INFORME

ALUMNO:

LEANDRO GABRIEL MIGLIORETTI

DNI: 42.290.683

MAIL: leandromiglioretti@gmail.com

Proyecto:

PPS

Práctica Profesional Supervisada

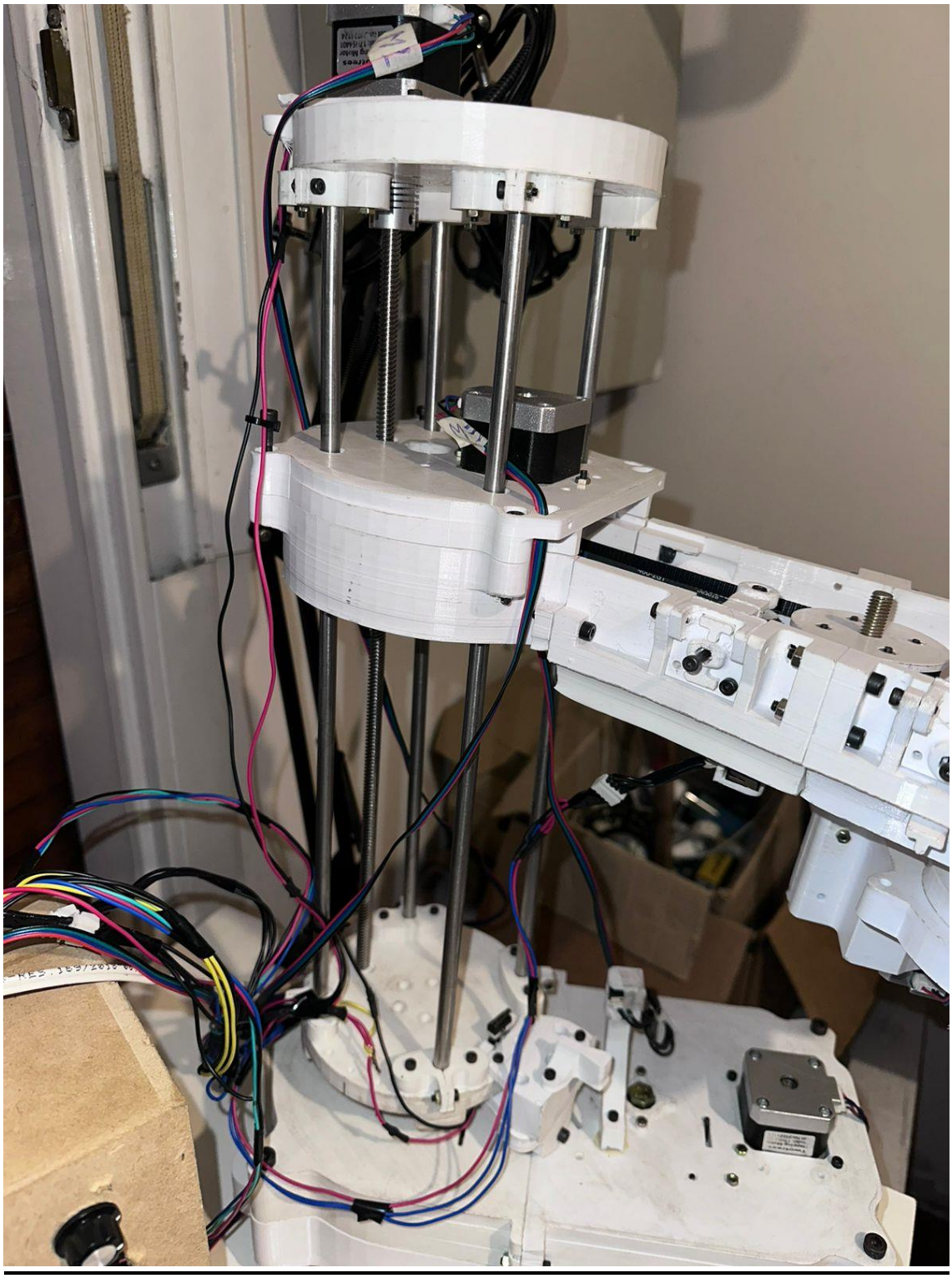
Carrera:

Ingenieria Mecatronica

Tutor:

Cristian Lukaszewicz

DISEÑO, DESARROLLO Y ARMADO DE PROTOTIPO DE BRAZO ROBOTICO TIPO SCARA



Índice

1. Presentación

1.1 Introducción

Se realizará el diseño y construcción de un prototipo desde 0 de un robot tipo “SCARA”, de 4 grados de libertad, orientado a aplicaciones de manipulación y ensamblaje en entornos industriales. Se desarrolló utilizando Arduino MEGA y motores nema 17 principalmente.

1.2 Objetivos

El principal objetivo es construir un robot SCARA 100% funcional capaz de realizar movimientos precisos en el espacio, mediante el control de los motores paso a paso y la implementación de cinemática directa e inversa.

1.3 Alcance

El proyecto abarca muchas áreas desde el diseño mecánico, modelado 3D, electrónica y diseño de software y programación de su sistema de control.

Se excluyen tareas avanzadas de automatización.

1.4 Resumen

Se ha logrado construir un prototipo funcional de robot SCARA capaz de ejecutar movimientos programados utilizando un sistema de control desarrollado por nosotros que basa sus cálculos en cinemática.

2. Ingeniería de la solucion

2.1 Especificaciones Técnicas

El robot se compone principalmente de un tablero, en el cual se incorpora una interfaz HMI capaz de funcionar como nexo entre el operador y el robot, utilizando una interacción sencilla e intuitiva. A su vez en el tablero se incorporan todos los componentes eléctricos y electrónicos para su funcionamiento.
La estructura física del robot fue enteramente diseñada e impresa en 3D, utilizando filamento PLA blanco. Consta de un sistema de sensado de posición (o referencias) mediante switchs finales de carrera y en la parte de actuadores tenemos los motores nema 17 que ejecutaran los movimientos controlados para cada articulación.

2.1.1 Materiales, características y criterios de selección

Arduino MEGA 2560:

Cantidad: 1

Criterio: Fue seleccionado por su gran cantidad de entradas y salidas

Características principales: 54 Entrdas y salidas digitales de 10 bits.; 16 entradas analógicas, 4 UART's, un cristal de 12MHZ

SHIELD PLACA DE DESARROLLO Arduino MEGA 2560:

Cantidad: 1

Criterio: Necesaria para la facilidad a la hora de montar la placa a los diversos componentes

Motores NEMA 17:

X 4 Motores bipolares

Criterio: Utilizado ya que con solamente ejecutar una secuencia de pulsos podemos tener un control preciso sin retroalimentación de las posiciones angulares de nuestro robot.

Características principales: paso 1,8°, bipolar.

Drivers TB6600:

Cantidad: x4

Criterio: Compatible con microcontroladores como arduino

Características principales: 9V-42V entrada, control de microstepping, control de corriente

Display LCD 16x02:

Cantidad: x1

Criterio: Utilizado para el control y visualización de nuestro robot HMI

Switch Encoder Rotativo:

Cantidad: x1

Criterio: Facil uso al tratarse de un solo elemento para controlar el software del robot

Boton Parada de Emergencia:

Cantidad: x1

Criterio: Bloqueo instanteaneo del robot en caso de emergencia

Características principales: Conectado a un pin de interrupción IRS de arduino, para mayor velocidad de reaccion

Buzzer:

Cantidad: x1

Criterio: Utilizado para señales sonoras en el uso del menú y de estados del robot.

Resistencias:

Cantidad: x10 10KOHM, x3 330OHM

Leds de indicacion:

Cantidad: x1 Rojo; x1 Amarillo; x1 Verde

Criterio: Utilizado para indicar señales visuales

Fuente 12V:

Cantidad: x1

Criterio: Fuente con potencia necesaria para alimentar todo el sistema

Características principales: 10A

Microelectronica:

Se han utilizado una serie de componentes necesarios para realizar las conexiones, entre ellos placas de desarrollo, borneras, resistencias para PullUP, conectores xh254 macho y hembra entre otros.

Tornillería:

Se ha usado una diversa y amplia gama de tornillos del tipo Alem de cabeza circular. Se han utilizado mayoritariamente tornillos M3 y en menor medida M4 y M5. Estos tornillos son ideales para nuestro robot ya que nos ofrecen estabilidad y precisión a las uniones entre las piezas impresas en 3d, haciendo al robot más robusto.

Transmisión:

- X2 Correa Cerrada GT2 6mm 400mm
- X1 Correa Cerrada GT2 6mm 254mm
- X1 Correa Cerrada GT2 6mm 314 mm

Criterio: Se utilizan esta serie de correas debido a su gran precisión a la hora de transmitir la potencia entre el motor y la articulación.

Características principales:

Paso: 2mm; Ancho: 6mm; Altura: 1,78mm; Altura de diente: 0,75mm

Rodamientos axiales:

- X4 Rodamiento Axial CRAPODINA 51107 35x62sx18
- X2 Rodamiento Axial CRAPODINA 51108 40x60x13

Criterios: Buen funcionamiento, bajo coeficiente de fricción y un desgaste reducido. Ideales para soportar el peso manteniendo una rotación suave y precisa.

Rodamientos radiales:

Cantidad: x5 Rodamiento Radial 608-2rs ; x1 6807-2rs ; x2 6806-2rs

Criterios: Soportan cargas que se encuentran principalmente en dirección perpendicular al eje, siendo útil para mantener el eje estable.

Switch Final de Carrera:

Cantidad: x10

Estructurales y dinámicas:

- Varilla Roscada 8mm x 40mm largo: Encargada de transformar el movimiento circular en un desplazamiento vertical a lo largo del eje Z.

- X4 Barras estructurales de acero inoxidable de 8mm x 40mm largo: Sirven de soporte para la torre del robot, proporcionando estabilidad y firmeza.

- Acople flexible 5mm a 8mm: Utilizado para transmitir la potencia del motor nema17 a la varilla roscada

- X3 Poleas GT2 20mm diámetro: Utilizadas para los tensores de los motores, para evitar falsos cdeslizamientos entre los dientes y la correa

- X3 Poleas dentadas GT2, 20 dientes, Eje 5mm, Correa 6mm: Utilizadas para transmitir la potencia desde los ejes de los motores, mediante la correa.

Rollo Plastico PLA blanco 0,25 x 1 metro:

Cantidad: 4

Criterio: El PLA ofrece buena resistencia y bajo peso.

2.1.2 Herramientas a utilizar

Se han utilizado diversas herramientas que van desde soldadores, pinzas, destornilladores hasta un juego de llaves alem de M2 a M7.

Para el desarrollo e impresión de las piezas se ha utilizado Solidworks 2013 para la creación y edición de las piezas, como asi una aplicación Creality Slicer para pasar a codigo G, el formato de archivo que lee nuestra impresora 3D, la CrealityEnder 3 v2.

Se ha utilizado para la programación herramientas como el entorno de desarrollo arduino IDE.

El rol de una computadora fue clave para la creación del robot.

2.3 Presupuesto

Elemento	**Cantidad**	**Precio**
Rollo 1kg PLA blanco	4	$80000
Varilla Roscada 8mm x 40 mm largo	1	$8000
Barra estructural de acero	4	$12000
Acople flexible 5 a 8 mm	1	$2000
Poleas gt2 20mm Diametro	3	$5000
Polea dentada GT2 z20 eje 5mm	3	$5000
Rodamiento Axial Crapodina 51107	4	$16000
Rodamiendo Axial Crapodina 51108	2	$10400
Rodamiento Radial 608-2rs	5	$10000
Rodamiento radial 6807-2rs	1	$6000
Rodamiento Radial 6806-2rs	2	$11000
Correa Cerrada GT2 6mm 400mm	2	$8600
Correa Cerrada GT2 6mm 254mm	1	$3200
Correa Cerrada GT2 6mm 314mm	1	$3600
Tornilleria		$70000
Fuente 12v 10A	1	$18000
Leds de indicacion	3	1000
Resistencias 10KOHM	10	$1500
Resistencias 330 OHM	3	$300
Buzzer	1	$3000
Boton Parada de Emergencia	1	$20000
Switch Encoder Rotativo	1	$5000
Display 1602	1	$6000
Drivers TB6600	4	$140000
Motores Nema 17	4	$120000
Arduino MEGA	1	$35000
Shield Placa de desarrollo arduino mega	1	$19000
Switch Final de carrera	10	$5000
TOTAL	aprox	$625000

3. Fundamentos Matemáticos

3.1 Características espaciales del robot SCARA

El robot SCARA (Selective Compilance Asembly Robot Arm) es un tipo de robot industrial diseñado para realizar movimientos precisos en un plano horizontal. Su estructura se caracteriza por tener dos brazos en serie conectados a un actuador final que se mueve en un espacio tridimensional, permitiendo desplazamientos rápidos y eficientes de los ejes X e Y.

En nuestro proyecto, el robot SCARA cuenta con 4 grados de libertad que no solo permiten movimientos en XY si no también en el eje Z.

3.2 Matemática para la resolución utilizando el método DH

El método de Denavit-Hartenberg (DH) es una convención utilizada en la robótica para describir la relación entre los eslabones de un manipulador mediante una serie de parámetros homogéneos. Este método permite representar las transformaciones de posición y orientación entre las distintas articulaciones de un robot de manera sistemática y uniforme, lo cual es esencial para la resolución de la cinemática directa e inversa del robot SCARA.

