tensión continua. Además, realizaremos un puente entre los terminales EN y GND, como se nos indica en la Imagen 41, extraída del *datasheet* del producto. Los cuatro cables del motor se conectan a los cuatro pines que se encuentran al lado de cada uno de los ejes. Debemos tener en cuenta puentear los dos pares de pines subrayados en blanco en la Imagen 41, de esta forma conseguimos que el eje A actué por su propia cuenta, sin copiar ninguno de los otros tres ejes restantes.

Observando el archivo *CNC Shield Pinout*, que aporta el vendedor, asociamos:

- Pin digital 2 placa → *Step* motor eje X
- Pin digital 5 placa → *Dir* motor eje X
- Pin digital 3 placa → *Step* motor eje Y
- Pin digital 6 placa → *Dir* motor eje Y
- Pin digital 4 placa → *Step* motor eje Z
- Pin digital 7 placa → *Dir* motor eje Z

Así como obtenemos del *datasheet* del módulo los pines D12 y D13, para el *Step* y *Dir*, respectivamente, del motor del eje A.

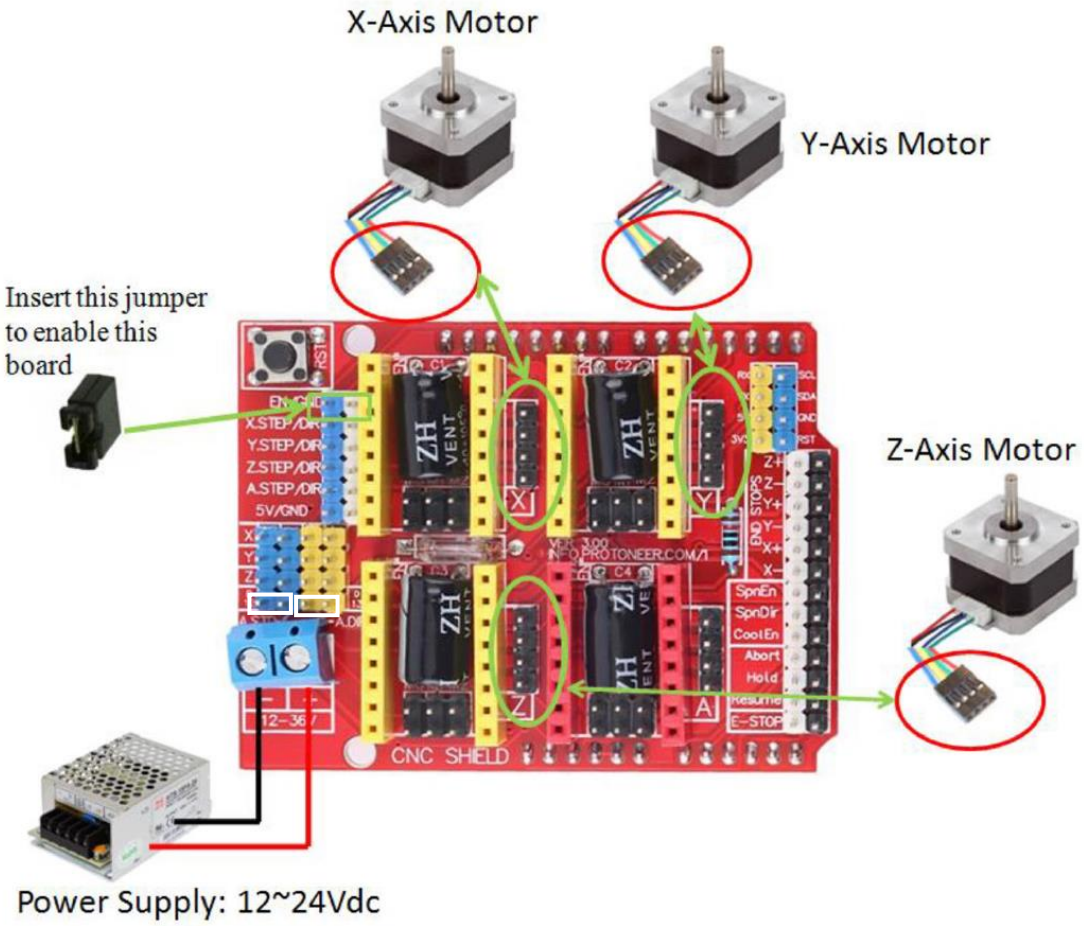



*Imagen 41. Esquema de conexiones del CNC Shield.*

Por último, faltaría conectar los dos controladores restantes a la placa y a los motores. Para ello, se hace uso de la *protoboard*. Introducimos los pines de los controladores en las ranuras de la *proto*. Los cuatro cables del motor se conectarán en los pines 2B, 2A, 1A y 1B. La fuente de tensión se conectará por los pines VMOT y GND, positivo y negativo, respectivamente. La tensión de 5V proporcionada por la placa ELEGOO se conectará por los pines VDD y GND. Los pines SLEEP y RESET deben puentearse. Por último, se conectarán los pines STEP y DIR a cualquiera de los pines digitales libres de nuestra placa.

Usaremos una de las ranuras laterales de la *protoboard* para colocar la salida de fuente de tensión, poniendo seguidamente un condensador electrolítico de desacoplamiento de 100 µF, para asegurar no dañar los controladores debido a picos altos de corriente salientes de la fuente. Las otras ranuras laterales las utilizaremos para la tensión de 5 V que nos proporciona la placa ELEGOO; para ello conectaremos los pines de 5V y GND presentes en el *Shield* a la entrada de estas ranuras laterales.

### *3.3.3 Diagrama de conexiones*

Una vez realizado todo el montaje, nuestro circuito tendrá las conexiones presentes en el siguiente diagrama:

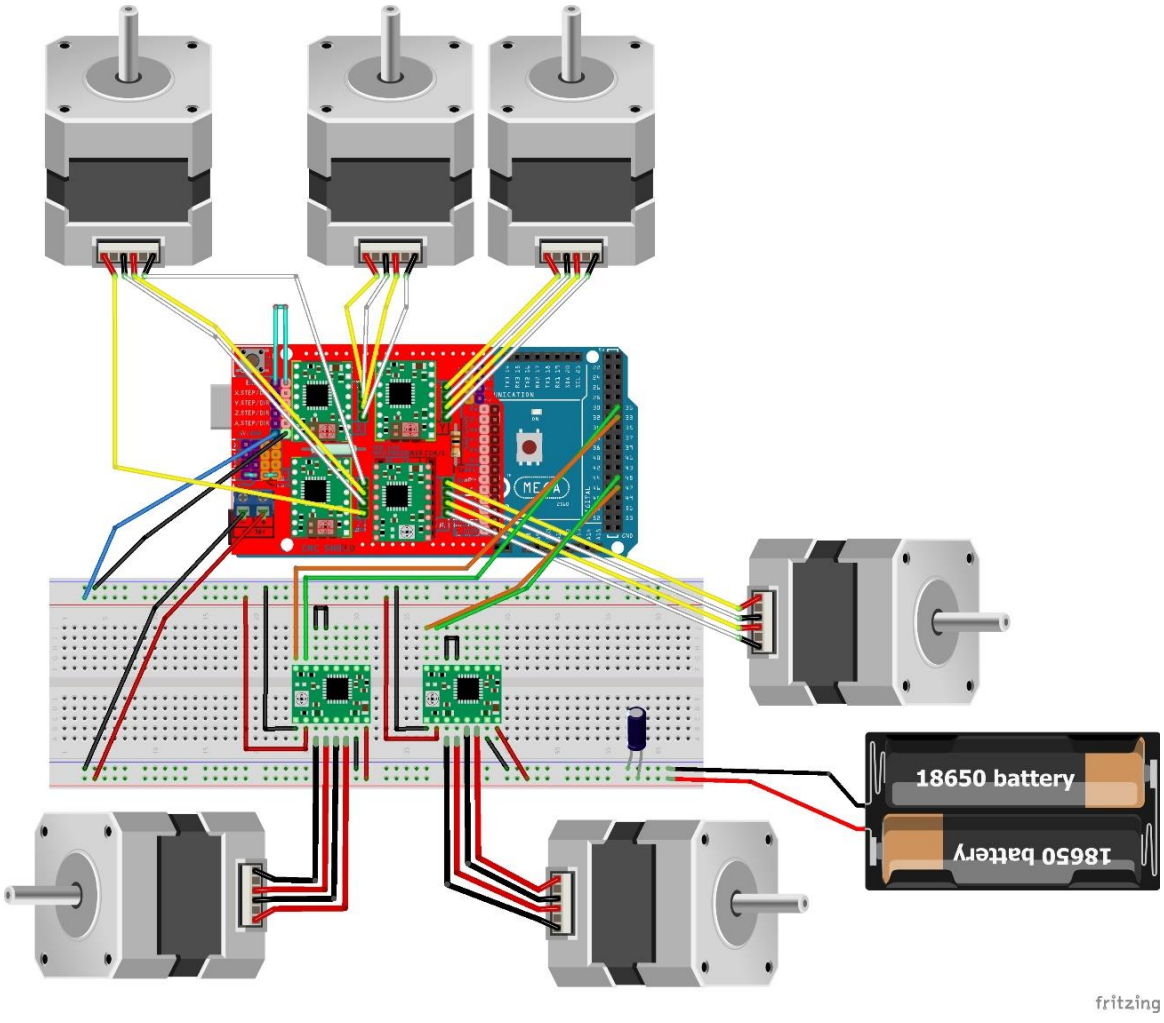



*Imagen 42. Diagrama de conexiones final.*

# 3.4 Software

Para el control del giro de los ejes usaremos la plataforma de desarrollo Arduino IDE. Esta nos ofrece un entorno de programación sencillo basado en el lenguaje C/C++, incluye varias bibliotecas integradas muy útiles que facilitan la programación de actuadores. Además, esta es compatible con nuestra placa ELEGOO y permite la comunicación serial entre el ordenador y la placa, mediante la cual podremos mandar información desde otro entorno de programación, en nuestro caso con Visual Studio Code, como veremos en el presente capítulo.

## *3.4.1 Control Motores*

Como ya se ha introducido, la programación que se encargue de controlar los motores se realizará en el entorno de Arduino IDE.

En primer lugar, añadiremos la librería que nos permitirá recoger la información enviada a través del puerto serie.

```
#include <SoftwareSerial.h>
```

Después, se definen, mediante constantes, los pines de la placa por los que se envía la información del sentido de giro y la cantidad de pasos que el eje debe realizar. La información de los pasos se almacena en la constante “pwm”, el sentido de giro en “m” y la última letra de ambas hace referencia a la cara del cubo a la que aplica.

```
// Control Motor Front (Green)
#define mF 7
#define pwmF 4
```

Además, se define otra constante que hace referencia a la velocidad de giro, cuanto mayor es este número, menor es la velocidad.

```
// Velocidad de giro Motores
#define vel 5000
```

Adicionalmente, añadimos 4 variables:

- La variable de tipo entero “steps”. Se encargará de albergar el valor de los pasos. Revisando el *datasheet* de nuestros motores, obtenemos la conversión de grados a pasos, en este caso están configurados para que, por cada paso, el eje rote 1.8°.
- La variable booleana “direction”. Esta tomará el valor *true* cuando el eje deba girar en sentido antihorario, tomando el valor *false* en el caso contrario.
- La variable de tipo cadena “turn”. Esta tomará el valor de giro que se extraerá desde la cadena de resolución del algoritmo.
- La variable de tipo cadena “face”. Esta tomará la letra de la cara que se extraerá desde la cadena de resolución del algoritmo.

```
// Variables
int steps;
bool direction;
String turn;
String face;
```

Dentro de la función de configuración (setup), fijaremos el valor de velocidad de la comunicación serial en 9600 bits por segundo, así como el valor de tiempo de espera máximo para las funciones de lectura serial en 10 milisegundos. Por otra parte, configuramos los pines digitales en modo salida (output), utilizando las constantes definidas anteriormente.

```
  // Velocidad de comunicación puerto serial
  Serial.begin(9600);
  Serial.setTimeout(10);

  // --------------- Pines de salida ---------------
  // Motor Front (Green)
  pinMode(mF, OUTPUT);
  pinMode(pwmF, OUTPUT);
```

Definimos 3 funciones principales:

**La función type_turn**. Esta recibe una cadena, que será “1”, “2” o “3”. Cuando reciba “1”, significará que la cara debe girar 90° en sentido horario. La variable “steps” tomará el valor de 50, que, realizando la conversión, significará un giro de 90°, además, la variable “direction” tomará el valor *false*, significando un giro horario. El valor “2” significará un giro de 180° (da igual el sentido de giro realmente) y el “3”, un giro de 90° en sentido antihorario.

```
void type_turn(char turn){
  if (turn == '1'){
      steps = 50;
      direction = false;
      Serial.print("Giramos 90ª clockwise");
  }
  else if (turn == '2'){
    steps = 100;
    direction = true;
    Serial.print("Giramos 180ª a-clockwise");
  }
  else if (turn == '3'){
    steps = 50;
    direction = true;
    Serial.print("Giramos 90ª a-clockwise");
  }
}
```

**La función *step***. Esta recoge el valor booleano del sentido de giro del motor, el número de pasos que se le aplicará al motor y los pines de salida a los que se aplicará. En primer lugar, se aplica al pin de sentido de giro el valor de la variable booleana "dir". Después, mediante un bucle *for* se activa y desactiva el pin de los pasos del motor (pwm), dejando entre ellos un tiempo de espera adjudicado en la constante "vel", anteriormente mencionada.

```
void step(bool dir, int steps, int m, int pwm) {
  digitalWrite(m, dir);
  delay(50);

  for (int i=0;i<steps;i++){
    digitalWrite(pwm, HIGH);
    delayMicroseconds(vel);
    digitalWrite(pwm, LOW);
    delayMicroseconds(vel);
  }
}
```

**La función what_face**. Esta recibe la variable "face". Depende de la letra que reciba, aplicará en la función "step" las constantes ligadas a la cara del cubo en concreto, consiguiendo accionar al motor asociado, con la rotación y el sentido correctos.

```
void what_face(char face) {
  if (face == 'U'){
    step(direction, steps, mU, pwmU);
  }
  else if (face == 'D'){
    step(direction, steps, mD, pwmD);
  }
  else if (face == 'F'){
    step(direction, steps, mF, pwmF);
  }
  else if (face == 'B'){
    step(direction, steps, mB, pwmB);
  }
  else if (face == 'R'){
    step(direction, steps, mR, pwmR);
  }
  else if (face == 'L'){
    step(direction, steps, mL, pwmL);
  }
  Serial.print(" cara ");
  Serial.println(face);
}
```

Dentro del *loop*, comprobaremos en primer lugar que se recibe algo mediante el puerto serie.

```
while(Serial.available() > 0){
```

Si es así, leeremos la cadena recibida y se la asociaremos a la variable “prompt”. Modificaremos la cadena para quedarnos solo con la secuencia de movimientos a realizar, ya que el algoritmo nos devuelve el número de movimientos realizados al final de la cadena.

```
String prompt = Serial.readStringUntil('\n');
    prompt = prompt.substring(0, prompt.length() - (prompt.length() -
      (prompt.indexOf("(")) + 1));
    Serial.println("Prompt: " + prompt);
```

Seguidamente, mediante el uso de un bucle recorremos la cadena recibida, carácter por carácter, almacenando en la variable “turn” cuando este sea un dígito y en la variable “face” cuando sea una letra.

```
for (int i=0; i <= prompt.length(); i++){
      if (isDigit(prompt.charAt(i))){
        turn = turn + prompt.charAt(i);
      }
      else if (isalpha(prompt.charAt(i))){
        face = face + prompt.charAt(i);
      }
    }
```

Una vez hemos obtenido las cadenas de letras y números, mediante un bucle recorremos la variable “face”. Aplicamos la función “type_turn”, introduciendo el número asociado a la letra de la variable “face”, para después aplicar la función “what_face”, introduciendo la letra; esto hará que se realice el primer giro de la secuencia. Se repetirán estas acciones hasta que la variable “face” se haya recorrido entera. Ajustando el valor del último *delay*, conseguiremos una mayor velocidad en la resolución de la secuencia de giros.

```
for (int i=0; i < face.length(); i++){
      // Establecemos los grados y sentido de giro
      type_turn(turn.charAt(i));
      // Establecemos el motor que vamos a usar
      what_face(face.charAt(i));
      Serial.print("Giro ");
      Serial.print(i + 1);
      Serial.println(" OK");
      delay(300);
    }
```

Por último, vaciamos todas las cadenas, para poder recibir otra instrucción distinta por el puerto serie.

```
prompt = "";
turn = "";
face = "";
```

### *3.4.2 Algoritmo de Kociemba*

En Python podemos encontrar una librería que implementa el algoritmo de dos fases de Kociemba. Como Python es un lenguaje con un procesado bastante más lento que otros lenguajes como Java o C, es necesario instalar unas tablas con un peso de unos 80 MB, que permitirán una implementación lo suficientemente eficiente, consiguiendo la secuencia de 20 o menos movimientos en menos de un segundo.

Para instalarla deberemos introducir en el compilador de Visual Studio Code:

```
$ pip install RubikTwoPhase
```

Una vez instalada, podremos importar el módulo en nuestro código, pudiendo usar las funciones de este.

```
import twophase.solver as sv
```

Para obtener la secuencia de movimientos que resuelvan el cubo deberemos definir una cadena que represente la configuración inicial del cubo. La cadena debe tener la siguiente estructura:

'UUUUUUUUURRRRRRRRRFFFFFFFFFDDDDDDDDDLLLLLLLLLBBBBBBBBB'

La letra representa la cara del cubo y el orden es el siguiente:

1. Esquina superior izquierda
2. Arista superior
3. Esquina superior derecha
4. Arista izquierda
5. Centro
6. Arista derecha
7. Esquina inferior izquierda
8. Arista inferior
9. Esquina inferior derecha

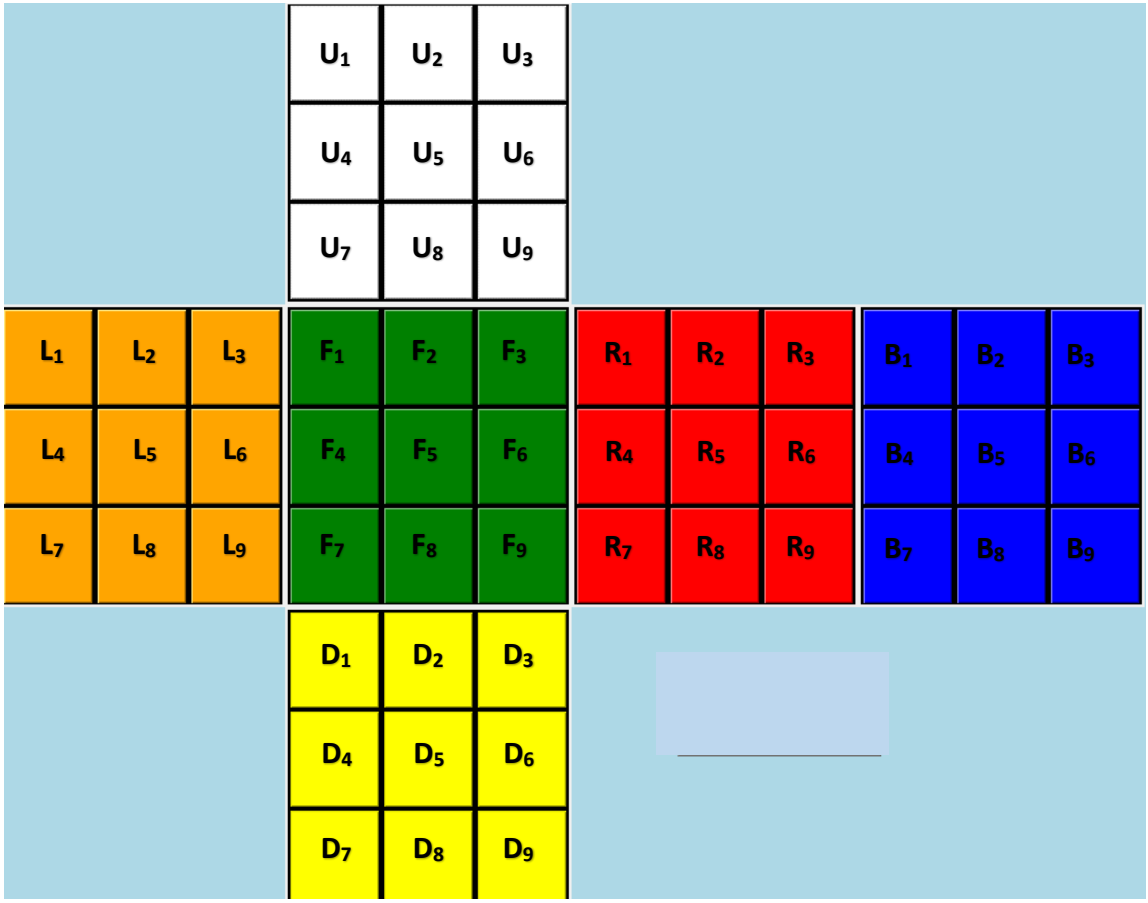



*Imagen 43. Orden de las caras en la cadena de secuencia de movimientos.*

Para resolver la configuración tendremos que introducir la siguiente instrucción:

```
sv.solve(CubeString)
```

Siendo *CubeString*, la variable que contiene la cadena de la configuración inicial. Esta función nos devolverá otra cadena, que nos indica la secuencia de movimientos a realizar para resolver el rompecabezas, con la siguiente forma:

```
'L3 U1 B1 R2 F3 L1 F3 U2 L1 U3 B3 U2 B1 L2 F1 U2 R2 L2 B2 (19f)'
```

Donde la letra corresponde a la cara del cubo a la que debe aplicarse el giro y los números a los giros de 90° horario, 90° antihorario y 180°, como ya se ha explicado anteriormente. El número entre paréntesis indica el número de movimientos empleados.

### *3.4.3 Conexión Python-Arduino*

Para poder conectar ambas plataformas, usaremos la conexión serial. Para ello deberemos instalar la librería “PySerial” en Python:

```
$ pip install pyserial
```

También será necesario importarla en nuestro programa, así como la librería “time”, que viene preinstalada, para poder manejar tiempos.

```
import serial, time
```

Antes de poder usar las funciones de la librería, debemos configurar el puerto serial.

```
com = serial.Serial("COM4", 9600, write_timeout= 10)
```

Donde el primer valor corresponde al puerto USB al que conectamos el controlador; el segundo valor corresponde a la tasa de velocidad de intercambio de bits, que debe coincidir con la configurada en ArduinoIDE; y la tercera es el tiempo máximo de escritura. Esta función la asociamos a una variable para poder manejarla eficientemente.

Para enviar la cadena que contiene la secuencia de movimientos por el puerto serie realizamos las siguientes instrucciones:

```
# Escribimos por el puerto serie la cadena a enviar a arduino
com.write(str2arduino.encode("ascii"))
time.sleep(0.1)
# Cerramos el puerto serie
com.close()
```

Siendo *str2arduino* la variable que contiene la cadena de secuencia de movimientos. Una parte importante es pasar la cadena a código ASCII, ya que este es el código que interpretará Arduino al recibir la cadena mediante comunicación serial. Con la función "time.sleep" esperamos un instante para asegurar que la cadena ha sido enviada correctamente. Por último, se cierra la comunicación con la función "com.close".

### *3.4.4 Interfaz usuario*

Con el objetivo de que el usuario controle el prototipo de manera visual y práctica, se diseña una interfaz sencilla que contará con varias funciones para controlar los motores de la máquina.

Para ello, se opta por utilizar un módulo dentro de las librerías del entorno de programación de Python: Tkinter. Esta es una de las herramientas más populares y utilizadas para desarrollar aplicaciones GUI (Graphical User Interface) en Python debido a su simplicidad y facilidad de uso.

Entre las diversas opciones que nos ofrece, nos interesan los *widgets*:

- Button: para crear botones.
- Label: para mostrar texto o imágenes.
- Entry: para campos de entrada de texto.
- Text: para áreas de texto multi-línea.
- Frame: para contener otros widgets.
- Canvas: para dibujar gráficos y manejar elementos gráficos.

Así como la gestión de estos: Tkinter nos ofrece varios métodos para gestionar la disposición de los widgets en la ventana, como pack(), grid() y place(). También nos

interesa la gestión de eventos y *callbacks*: nos permite manejar eventos de usuario como clics de ratón, pulsaciones de teclas, etc., y asociar funciones *callback* a estos eventos.

Por último, también cabe destacar la portabilidad de este módulo: las aplicaciones creadas con Tkinter son portátiles y pueden ejecutarse en múltiples plataformas como Windows, macOS y Linux sin necesidad de cambios en el código.

## Funciones interfaz

En el menú principal encontraremos los dos modos con los que contará el prototipo (Imagen 44). En el modo automático, podremos entrar al menú de Configuración Inicial, permitiendo obtener la secuencia de movimientos relacionada con la configuración inicial que hayamos introducido, además tendremos la opción de enviar la secuencia entera a ArduinoIDE (Enviar Solución a Arduino) o la de enviar tan solo un movimiento de la secuencia. Por otra parte, el modo manual nos permitirá entrar al menú de Giros Manuales, donde podremos enviar el movimiento que deseemos a cualquiera de los seis motores; o poder realizar una secuencia de giros aleatorios, entre 10 y 30.

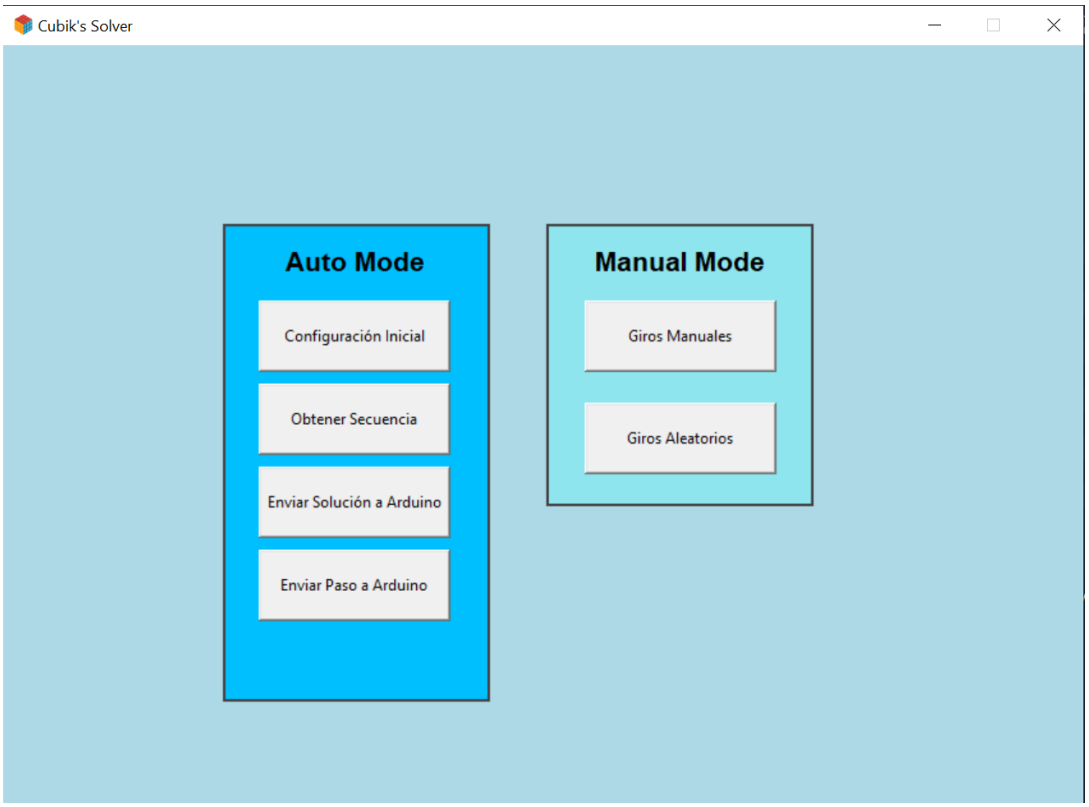



*Imagen 44. Menú principal GUI.*

**Configuración Inicial**. Dentro de este menú (Imagen 45) nos aparecerán las seis caras del cubo. Haciendo clic en las esquinas o aristas de cualquier cara podremos cambiar el color de estas, usando los *radiobutton* colocados a la derecha de la cara UP del cubo. Por supuesto, las piezas centrales no podrán cambiarse de color, ya que estas son las que definen las caras. Una vez hayamos definido todos los colores del cubo, podremos pulsar el botón de "Confirmar configuración", obteniendo así la cadena de configuración inicial precisada.