Método Denavit-Hartenberg (DH)

En el método DH, describimos la relación entre dos eslabones consecutivos en un robot usando cuatro parámetros:

1. θ_i: Ángulo de rotación alrededor del eje z_i (articulación rotativa).
2. d_i: Distancia a lo largo del eje z_i (articulación prismática, o desplazamiento para $q3$ en el SCARA).
3. a_i: Distancia a lo largo del eje x_i (longitud del eslabón).
4. α_i: Ángulo entre los ejes z_i y z_{i+1}, rotado alrededor de x_i.

Parámetros DH para el SCARA de 4 DOF

Para un robot SCARA con 4 grados de libertad, tenemos que aplicar el método DH paso a paso. Asumamos las siguientes asignaciones de los parámetros DH para el robot SCARA:

Eslabón	θ_i (Ángulo)	d_i (Desplazamiento)	a_i (Longitud)	α_i (Ángulo entre ejes)
1	$\theta_1 = q_1$	$d_1 = d_1$	$a_1 = L_1$	$\alpha_1 = 0$
2	$\theta_2 = q_2$	$d_2 = 0$	$a_2 = L_2$	$\alpha_2 = 0$
3	$\theta_3 = 0$	$d_3 = q_3$	$a_3 = 0$	$\alpha_3 = 0$
4	$\theta_4 = q_4$	$d_4 = 0$	$a_4 = 0$	$\alpha_4 = 0$

Matriz de Transformación Homogénea

El siguiente paso es definir la **matriz de transformación homogénea** T entre cada par de eslabones usando la siguiente fórmula general:

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

Paso 1: Transformación entre eslabones

T_1^0 (de la base al primer eslabón)

Para el eslabón 1, $\theta_1 = q_1$, $d_1 = d_1$, $a_1 = L_1$, y $\alpha_1 = 0$. La matriz de transformación homogénea es:

$$T_1^0 = \begin{pmatrix} \cos q_1 & -\sin q_1 & 0 & L_1 \cos q_1 \\ \sin q_1 & \cos q_1 & 0 & L_1 \sin q_1 \\ 0 & 0 & 1 & d_1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

T_2^1 (del primer eslabón al segundo eslabón)

Para el eslabón 2, $\theta_2 = q_2$, $d_2 = 0$, $a_2 = L_2$, y $\alpha_2 = 0$. La matriz de transformación es:

$$T_2^1 = \begin{pmatrix} \cos q_2 & -\sin q_2 & 0 & L_2 \cos q_2 \\ \sin q_2 & \cos q_2 & 0 & L_2 \sin q_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

T_3^2 (del segundo eslabón al tercero, que es el desplazamiento en Z)

Para el eslabón 3, es un desplazamiento vertical (q_3) sin rotación, por lo que la matriz es simple:

$$T_3^2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & q_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

T_4^3 (rotación del efector final)

Para el eslabón 4, es una rotación pura alrededor del eje z_4, por lo que la matriz de transformación es:

$$T_4^3 = \begin{pmatrix} \cos q_4 & -\sin q_4 & 0 & 0 \\ \sin q_4 & \cos q_4 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Paso 2: Matriz de transformación total

Ahora, para obtener la posición del efector final en el espacio, simplemente multiplicamos las matrices de transformación desde la base hasta el efector final:

$$T = T_1^0 \cdot T_2^1 \cdot T_3^2 \cdot T_4^3$$

La posición final del efector estará en la última columna de la matriz T, que nos da las coordenadas (X, Y, Z) del efector final.

Paso 3: Cinemática Directa desde DH

Al multiplicar las matrices y resolver, obtendrás las siguientes ecuaciones para la posición del efector:

1. **Posición en X**:

$$X = L_1 \cdot \cos(q_1) + L_2 \cdot \cos(q_1 + q_2)$$

2. **Posición en Y**:

$$Y = L_1 \cdot \sin(q_1) + L_2 \cdot \sin(q_1 + q_2)$$

$$Z = q_3$$

4. **Rotación del efector**:

$$\text{Orientación} = q_4$$

Paso 4: Cinemática Inversa desde DH

Dado que estas ecuaciones son las mismas que obtuvimos anteriormente, puedes usar los mismos métodos de cinemática inversa para calcular los ángulos q_1, q_2, q_3 y q_4 a partir de una posición objetivo en X, Y, Z.

3.3 Cinemática Inversa

Para encontrar los ángulos `q1`, `q2` y la distancia `q3` del robot SCARA a partir de una matriz de transformación homogénea T, deberemos descomponer T en función de las variables articulares del robot. Considerando un robot SCARA con tres grados de libertad, el cálculo se basa en la posición y orientación del efector final en función de sus articulaciones.

La matriz de transformación homogénea T se define como:

$$T = \begin{bmatrix} \cos(\theta_{12}) & -\sin(\theta_{12}) & 0 & x \\ \sin(\theta_{12}) & \cos(\theta_{12}) & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Donde:

- $\theta_{12} = \theta_1 + \theta_2$, es la suma de los ángulos de las articulaciones rotacionales $q1$ y $q2$.
- x, y, y z son las coordenadas del efector final en el espacio cartesiano.

A partir de T, se pueden deducir los valores de $q1$, $q2$ y $q3$:

1. **Encontrar** $q3$:

$$q3 = z - d1$$

   Donde $d1$ es la distancia constante del eje base al plano de trabajo.

2. **Encontrar** $q2$: La ecuación para $q2$ se deriva de la posición (x, y):

$$\cos(\theta_2) = \frac{x^2 + y^2 - L_1^2 - L_2^2}{2L_1L_2}$$

   Luego, $q2$ puede ser calculado usando:

$$q2 = \pm \arccos\left(\frac{x^2 + y^2 - L_1^2 - L_2^2}{2L_1L_2}\right)$$

   En la práctica, se selecciona el valor de $q2$ más adecuado para evitar posiciones no alcanzables (dependiendo de la configuración del robot).

3. **Encontrar** $q1$: Con el valor de $q2$, el valor de $q1$ se calcula como:

$$q1 = \arctan 2(y, x) - \arctan 2\left(L_2 \sin(q2), L_1 + L_2 \cos(q2)\right)$$

4. Diseño Mecánico

4.1 Criterio

El sistema de transmisión del robot SCARA es esencial para asegurar la precisión y eficiencia en los movimientos de los eslabones. En este proyecto, las transmisiones están diseñadas para maximizar el control sobre los motores paso a paso, permitiendo movimientos suaves y precisos en el plano XY y a lo largo del eje Z.

Para proteger el robot y asegurar que los movimientos se mantengan dentro de rangos seguros, se han instalado interruptores de límite (limit switches) en cada eslabón. Estos interruptores sirven tanto para la calibración inicial del robot como para la protección continua durante su operación:

- **Calibración Inicial**: Al inicio, el robot realiza un movimiento hacia los interruptores de límite para establecer su posición de origen. Esto garantiza

que todos los movimientos subsecuentes se realicen dentro de los límites seguros y permite una calibración automática del sistema.

- **Prevención de Daños**: Durante la operación, los interruptores de límite desactivan el motor correspondiente cuando se alcanza el límite físico de un eslabón. Esto evita que el motor o el sistema de transmisión sufran daños por movimientos fuera del rango permitido.

4.2 Transmisiones

En total se han diseñado 3 sistemas de transmisiones complejos, en los cuales en dos se hace una doble reducción utilizando una polea doble intermedia, resultando en:

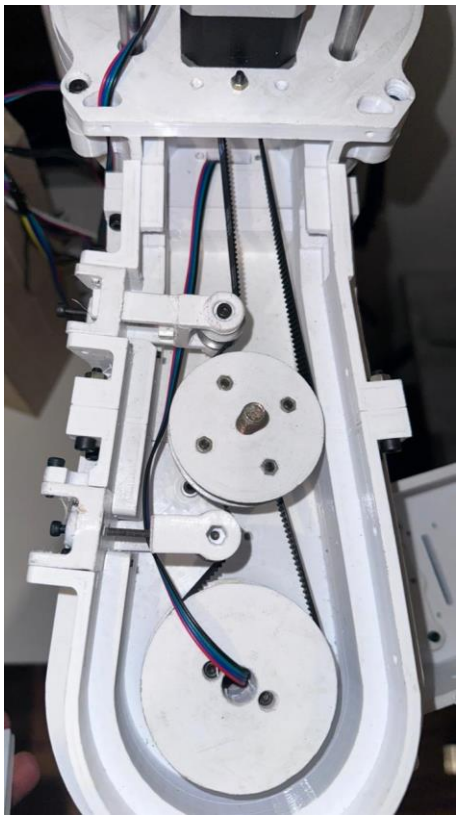
MOTOR 1: Relación 18:1

MOTOR 2: Relación 8mm avance x vuelta (Eje Z – Varilla roscada)

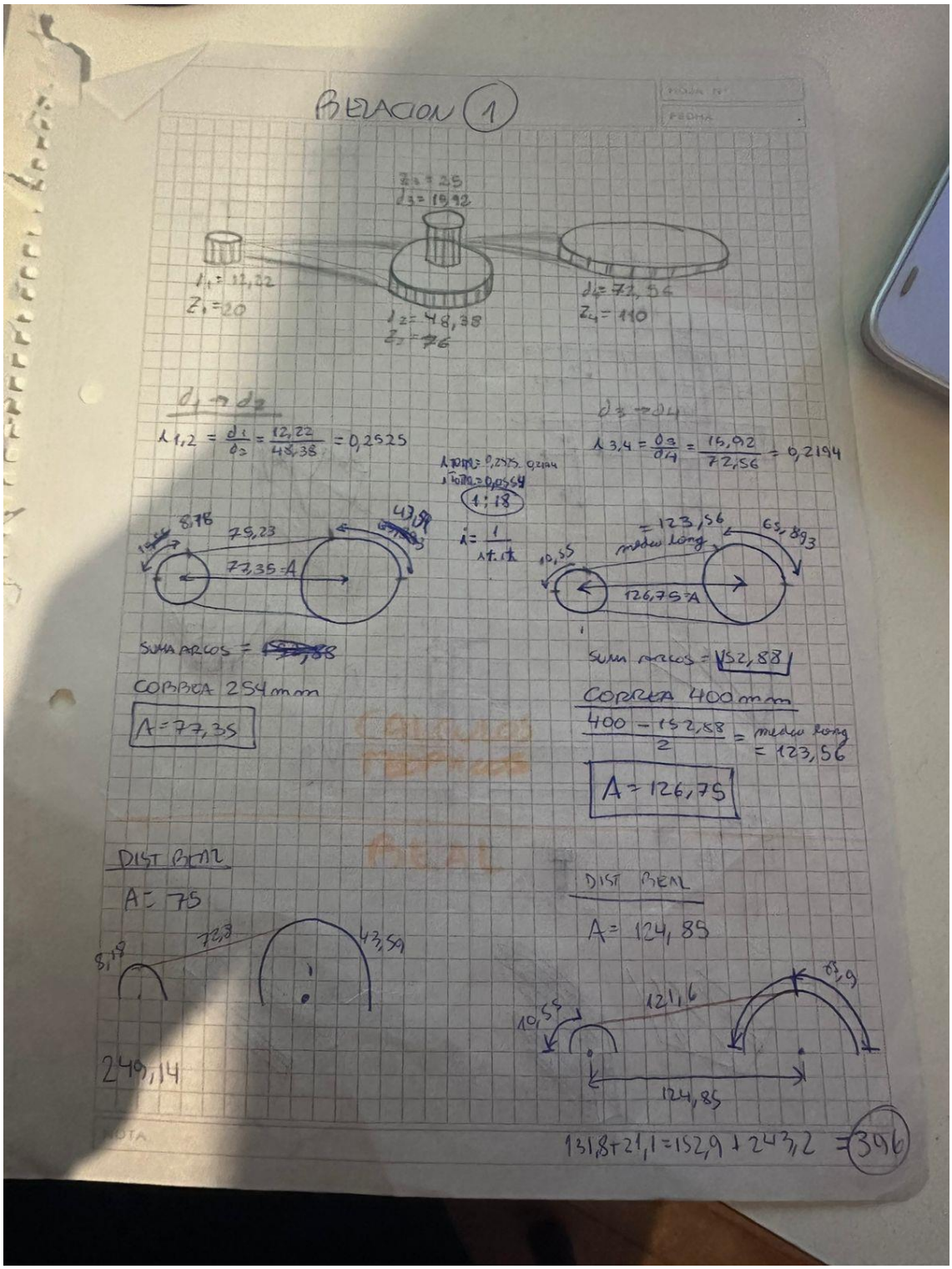
MOTOR 3: Relacion 15:1

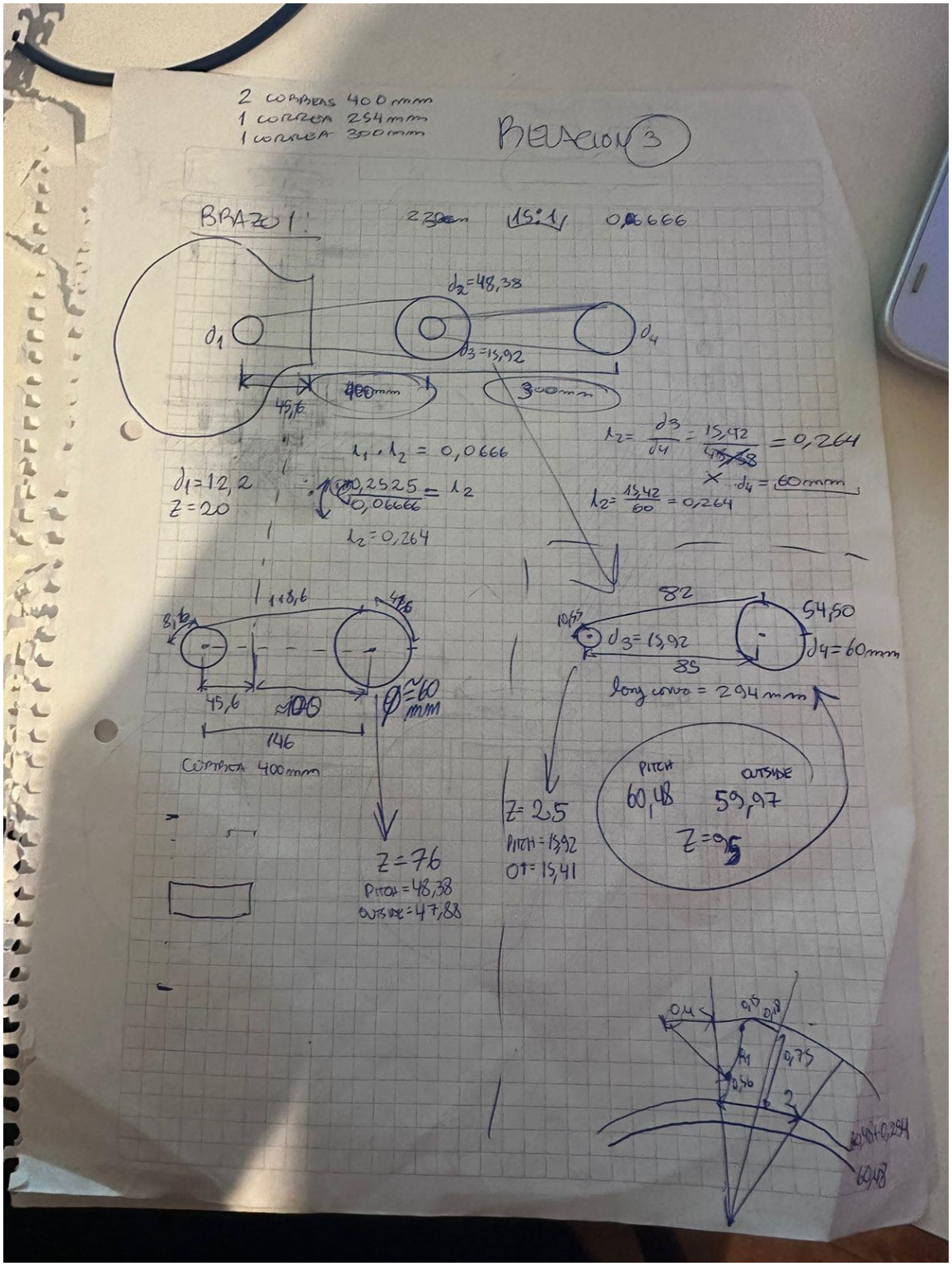
MOTOR 4: Relacion 3:1

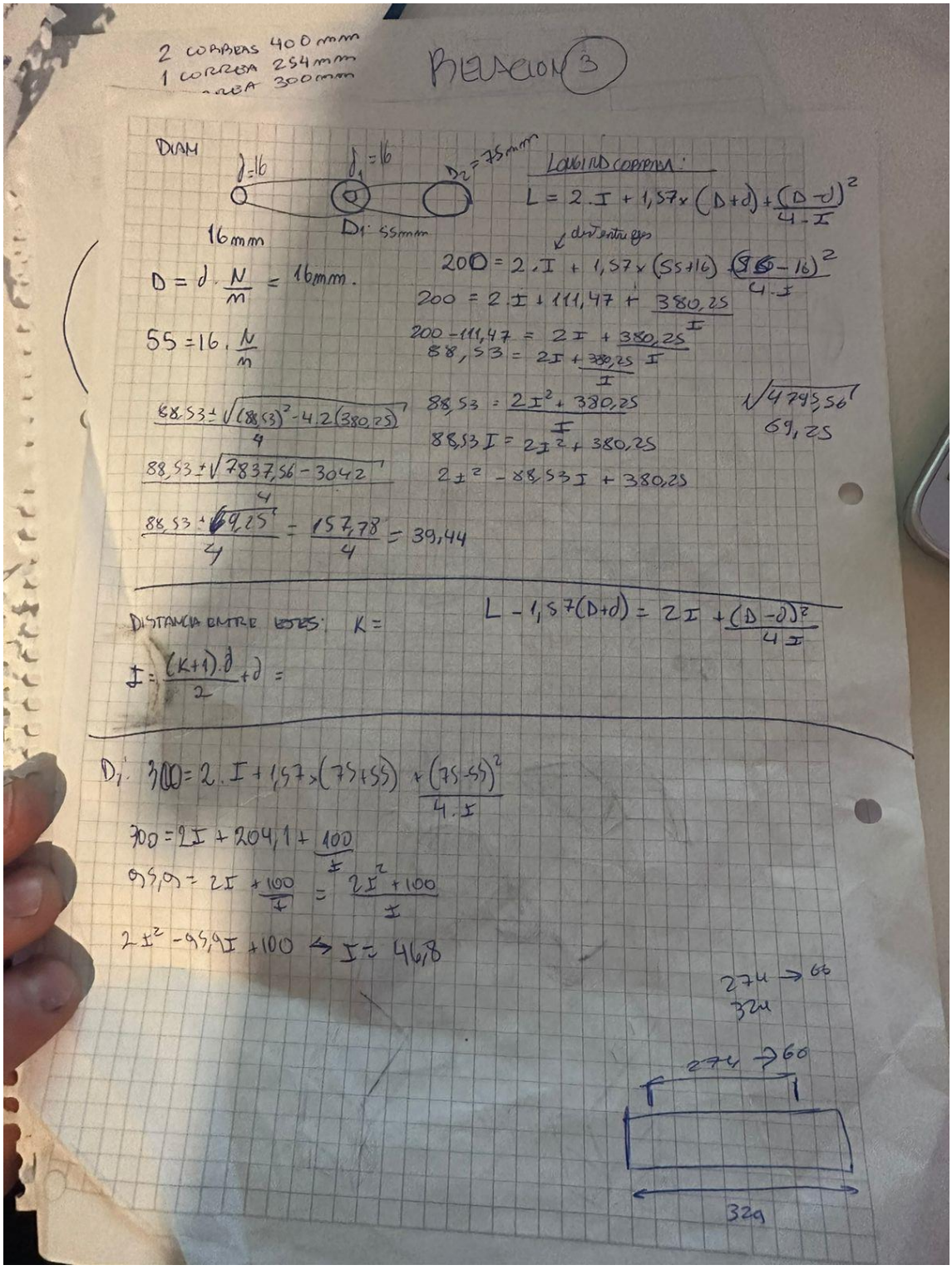
Estas relaciones de transmisión han sido seleccionadas para satisfacer las necesidades específicas de cada grado de libertad, teniendo en cuenta los límites de espacio y los requerimientos de movimiento del robot.



4.3 Calculos


RELACION (1)
Z3 = 25
d3 = 19,92
d1 = 12,22
Z1 = 20
d2 = 48,38
Z2 = 76
d4 = 72,56
Z4 = 110
d1 → d2
λ1,2 = d1/d2 = 12,22/48,38 = 0,2525
d3 → d4
λ3,4 = d3/d4 = 19,92/72,56 = 0,2194
λ total = 0,2525 . 0,2194
λ total = 0,0554
1 : 18
i = 1/λ total
8,78
75,23
77,35 = A
SUMA ARCOS =
COBREA 254 mm
A = 77,35
123,56 = medio long
63,89 3
10,55
126,75 = A
SUMA ARCOS = 152,88
COBREA 400 mm
400 − 152,58 / 2 = medio long = 123,56
A = 126,75
DIST REAL
A = 75
8,78
72,8
43,54
249,14
DIST REAL
A = 124,85
10,55
121,6
63,9
124,85
131,8 + 21,1 = 152,9 + 243,2 = 396


2 CORREAS 400 mm
1 CORREA 254 mm
1 CORREA 300 mm
REDUCCION 3
BRAZO 1
15:1
0,0666
d_2=48,38
d_1
d_3=15,92
d_4
400mm
300mm
45,6
$i_1 \cdot i_2 = 0,0666$
d_1=12,2
Z=20
0,2525 = i_2
0,0666
i_2=0,264
$i_2 = \frac{d_3}{d_4} = \frac{15,42}{48,38} = 0,264$
$\times d_4$ = 60mm
$i_2 = \frac{15,42}{60} = 0,264$
146
Ø=60 mm
Correa 400mm
Z=76
Pitch=48,38
Outside=47,88
82
54,50
d_3=15,92
d_4=60mm
85
long curva = 294 mm
Z=25
Pitch=15,92
OT=15,41
PITCH
OUTSIDE
60,48
59,97
Z=95
0,4
0,75
0,56
60,48+0,254
60,48


2 CORREAS 400 mm
1 CORREA 254 mm
1 CORREA 300 mm
RELACION 3
DIAM
d=16
d=16
D2=75mm
D1=55mm
16mm
D = d . N/n = 16mm.
55 = 16 . N/n
Longitud correas:
L = 2.I + 1,57 x (D+d) + (D-d)^2 / 4.I
distancia ejes
200 = 2.I + 1,57 x (55+16) + (55-16)^2 / 4.I
200 = 2.I + 111,47 + 380,25/I
200 - 111,47 = 2I + 380,25/I
88,53 = 2I + 380,25/I
88,53 = 2I^2 + 380,25 / I
88,53 I = 2I^2 + 380,25
2I^2 - 88,53 I + 380,25
√4745,56
69,25
DISTANCIA ENTRE EJES: K =
L - 1,57(D+d) = 2I + (D-d)^2 / 4I
I = (K+1).d/2 + d =
D1: 300 = 2.I + 1,57 x (75+55) + (75-55)^2 / 4.I
300 = 2I + 204,1 + 100/I
95,9 = 2I + 100/I = 2I^2 + 100 / I
2I^2 - 95,9I + 100 ⇔ I = 46,8
27u → 66
32u
27u → 60
329

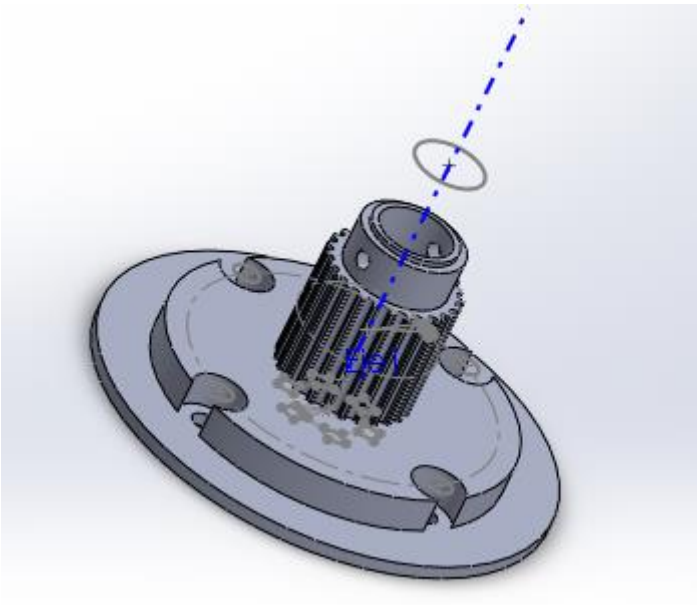
5. Desarrollo 3D - Solidworks

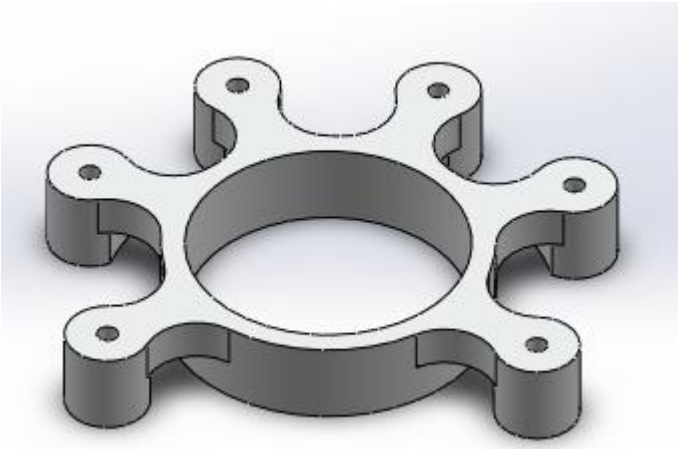
5.1 Desarrollo de las piezas

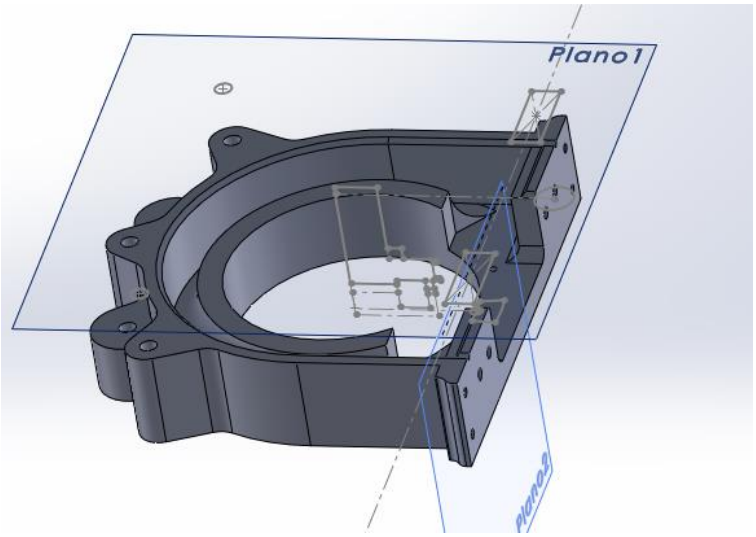
El diseño de las piezas y el ensamblaje del robot SCARA se ha realizado utilizando el software de modelado 3D Solidworks. Esta herramienta permite crear modelos detallados de cada componente y visualizar el ensamblaje completo antes de la fabricación, asegurando que todas las partes se ajusten y funcionen correctamente.