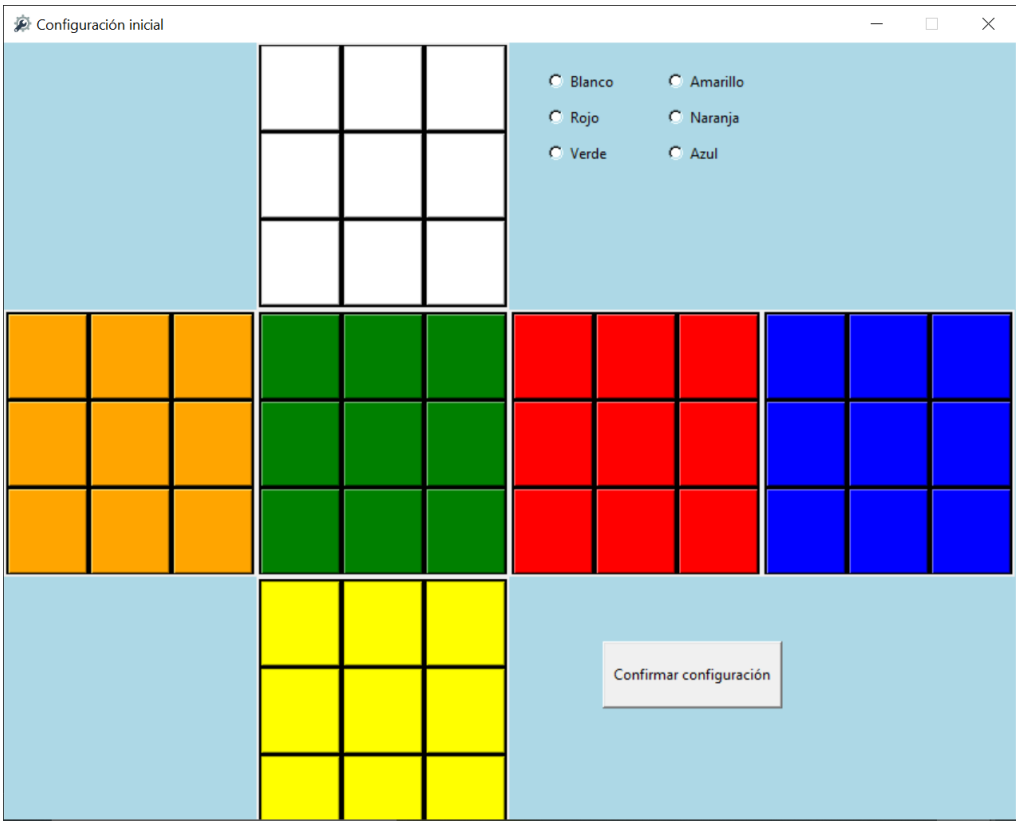



*Imagen 45. Menú de Configuración Inicial GUI.*

Si hemos introducido una configuración errónea, al pulsar el botón de confirmar nos saltará un error con el mensaje: “Hay algún error en la configuración inicial. Revísalo”.

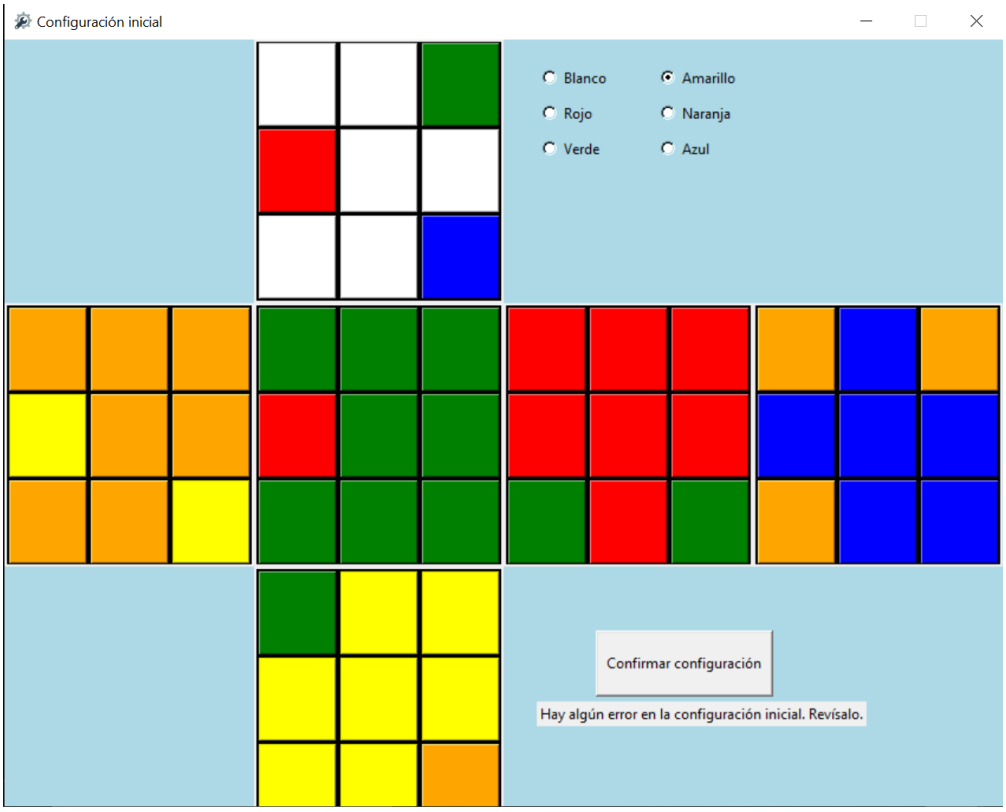



*Imagen 46. Mensaje de error en la Configuración Inicial GUI.*

Si la configuración es posible (es correcta), al pulsar el botón de confirmar el programa devolverá la cadena de la configuración introducida.

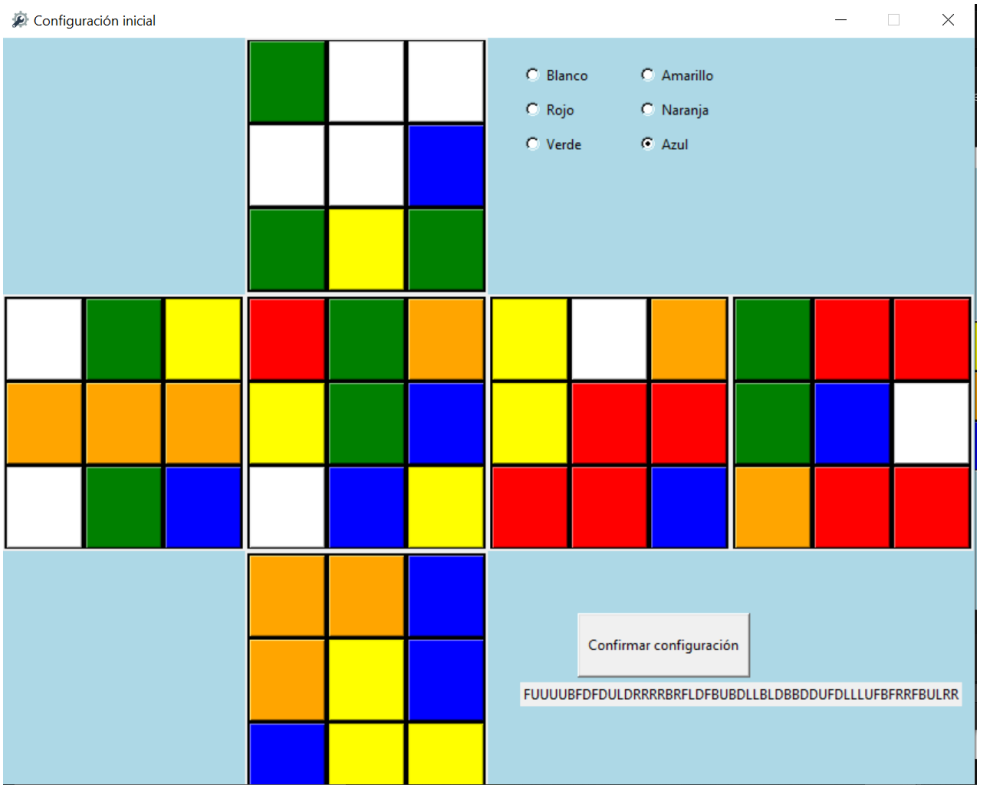



*Imagen 47. Configuración correcta en Configuración Inicial GUI.*

**Obtener Secuencia**. Cuando presionemos esta opción nos aparecerá en un recuadro la secuencia de movimientos necesaria para resolver el cubo según la configuración inicial que hayamos introducido (Imagen 48). Esta opción solo estará disponible cuando hayamos confirmado alguna configuración.

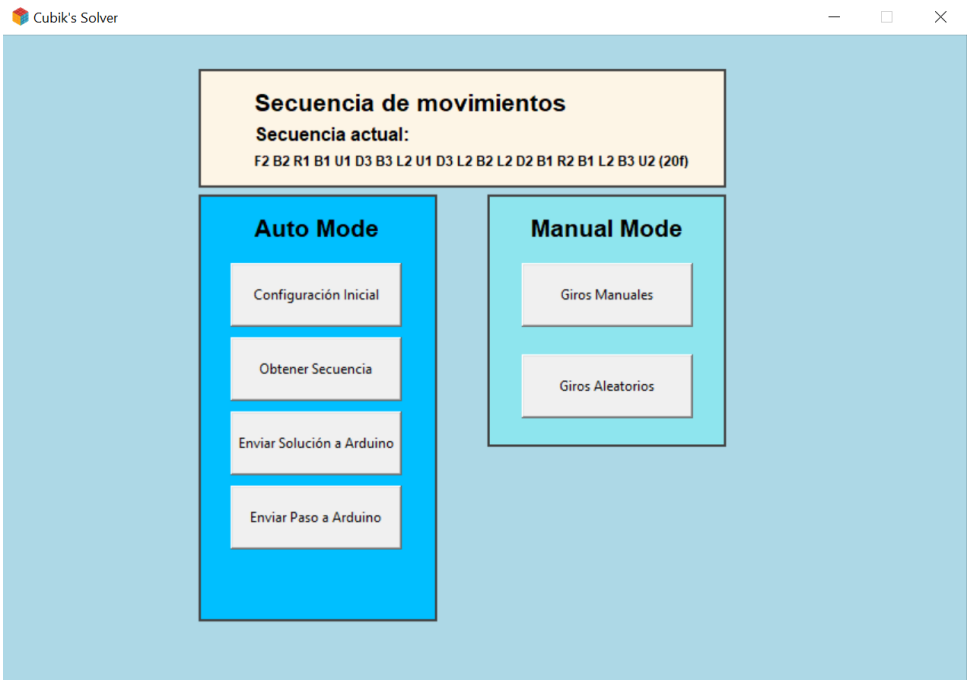



*Imagen 48. Obtener Secuencia GUI.*

**Enviar Paso a Arduino**. Al presionar sobre esta opción, Python enviará mediante el puerto serie el movimiento que toque dentro de la secuencia actual. Los pasos restantes aparecerán debajo del botón (Imagen 49).

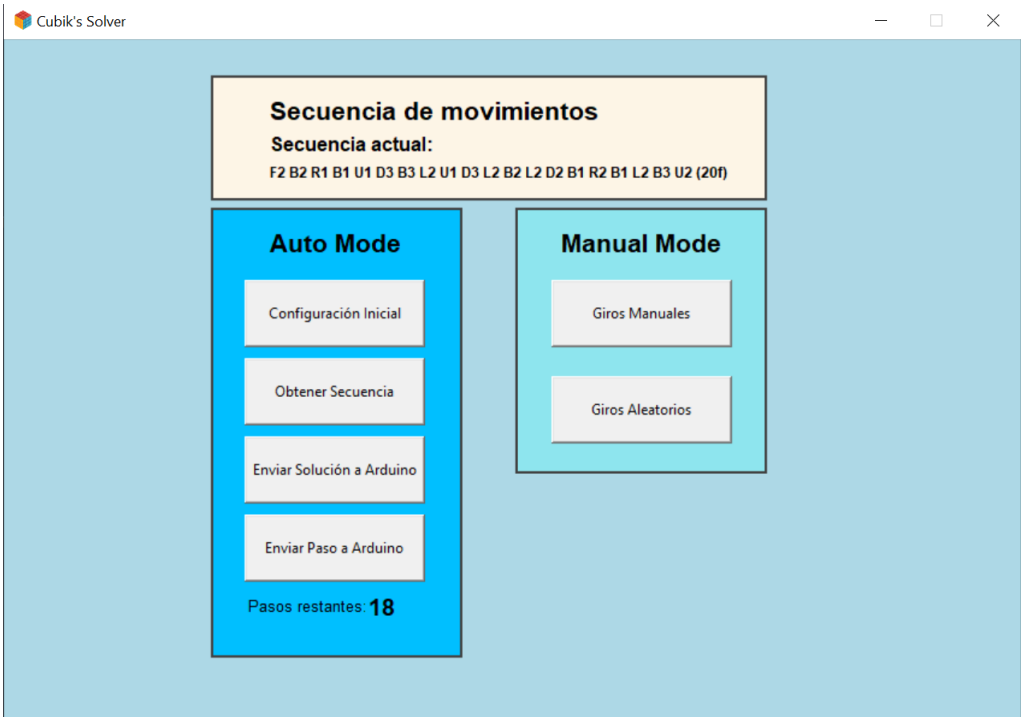



*Imagen 49. Enviar Paso a Arduino GUI.*

En caso de no tener conectado el puerto USB o simplemente no se pueda establecer la conexión mediante el puerto serie, saltará un mensaje de error: “No se encuentra el puerto serie. Revísalo”.

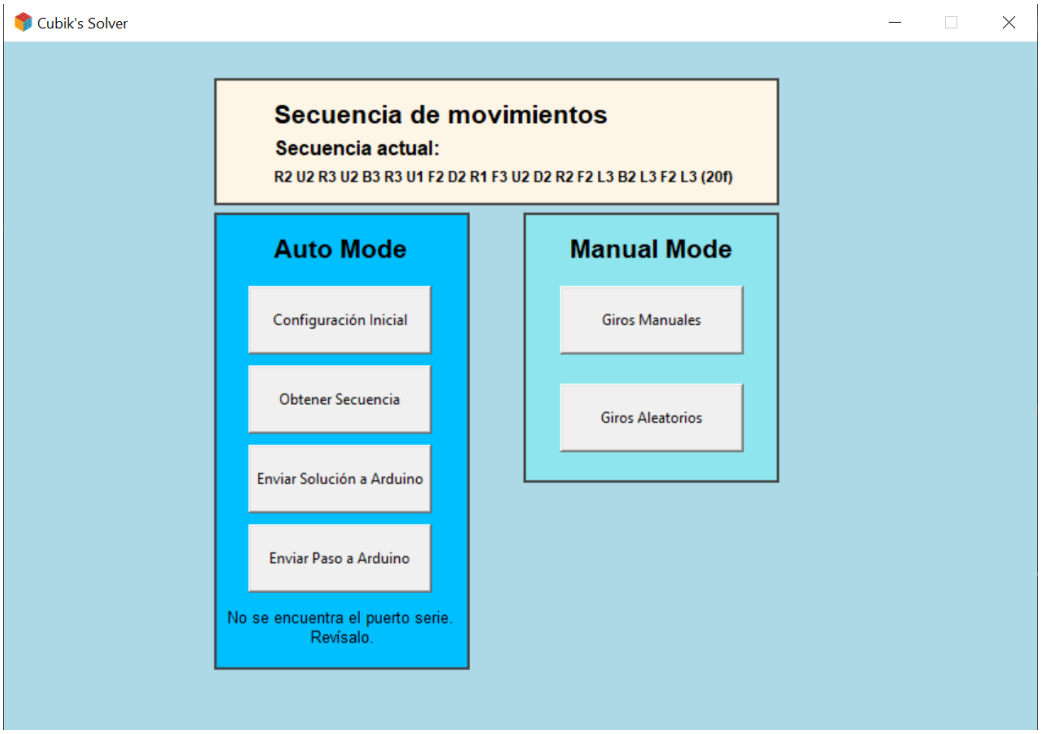



*Imagen 50. Mensaje error conexión serial GUI.*

**Giros Manuales**. En este menú (Imagen 51) nos aparecerán de nuevo las seis caras del cubo. En cada una de ellas tendremos las opciones para realizar cada uno de los tres giros posibles. Presionando sobre el botón de cada uno mandaremos mediante conexión serial el giro y el sentido que debe realizar el motor correspondiente a la cara seleccionada. Del mismo modo que antes, si no se puede establecer conexión saltará un mensaje de error, advirtiendo así al usuario.

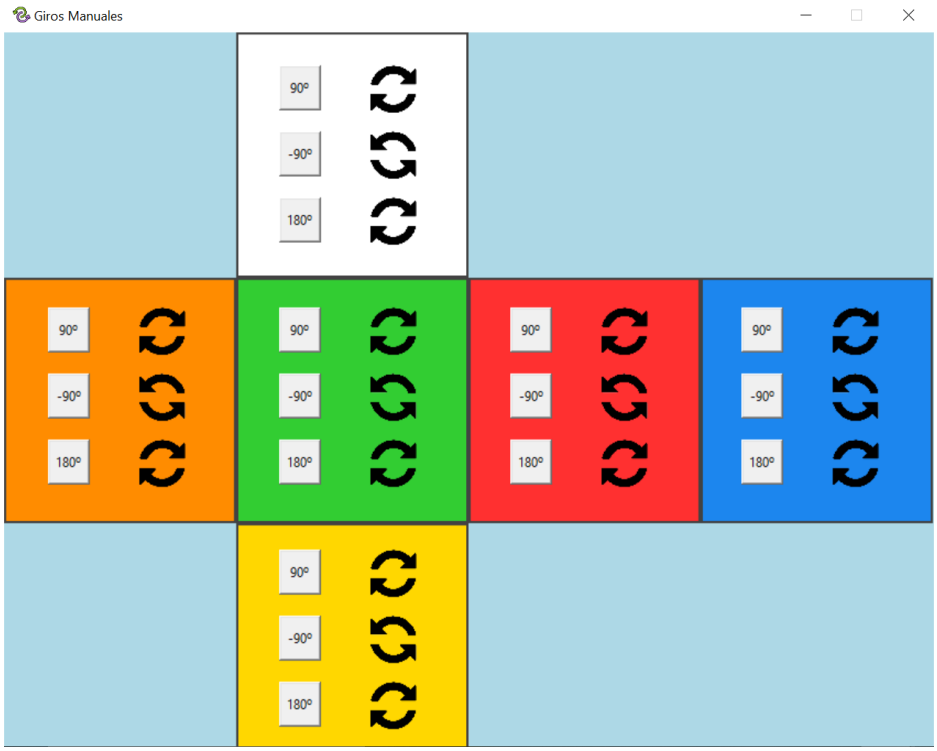



*Imagen 51. Menú de Giros Manuales GUI.*

**Giros Aleatorios**. Al presionar sobre esta opción se abrirá el menú de la Imagen 52. Aquí podremos seleccionar la cantidad de movimientos, entre un rango de 10 a 30, a través de la *slider*. Cuando hayamos seleccionado esta cantidad, pulsamos sobre el botón "Generar movimientos". En el menú principal seleccionamos la opción "Generar secuencia" para cargar la secuencia de movimientos aleatorios recién generados. Podemos enviárselo a Arduino con las dos opciones ya explicadas anteriormente.

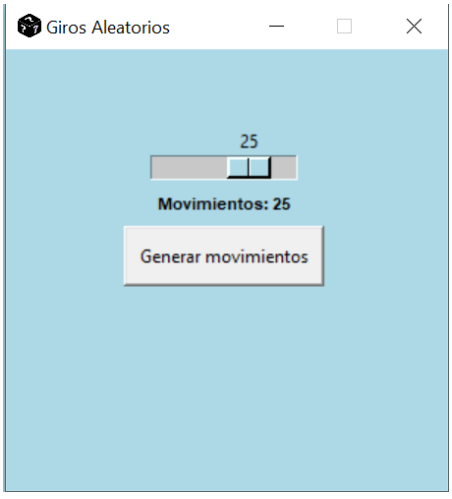



*Imagen 52. Menú Giros Aleatorios GUI.*

## Código

En primer lugar, para poder usar el módulo de Tkinter debemos importar la librería:

```
import tkinter as tk
```

Con el objetivo de tener una mejor organización en nuestro script, creamos una clase para cada menú. Definimos la clase "MainWindow" para el menú principal, pasándole como entrada la función Tk de la librería, que define la que será la "raíz" del programa.

```
class MainWindow(tk.Tk):
```

Al resto de ventanas se le pasa el parámetro Toplevel, función de la librería también, que define a estos menús como ventanas de segundo nivel, por debajo de la principal.

```
class ConfigWindow(tk.Toplevel): […]
class ManualWindow(tk.Toplevel): […]
class RandomWindow(tk.Toplevel): […]
```

Dentro de cada clase definimos una función que hereda de la clase las propiedades. En esta definimos todos los widgets que vamos a incluir en el menú: sus parámetros, color, tamaño, variable asociada, función asociada, etc.

```
def __init__(self, *args, **kwargs):
```

Además, definimos las funciones que se asociarán a los botones. Por ejemplo, en la clase de la ventana principal definimos la función "python2arduino". Esta será la encargada de que al pulsar el botón "Enviar Solución a Arduino" se establezca una conexión mediante el puerto serie, enviando la secuencia de movimientos generada, almacenada en la variable "sequence". Además, mediante un *try except* se generará un mensaje de error en caso de no poder establecer la conexión.

```
def python2arduino(self):
        global sequence

        try:
            # Configuración puerto Serial
            com = serial.Serial("COM4", 9600, write_timeout= 10)
            time.sleep(1)
            # Escribimos por el puerto serie la cadena a enviar a
arduino
            com.write(sequence.encode("ascii"))
            time.sleep(0.1)
            # Cerramos el puerto serie
            com.close()
        except:
            self.title_steps.config(text="No se encuentra el puerto
serie.\n Revísalo.")
```

```
            self.title_steps.place(x=180, y=460)
```

En la función __init__ de cada una de las ventanas secundarias se incluyen una serie de acciones. La función “focus()” hará que la ventana secundaria se ponga en primer plano en nuestro equipo. Se crea un atributo de clase de tipo booleano para saber cuando una ventana secundaria ha sido abierta, de este modo, hasta que no se cierre la actual no se podrá abrir otra secundaria. Además, mediante la función “protocol()” se asigna la función “acción_al_cerrar” a la acción de cerrar la ventana mediante la cruz del menú de herramientas de la ventana (“WM_DELETE_WINDOW”).

```
# Acciones
        self.focus()
        self.__class__.loading = True

        # Asociar la función accion_al_cerrar() al cierre de la
ventana
        self.protocol("WM_DELETE_WINDOW", self.accion_al_cerrar)
```

Esta función comentada, se encontrará en las tres clases de ventana secundaria, encargándose de cambiar el valor del atributo de clase a False, además de invocar el método “destroy()”, que destruirá la ventana secundaria.

```
def accion_al_cerrar(self):
        self.__class__.loading = False
        return super().destroy()
```

En el script principal simplemente crearemos una variable de la clase “MainWindow”, encargada de invocar el menú principal. Además, se definen las variables globales que usaremos en varias clases y por último, se invoca el método “mainloop()”, imprescindible cuando estamos usando la librería Tkinter.

```
# Main
main_window = MainWindow()

# Variables globales
InitialStr = tk.StringVar()
cara = tk.StringVar()
letra_cara = tk.StringVar()
name_color = tk.StringVar()
sequence = tk.StringVar()
color = tk.IntVar()
stepsleft = tk.IntVar()

main_window.mainloop()
```

# 4 Conclusión y desarrollos futuros

En el presente trabajo se ha conseguido desarrollar un prototipo que cumple con los objetivos planteados satisfactoriamente.

Se ha desarrollado un **diseño 3D rígido**, **efectivo** y **económico**, ya que el material escogido ha sido la madera de contrachapado de abedul, optando por un diseño con piezas planas, ahorrando en plástico y tiempo de impresión. El diseño propuesto funciona correctamente tanto en la simulación como en la implementación real.

Se ha conseguido **simular** el **comportamiento** del **sistema**, obteniendo gráficas del movimiento de los motores en el tiempo. Además, se comprueba que las uniones de piezas diseñadas y la colocación de motores y cubo son correctas.

La **conexión** entre la **interfaz de usuario** y el **sistema de motores** es satisfactoria, reaccionando los motores de forma correcta antes las instrucciones de cada una de las opciones que nos permite utilizar la interfaz.

El prototipo permite sacar e introducir el cubo en él de forma sencilla, teniendo que mover tan solo la pieza superior que alberga el motor de la cara blanca. Finalmente, este consigue **resolver** el cubo **desde cualquier configuración inicial** en menos de **10 segundos**. Además, el coste de este no es muy elevado, estando sobre los 110€.

No obstante, el prototipo consta de una serie de limitaciones, que pueden resolverse realizando **futuras mejoras**:

- En ocasiones, el eje del motor no realiza el giro de 90 o 180° de forma precisa, arrastrando un error en el giro de la cara que acaba atascando al mecanismo, no permitiendo que las caras giren. Esto puede ser debido al propio motor (es de los paso a paso más baratos) o a la calidad del cubo (puede mejorar con un girado más suave o un cubo magnético). Podría solucionarse totalmente con el uso de servomotores potentes con encoder. Esto permitiría enviar a nuestra controladora la posición angular de los motores constantemente, pudiendo así utilizar reguladores con una retroalimentación de la posición y permitiendo reajustar la posición de la cara del cubo, quedando siempre alineada para permitir un giro correcto.
- Mejorar la velocidad de resolución. La opción descrita anteriormente también permitiría aumentar la velocidad de giro de los motores, ya que los actuales se descontrolan más al aumentar la velocidad de giro fijada en Arduino.
- Se ha intentado implementar un **sistema de visión artificial** usando la librería OpenCV de Python. Esta nos permite programar un sistema que procese en tiempo real información a través de una cámara. El problema reside en encontrar una buena cámara y, sobre todo, una iluminación que sea siempre igual, ya que, los parámetros que se usan en el filtro no pueden cambiarse en tiempo real según la iluminación que incida en la cara del cubo. Es por ello, que una posible mejora sería montar un sistema de iluminación que sea constante y permita al algoritmo de visión reconocer los colores de las piezas del cubo siempre. Con esto, ganaríamos en velocidad de preparación, ya que resulta más cómodo que introducir los colores manualmente.

# 5 Bibliografía

[1] SolidWorks, «Requisitos del sistema de gestión de datos de SOLIDWORKS y SW» [En línea]. Disponible en: https://www.solidworks.com/es/support/system-requirements [Última fecha de consulta: 20.06.24].

[2] MATLAB, «MATLAB R2024a System Requirements for Windows» [En línea]. Disponible en: https://es.mathworks.com/support/requirements/matlab-system-requirements.html [Última fecha de consulta: 20.06.24].

[3] Visual Studio Code, «Requirements for Visual Studio Code» [En línea]. Disponible en: https://code.visualstudio.com/docs/supporting/requirements [Última fecha de consulta: 20.06.24].

[4] ArduinoIDE, «Como instalar Arduino en Windows» [En línea]. Disponible en: https://arduino.cl/como-instalar-arduino-en-windows/ [Última fecha de consulta: 20.06.24].

[5] Kubekings, «Notación Cubo de Rubik» [En línea]. Disponible en: https://kubekings.com/blog/post/notacion-cubo-de-rubik [Última fecha de consulta: 29.05.24].

[6] Kubekings, «Notación Cubo de Rubik» [En línea]. Disponible en: https://kubekings.com/blog/post/notacion-cubo-de-rubik [Última fecha de consulta: 29.05.24].

[7] Algoritmo Kociemba de dos fases, «RubiksCube-TwophaseSolver» [En línea]. Disponible en: https://github.com/hkociemba/RubiksCube-TwophaseSolver/blob/master/README.md [Última fecha de consulta: 24.04.24].

[8] Algoritmo de Dios, «El número de Dios es 20» [En línea]. Disponible en: https://www-cube20-org.translate.goog/?_x_tr_sch=http&_x_tr_sl=auto&_x_tr_tl=es&_x_tr_hl=en [Última fecha de consulta: 15.05.24].

[9] Simscape Multibody, «Simscape - Modele y simule sistemas físicos multidominio» [En línea]. Disponible en: https://es.mathworks.com/products/simscape.html [Última fecha de consulta: 28.03.24].

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ETSI Aeroespacial y Diseño Industrial

# UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Escuela Técnica Superior de Ingeniería Aeroespacial y Diseño Industrial**

## DISEÑO, SIMULACIÓN E IMPLEMENTACIÓN DE UN ROBOT PARA RESOLVER EL CUBO DE RUBIK

## DOCUMENTO II: PLIEGO DE CONDICIONES

*TRABAJO FINAL DEL*

**Máster en Ingeniería Mecatrónica**

*REALIZADO POR*

**Sergio Martínez Olmos**

*TUTORIZADO POR*

**Vicente Fermín Casanova Calvo**

**CURSO ACADÉMICO: 2023/2024**

# Índice del pliego de condiciones

# 1. Objeto

El objeto del presente documento se refiere a las condiciones generales y específicas mínimas requeridas para desarrollar el diseño, simulación e implementación del prototipo descrito en el proyecto.

En este caso, al tratarse de un proyecto destinado a la docencia y no tratarse de un producto que vaya a comercializarse, el ámbito de aplicación se extenderá a la descripción de los equipos de simulación y control de motores, así como de los materiales empleados y requisitos para la correcta puesta en marcha del prototipo. No se establecerán condiciones económicas y legales.