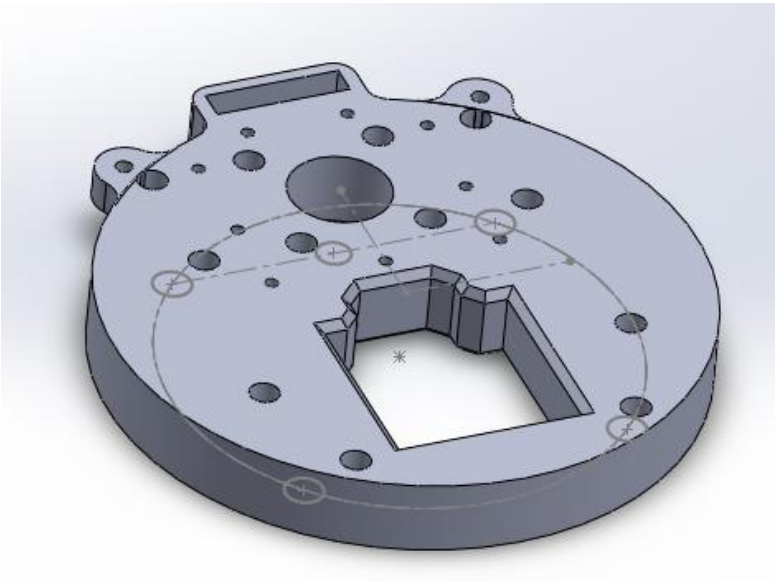
Se han creado una gama diversa de piezas que van desde soportes, carcazas, poleas, retenes, tensores, sujetadores, acoples, etc, teniendo en cuenta factores como la resistencia estructural, el peso y la facilidad de impresión.

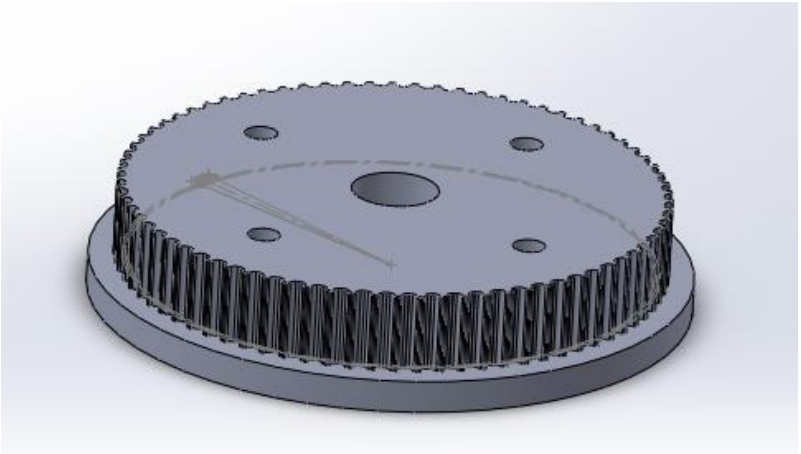
Se visualizan a continuación fotos de algunas.

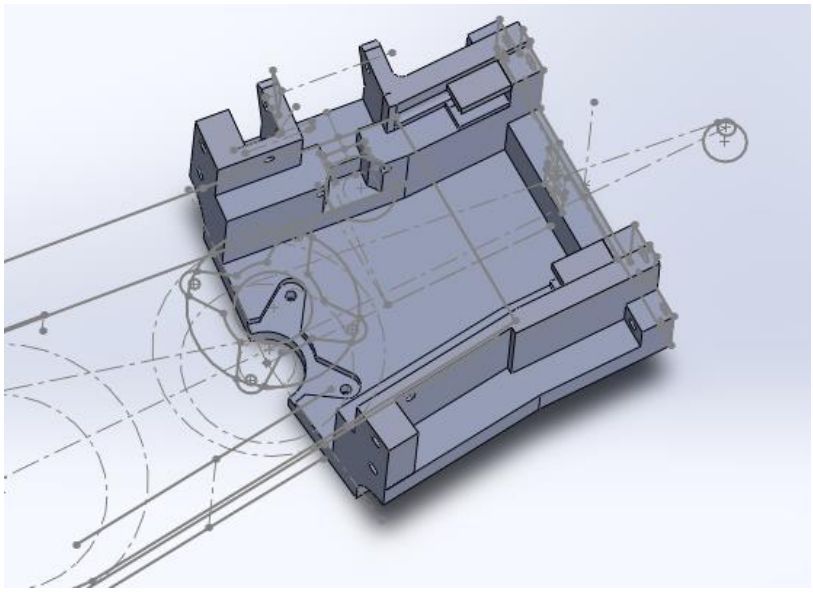





Plano1
Plano2







5.2 Desarrollo de los ensamblajes

Una vez diseñadas las piezas individuales, se procedió a realizar el ensamblaje en Solidworks para verificar que todas encajaran correctamente y que el diseño general del robot fuera funcional.

- **Simulación de Movimientos**: Antes de la fabricación, se realizó una simulación de los movimientos del robot en Solidworks para asegurarse de que todos los eslabones tuvieran el rango de movimiento adecuado y que no hubiera interferencias entre las piezas. Esto incluyó la verificación del ángulo de rotación de los motores y la libertad de movimiento del efector final.
- **Validación de Espacios y Colisiones**: Se utilizó la herramienta de detección de colisiones de Solidworks para confirmar que el ensamblaje completo operara sin interferencias entre las piezas durante el movimiento. Se hicieron ajustes en los eslabones y en la carcasa de los motores para asegurar que no hubiera colisiones no deseadas.
- **Optimización de Peso y Resistencia**: Solidworks permite realizar análisis de tensión y peso en cada componente. Se utilizó esta función para identificar posibles mejoras en el diseño, ajustando el grosor y el patrón de relleno de las piezas para garantizar que fueran lo suficientemente resistentes sin añadir peso innecesario.

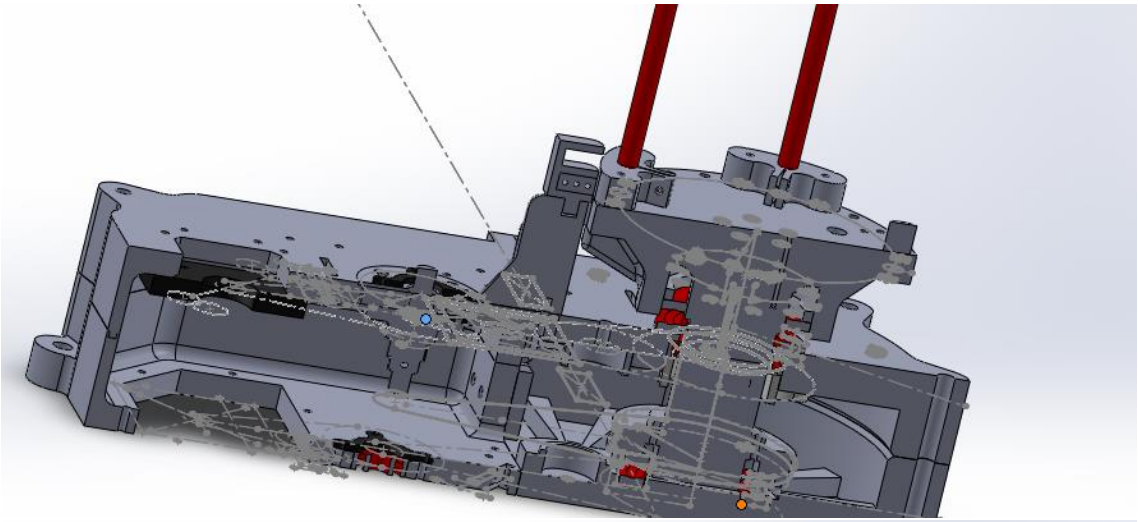
Fabricación de Piezas

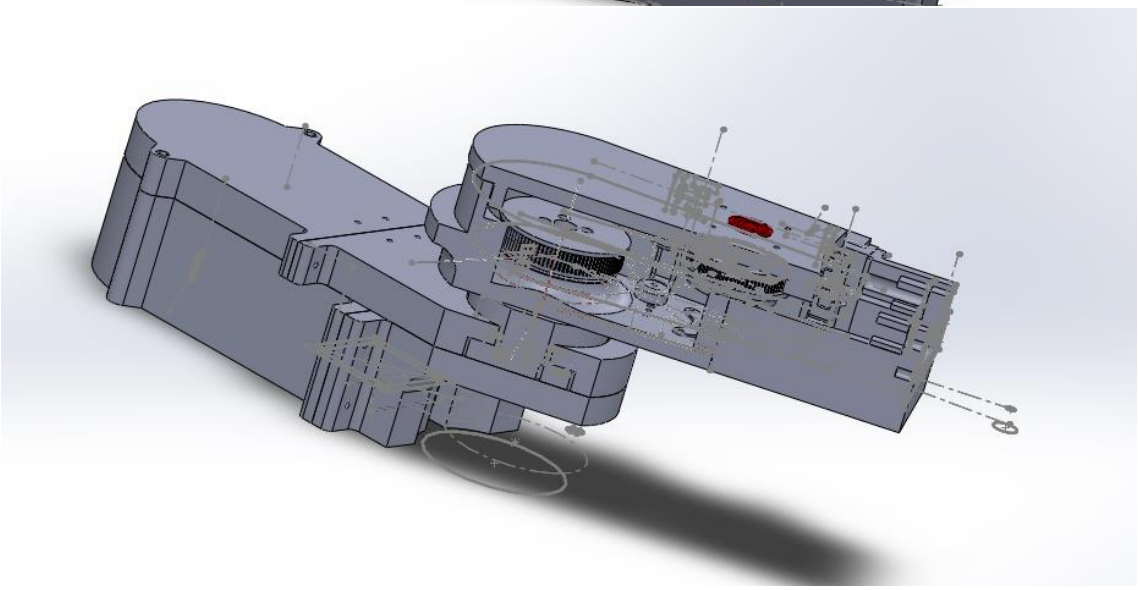
Las piezas diseñadas en Solidworks fueron exportadas como archivos STL para ser impresas en 3D. El material seleccionado para la mayoría de las piezas fue PLA debido a su facilidad de impresión y suficiente resistencia para el robot en condiciones normales.

Ensamblaje Final

El ensamblaje final del robot SCARA fue realizado siguiendo el modelo desarrollado en Solidworks, lo cual permitió que el proceso fuera ágil y preciso. Gracias al modelado en 3D, todas las piezas se ensamblaron sin necesidad de realizar ajustes significativos, demostrando la eficacia del proceso de diseño y simulación previa.

A continuación se muestran algunas imágenes:



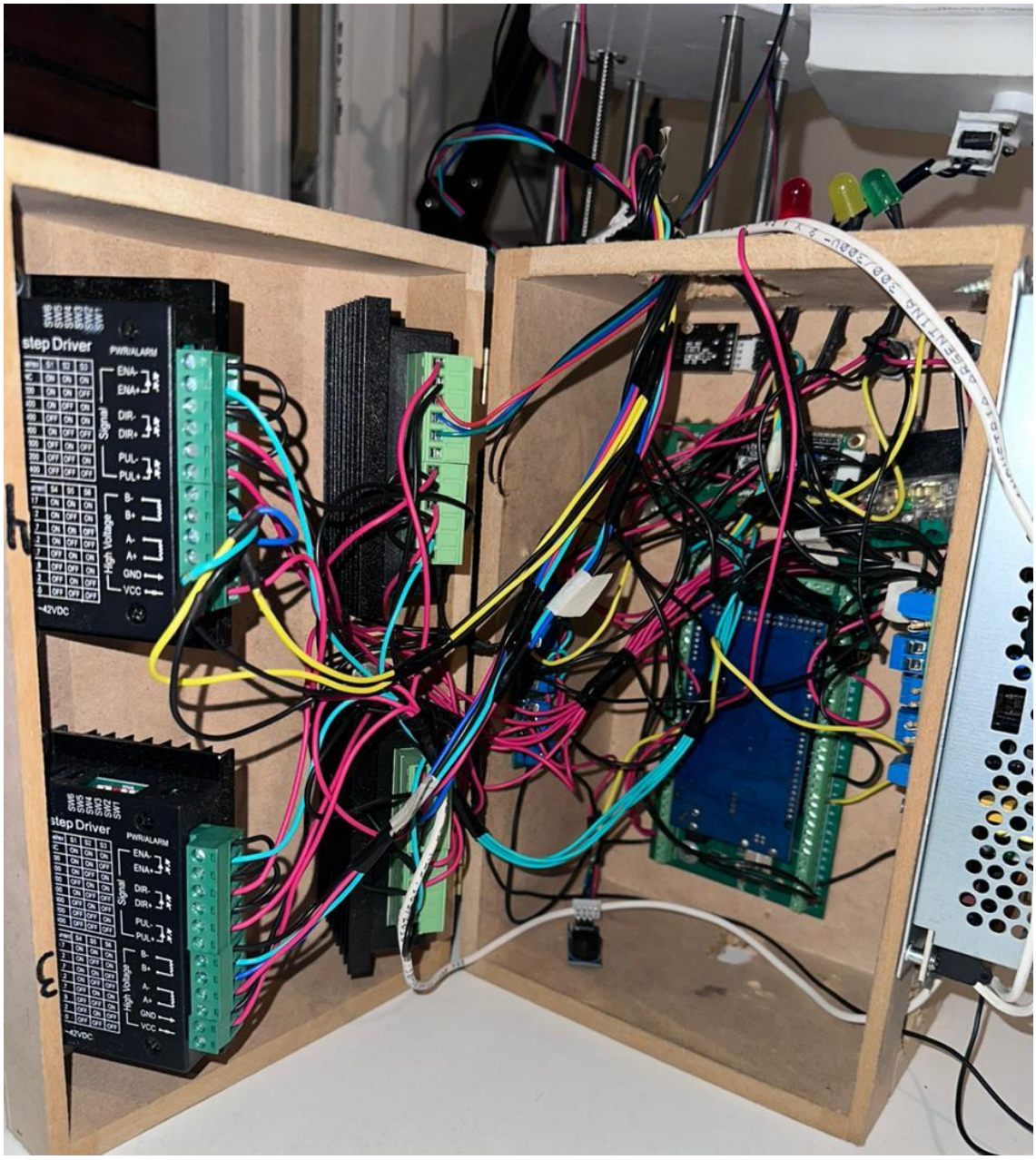


6. Diseño Electrónico

6.1 Criterios

El diseño electrónico del robot SCARA se centra en la integración y control de los motores paso a paso, los controladores de motor, y otros componentes periféricos como los interruptores de límite y la pantalla LCD. El sistema de control se basa en una placa Arduino Mega 2560, que proporciona la potencia de procesamiento necesaria para gestionar simultáneamente los movimientos del robot y la interfaz de usuario.

6.2 Fotos del conexionado



7. Diseño Interfaz HDI

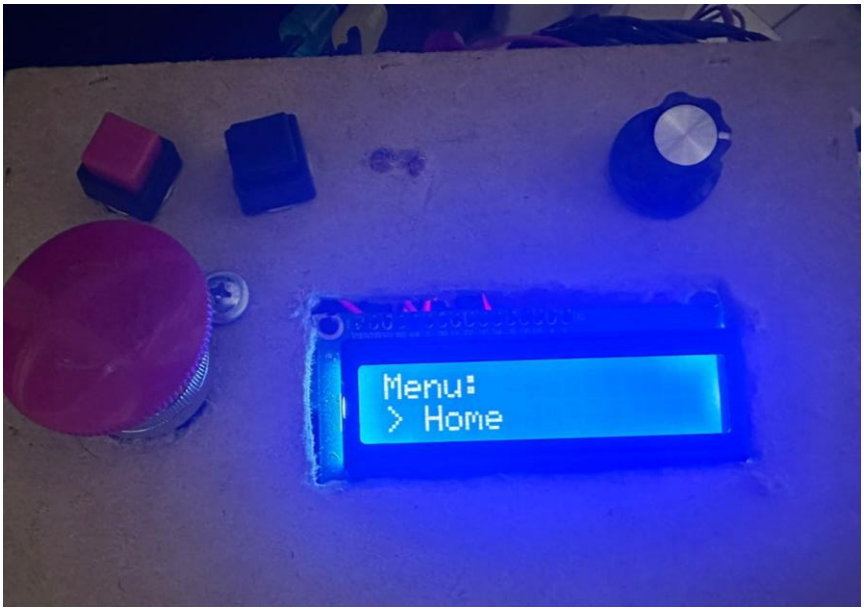
7.1 Interfaz Principal

El diseño de la interfaz HMI del robot SCARA se centra en facilitar la interacción entre el usuario y el sistema de control del robot. La interfaz se implementa a través de una pantalla LCD junto con un codificador rotativo que permite navegar por el menú, seleccionar opciones y ajustar parámetros de control. Esta configuración asegura que el usuario pueda monitorear y controlar el robot de manera intuitiva y eficiente.

La interfaz principal se estructura en un menú jerárquico que se despliega en la pantalla LCD. Este menú incluye opciones para controlar y configurar el robot SCARA de acuerdo a diferentes necesidades operativas. Las opciones principales del menú incluyen:

- **Home**: Permite al usuario ejecutar la función de homing del robot, posicionándolo en su origen.
- **PuntoToGo**: Proporciona acceso a la función de movimiento controlado a coordenadas X, Y, Z definidas por el usuario, con límites establecidos para cada eje.
- **Set y Calibración**: Facilita la configuración de parámetros iniciales, incluyendo la dirección de calibración y la ejecución de la verificación.
- **Ajuste de Parámetros**: Permite ajustar la velocidad, aceleración y curva de aceleración del robot para personalizar su desempeño.
- **Modelo de Movimiento**: Ofrece opciones para seleccionar entre los modos de cinemática directa o inversa, de acuerdo a la operación deseada.
- **Modo Manual**: Permite realizar control manual de los motores, ya sea energizándolos o liberándolos, o ajustando individualmente cada motor.
- **Set Origen**: Permite al usuario llevar el robot a su origen, establecer un nuevo origen o recalibrar el sistema absoluto.
- **Ejecutar Movimiento Paralelo**: Ofrece opciones para configurar y ejecutar movimientos paralelos en el espacio.
- **Rutas**: Da acceso a funciones avanzadas de rutas, incluyendo la creación, visualización y ejecución de trayectorias personalizadas.

El codificador rotativo permite seleccionar opciones del menú girándolo para navegar y presionándolo para confirmar una selección. Las opciones seleccionadas se desplazan automáticamente en la pantalla, lo cual facilita la lectura y la navegación en menús de varias opciones.