# 2. Normativa

En el siguiente apartado se mencionará toda la normativa vigente en España de carácter general relacionada con el proyecto. Todo proyecto industrial debe cumplir con las normas y condiciones de carácter general dictadas por el Organismo de Normalización en España, con el fin de asegurar al autor y a aquellas personas que participan en el proyecto.

| Código | Nombre | Fecha publicación |
|---|---|---|
| UNE 157001:2014 | Criterios generales para la elaboración formal de los documentos que constituyen un proyecto técnico | 2014/06/18 (Ratificada) |
| UNE-EN 61508:2011 | Seguridad funcional de los sistemas eléctricos/electrónicos/electrónicos programables relacionados con la seguridad. | 2011/03/30 (Ratificada) |
| UNE-EN ISO 11442:2006 | Documentación técnica de productos. Gestión de documentos | 2006/06/28 (Ratificada) |
| UNE-EN ISO 16090-1:2023 | Seguridad de las máquinas herramienta. Centros de mecanizado, centros de fresado, máquinas transfer. Parte 1: Requisitos de seguridad. | 2023/11/08 (Ratificada) |
| UNE-EN ISO 888:2019 | Elementos de fijación. Pernos, tornillos y espárragos. Longitudes nominales y longitudes roscadas. | 2019/04/24 (Ratificada) |
| UNE 1027:1995 | Dibujos técnicos. Plegado de planos. | 1995/01/30 (Ratificada) |

# 3. Condiciones de los equipos de simulación, diseño y control

En este apartado se realiza una descripción de las condiciones que debe cumplir el equipo con el que se realiza la simulación, el diseño y el control del prototipo.

Requisitos mínimos del PC para el empleo del software MATLAB R2024a

- Sistema operativo: Windows 10 (versión 21H2 o superior) o superior.
- CPU: Mínimo procesador Intel o AMD x86-64 de dos o más núcleos.
- GPU: 1GB de memoria GPU recomendada.
- RAM: 8 GB mínimo, 16 GB recomendado.
- Memoria: 3.8 GB solo MATLAB, 4-6 GB para una instalación típica y 23 GB para una instalación de todos los productos.

Requisitos mínimos del PC para el empleo del software SolidWorks 2024

- Sistema operativo: Windows 10 64 bits o superior.
- CPU: Intel o AMD.
- GPU: Tarjetas y controladores recomendadas en su página web.
- RAM: 16 GB o más.
- Memoria: 5 GB.

Requisitos mínimos del PC para el empleo del software ArduinoIDE

- Sistema operativo: Windows 10 64 bits o superior.
- CPU: Pentium 4 o superior.
- RAM: 256 MB o más.
- Memoria: 1 GB.

Requisitos mínimos del PC para el empleo del software Visual Studio Code

- Sistema operativo: Windows 10 64 bits o superior.
- CPU: Procesador de 1.6 GHz o más rápido.
- RAM: 1 GB.
- Memoria: 500 MB.

# 4. Condiciones de ejecución del prototipo

En este apartado se describen las condiciones necesarias para poder ejecutar la puesta en marcha del prototipo.

En primer lugar, debemos asegurar que la tarjeta ELEGOO esté conectada correctamente al ordenador, asegurando que el puerto USB coincide con el puerto COM descrito en ambos programas (ArduinoIDE y Visual Studio Code). También se comprueba que el programa se haya enviado a la tarjeta correctamente.

Para comprobar que ambos programas se conectan correctamente, se procede a conectar la alimentación y posteriormente, entramos en el modo manual de la interfaz de usuario y enviamos una señal a uno de los motores, comprobando que este recibe la información y ejecuta el movimiento seleccionado.

Una vez comprobada la electrónica, comprobamos que los motores se encuentran fijos en la estructura y se procede a colocar el cubo, comprobando que cada una de las caras coincide con su motor destinado. Antes de iniciar la secuencia de movimientos se comprueba que todas las piezas agarraderas se encuentran bien sujetas a las piezas solidarias a los ejes de los motores.

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ETSI Aeroespacial y Diseño Industrial

# UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Escuela Técnica Superior de Ingeniería Aeroespacial y Diseño Industrial**

## DISEÑO, SIMULACIÓN E IMPLEMENTACIÓN DE UN ROBOT PARA RESOLVER EL CUBO DE RUBIK

## DOCUMENTO III: PRESUPUESTO

*TRABAJO FINAL DEL*

**Máster en Ingeniería Mecatrónica**

*REALIZADO POR*

**Sergio Martínez Olmos**

*TUTORIZADO POR*

**Vicente Fermín Casanova Calvo**

**CURSO ACADÉMICO: 2023/2024**

# Índice del presupuesto

# 1. Introducción

En el siguiente documento se expondrá el presupuesto del proyecto, incluyendo los costes parciales del diseño 3D, la simulación del modelo propuesto, la programación del control de los motores y de la interfaz de usuario y del montaje del prototipo, así como los costes de licencia de los softwares empleados y la amortización del hardware vinculado al proyecto.

También, añadiremos los gastos de material que han sido comprados para la creación del prototipo, descartando los gastos de impresión 3D, para las agarraderas, y los gastos del cortado láser, para el resto de las piezas, ya que estos costes están a cargo de la UPV. Algún elemento más ha sido reciclado o prestado por la universidad, como veremos más adelante.

Dos de los softwares empleados, en concreto Visual Studio Code y Arduino IDE, son plataformas de código abierto gratuitas, por lo que estas no entran en el apartado de software. Para el resto contaremos con su amortización, aunque el alumno puede usarlas de forma gratuita gracias a los convenios de la universidad con las empresas poseedoras de estas plataformas software.

A continuación, se deja una tabla resumen que describe las distintas secciones en las que se divide el desarrollo del proyecto, así como el tiempo invertido en cada una de estas.

| **Sección** | **Tiempo invertido (h)** |
|---|---:|
| Diseño 3D | 35 |
| Simulación | 25 |
| Programación | 120 |
| Montaje | 25 |
| Redacción del proyecto | 35 |
| **Tiempo total** | **240** |

*Tabla 1. Desglose en secciones del desarrollo del proyecto.*

# 2. Precios elementales

## 2.1 Mano de obra

| Código | Descripción | Salario (€/h) |
|---|---|---|
| **MO.MIM** | Ingeniero máster en ingeniería mecatrónica | 16 |
| **MO.TI** | Técnico de impresoras | 12 |

*Tabla 2. Coste de mano de obra.*

## 2.2 Maquinaria utilizada

En este apartado se incluirá la amortización del hardware utilizado durante el desarrollo del prototipo, en este caso un ordenador portátil *ASUS harman/kardon i7 8th Gen*, valorado en el año de su compra en 750€, así como la máquina de cortado láser, valorada en 4500€ y la impresora 3D modelo Bambu Lab P1S, valorada en 860€. Además, se incluye la amortización del software empleado, en este caso SolidWorks, MATLAB Simulink y Fritzing, cuyas licencias son de pago.

### Hardware

Para el portátil se estima un uso de medio de 5 horas diarias, en el periodo de 225 días laborales. Además, se estima una vida útil del ordenador de 4 años. Se procede a calcular la amortización del ordenador:

$$Amort.MU.OP = \frac{base\ de\ amort.\ (€)}{vida\ útil\ (h)} = \frac{750€}{4\ años \cdot 225\ días \cdot 5\ h/día} = 0.167\ {}^{€}/_{h}$$

De igual manera, para la máquina de cortado láser, se estima una vida útil de entre 10000 y 15000 horas. Con estos datos se calcula la amortización de la máquina:

$$Amort.MU.MCL = \frac{base\ de\ amort.\ (€)}{vida\ útil\ (h)} = \frac{4500€}{12500\ h} = 0.36\ {}^{€}/_{h}$$

Estimando una vida útil de 20000 horas para una impresora 3D que no es de uso industrial, se procede a calcular la amortización de esta:

$$Amort.MU.I3D = \frac{base\ de\ amort.\ (€)}{vida\ útil\ (h)} = \frac{860€}{20000\ h} = 0.043\ {}^{€}/_{h}$$

### Software

La licencia de SolidWorks es de 4000€ anuales. Estimando un uso medio de 5 horas durante 225 días laborales, se calcula la amortización del programa:

$$Amort.MU.SW = \frac{base\ de\ amort.\ (€)}{vida\ útil\ (h)} = \frac{4000€}{1\ año \cdot 225\ días \cdot 5\ h/día} = 3.55\ {}^{€}/_{h}$$

La licencia de MATLAB es de 190€ anuales. Estimando un uso medio de 5 horas durante 225 días laborales, se calcula la amortización del programa:

$$Amort.MU.MS = \frac{base\ de\ amort.(€)}{vida\ útil\ (h)} = \frac{190€}{1\ año \cdot 225\ días \cdot 5\ h/día} = 0.169\ {}^{€}/_{h}$$

La licencia de Fritzing es de 25€. Estimando un uso medio de 2 horas durante 225 días laborales, calculamos la amortización:

$$Amort.MU.FZ = \frac{base\ de\ amort.(€)}{vida\ útil\ (h)} = \frac{25€}{1\ año \cdot 225\ días \cdot 2\ h/día} = 0.056\ {}^{€}/_{h}$$

| Código | Descripción | Coste (€/h) |
|---|---|---|
| **MU.OP** | Ordenador portátil ASUS | 0.167 |
| **MU.MCL** | Máquina de cortado láser | 0.36 |
| **MU.I3D** | Impresora 3D | 0.043 |
| **MU.SW** | Software SolidWorks | 3.55 |
| **MU.MS** | Software MATLAB Simulink | 0.169 |
| **MU.FZ** | Software Fritzing | 0.056 |

*Tabla 3. Coste de maquinaria utilizada.*

## 2.3 Materiales

El condensador y el multímetro digital han sido prestados por el departamento de automática de la UPV. El cargador portátil ha sido reciclado de un antiguo ordenador ya en desuso.

| Código | Descripción | Unidades | Precio unitario(€) | Coste (€) |
|---|---|---|---|---|
| **MU.MN** | Motor paso a paso Nema 17 Bipolar | 6 | 8.37 | 50.24 |
| **MU.DR** | Driver A4988 | 6 | 2.3 | 13.8 |
| **MU.CNC** | CNC Shield v3 | 1 | 6.8 | 6.8 |
| **MU.EM** | Tarjeta ELEGOO Mega 2560 R3 | 1 | 25.99 | 25.99 |
| **MU.CC** | Condensador 100 µF | 1 | - | - |
| **MU.CMM** | Cables macho-macho | 40 | - | 4.99 |
| **MU.CHH** | Cables hembra-hembra | 40 | - | 4.99 |
| **MU.PB** | Placa protoboard | 1 | 4 | 4 |
| **MU.CP** | Cargador portátil ASUS | 1 | - | - |
| **MU.MD** | Multímetro digital | 1 | - | - |
| **MU.TM3** | Tornillos M3 | 20 | - | 2.69 |
| **MU.TU6** | Tuercas M6 | 20 | - | 2.19 |
| **MU.VM** | Varilla de metal 6 mm de diámetro | 1 | 1.19 | 1.19 |

*Tabla 4. Coste de materiales utilizados.*

# 3. Precios descompuestos

A continuación, se mostrarán las tablas del precio descompuesto de cada una de las partes que forman el proyecto, indicando el código, descripción, rendimiento en horas, coste en euro la hora e importe total en euros, de cada elemento que interviene.

Justificación del coste del Diseño 3D:

| MO.1 – Diseño 3D | | | | |
|---|---|---|---|---|
| **Código** | **Descripción** | **Rendimiento (h)** | **Coste(€/h)** | **Importe (€)** |
| **MO.MIM** | Ingeniero máster en ingeniería mecatrónica | 35 | 16 | 560 |
| **MO.TI** | Técnico de impresoras | 2 | 12 | 24 |
| **MU.OP** | Ordenador portátil ASUS | 33 | 0.167 | 5.52 |
| **MU.MCL** | Máquina de cortado láser | 1 | 0.36 | 0.36 |
| **MU.I3D** | Impresora 3D | 1 | 0.043 | 0.05 |
| **MU.SW** | Software SolidWorks | 33 | 3.55 | 117.15 |

*Tabla 5. Precio descompuesto de Diseño 3D (MO.1).*

Justificación del coste de Simulación:

| MO.2 – Simulación | | | | |
|---|---|---|---|---|
| **Código** | **Descripción** | **Rendimiento (h)** | **Coste(€/h)** | **Importe (€)** |
| **MO.MIM** | Ingeniero máster en ingeniería mecatrónica | 25 | 16 | 400 |
| **MU.OP** | Ordenador portátil ASUS | 25 | 0.167 | 4.18 |
| **MU.MS** | Software MATLAB Simulink | 25 | 0.169 | 4.23 |

*Tabla 6. Precio descompuesto de Simulación (MO.2).*

Justificación del coste de Programación:

| MO.3 – Programación | | | | |
|---|---|---|---|---|
| **Código** | **Descripción** | **Rendimiento (h)** | **Coste(€/h)** | **Importe (€)** |
| **MO.MIM** | Ingeniero máster en ingeniería mecatrónica | 120 | 16 | 1920 |
| **MU.OP** | Ordenador portátil ASUS | 120 | 0.167 | 20.04 |

*Tabla 7. Precio descompuesto de Programación (MO.3).*

Justificación del coste de Montaje:

| MO.4 – Montaje | | | | |
|---|---|---|---|---|
| **Código** | **Descripción** | **Rendimiento (h)** | **Coste(€/h)** | **Importe (€)** |
| **MO.MIM** | Ingeniero máster en ingeniería mecatrónica | 25 | 16 | 400 |
| **MU.OP** | Ordenador portátil ASUS | 20 | 0.167 | 3.34 |

*Tabla 8. Precio descompuesto de Montaje (MO. 4).*

Justificación del coste de Redacción del proyecto:

| MO.5 – Redacción del proyecto | | | | |
|---|---|---|---|---|
| **Código** | **Descripción** | **Rendimiento (h)** | **Coste(€/h)** | **Importe (€)** |
| **MO.MIM** | Ingeniero máster en ingeniería mecatrónica | 35 | 16 | 560 |
| **MU.OP** | Ordenador portátil ASUS | 35 | 0.167 | 5.85 |

*Tabla 9. Precio descompuesto de Redacción del proyecto (MO.5).*

# 4. Precios unitarios

En este apartado se muestra una tabla con los precios unitarios de cada una de las secciones del proyecto.

| Código | Descripción | Coste (€) |
|---|---|---|
| **MO.1** | Diseño 3D | 707.08 |
| **MO.2** | Simulación | 408.41 |
| **MO.3** | Programación | 1940.04 |
| **MO.4** | Montaje | 403.34 |
| **MO.5** | Redacción del proyecto | 565.85 |

*Tabla 10. Precio unitario de cada sección.*

# 5. Resumen de presupuesto

| Secciones | Importe (€) |
|---|---|
| Sección 1. Diseño 3D | 707.08 |
| Sección 2. Simulación | 408.41 |
| Sección 3. Programación | 1940.04 |
| Sección 4. Montaje | 403.34 |
| Sección 5. Redacción del proyecto | 565.85 |
| **Materiales** | 116.88 |
| **Presupuesto de ejecución material** | 4139.60 |
| Gastos generales (13%) | 538.148 |
| Beneficio industrial (6%) | 248.376 |
| Suma | 4926.12 |
| IVA (21%) | 1034.48604 |
| **Presupuesto de ejecución por contrata** | **5960.61** |

*Tabla 11. Resumen del presupuesto.*

El presupuesto final asciende a la cantidad de CINCO MIL NOVECIENTOS SESENTA EUROS CON SESENTA Y UN CÉNTIMOS.