8. Diseño del Software

8.1 Código para el Arduino MEGA

```
// INCLUSION DE LIBRERIAS A UTILIZAR
 #include <AccelStepper.h>
 #include <Wire.h>
 #include <LiquidCrystal_I2C.h>
 #include <Encoder.h>
 #include <math.h>
//




bool mensajeMostrado = false;
unsigned long mensajeStartTime = 0; // Temporización para el mensaje
bool enMovimiento = false;    // Indica si los motores están en movimiento
bool mostrarMensaje = false;  // Indica si se debe mostrar el mensaje de "Punto Alcanzado"
bool movimientoCompletado = false;   // Indica si el movimiento se ha completado
unsigned long tiempoMensaje = 0;     // Para controlar el tiempo de mostrar el mensaje
unsigned long mensajeCompletoStartTime = 0;
bool mensajeCompletoMostrado = false;
```

```
unsigned long mostrarMensajeStartTime = 0;
const unsigned long mostrarMensajeDuration = 2000;




// Declaraciones de funciones



// DEFINICION DE PIN'S DE INPUT/OUTPUT
 //DEFINICION DE PIN'S DISPLAY, BUSSER Y ENCODER
  LiquidCrystal_I2C lcd(0x27, 16, 2);
  Encoder myEnc(51, 52);
  const int SwitchEncPin = 53;
  #define BUZZER 5

 /// DEFINIMOS LOS PINES DE CONTROL DE LOS MOTORES
  // PINES DE HABILITACION
   #define ENAPIN1 39
   #define ENAPIN2 41
   #define ENAPIN3 43
   #define ENAPIN4 45
  // PINES DE PULSO Y DIRECCION
   #define PULPIN1 37
   #define PULPIN2 35
   #define PULPIN3 33
   #define PULPIN4 31
   #define DIRPIN1 36
   #define DIRPIN2 34
   #define DIRPIN3 32
   #define DIRPIN4 30
 // DEFINIMOS LOS PINES DE LOS FINALES DE CARRERA
   #define LIMMIN1 22
   #define LIMMAX1 23
   #define LIMMIN2 24
   #define LIMMAX2 25
   #define LIMMIN3 26
   #define LIMMAX3 27
   #define LIMMIN4 28
```

```
#define LIMMAX4 29

// DEFINICION A MAS BAJO NIVEL DE LOS PIN'S DE LOS SWITCHS
#define LIMIT_SWITCH_PIN PINA
#define LIMIT_SWITCH_DDRA DDRA
#define LIMIT_SWITCH_PORT PORTA

// DEFINICION A MAS BAJO NIVEL DE LOS PIN'S PARA DIR Y PUL EN PUERTO C
#define MOTOR_PIN_DIR_PUL PORTC
#define MOTOR_PIN_DDR DDRC

// Indicadores visuales (LED)
#define RED_LED_PIN 48
#define YELLOW_LED_PIN 49
#define GREEN_LED_PIN 47

// PULSADORES
#define EMERGENCY_STOP_PIN 2
#define BOTON1 40
#define BOTON2 42
//

//CONTROL DEL ROBOT-VARIABLES
//Variables y constantes del motor
const int stepsPerRevolution = 200;
const int RETROTRAER_STEPS[4]= {40,30,30,10}; // Cantidad de pasos para retroceder
//const int retrotraer_1 = 40;
//const int retrotraer_2 = 30;
//const int retrotraer_3 = 30;
//const int retrotraer_4 = 10;
const uint8_t MOTOR1_DIR_MASK = 0b00000001;
const uint8_t MOTOR1_PUL_MASK = 0b00000010;
const uint8_t MOTOR2_DIR_MASK = 0b00000100;
const uint8_t MOTOR2_PUL_MASK = 0b00001000;
const uint8_t MOTOR3_DIR_MASK = 0b00010000;
const uint8_t MOTOR3_PUL_MASK = 0b00100000;
const uint8_t MOTOR4_DIR_MASK = 0b01000000;
const uint8_t MOTOR4_PUL_MASK = 0b10000000;
```

```
AccelStepper MOTORS[4]= {
  AccelStepper (AccelStepper::DRIVER, PULPIN1, DIRPIN1),
  AccelStepper (AccelStepper::DRIVER, PULPIN2, DIRPIN2),
  AccelStepper (AccelStepper::DRIVER, PULPIN3, DIRPIN3),
  AccelStepper (AccelStepper::DRIVER, PULPIN4, DIRPIN4),
};

//Relaciones de transmision
  const float relacionMOTOR1 = 475.0 / 32.0;
  const float relacionMOTOR2 = 1200.0 / 44.0;
  const float relacionMOTOR3 = 700.0 / 90.0;
//

// VARIABLES DE CINEMATICA
  //Longitud de eslabones
    float L1 = 260;
    float L2 = 210;
    float d1 = 120;
  //

  //Variables para los limites de los angulos/desplazamientos
    // Límites de movimiento
      const float q1Min = -90.0 * DEG_TO_RAD ;
      const float q1Max = 90.0 * DEG_TO_RAD ;
      const float q2Min = -135.0 * DEG_TO_RAD ;
      const float q2Max = 135.0 * DEG_TO_RAD ;
      const float q3Min = 0.0;
      const float q3Max = 440.0;
      long pasosMotor1 = 0, pasosMotor2 = 0, pasosMotor3 = 0;
    //

    float q1 = 0, q2 = 0, q3 = 0;

    float X_destino = 0.0;
    float Y_destino = 0.0;
    float Z_destino = 0.0;
  //

  //
```

```
    unsigned long PTGStartTime = 0;
    bool PTGMostrar = false;
    bool iniciarPTG = false;
    bool PTGCompleta = false;
  //

  // Variables para gestión no bloqueante
   unsigned long lastRunTime = 0;
   const unsigned long runInterval = 20; // Intervalo de actualización
  //
 //
//

//Variables de los Switchs
  const long delay_switchs = 250;

 const uint8_t LIMIT_SWITCH_MASKS[4][2] = {
  {0b00000001, 0b00000010}, // Switches para el motor 1
  {0b00000100, 0b00001000}, // Switches para el motor 2
  {0b00010000, 0b00100000}, // Switches para el motor 3
  {0b01000000, 0b10000000}  // Switches para el motor 4
  };
//

// Variables MODO MANUAL

 //Variables estado motores
 //

 //Variables Mover motores individualmente
 int ang_m1 = 0;
 int pasos_m1 = 0;

 int ang_m2 = 0;
 int pasos_m2 = 0;

 int ang_m3 = 0;
 int pasos_m3 = 0;
```

```
 int ang_m4 = 0;
 int pasos_m4 = 0;
 //

//

// Variables de SET ORIGEN
 // Variable global para el tiempo no bloqueante
   unsigned long origenAlcanzadoStartTime = 0;
   bool origenAlcanzadoMostrar = false;
 //

 bool iniciarEstablecerOrigen = false;
 bool EstablecerOrigenCompleta = false;

 bool iniciarIrAlOrigen = false;
 bool IrAlOrigenCompleta = false;
//

// Variables RECALIBRACION

 // Variable global para el tiempo no bloqueante
   unsigned long recalibracionAbsolutaStartTime = 0;
   bool recalibracionAbsolutaMostrar = false;
 //

 bool iniciarRecalibracionAbsoluta = false;
 bool recalibracionAbsolutaCompleta = false;
//


//Variables de SetYCalibracion
   long calib_m1 = 0;
   long calib_m2 = 0;
   long calib_m3 = 0;
   long calib_m4 = 0;

   const int verif_speed=330; //VELOCIDAD PARA "SET Y CALIBRACION"
```

```
  uint8_t global_first_verify;
  uint8_t global_first_retro;
  int global_steps_retro;
  bool verificacionCompleta = false;
  bool iniciarVerificacion = false;  // flag para controlar cuándo iniciar la verificación

 volatile int POSICIONES_VYC[4][2] = {
  {0, 0}, // Switches para el motor 1
  {0, 0}, // Switches para el motor 2
  {0, 0}, // Switches para el motor 3
  {0, 0}  // Switches para el motor 4
  };
//

//Variables y constantes de movimiento
  const int normalSpeed = 400; //VELOCIDAD DEFAULT DEL ROBOT
  const int slowSpeed = 20;
  const int maxSpeed = 200;
  const int acceleration = 100;
  const int VELOCIDAD_MIN = 1;
  const int VELOCIDAD_MAX = 1000;
  const int ACELERACION_MIN = 1;
  const int ACELERACION_MAX = 1000;

  // Variables "X, Y, Z" para la opción "PuntoToGo"  DEFAULTT
  int X = 140;
  int Y = 100;
  int Z = 120;
//

//Variables y constantes de interpolacion
 long pasosEntreSwitches[4] = {0, 0, 0, 0};  // Un valor para cada motor
 const int PASOS_PRUEBA[4]= {500,200,300,100};
//

// Flags, auxiliares, generales, etc
  bool flag6 = false;
  bool flagProteccion = false;
  bool mostrandoPasos = false;
```

```
unsigned long mostrarPasosStartTime = 0;
int motorMostrandoIndex = 0;
const unsigned long tiempoMostrarMotor = 2000;  // Tiempo de 2 segundos para mostrar cada motor
bool  moveFlag = true;
bool flagSTR = false;
bool proteccion = false;
char direccion='L'; ///por defecto nuestro robot calibra hacia L


bool emergencyStop = false;
bool ldr_proteccion = false;
bool resumePressed = false;

// Variables para el Buzzer (EMERGENCIA)
unsigned long buzzerEmergencyEndTime = 0; // Tiempo para el zumbido de emergencia
const unsigned long buzzerEmergencyInterval = 500; // Intervalo del zumbido (en ms)

// Variables para el Buzzer (DISPLAY)
unsigned long buzzerStartTime = 0;
unsigned long buzzerDuration = 100;  // Duración del zumbido en milisegundos
bool isBuzzerOn = false;

//Variables para control del Encoder
long oldPosition = 0; // Para almacenar la posición anterior del encoder
unsigned long lastEncoderChangeTime = 0; // Última vez que se cambió la posición del encoder
//




//

//CONSTANTES Y VARIABLES ASOCIADAS AL DISPLAY LCD

 //MENU ESTATUS GENERAL

  //ESTADOS:
  enum MenuState { MAIN_MENU,
```

```
                SUB_MENU_HOME,
                SUB_MENU_PUNTOTOGO,
                SUB_MENU_MODELO,
                 CONFIRM_MODELO,
                SUB_MENU_AJUSTE_PARAMETROS,
                SUB_MENU_SETCALIBRACION,
                 SUB_MENU_CAMBIAR_DIRECCION,
                SUB_MENU_SET_ORIGEN,
                SUB_MENU_MOV_PARALELO,
                SUB_MENU_ESTABLECER_RUTA,
                SUB_MENU_MODO_MANUAL,
                 SUB_MENU_ESTADO_MOTORES,
                 SUB_MENU_MOV_INDIVIDUAL,
                   SUB_MENU_M1,
                   SUB_MENU_M2,
                   SUB_MENU_M3,
                   SUB_MENU_M4
                }

                currentMenuState = MAIN_MENU;
        //



        //MENU PRINCIPAL ----SECCIONES----
        String mainMenu[9] = {"Home",
                         "PuntoToGo",
                         "Set y Calibracion",
                         "Ajuste de parametros",
                         "Modelo de movimiento",
                         "Modo Manual",
                         "SET Origen",
                         "Ej.M Paralelo",
                         "Rutas"};
        int menuIndex = 0;
        //


        // SUBMENU: "MODELO DE MOVIMIENTO"
          int MODELO = 0; // 0 = Cinemática Directa, 1 = Cinemática Inversa
          String modeloMovimientoMenu[3] = {"Cinematica Directa", "Cinematica Inversa", "<- ATRAS"}; //TEXTO
VISUALIZADO EN EL MENU
```

```
  int modeloMovimientoIndex = 0; // Índice del menú "Modelo de Movimiento"
  bool confirmChange = false; // Estado para confirmar cambio de modelo
 //


 // SUBMENU "AJUSTE DE PARAMETROS"
  String ajusteParametrosMenu[5] = {"Velocidad", "Aceleracion", "Curva de Aceleracion", "Inicio Set", "<- ATRAS"};
//TEXTO VISUALIZADO EN EL MENU
  int ajusteParametrosIndex = 0; // Índice del menú de Ajuste de Parametros
  bool editingParametros = false; // Estado para saber si se está editando una variable de Ajuste de Parametros

  int velocidad = 50;  // Rango 0 a 100
  int aceleracion = 50; // Rango 0 a 100

  //SUB-SUBMENU "Curvas de aceleracion"
   String curvasAceleracion[5] = {"Tipo 1", "Tipo 2", "Tipo 3", "Tipo 4", "<- ATRAS"}; //TEXTO VISUALIZADO EN EL
MENU
   int curvaAceleracionIndex = 0; // Índice para la curva de aceleración
  //
 //

 //SUBMENU "SET Y CALIBRACION"
 String setCalibracionMenu[3] = {"Cambiar Direccion", "Ejecutar", "<- ATRAS"}; //TEXTO VISUALIZADO EN EL MENU
 int setCalibracionIndex = 0; // Índice del menú "Set y Calibracion"
 String inicioSetCalibracion = "LEFT"; // Opciones: LEFT or RIGHT
  //SUB-SUBMENU "CAMBIAR DIRECCION"
   String cambiarDireccionMenu[3] = {"LEFT", "RIGHT", "<- ATRAS"};
   int cambiarDireccionIndex = 0; // Índice para el submenú "Cambiar Direccion"
  //
 //

 //SUBMENU "PUNTO TO GO"
  String puntoToGoMenu[5] = {"X", "Y", "Z", "Confirmar", "<- ATRAS"};
  int puntoToGoIndex = 0; // Índice del menú de PuntoToGo
  bool editing = false; // Estado para saber si se está editando una variable

  // Límites de las variables
  const int minValXPTG = 50;
  const int maxValXPTG = 470;
  const int minValYPTG = 50;
  const int maxValYPTG = 470;
```

```
  const int minValZPTG = 120;
  const int maxValZPTG = 560;



//

//SUBMENU "HOME"
  String homeMenu[3] = {"Ejecutar", "NO", "<- ATRAS"};
  int homeMenuIndex = 0; // Índice del menú de HOME
//

//SUBMENU "MODO MANUAL"
  String modoManualMenu[3] = {"Estado Motores", "Mover individualmente", "<- ATRAS"};
  int modoManualIndex = 0; // Índice del menú "Modelo de Movimiento"

   //SUB-SUBMENU ESTADO MOTORES
     String estadoMotoresMenu[3] = {"Liberar", "Energizar", "<- ATRAS"};
     int estadoMotoresIndex = 0; // Índice del menú de HOME
   //

   //SUB-SUBMENU MOVIMIENTO INDIVIDUAL
     String movIndividualMenu[5] = {"MOTOR 1", "MOTOR 2", "MOTOR 3", "MOTOR 4", "<- ATRAS"};
     int movIndividualIndex = 0; // Índice del menú de HOME

       //SUB-SUB-SUBMENU MOTORES
         //MOTOR1
           String motor1Menu[4] = {"Ang", "Pasos", "Mover", "<- ATRAS"};
           int motor1Index = 0;
           bool editingM1 = false;
           const int minValANG1 = -90;
           const int maxValANG1 = 90;
           const int minValPAS1 = -1336;
           const int maxValPAS1 = 1336;

         //MOTOR2
           String motor2Menu[4] = {"Ang", "Pasos", "Mover", "<- ATRAS"};
           int motor2Index = 0;
           bool editingM2 = false;
           const int minValANG2 = -220;
```

```
  const int maxValANG2 = 220;
  const int minValPAS2 = -6000;
  const int maxValPAS2 = 6000;

 //MOTOR3
  String motor3Menu[4] = {"Ang", "Pasos", "Mover", "<- ATRAS"};
  int motor3Index = 0;
  bool editingM3 = false;
  const int minValANG3 = -135;
  const int maxValANG3 = 135;
  const int minValPAS3 = -1050;
  const int maxValPAS3 = 1050

  ;

 //MOTOR4
  String motor4Menu[4] = {"Ang", "Pasos", "Mover", "<- ATRAS"};
  int motor4Index = 0;
  bool editingM4 = false;
  const int minValANG4 = -6000;
  const int maxValANG4 = 6000;
  const int minValPAS4 = -6000;
  const int maxValPAS4 = 6000;
 //
//
 //
//

//SUBMENU "SET ORIGEN"
 String setOrigenMenu[4] = {"Ir a Origen", "Establecer nuevo Origen", "Ir a Origen Absoluto", "<- ATRAS"};
 int setOrigenIndex = 0; // Índice del menú de HOME
//

//SUBMENU "EJECUTAR MOVIMIENTO PARALELO"
 String movParaleloMenu[4] = {"Establecer plano", "Establecer puntos", "Ejecutar", "<- ATRAS"};
 int movParaleloIndex = 0; // Índice del menú de HOME

 //SUB-SUBMENU ESTABLECER PLANO
```

```
    String planoParalelo[3] = {"X Constante (YZ)", "Y Constante (XZ)", "<- ATRAS"};
    int planoParaleloIndex = 0; // Índice del menú de HOME
   //

   //SUB-SUBMENU ESTABLECER PUNTOS
    // X CONSTANTE
     String paraleloX[5] = { "Est. Origen","Est. Destino", "<- ATRAS", "Menu Principal"};
     int paraleloXIndex = 0; // Índice del menú de PuntoToAdd
     bool editingParaleloX = false; // Estado para saber si se está editando una variable

     // SUB-SUB-SUBMENU ESTABLECER ORIGEN

     //

     // SUB-SUB-SUBMENU ESTABLECER DESTINO


    // Y CONSTANTE
     String paraleloY[5] = {"Est. Origen","Est. Destino", "<- ATRAS"};
     int paraleloYIndex = 0; // Índice del menú de PuntoToAdd
     bool editingParaleloY = false; // Estado para saber si se está editando una variable
    //
   //

  // Límites de las variables
   const int minValXParalelo = 20;
   const int maxValXParalelo = 60;
   const int minValYParalelo = 20;
   const int maxValYParalelo = 60;
   const int minValZParalelo = 20;