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ETSI Aeroespacial y Diseño Industrial

# UNIVERSITAT POLITÈCNICA DE VALÈNCIA

**Escuela Técnica Superior de Ingeniería Aeroespacial y Diseño Industrial**

## DISEÑO, SIMULACIÓN E IMPLEMENTACIÓN DE UN ROBOT PARA RESOLVER EL CUBO DE RUBIK

## DOCUMENTO IV: ANEXOS

*TRABAJO FINAL DEL*

**Máster en Ingeniería Mecatrónica**

*REALIZADO POR*

**Sergio Martínez Olmos**

*TUTORIZADO POR*

**Vicente Fermín Casanova Calvo**

**CURSO ACADÉMICO: 2023/2024**

# Índice de los anexos

# 1. Script Arduino

```
// Librerías
#include <SoftwareSerial.h>

// Control Motor Front (Green) --> Driver green wires
#define mF 7
#define pwmF 4
// Control Motor Back (Blue) --> Driver blue wires
#define mB 5
#define pwmB 2
// Control Motor Down (Yellow) --> Y axis CNC shield
#define mD 33
#define pwmD 31
// Control Motor Up (White) --> X axis CNC shield
#define mU 6
#define pwmU 3
// Control Motor Right (Red) --> A axis CNC shield
#define mR 47
#define pwmR 45
// Control Motor Left (Orange) --> Z axis CNC shield
#define mL 13
#define pwmL 12

// Velocidad de giro Motores
#define vel 3000 // Cuanto mayor sea el número, menor es la
velocidad

// Variables
int steps;
bool direction;
String turn;
String face;

void setup() {
  // Velocidad de comunicación puerto serial
  Serial.begin(9600);
  Serial.setTimeout(10);

  // --------------- Pines de salida ---------------
  // Motor Front (Green)
  pinMode(mF, OUTPUT);
  pinMode(pwmF, OUTPUT);
  // Motor Back (Blue)
  pinMode(mB, OUTPUT);
  pinMode(pwmB, OUTPUT);
  // Motor Down (Yellow)
  pinMode(mD, OUTPUT);
```

```
  pinMode(pwmD, OUTPUT);
  // Motor Up (White)
  pinMode(mU, OUTPUT);
  pinMode(pwmU, OUTPUT);
  // Motor Right (Red)
  pinMode(mR, OUTPUT);
  pinMode(pwmR, OUTPUT);
  // Motor Left (Orange)
  pinMode(mL, OUTPUT);
  pinMode(pwmL, OUTPUT);
}

void step(bool dir, int steps, int m, int pwm) {
  digitalWrite(m, dir);
  delay(50);

  for (int i=0;i<steps;i++){
    digitalWrite(pwm, HIGH);
    delayMicroseconds(vel);
    digitalWrite(pwm, LOW);
    delayMicroseconds(vel);
  }
}

void what_face(char face) {
  if (face == 'U'){
    step(direction, steps, mU, pwmU);
  }
  else if (face == 'D'){
    step(direction, steps, mD, pwmD);
  }
  else if (face == 'F'){
    step(direction, steps, mF, pwmF);
  }
  else if (face == 'B'){
    step(direction, steps, mB, pwmB);
  }
  else if (face == 'R'){
    step(direction, steps, mR, pwmR);
  }
  else if (face == 'L'){
    step(direction, steps, mL, pwmL);
  }
  Serial.print(" cara ");
  Serial.println(face);
}

void type_turn(char turn){
  if (turn == '1'){
```

```
      steps = 50;
      direction = false;
      Serial.print("Giramos 90ª clockwise");
  }
  else if (turn == '2'){
    steps = 100;
    direction = true;
    Serial.print("Giramos 180ª a-clockwise");
  }
  else if (turn == '3'){
    steps = 50;
    direction = true;
    Serial.print("Giramos 90ª a-clockwise");
  }
}

void loop() {
  // Cada paso son 1.8 grados. 100 pasos = 180 grados
  // dir true == sentido horario
  while(Serial.available() > 0){
    String prompt = Serial.readStringUntil('\n');
    prompt = prompt.substring(0, prompt.length() - (prompt.length()
- (prompt.indexOf("(")) + 1)); // Nos quedamos solo con los
movimientos del cubo
    Serial.println("Prompt: " + prompt);

    // Recorremos la string recibida
    for (int i=0; i <= prompt.length(); i++){
      if (isDigit(prompt.charAt(i))){
        turn = turn + prompt.charAt(i);
      }
      else if (isalpha(prompt.charAt(i))){
        face = face + prompt.charAt(i);
      }
    }

    for (int i=0; i < face.length(); i++){
      // Establecemos los grados y sentido de giro
      type_turn(turn.charAt(i));
      // Establecemos el motor que vamos a usar
      what_face(face.charAt(i));
      Serial.print("Giro ");
      Serial.print(i + 1);
      Serial.println(" OK");
      delay(300);
    }

    Serial.println("Secuencia finalizada.");
    Serial.println();
```

```
    // Vacíamos las cadenas
    prompt = "";
    turn = "";
    face = "";
    delay(500);
  }
}
```