   const int maxValZParalelo = 60;
  //
 //
 //

 //SUBMENU "ESTABLECER RUTA"
  String rutaMenu[5] = {"Nueva ruta", "Ver rutas", "Añadir puntos", "Ejecutar rutas", "<- ATRAS"};
  int rutaMenuIndex = 0; // Índice del menú de HOME
```

```
  //SUB-SUBMENU AÑADIR PUNTOS
  String puntoToAdd[6] = {"X", "Y", "Z", "Sel. Ruta", "Añadir", "<- ATRAS"};
   int puntoToAddIndex = 0; // Índice del menú de PuntoToAdd
   bool editingAdd = false; // Estado para saber si se está editando una variable

   // Límites de las variables
   const int minValXAdd = 50;
   const int maxValXAdd = 470;
   const int minValYAdd = 50;
   const int maxValYAdd = 470;
   const int minValZAdd = 120;
   const int maxValZAdd = 560;
  //
 //


//

 //DESPLAZAMIENTO DE TEXTO EN EL DISPLAY
  unsigned long lastScrollTime = 0; // Tiempo de la última actualización del desplazamiento
  int scrollIndex = 0; // Índice actual de desplazamiento
  String currentScrollingText = ""; // Texto actualmente en desplazamiento
  bool isScrolling = false; // Estado de si el texto debe desplazarse

 //
//

// Variables para temporización no bloqueante
 unsigned long lastDebounceTime = 0; // Tiempo de la última interacción con el encoder
 const unsigned long debounceDelay = 300; // Retraso de debounce para el botón
 unsigned long buzzerEndTime = 0; // Tiempo de finalización para el BUZZER
 unsigned long puntoToGoEndTime = 0; // Tiempo de finalización para la función PUNTOTOGO
 bool isPuntoToGoRunning = false; // Estado de ejecución de PUNTOTOGO
//
```

```
///----------------------------------------------------FUNCIONES-----------------------------------------------------------


void selectValuesMotor(int motorIndex, char replace_direction, char sentido_retraccion) {

 global_first_verify = (replace_direction == 'L') ? LIMIT_SWITCH_MASKS[motorIndex][0] :
LIMIT_SWITCH_MASKS[motorIndex][1];

 global_first_retro = (sentido_retraccion == 'R') ? LIMIT_SWITCH_MASKS[motorIndex][1] :
LIMIT_SWITCH_MASKS[motorIndex][0];

 global_steps_retro = RETROTRAER_STEPS[motorIndex];


 // Mostrar el mensaje en el LCD de forma no bloqueante

  static unsigned long displayStartTime = 0;

  displayStartTime = millis(); // Marca el tiempo actual para la visualización del mensaje

}



//FUNCION CHEQUEO PARADA DE EMERGENCIA

 ISR(TIMER1_COMPA_vect) {

    moveFlag = true; // Establece la bandera de movimiento

 }



 void checkEmergencyStop() {

   if (digitalRead(EMERGENCY_STOP_PIN) == LOW) {  // Si el botón de emergencia está presionado

      emergencyStop = true;  // Activa la parada de emergencia


      // Mostrar "EMERGENCIA" en el LCD

      lcd.clear();

      lcd.setCursor(0, 0);

      lcd.print("EMERGENCIA");

      lcd.setCursor(0, 1);

      lcd.print("Stopped!");


      // Detener los motores

      for (int i = 0; i < 4; i++) {

        MOTORS[i].stop();  // Detener el motor

        digitalWrite(ENAPIN1 + (i * 2), HIGH);  // Deshabilitar el motor

      }


      // Iniciar el zumbido de emergencia

   } else {
```

```
            if (emergencyStop) {
              emergencyStop = false;

              // Mostrar mensaje en el LCD para reiniciar
              lcd.clear();
              lcd.setCursor(0, 0);
              lcd.print("Reinicie el");
              lcd.setCursor(0, 1);
              lcd.print("sistema");

              // Regresar al menú principal
              currentMenuState = MAIN_MENU;
              updateMenu();
            }
          }
        }
      //
```

```
      void setup() {

        //CONFIGURACION DE LOS PINES BUZZER Y ENCODER INPUT/OUTPUT
          pinMode(SwitchEncPin, INPUT_PULLUP);
          pinMode(BUZZER, OUTPUT);  // Configura el pin del zumbador como salida
          digitalWrite(BUZZER, HIGH);  // Asegura que el buzzer esté apagado
        //

        // PARADA DE EMERGENCIA
          //sei(); cli();
          pinMode(EMERGENCY_STOP_PIN, INPUT);
          attachInterrupt(digitalPinToInterrupt(EMERGENCY_STOP_PIN), checkEmergencyStop, FALLING); // Asocia la
interrupcion al pin de la parada de emergencia
        //

        // CONFIGURACION INPUT/OUTPUT PINES
```

```
  pinMode(EMERGENCY_STOP_PIN, OUTPUT); //pin del led de parada de emergencia definido como OUTPUT
  digitalWrite(EMERGENCY_STOP_PIN, LOW); // led parada de emergencia inicio apagado
  DDRA = 0x00;  // Confugura todos los pines del puerto A como entradas ---> switchs
  DDRC = 0xFF;  // Configura todos los pines del puerto C como salidas ---> DIR Y PUL
 //

 // SETEO DE LAS VELOCIDADES MIN Y MAX DE LOS MOTORES x default
  for (int i = 0; i < 4; i++) {
    MOTORS[i].setMaxSpeed(1000);
    MOTORS[i].setAcceleration(100);
  }
 //

 // LCD
  lcd.init();  // Inicializa el LCD
  lcd.backlight();  // Enciende la luz de fondo del LCD
  updateMenu();  // Muestra el menú inicial
 //

}


//FUNCION QUE MUEVE LOS MOTORES HASTA SU DETECCION
 void giroSTR(char replace_direction, char sentido_retraccion) {

  //CICLO FOR PARA LOS 4 MOTORES
    for (int motorIndex = 0; motorIndex < 3; motorIndex++) {
      AccelStepper &num_motor = MOTORS[motorIndex];
      bool firstSwitchDetected = false;
      bool secondSwitchDetected = false;

      // Seleccionamos los valores correspondientes para este motor
      selectValuesMotor(motorIndex, replace_direction, sentido_retraccion);

      // Invertimos los switches del motor 1
      if (motorIndex == 0) {
        // Intercambiar los switches del motor 1
        uint8_t temp = global_first_verify;
        global_first_verify = global_first_retro;
```

```
    global_first_retro = temp;
}

digitalWrite(ENAPIN1 + (motorIndex * 2), LOW);  // Habilita el motor correspondiente

//DISPLAY LCD
lcd.clear();
lcd.setCursor(0, 0);
lcd.print("Motor ");
lcd.print(motorIndex + 1);
lcd.setCursor(0, 1);
lcd.print("Verificando...");

// Reiniciamos la posición del motor para contar pasos
num_motor.setCurrentPosition(0);

// Fase 1: Mover en la dirección inicial hasta detectar el primer switch
 num_motor.setSpeed(verif_speed); // Ajusta la velocidad de verificación
 unsigned long debounceStart = millis();

 while (!firstSwitchDetected) {
    num_motor.runSpeed();  // Mueve el motor en el primer sentido

    // Verificamos si el switch ha sido presionado (lógica inversa)
    if ((PINA & global_first_verify) != 0) {  // Si el switch es activado
       if (millis() - debounceStart > 50) {  // Debounce no bloqueante
          num_motor.stop();  // Detenemos el motor
          firstSwitchDetected = true;

          // Retrocedemos una cantidad de pasos
          num_motor.move(-global_steps_retro);
          while (num_motor.distanceToGo() != 0) {
             num_motor.run();  // Retrocedemos
          }

          // Cambiamos de dirección para la siguiente verificación
          lcd.clear();
          lcd.setCursor(0, 0);
          lcd.print("Motor ");
```

```
            lcd.print(motorIndex + 1);
            lcd.setCursor(0, 1);
            lcd.print("Verif otro sw.");
          }
        }
      }
      // Reiniciamos la posición antes de mover en el sentido contrario
      num_motor.setCurrentPosition(0);
    //

    // Fase 2: Mover en el sentido contrario hasta detectar el segundo switch
      num_motor.setSpeed(-verif_speed);  // Cambiamos de dirección
      debounceStart = millis();  // Reiniciamos debounce para el segundo switch

      while (!secondSwitchDetected) {
        num_motor.runSpeed();  // Mueve el motor en el sentido contrario

        // Verificamos si el segundo switch ha sido presionado
        if ((PINA & global_first_retro) != 0) {  // Si el segundo switch es activado
          if (millis() - debounceStart > 50) {  // Debounce no bloqueante
            num_motor.stop();  // Detenemos el motor
            secondSwitchDetected = true;

                        pasosEntreSwitches[motorIndex] = abs(num_motor.currentPosition());

            // Retrocedemos una pequeña cantidad de pasos
            num_motor.move(global_steps_retro);
            while (num_motor.distanceToGo() != 0) {
              num_motor.run();  // Retrocedemos
            }

            // Guardamos la posición final del motor
            POSICIONES_VYC[motorIndex][flagSTR ? 1 : 0] = num_motor.currentPosition();
          }
        }
      }
    //

    // Apagar el motor una vez que termine la verificación
```

```
            digitalWrite(ENAPIN1 + (motorIndex * 2), HIGH);  // Deshabilita el motor
            verificacionCompleta = true;  // Marcar la verificación como completa
          }
         //

         // Al terminar la verificación de todos los motores, iniciar la visualización de los pasos
          motorMostrandoIndex = 0;
          mostrandoPasos = true;
          mostrarPasosStartTime = millis();
          flagSTR = !flagSTR;  // Alterna entre la primera y segunda verificación
         //
        }
       //

      // FUNCION PARA MOSTRAR LOS PASOS DE TODOS LOS MOTORES "SET Y VERIFICACION" EN EL LCD -> SIN BLOQUEAR
EL MENU
       void mostrarPasosMotores() {
          lcd.clear();
          lcd.setCursor(0, 0);
          for (int i = 0; i < 4; i++) {
            lcd.setCursor(0, 0);
            lcd.print("Motor ");
            lcd.print(i + 1);
            lcd.setCursor(0, 1);
            lcd.print("Pasos: ");
            lcd.print(pasosEntreSwitches[i]);

            // Espera breve no bloqueante para mostrar los pasos de cada motor (controlado con millis)
            unsigned long startTime = millis();
            while (millis() - startTime < 2000) {  // Mostrar cada motor durante 2 segundos
              // Loop vacío para evitar bloqueos del sistema
            }

            lcd.clear();  // Limpiar la pantalla para mostrar el siguiente motor
          }

          verificacionCompleta = false;  // Reiniciar el estado para la próxima verificación
        }
       //
```

```
//FUNCION PARA LA EJECUTACION DE LA VERIFICACION Y CALIBRACION
void VYC() {

  stopBuzzer(); //detiene el buzzer una vez ejecutada la operacion
  static unsigned long startTime = 0;

  if (millis() - startTime >= 1000) {  // Muestra el mensaje durante 1 segundo
    lcd.clear();
    lcd.setCursor(0, 0);
    lcd.print("Calibrando...");
    startTime = millis();  // Actualiza el tiempo para que el mensaje dure 1 segundo
  }

  //verificacion ycalibracion
  uint8_t switchStates = ~LIMIT_SWITCH_PIN;
  int verif_speed=100;
  String replace_L=String('L');
  String replace_R=String('R');



  char replace_direction = '\0';
  char sentido_retraccion ='\0';

  switch (direccion) {
   case 'L':

    replace_direction='L';
    sentido_retraccion='R';
    break;

   case 'R':
    replace_direction='R';
    sentido_retraccion='L';
    break;

    //// LLAVE FINAL SWITCH
   default:
   Serial.println("Direccion invalida");
   return;
```

```
  }

  giroSTR(replace_direction, sentido_retraccion);    //EHECUTA PARA REALIZAR LA PRIMERA VERIFICACION
  char temp = replace_direction;                    //INVERTIMOS EL SENTIDO
  replace_direction = sentido_retraccion;
  sentido_retraccion = temp;

}



// FUNCION PARA VERIFICAR ESTADO DEL BUZZER
  void updateLCDForDebugging() {
     if (isBuzzerOn) {
        lcd.clear();
        lcd.setCursor(0, 0);
        lcd.print("Buzzer ON!");
        lcd.setCursor(0, 1);
        lcd.print("Time: ");
        lcd.print(millis() - buzzerStartTime);
     } else {
        lcd.clear();
        lcd.setCursor(0, 0);
        lcd.print("Buzzer OFF");
     }
  }
//




void loop() {
  //Lectura continua de la posicion del encoder
  long newPosition = myEnc.read();  // Lee la posición del encoder sin dividir



    // Control de inicio de PUNTOTOGO
    if (iniciarPTG && !PTGCompleta) {
       PUNTOTOGO();
    } else if (PTGCompleta && millis() - mostrarMensajeStartTime >= mostrarMensajeDuration) {
```

```
    // Volver a mostrar el menú principal u otras acciones después de mostrar el mensaje
    lcd.clear();
    iniciarPTG = false;
    PTGCompleta = false;
  }

  moverMotoresSincronizados(); // Ejecuta el movimiento y controla el mensaje sin bloquear

  if (mensajeCompletoMostrado && millis() - mensajeCompletoStartTime > 2000) { // 2 segundos de pausa
    mensajeCompletoMostrado = false;
    lcd.clear(); // Limpia el mensaje
    // Continuar al menú principal o lo que sea necesario
    updateMenu(); //
  }
  //


 // Ejecucion de la verificacion y calibracion
  if (iniciarVerificacion && !verificacionCompleta) {
    // Llama a la función de verificación de switches si no está completa
    giroSTR('L', 'R');  // Usamos la dirección inicial para la verificación
  } else if (verificacionCompleta) {
    // Si la verificación está completa, mostramos los pasos
    mostrarPasosMotores();

    // Una vez que se muestran los pasos, desactivamos la verificación
    iniciarVerificacion = false;  // Reinicia la bandera para la próxima ejecución
  }
 //


 // Ejecucion de origenes
  if (iniciarIrAlOrigen && !IrAlOrigenCompleta) {
    //LLama a la funcion de "Ir al Origen"
    IrAlOrigen();
  } else if (IrAlOrigenCompleta) {
    iniciarIrAlOrigen = false; //Reiniciamos la bandera
  }

  // Lógica no bloqueante para mostrar el mensaje "Origen Alcanzado" durante 2 segundos
```

```
  if (origenAlcanzadoMostrar && millis() - origenAlcanzadoStartTime >= 2000) {
    lcd.clear();  // Limpiar el mensaje después de 2 segundos
    origenAlcanzadoMostrar = false;  // Desactivar la visualización del mensaje
    IrAlOrigenCompleta = true;  // Marcar la función como completada
  }
 //


 // Ejecucion de recalibracion
 if (iniciarRecalibracionAbsoluta && !recalibracionAbsolutaCompleta) {
  //LLama a la funcion de "Ir al Origen"
  RecalibrarAbsoluto();
 } else if (recalibracionAbsolutaCompleta) {
  iniciarRecalibracionAbsoluta = false; //Reiniciamos la bandera
 }

 // Lógica no bloqueante para mostrar el mensaje "Origen Alcanzado" durante 2 segundos
  if (recalibracionAbsolutaMostrar && millis() - recalibracionAbsolutaStartTime >= 2000) {
    lcd.clear();  // Limpiar el mensaje después de 2 segundos
    recalibracionAbsolutaMostrar = false;  // Desactivar la visualización del mensaje
    recalibracionAbsolutaCompleta = true;  // Marcar la función como completada
  }
 //

//


// Controlar el zumbido breve del buzzer de forma no bloqueante
 if (isBuzzerOn && millis() - buzzerStartTime >= buzzerDuration) {