# 2. Script Python

```
import tkinter as tk
import twophase.solver as sv
import serial, time
import random

def cadena_resolver (Config_inicial):
    Moves = sv.solve(Config_inicial)
    print(Moves)
    return Moves

def modificar_caracter(cadena, indice, nuevo_caracter):
    lista_caracteres = list(cadena)
    lista_caracteres[indice] = nuevo_caracter
    cadena_modificada = ''.join(lista_caracteres)
    return cadena_modificada

def contar_caracteres(cadena):
    recuentos = {}
    for caracter in cadena:
        recuentos[caracter] = recuentos.get(caracter, 0) + 1
    return recuentos

class ConfigWindow(tk.Toplevel):

    # Atributo de clase que indica si la ventana de configuración
está en uso
    loading = False

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.config(width=MainWindow.wroot,
height=MainWindow.hroot)
        self.title("Configuración inicial")
        self.geometry(str(MainWindow.wroot) + "x" +
str(MainWindow.hroot) + "+" +
                      str(MainWindow.x_root)  + "+" +
str(MainWindow.y_root))
        self.resizable(0,0)
        self.iconbitmap('config.ico')

        global color
        global cara
        global letra_cara
        global InitialStr

        # Configuración del marco
```

```
        self.frame = tk.Frame(self)
        self.frame.pack(fill='both', expand=1)
        self.frame.config(bg="lightblue")

        # Cuadrados caras
        self.canvas0 = tk.Canvas(self.frame, width=205, height=217,
background="lightblue", borderwidth=1,
highlightbackground="lightblue")
        self.canvas1 = tk.Canvas(self.frame, width=205, height=217,
background="black", borderwidth=1)
        self.canvas2 = tk.Canvas(self.frame, width=205, height=217,
background="black", borderwidth=1)
        self.canvas3 = tk.Canvas(self.frame, width=205, height=217,
background="black", borderwidth=1)
        self.canvas4 = tk.Canvas(self.frame, width=205, height=217,
background="black", borderwidth=1)
        self.canvas5 = tk.Canvas(self.frame, width=205, height=217,
background="black", borderwidth=1)
        self.canvas6 = tk.Canvas(self.frame, width=205, height=217,
background="lightblue", borderwidth=1,
highlightbackground="lightblue")
        self.canvas7 = tk.Canvas(self.frame, width=205, height=217,
background="black", borderwidth=1)

        self.canvas0.grid(column=0, row=0)
        self.canvas1.grid(column=1, row=0)
        self.canvas2.grid(column=0, row=1)
        self.canvas3.grid(column=1, row=1)
        self.canvas4.grid(column=2, row=1)
        self.canvas5.grid(column=3, row=1)
        self.canvas6.grid(column=0, row=2)
        self.canvas7.grid(column=1, row=2)

        # ----------------------- Caras -------------------------
-
        # Cara Blanca
        self.ubtn1 = tk.Button(self.frame, bg="white", width=8,
height=4, command=lambda: self.show_rb("u1"))
        self.ubtn1.place(x=215, y=3)
        self.ubtn2 = tk.Button(self.frame, bg="white", width=8,
height=4, command=lambda: self.show_rb("u2"))
        self.ubtn2.place(x=284, y=3)
        self.ubtn3 = tk.Button(self.frame, bg="white", width=8,
height=4, command=lambda: self.show_rb("u3"))
        self.ubtn3.place(x=284+69, y=3)
        self.ubtn4 = tk.Button(self.frame, bg="white", width=8,
height=4, command=lambda: self.show_rb("u4"))
        self.ubtn4.place(x=215, y=76)
```

```
        self.ubtn5 = tk.Button(self.frame, bg="white", width=8,
height=4, state="disabled").place(x=284, y=76)
        self.ubtn6 = tk.Button(self.frame, bg="white", width=8,
height=4, command=lambda: self.show_rb("u6"))
        self.ubtn6.place(x=284+69, y=76)
        self.ubtn7 = tk.Button(self.frame, bg="white", width=8,
height=4, command=lambda: self.show_rb("u7"))
        self.ubtn7.place(x=215, y=76+73)
        self.ubtn8 = tk.Button(self.frame, bg="white", width=8,
height=4, command=lambda: self.show_rb("u8"))
        self.ubtn8.place(x=284, y=76+73)
        self.ubtn9 = tk.Button(self.frame, bg="white", width=8,
height=4, command=lambda: self.show_rb("u9"))
        self.ubtn9.place(x=284+69, y=76+73)
        # Cara Roja
        self.rbtn1 = tk.Button(self.frame, bg="red", width=8,
height=4, command=lambda: self.show_rb("r1"))
        self.rbtn1.place(x=426, y=227)
        self.rbtn2 = tk.Button(self.frame, bg="red", width=8,
height=4, command=lambda: self.show_rb("r2"))
        self.rbtn2.place(x=426+69, y=227)
        self.rbtn3 = tk.Button(self.frame, bg="red", width=8,
height=4, command=lambda: self.show_rb("r3"))
        self.rbtn3.place(x=426+69+69, y=227)
        self.rbtn4 = tk.Button(self.frame, bg="red", width=8,
height=4, command=lambda: self.show_rb("r4"))
        self.rbtn4.place(x=426, y=227+73)
        self.rbtn5 = tk.Button(self.frame, bg="red", width=8,
height=4, state="disabled").place(x=426+69, y=227+73)
        self.rbtn6 = tk.Button(self.frame, bg="red", width=8,
height=4, command=lambda: self.show_rb("r6"))
        self.rbtn6.place(x=426+69+69, y=227+73)
        self.rbtn7 = tk.Button(self.frame, bg="red", width=8,
height=4, command=lambda: self.show_rb("r7"))
        self.rbtn7.place(x=426, y=227+73+73)
        self.rbtn8 = tk.Button(self.frame, bg="red", width=8,
height=4, command=lambda: self.show_rb("r8"))
        self.rbtn8.place(x=426+69, y=227+73+73)
        self.rbtn9 = tk.Button(self.frame, bg="red", width=8,
height=4, command=lambda: self.show_rb("r9"))
        self.rbtn9.place(x=426+69+69, y=227+73+73)
        # Cara Verde
        self.fbtn1 = tk.Button(self.frame, bg="green", width=8,
height=4, command=lambda: self.show_rb("f1"))
        self.fbtn1.place(x=215, y=227)
        self.fbtn2 = tk.Button(self.frame, bg="green", width=8,
height=4, command=lambda: self.show_rb("f2"))
        self.fbtn2.place(x=215+69, y=227)
```

```
        self.fbtn3 = tk.Button(self.frame, bg="green", width=8,
height=4, command=lambda: self.show_rb("f3"))
        self.fbtn3.place(x=215+69+69, y=227)
        self.fbtn4 = tk.Button(self.frame, bg="green", width=8,
height=4, command=lambda: self.show_rb("f4"))
        self.fbtn4.place(x=215, y=227+73)
        self.fbtn5 = tk.Button(self.frame, bg="green", width=8,
height=4, state="disabled").place(x=215+69, y=227+73)
        self.fbtn6 = tk.Button(self.frame, bg="green", width=8,
height=4, command=lambda: self.show_rb("f6"))
        self.fbtn6.place(x=215+69+69, y=227+73)
        self.fbtn7 = tk.Button(self.frame, bg="green", width=8,
height=4, command=lambda: self.show_rb("f7"))
        self.fbtn7.place(x=215, y=227+73+73)
        self.fbtn8 = tk.Button(self.frame, bg="green", width=8,
height=4, command=lambda: self.show_rb("f8"))
        self.fbtn8.place(x=215+69, y=227+73+73)
        self.fbtn9 = tk.Button(self.frame, bg="green", width=8,
height=4, command=lambda: self.show_rb("f9"))
        self.fbtn9.place(x=215+69+69, y=227+73+73)
        # Cara Amarilla
        self.dbtn1 = tk.Button(self.frame, bg="yellow", width=8,
height=4, command=lambda: self.show_rb("d1"))
        self.dbtn1.place(x=215, y=450)
        self.dbtn2 = tk.Button(self.frame, bg="yellow", width=8,
height=4, command=lambda: self.show_rb("d2"))
        self.dbtn2.place(x=215+69, y=450)
        self.dbtn3 = tk.Button(self.frame, bg="yellow", width=8,
height=4, command=lambda: self.show_rb("d3"))
        self.dbtn3.place(x=215+69+69, y=450)
        self.dbtn4 = tk.Button(self.frame, bg="yellow", width=8,
height=4, command=lambda: self.show_rb("d4"))
        self.dbtn4.place(x=215, y=450+73)
        self.dbtn5 = tk.Button(self.frame, bg="yellow", width=8,
height=4, state="disabled").place(x=215+69, y=450+73)
        self.dbtn6 = tk.Button(self.frame, bg="yellow", width=8,
height=4, command=lambda: self.show_rb("d6"))
        self.dbtn6.place(x=215+69+69, y=450+73)
        self.dbtn7 = tk.Button(self.frame, bg="yellow", width=8,
height=4, command=lambda: self.show_rb("d7"))
        self.dbtn7.place(x=215, y=450+73+73)
        self.dbtn8 = tk.Button(self.frame, bg="yellow", width=8,
height=4, command=lambda: self.show_rb("d8"))
        self.dbtn8.place(x=215+69, y=450+73+73)
        self.dbtn9 = tk.Button(self.frame, bg="yellow", width=8,
height=4, command=lambda: self.show_rb("d9"))
        self.dbtn9.place(x=215+69+69, y=450+73+73)
        # Cara Naranja
```

```
        self.lbtn1 = tk.Button(self.frame, bg="orange", width=8,
height=4, command=lambda: self.show_rb("l1"))
        self.lbtn1.place(x=4, y=227)
        self.lbtn2 = tk.Button(self.frame, bg="orange", width=8,
height=4, command=lambda: self.show_rb("l2"))
        self.lbtn2.place(x=4+69, y=227)
        self.lbtn3 = tk.Button(self.frame, bg="orange", width=8,
height=4, command=lambda: self.show_rb("l3"))
        self.lbtn3.place(x=4+69+69, y=227)
        self.lbtn4 = tk.Button(self.frame, bg="orange", width=8,
height=4, command=lambda: self.show_rb("l4"))
        self.lbtn4.place(x=4, y=227+73)
        self.lbtn5 = tk.Button(self.frame, bg="orange", width=8,
height=4, state="disabled").place(x=4+69, y=227+73)
        self.lbtn6 = tk.Button(self.frame, bg="orange", width=8,
height=4, command=lambda: self.show_rb("l6"))
        self.lbtn6.place(x=4+69+69, y=227+73)
        self.lbtn7 = tk.Button(self.frame, bg="orange", width=8,
height=4, command=lambda: self.show_rb("l7"))
        self.lbtn7.place(x=4, y=227+73+73)
        self.lbtn8 = tk.Button(self.frame, bg="orange", width=8,
height=4, command=lambda: self.show_rb("l8"))
        self.lbtn8.place(x=4+69, y=227+73+73)
        self.lbtn9 = tk.Button(self.frame, bg="orange", width=8,
height=4, command=lambda: self.show_rb("l9"))
        self.lbtn9.place(x=4+69+69, y=227+73+73)
        # Cara Azul
        self.bbtn1 = tk.Button(self.frame, bg="blue", width=8,
height=4, command=lambda: self.show_rb("b1"))
        self.bbtn1.place(x=637, y=227)
        self.bbtn2 = tk.Button(self.frame, bg="blue", width=8,
height=4, command=lambda: self.show_rb("b2"))
        self.bbtn2.place(x=637+69, y=227)
        self.bbtn3 = tk.Button(self.frame, bg="blue", width=8,
height=4, command=lambda: self.show_rb("b3"))
        self.bbtn3.place(x=637+69+69, y=227)
        self.bbtn4 = tk.Button(self.frame, bg="blue", width=8,
height=4, command=lambda: self.show_rb("b4"))
        self.bbtn4.place(x=637, y=227+73)
        self.bbtn5 = tk.Button(self.frame, bg="blue", width=8,
height=4, state="disabled").place(x=637+69, y=227+73)
        self.bbtn6 = tk.Button(self.frame, bg="blue", width=8,
height=4, command=lambda: self.show_rb("b6"))
        self.bbtn6.place(x=637+69+69, y=227+73)
        self.bbtn7 = tk.Button(self.frame, bg="blue", width=8,
height=4, command=lambda: self.show_rb("b7"))
        self.bbtn7.place(x=637, y=227+73+73)
        self.bbtn8 = tk.Button(self.frame, bg="blue", width=8,
height=4, command=lambda: self.show_rb("b8"))
```

```
        self.bbtn8.place(x=637+69, y=227+73+73)
        self.bbtn9 = tk.Button(self.frame, bg="blue", width=8,
height=4, command=lambda: self.show_rb("b9"))
        self.bbtn9.place(x=637+69+69, y=227+73+73)

        # Selección de color
        self.r1 = tk.Radiobutton(self.frame, text="Blanco",
variable=color, value=1, command=self.select,
                    bg="lightblue")
        self.r2 = tk.Radiobutton(self.frame, text="Rojo",
variable=color, value=2, command=self.select,
                    bg="lightblue")
        self.r3 = tk.Radiobutton(self.frame, text="Verde",
variable=color, value=3, command=self.select,
                    bg="lightblue")
        self.r4 = tk.Radiobutton(self.frame, text="Amarillo",
variable=color, value=4, command=self.select,
                    bg="lightblue")
        self.r5 = tk.Radiobutton(self.frame, text="Naranja",
variable=color, value=5, command=self.select,
                    bg="lightblue")
        self.r6 = tk.Radiobutton(self.frame, text="Azul",
variable=color, value=6, command=self.select,
                    bg="lightblue")

        # Botón confirmar
        self.conf_btn = tk.Button(self.frame, width=20, height=3,
text="Confirmar configuración",
                                  command=self.confirmar).place(x=
500, y=500)
        self.lab = tk.Label(self.frame)

        # Acciones
        self.focus()
        self.__class__.loading = True

        # Asociar la función accion_al_cerrar() al cierre de la
ventana
        self.protocol("WM_DELETE_WINDOW", self.accion_al_cerrar)

    def select(self):

        global letra_cara
        global InitialStr
        global cara
        global color
        global name_color

        # Función para cambiar el color a las caras
```

```
        fichero = open("InitialConfig.txt", "r")
        InitialStr = fichero.read()
        fichero.close()

        match color.get():
            case 1:
                name_color = "white"
                letra_cara = "U"
            case 2:
                name_color = "red"
                letra_cara = "R"
            case 3:
                name_color = "green"
                letra_cara = "F"
            case 4:
                name_color = "yellow"
                letra_cara = "D"
            case 5:
                name_color = "orange"
                letra_cara = "L"
            case 6:
                name_color = "blue"
                letra_cara = "B"

        match cara:
            case "u1":
                self.ubtn1.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 0,
letra_cara)
            case "u2":
                self.ubtn2.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 1,
letra_cara)
            case "u3":
                self.ubtn3.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 2,
letra_cara)
            case "u4":
                self.ubtn4.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 3,
letra_cara)
            case "u6":
                self.ubtn6.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 5,
letra_cara)
            case "u7":
                self.ubtn7.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 6,
letra_cara)
```

```
            case "u8":
                self.ubtn8.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 7,
letra_cara)
            case "u9":
                self.ubtn9.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 8,
letra_cara)
            case "r1":
                self.rbtn1.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 9,
letra_cara)
            case "r2":
                self.rbtn2.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 10,
letra_cara)
            case "r3":
                self.rbtn3.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 11,
letra_cara)
            case "r4":
                self.rbtn4.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 12,
letra_cara)
            case "r6":
                self.rbtn6.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 14,
letra_cara)
            case "r7":
                self.rbtn7.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 15,
letra_cara)
            case "r8":
                self.rbtn8.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 16,
letra_cara)
            case "r9":
                self.rbtn9.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 17,
letra_cara)
            case "f1":
                self.fbtn1.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 18,
letra_cara)
            case "f2":
                self.fbtn2.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 19,
letra_cara)
            case "f3":
```

```
                self.fbtn3.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 20,
letra_cara)
            case "f4":
                self.fbtn4.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 21,
letra_cara)
            case "f6":
                self.fbtn6.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 23,
letra_cara)
            case "f7":
                self.fbtn7.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 24,
letra_cara)
            case "f8":
                self.fbtn8.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 25,
letra_cara)
            case "f9":
                self.fbtn9.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 26,
letra_cara)
            case "d1":
                self.dbtn1.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 27,
letra_cara)
            case "d2":
                self.dbtn2.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 28,
letra_cara)
            case "d3":
                self.dbtn3.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 29,
letra_cara)
            case "d4":
                self.dbtn4.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 30,
letra_cara)
            case "d6":
                self.dbtn6.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 32,
letra_cara)
            case "d7":
                self.dbtn7.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 33,
letra_cara)
            case "d8":
                self.dbtn8.config(bg=name_color)
```

```
                InitialStr = modificar_caracter(InitialStr, 34,
letra_cara)
            case "d9":
                self.dbtn9.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 35,
letra_cara)
            case "l1":
                self.lbtn1.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 36,
letra_cara)
            case "l2":
                self.lbtn2.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 37,
letra_cara)
            case "l3":
                self.lbtn3.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 38,
letra_cara)
            case "l4":
                self.lbtn4.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 39,
letra_cara)
            case "l6":
                self.lbtn6.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 41,
letra_cara)
            case "l7":
                self.lbtn7.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 42,
letra_cara)
            case "l8":
                self.lbtn8.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 43,
letra_cara)
            case "l9":
                self.lbtn9.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 44,
letra_cara)
            case "b1":
                self.bbtn1.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 45,
letra_cara)
            case "b2":
                self.bbtn2.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 46,
letra_cara)
            case "b3":
                self.bbtn3.config(bg=name_color)
```

```
                InitialStr = modificar_caracter(InitialStr, 47,
letra_cara)
            case "b4":
                self.bbtn4.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 48,
letra_cara)
            case "b6":
                self.bbtn6.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 50,
letra_cara)
            case "b7":
                self.bbtn7.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 51,
letra_cara)
            case "b8":
                self.bbtn8.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 52,
letra_cara)
            case "b9":
                self.bbtn9.config(bg=name_color)
                InitialStr = modificar_caracter(InitialStr, 53,
letra_cara)

        fichero = open("InitialConfig.txt", "w")
        fichero.write(InitialStr)
        fichero.close()

    def show_rb(self, num_face):

        global cara
        # Mostrar la selección de colores para las caras
        self.r1.place(x=450, y=20)
        self.r2.place(x=450, y=50)
        self.r3.place(x=450, y=80)
        self.r4.place(x=550, y=20)
        self.r5.place(x=550, y=50)
        self.r6.place(x=550, y=80)
        cara = num_face

    def confirmar(self):

        # Crear la matriz de colores de las caras del cubo
        self.lab.place(x=450, y=560)
        with open("InitialConfig.txt", "r") as fichero:
                labeltext = fichero.read()
        c_cont = contar_caracteres(labeltext)
        for c, num in c_cont.items():
            if num != 9:
```

```
                self.lab.config(text="Hay algún error en la
configuración inicial. Revísalo.")
                return

        self.lab.config(text=labeltext)
        with open("Config2Alogrithm.txt","w") as fichero:
            fichero.write(labeltext)

        RandomWindow.random_seq = False

    def accion_al_cerrar(self):
        # Restauramos la cadena de la configuración inicial al
cerrar la ventana
        with open("InitialConfig.txt", "w") as fichero:
            fichero.write('UUUUUUUUURRRRRRRRRFFFFFFFFFDDDDDDDDDLLLL
LLLLLBBBBBBBBB')

        self.__class__.loading = False
        return super().destroy()

class ManualWindow(tk.Toplevel):

    # Atributo de clase que indica si la ventana de configuración
está en uso
    loading = False

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.config(width=MainWindow.wroot,
height=MainWindow.hroot)
        self.title("Giros Manuales")
        self.geometry(str(MainWindow.wroot) + "x" +
str(MainWindow.hroot) + "+" +
                      str(MainWindow.x_root)  + "+" +
str(MainWindow.y_root))
        self.resizable(0,0)
        self.iconbitmap('giro.ico')

        # Configuración del marco
        self.frame = tk.Frame(self)
        self.frame.pack(fill='both', expand=1)
        self.frame.config(bg="lightblue")

        # ---------------------- Caras -------------------------
        self.canvas0 = tk.Canvas(self.frame, width=205, height=217,
background="lightblue", borderwidth=1,
highlightbackground="lightblue")
        self.canvas1 = tk.Canvas(self.frame, width=205, height=217,
background="white",
```

```
                                    borderwidth=1,
highlightbackground="gray26")
        self.canvas2 = tk.Canvas(self.frame, width=205, height=217,
background="darkorange",
                                    borderwidth=1,
highlightbackground="gray26")
        self.canvas3 = tk.Canvas(self.frame, width=205, height=217,
background="limegreen",
                                    borderwidth=1,
highlightbackground="gray26")
        self.canvas4 = tk.Canvas(self.frame, width=205, height=217,
background="firebrick1",
                                    borderwidth=1,
highlightbackground="gray26")
        self.canvas5 = tk.Canvas(self.frame, width=205, height=217,
background="dodgerblue2",
                                    borderwidth=1,
highlightbackground="gray26")
        self.canvas6 = tk.Canvas(self.frame, width=205, height=217,
background="lightblue", borderwidth=1,
highlightbackground="lightblue")
        self.canvas7 = tk.Canvas(self.frame, width=205, height=217,
background="gold",
                                    borderwidth=1,
highlightbackground="gray26")

        self.canvas0.grid(column=0, row=0)
        self.canvas1.grid(column=1, row=0)
        self.canvas2.grid(column=0, row=1)
        self.canvas3.grid(column=1, row=1)
        self.canvas4.grid(column=2, row=1)
        self.canvas5.grid(column=3, row=1)
        self.canvas6.grid(column=0, row=2)
        self.canvas7.grid(column=1, row=2)

        # Cargamos las imagenes de giro
        self.cw_img =
tk.PhotoImage(file="horario.png").subsample(12,12)
        self.acw_img =
tk.PhotoImage(file="antihorario.png").subsample(12,12)
        # Cara Blanca
        self.wbtn1 = tk.Button(self.frame, width=4, height=2,
text="90º", command=lambda: self.turn("u1"))
        self.wbtn1.place(x=250, y=30)
        self.wbtn2 = tk.Button(self.frame, width=4, height=2,
text="-90º", command=lambda: self.turn("u3"))
        self.wbtn2.place(x=250, y=90)
        self.wbtn3 = tk.Button(self.frame, width=4, height=2,
text="180º", command=lambda: self.turn("u2"))
```

```
        self.wbtn3.place(x=250, y=150)
        tk.Label(self.frame, image=self.cw_img,
bg="white").place(x=330, y=28)
        tk.Label(self.frame, image=self.acw_img,
bg="white").place(x=330, y=88)
        tk.Label(self.frame, image=self.cw_img,
bg="white").place(x=330, y=148)
        # Cara Naranja
        self.obtn1 = tk.Button(self.frame, width=4, height=2,
text="90º", command=lambda: self.turn("l1"))
        self.obtn1.place(x=250-210, y=30+220)
        self.obtn2 = tk.Button(self.frame, width=4, height=2,
text="-90º", command=lambda: self.turn("l3"))
        self.obtn2.place(x=250-210, y=90+220)
        self.obtn3 = tk.Button(self.frame, width=4, height=2,
text="180º", command=lambda: self.turn("l2"))
        self.obtn3.place(x=250-210, y=150+220)
        tk.Label(self.frame, image=self.cw_img,
bg="darkorange").place(x=330-210, y=28+220)
        tk.Label(self.frame, image=self.acw_img,
bg="darkorange").place(x=330-210, y=88+220)
        tk.Label(self.frame, image=self.cw_img,
bg="darkorange").place(x=330-210, y=148+220)
        # Cara Verde
        self.gbtn1 = tk.Button(self.frame, width=4, height=2,
text="90º", command=lambda: self.turn("f1"))
        self.gbtn1.place(x=250, y=30+220)
        self.gbtn2 = tk.Button(self.frame, width=4, height=2,
text="-90º", command=lambda: self.turn("f3"))
        self.gbtn2.place(x=250, y=90+220)
        self.gbtn3 = tk.Button(self.frame, width=4, height=2,
text="180º", command=lambda: self.turn("f2"))
        self.gbtn3.place(x=250, y=150+220)
        tk.Label(self.frame, image=self.cw_img,
bg="limegreen").place(x=330, y=28+220)
        tk.Label(self.frame, image=self.acw_img,
bg="limegreen").place(x=330, y=88+220)
        tk.Label(self.frame, image=self.cw_img,
bg="limegreen").place(x=330, y=148+220)
        # Cara Roja
        self.rbtn1 = tk.Button(self.frame, width=4, height=2,
text="90º", command=lambda: self.turn("r1"))
        self.rbtn1.place(x=250+210, y=30+220)
        self.rbtn2 = tk.Button(self.frame, width=4, height=2,
text="-90º", command=lambda: self.turn("r3"))
        self.rbtn2.place(x=250+210, y=90+220)
        self.rbtn3 = tk.Button(self.frame, width=4, height=2,
text="180º", command=lambda: self.turn("r2"))
        self.rbtn3.place(x=250+210, y=150+220)
```

```
        tk.Label(self.frame, image=self.cw_img,
bg="firebrick1").place(x=330+210, y=28+220)
        tk.Label(self.frame, image=self.acw_img,
bg="firebrick1").place(x=330+210, y=88+220)
        tk.Label(self.frame, image=self.cw_img,
bg="firebrick1").place(x=330+210, y=148+220)
        # Cara Azul
        self.bbtn1 = tk.Button(self.frame, width=4, height=2,
text="90º", command=lambda: self.turn("b1"))
        self.bbtn1.place(x=250+420, y=30+220)
        self.bbtn2 = tk.Button(self.frame, width=4, height=2,
text="-90º", command=lambda: self.turn("b3"))
        self.bbtn2.place(x=250+420, y=90+220)
        self.bbtn3 = tk.Button(self.frame, width=4, height=2,
text="180º", command=lambda: self.turn("b2"))
        self.bbtn3.place(x=250+420, y=150+220)
        tk.Label(self.frame, image=self.cw_img,
bg="dodgerblue2").place(x=330+420, y=28+220)
        tk.Label(self.frame, image=self.acw_img,
bg="dodgerblue2").place(x=330+420, y=88+220)
        tk.Label(self.frame, image=self.cw_img,
bg="dodgerblue2").place(x=330+420, y=148+220)
        # Cara Amarilla
        self.ybtn1 = tk.Button(self.frame, width=4, height=2,
text="90º", command=lambda: self.turn("d1"))
        self.ybtn1.place(x=250, y=30+440)
        self.ybtn2 = tk.Button(self.frame, width=4, height=2,
text="-90º", command=lambda: self.turn("d3"))
        self.ybtn2.place(x=250, y=90+440)
        self.ybtn3 = tk.Button(self.frame, width=4, height=2,
text="180º", command=lambda: self.turn("d2"))
        self.ybtn3.place(x=250, y=150+440)
        tk.Label(self.frame, image=self.cw_img,
bg="gold").place(x=330, y=28+440)
        tk.Label(self.frame, image=self.acw_img,
bg="gold").place(x=330, y=88+440)
        tk.Label(self.frame, image=self.cw_img,
bg="gold").place(x=330, y=148+440)

        # Label de mensaje
        self.label = tk.Label(self.frame, text="", font=("Arial",
10), bg="lightblue")
        self.label.place(x=450, y=50)

        # Acciones
        self.focus()
        self.__class__.loading = True
```

```
        # Asociar la función accion_al_cerrar() al cierre de la
ventana
        self.protocol("WM_DELETE_WINDOW", self.accion_al_cerrar)

    def accion_al_cerrar(self):
        self.__class__.loading = False
        return super().destroy()

    def turn(self, face_turn):
        chain2arduino = face_turn.upper() + " (1f)"

        try:
            # Configuración puerto Serial
            com = serial.Serial("COM4", 9600, write_timeout= 10)
            time.sleep(2)
            # Escribimos por el puerto serie la cadena a enviar a
arduino
            com.write(chain2arduino.encode("ascii"))
            # Cerramos el puerto serie
            com.close()
            print(chain2arduino)
            self.label.config(text=f"Movimiento {face_turn.upper()}
enviado correctamente.")
            time.sleep(2)
        except:
            self.label.config(text="No se encuentra el puerto
serie. Revísalo.")

class RandomWindow(tk.Toplevel):

    # Atributo de clase que indica si la ventana de configuración
está en uso
    loading = False
    random_seq = False

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.config(width=MainWindow.wroot,
height=MainWindow.hroot)
        self.title("Giros Aleatorios")
        self.geometry("300x300+" + str(MainWindow.x_root)  + "+" +
str(MainWindow.y_root))
        self.resizable(0,0)
        self.iconbitmap('random.ico')

        # Configuración del marco
        self.frame = tk.Frame(self)
        self.frame.pack(fill='both', expand=1)
        self.frame.config(bg="lightblue")
```

```
        # Slider para seleccionar numero de movimientos
        self.valor = "10"
        self.slider = tk.Scale(self.frame, from_=10, to=30,
orient="horizontal",
                               bg="lightblue",
highlightbackground="lightblue", command=self.valor_slider)
        self.slider.place(x=95, y=50)
        self.label_slider = tk.Label(self.frame, text="Movimientos:
10", bg="lightblue", font=("Arial", 8, "bold"))
        self.label_slider.place(x=100, y=95)

        # Botón para generar los movimientos aleatorios
        self.generate = tk.Button(self.frame, width=18, height=2,
text="Generar movimientos",
                                  command= self.generar_random)
        self.generate.place(x=80, y=120)

        # Acciones
        self.focus()
        self.__class__.loading = True

        # Asociar la función accion_al_cerrar() al cierre de la
ventana
        self.protocol("WM_DELETE_WINDOW", self.accion_al_cerrar)

    def generar_random(self):
        global sequence
        self.alpha_str = "UDRLFB"
        self.num_str = "123"
        self.random_str = ""
        for i in range(int(self.valor)):
            self.random_str = self.random_str +
random.choice(self.alpha_str) + random.choice(self.num_str) + " "
        self.random_str = self.random_str + f"({self.valor}f)"
        sequence = self.random_str
        RandomWindow.random_seq = True

    def valor_slider(self, valor):
        self.valor = valor
        self.label_slider.config(text=f"Movimientos: {self.valor}")

    def accion_al_cerrar(self):
        self.__class__.loading = False
        return super().destroy()

class MainWindow(tk.Tk):

    # Atributos de la clase, medidas de la raíz
```

```
    wroot = 845
    hroot = 670
    x_root = 0
    y_root = 0

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.config(width=self.wroot, height=self.hroot,
bg="lightblue")
        self.title("Cubik's Solver")

        self.x_root = self.winfo_screenwidth() // 2 - self.wroot //
2
        self.y_root = self.winfo_screenheight() // 2 - self.hroot
// 2
        self.geometry(str(self.wroot) + "x" + str(self.hroot) + "+"
                      + str(self.x_root)  + "+" + str(self.y_root-
35))
        self.resizable(0,0)
        self.iconbitmap('rubik.ico')

        # Configuración del marco
        self.frame = tk.Frame(self)
        self.frame.pack(fill='both', expand=1)
        self.frame.config(bg="lightblue")

        # ---------------------- Modo auto -----------------------
-
        self.canvas_auto = tk.Canvas(self.frame, width=205,
height=370, background="deepskyblue",
                                     highlightbackground="gray26")
        self.canvas_auto.place(x=172, y=140)
        # Label nombre marco
        self.label_auto = tk.Label(self.frame, text="Auto Mode",
font=("Arial", 16, "bold"), bg="deepskyblue")
        self.label_auto.place(x=218, y=155)
        # Botón para abrir la ventana de configuración
        self.open_config_btn = tk.Button(self.frame, width=20,
height=3, text="Configuración Inicial",
                                         command=self.open_config_
window)
        self.open_config_btn.place(x=200, y=200)
        # Botón para obtener secuencia de movimientos
        self.seq_btn = tk.Button(self.frame, width=20, height=3,
text="Obtener Secuencia",
                                 command=self.movement_seq
uence)
        self.seq_btn.place(x=200, y=265)
        # Botón para enviar la cadena resolvedora a Arduino
```

```
        self.py2ard_btn = tk.Button(self.frame, width=20, height=3,
text="Enviar Solución a Arduino",
                                    command=self.python2ardui
no)
        self.py2ard_btn.place(x=200, y=330)
        # Botón para enviar un paso a Arduino
        self.step2ard_btn = tk.Button(self.frame, width=20,
height=3, text="Enviar Paso a Arduino",
                                    command=self.step2arduino
)
        self.step2ard_btn.place(x=200, y=395)
        # Label pasos
        self.title_steps = tk.Label(self.frame, text="",
font=("Arial", 10), bg="deepskyblue")
        self.title_steps.place(x=200, y=460)
        self.label_steps = tk.Label(self.frame, text="",
font=("Arial", 14, "bold"), bg="deepskyblue")

        # ---------------------- Modo manual ----------------------
---
        self.canvas_manual = tk.Canvas(self.frame, width=205,
height=217, background="cadetblue2",
                                       highlightbackground="gray26")
        self.canvas_manual.place(x=425, y=140)
        # Label nombre marco
        self.label_manual = tk.Label(self.frame, text="Manual
Mode", font=("Arial", 16, "bold"), bg="cadetblue2")
        self.label_manual.place(x=460, y=155)
        # Botón para abrir la ventana de giros manuales
        self.btn_manual = tk.Button(self.frame, width=20, height=3,
text="Giros Manuales",
                                    command=self.open_manual_
window)
        self.btn_manual.place(x=455, y=200)
        # Botón para abrir la ventana de giros random
        self.btn_random = tk.Button(self.frame, width=20, height=3,
text="Giros Aleatorios",
                                    command=self.open_random_
window)
        self.btn_random.place(x=455, y=280)

        # ---------------------- Secuencia -----------------------
-
        # Creamos un marco para mostrar la secuencia de movimientos
        self.canvas_seq = tk.Canvas(self.frame, width=558,
height=100, background="oldlace",
                                    highlightbackground="gray26")
        self.canvas_seq.place(x=122, y=30)
        # Label nombre marco
```

```
        self.title_seq = tk.Label(self.frame, text="Secuencia de
movimientos", font=("Arial", 16, "bold"), bg="oldlace")
        self.title_seq.place(x=168, y=45)
        # Label para mostrar la secuencia actual
        self.label_actual = tk.Label(self.frame, text="",
font=("Arial", 12, "bold"), bg="oldlace")
        self.label_actual.place(x=168, y=75)
        self.label_seq = tk.Label(self.frame, text="",
font=("Arial", 9, "bold"), bg="oldlace")
        self.label_seq.place(x=168, y=100)

    def open_config_window(self):
        if not ConfigWindow.loading and not ManualWindow.loading
and not RandomWindow.loading:
            self.config_window = ConfigWindow()

    def open_manual_window(self):
        if not ManualWindow.loading and not ConfigWindow.loading
and not RandomWindow.loading:
            self.manual_window = ManualWindow()

    def open_random_window(self):
        if not ManualWindow.loading and not ConfigWindow.loading
and not RandomWindow.loading:
            self.random_window = RandomWindow()

    def movement_sequence(self):
        global stepsleft
        global sequence

        if RandomWindow.random_seq == False:
            with open("Config2Alogrithm.txt", "r") as fichero:
                str2solve = fichero.read()
            sequence = cadena_resolver(str2solve)
        # Obtenemos el número de pasos de la secuencia
        if sequence[-4] == "(":
            stepsleft = int(sequence[-3])
        else:
            stepsleft = int(sequence[-4:-2])
        # Actualizamos label
        self.label_actual.config(text="Secuencia actual:")
        self.label_seq.config(text=sequence)

    def python2arduino(self):
        global sequence

        try:
            # Configuración puerto Serial
            com = serial.Serial("COM4", 9600, write_timeout= 10)
```

```
            time.sleep(1)
            # Escribimos por el puerto serie la cadena a enviar a
arduino
            com.write(sequence.encode("ascii"))
            time.sleep(0.1)
            # Cerramos el puerto serie
            com.close()
        except:
            self.title_steps.config(text="No se encuentra el puerto
serie.\n Revísalo.")
            self.title_steps.place(x=180, y=460)

    def step2arduino(self):
        global sequence
        global stepsleft
        # Almacenamos los movimientos de la secuencia en una lista
        seq_list = sequence.split()
        seq_list.pop()
        # Configuración puerto Serial
        try:
            com = serial.Serial("COM4", 9600, write_timeout= 10)
            time.sleep(1)
            # Escribimos por el puerto serie el movimiento que toca
            com.write(seq_list[-stepsleft].encode("ascii"))
            time.sleep(0.1)
            # Cerramos el puerto serie
            com.close()
            # Quitamos un paso a la variable
            stepsleft = stepsleft - 1
            # Mostramos los pasos que faltan
            if stepsleft != 0:
                self.title_steps.config(text="Pasos restantes:")
                self.label_steps.config(text=str(stepsleft))
                self.label_steps.place(x=300, y=458)
            else:
                self.title_steps.config(text="Secuencia
finalizada.")
                self.label_steps.pack_forget()
        except:
            self.title_steps.config(text="No se encuentra el puerto
serie.\n Revísalo.")
            self.title_steps.place(x=180, y=460)
            self.label_steps.pack_forget()

# Main
main_window = MainWindow()

# Variables globales
InitialStr = tk.StringVar()
```

```
cara = tk.StringVar()
letra_cara = tk.StringVar()
name_color = tk.StringVar()
sequence = tk.StringVar()

color = tk.IntVar()
stepsleft = tk.IntVar()

main_window.mainloop()
```

# 3. Vídeos

- Preparación para funcionamiento: https://youtube.com/shorts/CO8jt0L6USY?feature=share
- Funcionamiento prototipo: https://youtu.be/hDCo4Mjeid8
- Simulación con cubo: https://youtu.be/qq0S5OXYwAo
  Simulación sin cubo, secuencia de las gráficas: https://youtu.be/xMFx_HTubgU
  Debido a la naturaleza de el tipo de conexiones que permite el programa, no puede simularse el giro de todas las caras del cubo. Sí que puede simularse el movimiento rotatorio de todos los motores, pero no cuando se conecta el cubo a ellos.

# 4. Planos

Adjuntados en un archivo externo: “Planos TFM.pdf”