    digitalWrite(BUZZER, HIGH);  // Apagar el buzzer después de que haya pasado la duración
    isBuzzerOn = false;
 }
//

// Zumbido periódico durante la emergencia

//
```

```
              // Detecta cambios en la posición del encoder
              if (newPosition != oldPosition) {

                // DETECCION DE CAMBIOS EN EL DISPLAY LCD DEBIDO A LA INTERACCION DEL ENCODER
                if (abs(newPosition - oldPosition) >= 2) { // Detectar cambios de al menos 2 pasos

                  // Si estamos en el MENU "MENU PRINCIPAL"
                  if (currentMenuState == MAIN_MENU) {
                    menuIndex = (menuIndex + (newPosition > oldPosition ? 1 : -1) + 9) % 9;  // Mueve el índice circularmente
                  }
                  //

                  // Si estamos en el SUBMENU "HOME"
                  else if (currentMenuState == SUB_MENU_HOME) {
                    homeMenuIndex = (homeMenuIndex + (newPosition > oldPosition ? 1 : -1) + 3) % 3;  // Mueve el índice
circularmente
                  }
                  //

                  // Si estamos en el SUBMENU "PUNTOTOGO"
                  else if (currentMenuState == SUB_MENU_PUNTOTOGO) {
                    if (!editing) {
                      puntoToGoIndex = (puntoToGoIndex + (newPosition > oldPosition ? 1 : -1) + 5) % 5;  // Mueve el índice
circularmente
                    } else {
                      // Si estamos editando, ajusta el valor de la variable seleccionada
                      adjustVariable(newPosition);
                    }
                  }
                  //

                  // Si estamos en el SUBMENU "MODELO DE MOVIMIENTO"
                  else if (currentMenuState == SUB_MENU_MODELO) {
                    modeloMovimientoIndex = (modeloMovimientoIndex + (newPosition > oldPosition ? 1 : -1) + 3) % 3;  // Mueve el
índice circularmente
                  }


                  // Si estamos en el SUB-SUBMENU de confirmación de cambio de modelo
                  else if (currentMenuState == CONFIRM_MODELO) {
```

```
                confirmChange = !confirmChange; // Alternar entre SI / NO
              }
             //
            //

          // Si estamos en el SUBMENU "AJUSTE DE PARAMETROS"
           else if (currentMenuState == SUB_MENU_AJUSTE_PARAMETROS) {
            if (!editingParametros) {
             ajusteParametrosIndex = (ajusteParametrosIndex + (newPosition > oldPosition ? 1 : -1) + 5) % 5;  // Mueve el
índice circularmente
            } else {
             // Ajusta el valor de la variable seleccionada
             adjustParametros(newPosition);
            }
           }
          //

          // Si estamos en el SUBMENU "SET Y CALIBRACION"
           else if (currentMenuState == SUB_MENU_SETCALIBRACION) {
            setCalibracionIndex = (setCalibracionIndex + (newPosition > oldPosition ? 1 : -1) + 3) % 3;  // Mueve el índice
circularmente
            }

           // Si estamos en el SUB-SUBMENU "CAMBIAR DIRECCION"
            else if (currentMenuState == SUB_MENU_CAMBIAR_DIRECCION) {
            cambiarDireccionIndex = (cambiarDireccionIndex + (newPosition > oldPosition ? 1 : -1) + 3) % 3;  // Mueve el
índice circularmente
            }
           //
          //

          // Si estamos en el SUBEMNU "SET ORIGEN"
            else if(currentMenuState == SUB_MENU_SET_ORIGEN){
            setOrigenIndex = (setOrigenIndex + (newPosition > oldPosition ? 1 : -1) + 4) % 4; //Mueve el indice circularmente
            }
          //


          // Si estamos en el SUBMENU "MODO MANUAL"
            else if(currentMenuState == SUB_MENU_MODO_MANUAL){
```

```
                modoManualIndex = (modoManualIndex + (newPosition > oldPosition ? 1 : -1) + 3) % 3; //Mueve el indice
circularmente
                }

                // Si estamos en el SUB-SUBMENU "ESTADO MOTORES"
                 else if(currentMenuState == SUB_MENU_ESTADO_MOTORES){
                  modoManualIndex = (modoManualIndex + (newPosition > oldPosition ? 1 : -1) + 3) % 3; //Mueve el indice
circularmente
                 }
                //

               // Si estamos en el SUB-SUBMENU "MOVIMIENTO INDIVIDUAL"
                 else if(currentMenuState == SUB_MENU_MOV_INDIVIDUAL){
                 movIndividualIndex = (movIndividualIndex + (newPosition > oldPosition ? 1 : -1) + 5) % 5; //Mueve el indice
circularmente
                 }

                 // Si estamos en M1
                  else if(currentMenuState == SUB_MENU_M1){
                   if (!editingM1) {
                   motor1Index = (motor1Index + (newPosition > oldPosition ? 1 : -1) + 4) % 4;  // Mueve el índice circularmente
                   } else {
                   // Si estamos editando, ajusta el valor de la variable seleccionada
                   adjustVariableMOTOR(newPosition);
                   }
                  }
                 //

                 // Si estamos en M2
                  else if(currentMenuState == SUB_MENU_M2){
                   if (!editingM2) {
                   motor2Index = (motor2Index + (newPosition > oldPosition ? 1 : -1) + 4) % 4;  // Mueve el índice circularmente
                   } else {
                   // Si estamos editando, ajusta el valor de la variable seleccionada
                   adjustVariableMOTOR(newPosition);
                   }
                  }
                 //

                 // Si estamos en M3
                  else if(currentMenuState == SUB_MENU_M3){
```

```
                    if (!editingM3) {
                    motor3Index = (motor3Index + (newPosition > oldPosition ? 1 : -1) + 4) % 4;  // Mueve el índice circularmente
                    } else {
                    // Si estamos editando, ajusta el valor de la variable seleccionada
                    adjustVariableMOTOR(newPosition);
                    }
                   }
                  //

                  // Si estamos en M4
                   else if(currentMenuState == SUB_MENU_M4){
                    if (!editingM4) {
                    motor4Index = (motor1Index + (newPosition > oldPosition ? 1 : -1) + 4) % 4;  // Mueve el índice circularmente
                    } else {
                    // Si estamos editando, ajusta el valor de la variable seleccionada
                    adjustVariableMOTOR(newPosition);
                    }
                   }
                  //

                 //

                //

               // Si estamos en el SUBEMNU "ESTABLECER RUTA"
                 else if(currentMenuState == SUB_MENU_ESTABLECER_RUTA){
                  rutaMenuIndex = (rutaMenuIndex + (newPosition > oldPosition ? 1 : -1) + 3) % 3; //Mueve el indice circularmente
                 }
               //

               // Si estamos en el SUBEMNU "EJECUTAR MOVIMIENTO PARALELO"
                 else if(currentMenuState == SUB_MENU_MOV_PARALELO){
                  movParaleloIndex = (movParaleloIndex + (newPosition > oldPosition ? 1 : -1) + 3) % 3; //Mueve el indice
circularmente
                 }
               //


               //ACTUALIZACION CONTINUA Y LECTURA DEL ENCODER Y EL DISPLAY
                 oldPosition = newPosition; // Actualiza la posición anterior
```

```
      updateMenu();  // Actualiza la pantalla solo cuando hay un cambio
     //

   }
   //

}
//

// Detección del botón del encoder con DELAY --> sin bloquear
  if (!digitalRead(SwitchEncPin) && (millis() - lastDebounceTime) > debounceDelay) {
    lastDebounceTime = millis(); // Actualizar tiempo de debounce
    // Maneja la selección según el estado actual del menú
    if (currentMenuState == MAIN_MENU) {
      handleMainMenuSelection();

    } else if (currentMenuState == SUB_MENU_HOME) {
      handleHomeSelection();

    } else if (currentMenuState == SUB_MENU_PUNTOTOGO) {
      handlePuntoToGoSelection();

    } else if (currentMenuState == SUB_MENU_MODELO) {
      handleModeloSelection();
       } else if (currentMenuState == CONFIRM_MODELO) {
       handleConfirmModeloSelection();

    } else if (currentMenuState == SUB_MENU_SETCALIBRACION) {
      handleSetCalibracionSelection();
      } else if (currentMenuState == SUB_MENU_CAMBIAR_DIRECCION) {
      handleCambiarDireccionSelection();

    } else if (currentMenuState == SUB_MENU_AJUSTE_PARAMETROS) {
      handleAjusteParametrosSelection();

    } else if (currentMenuState == SUB_MENU_MODO_MANUAL) {
      handleModoManualSelection();
       } else if (currentMenuState == SUB_MENU_ESTADO_MOTORES) {
       handleEstadoMotoresSelection();
```

```
    } else if (currentMenuState == SUB_MENU_MOV_INDIVIDUAL) {
     handleMovimientoMotoresSelection();

      } else if (currentMenuState == SUB_MENU_M1) {
       handleM1Selection();
      } else if (currentMenuState == SUB_MENU_M2) {
       handleM2Selection();
      } else if (currentMenuState == SUB_MENU_M3) {
       handleM3Selection();
      } else if (currentMenuState == SUB_MENU_M4) {
       handleM4Selection();

   } else if (currentMenuState == SUB_MENU_SET_ORIGEN) {
     handleSetOrigenSelection();

   } else if (currentMenuState == SUB_MENU_MOV_PARALELO) {
     handleMovParaleloSelection();

   } else if (currentMenuState == SUB_MENU_ESTABLECER_RUTA) {
     handleEstablecerRutaSelection();


   }
  }
//


// Actualizar el desplazamiento del texto de forma continua
  updateScroll();
//

//CONTROLADORES NO BLOQUIANTES
  // Controlar el zumbido breve del buzzer de forma no bloqueante
    if (isBuzzerOn && millis() - buzzerStartTime >= buzzerDuration) {
        digitalWrite(BUZZER, HIGH);  // Apagar el buzzer después de que haya pasado la duración
        isBuzzerOn = false;
    }
  //
```

```
  // Controlar la ejecución de PUNTOTOGO de forma no bloqueante
  if (isPuntoToGoRunning && millis() >= puntoToGoEndTime) {
    isPuntoToGoRunning = false;
    currentMenuState = MAIN_MENU; // Vuelve al menú principal después de PUNTOTOGO
    updateMenu();
  }
 //

} //FIN DEL VOID LOOP ----------------------------------------------------------------------------



// Función para ajustar las variables X, Y, Z
void adjustVariable(long newPosition) {

 // Invertir la dirección
  int increment = (newPosition < oldPosition) ? 1 : -1; // Invertimos el sentido de ajuste
 //

 // Calcular la velocidad del giro del encoder
  unsigned long currentTime = millis();
  unsigned long timeDiff = currentTime - lastEncoderChangeTime;
 //

 // Ajuste de escala de acuerdo a la velocidad del giro
  int scale = 1;
  if (timeDiff < 100) { // Si el tiempo entre cambios es menor a 100 ms
    scale = 5; // Incremento más rápido
  } else if (timeDiff < 200) { // Si es menor a 200 ms
    scale = 3; // Incremento medio
  }
 //

 // Ajusta el valor según la escala
 if (puntoToGoIndex == 0) { // Editando X
  X = constrain(X + (increment * scale), minValXPTG, maxValXPTG);
 } else if (puntoToGoIndex == 1) { // Editando Y
```

```
    Y = constrain(Y + (increment * scale), minValYPTG, maxValYPTG);
  } else if (puntoToGoIndex == 2) { // Editando Z
    Z = constrain(Z + (increment * scale), minValZPTG, maxValZPTG);
  }
  //

  oldPosition = newPosition; // Actualiza la posición anterior
  lastEncoderChangeTime = currentTime; // Actualiza el tiempo del cambio
  updateMenu();  // Actualiza la pantalla para mostrar el valor ajustado
}



// Funcion para ajustar movimiento de motores individualmente
 void adjustVariableMOTOR(long newPosition) {

   // Invertir la dirección
   int increment = (newPosition < oldPosition) ? 1 : -1; // Invertimos el sentido de ajuste
   //

   // Calcular la velocidad del giro del encoder
    unsigned long currentTime = millis();
    unsigned long timeDiff = currentTime - lastEncoderChangeTime;
  //

   // Ajuste de escala de acuerdo a la velocidad del giro
    int scale = 1;
    if (timeDiff < 100) { // Si el tiempo entre cambios es menor a 100 ms
      scale = 5; // Incremento más rápido
    } else if (timeDiff < 200) { // Si es menor a 200 ms
      scale = 3; // Incremento medio
    }
   //

   switch (currentMenuState) {
      case SUB_MENU_M1:
         if (motor1Index == 0) {  // Editar Ángulo para M1
            ang_m1 = constrain(ang_m1 + (increment * scale), minValANG1, maxValANG1);
         } else if (motor1Index == 1) {  // Editar Pasos para M1
            pasos_m1 = constrain(pasos_m1 + (increment * scale), minValPAS1, maxValPAS1);
```

```
            }
            break;
        case SUB_MENU_M2:
            if (motor2Index == 0) {  // Editar Ángulo para M1
                ang_m2 = constrain(ang_m2 + (increment * scale), minValANG2, maxValANG2);
            } else if (motor1Index == 1) {  // Editar Pasos para M1
                pasos_m2 = constrain(pasos_m2 + (increment * scale), minValPAS2, maxValPAS2);
            }
            break;
        case SUB_MENU_M3:
            if (motor3Index == 0) {  // Editar Ángulo para M1
                ang_m3 = constrain(ang_m3 + (increment * scale), minValANG3, maxValANG3);
            } else if (motor1Index == 1) {  // Editar Pasos para M1
                pasos_m3 = constrain(pasos_m3 + (increment * scale), minValPAS3, maxValPAS3);
            }
            break;
        case SUB_MENU_M4:
            if (motor4Index == 0) {  // Editar Ángulo para M1
                ang_m4 = constrain(ang_m4 + (increment * scale), minValANG4, maxValANG4);
            } else if (motor1Index == 1) {  // Editar Pasos para M1
                pasos_m4 = constrain(pasos_m4 + (increment * scale), minValPAS4, maxValPAS4);
            }
            break;
    }


    oldPosition = newPosition; // Actualiza la posición anterior
    lastEncoderChangeTime = currentTime; // Actualiza el tiempo del cambio
    updateMenu();  // Actualiza la pantalla para mostrar el valor ajustado
  }

//




// Función para ajustar los parámetros de "Ajuste de Parametros"
void adjustParametros(long newPosition) {
```

```
  int increment = (newPosition < oldPosition) ? 1 : -1; // Ajustar la dirección

  // Ajustar las variables seleccionadas
    if (ajusteParametrosIndex == 0) { // Editando Velocidad
      velocidad = constrain(velocidad + increment, 0, 100);
    } else if (ajusteParametrosIndex == 1) { // Editando Aceleracion
      aceleracion = constrain(aceleracion + increment, 0, 100);
    } else if (ajusteParametrosIndex == 2) { // Seleccionando Curva de Aceleracion
      curvaAceleracionIndex = (curvaAceleracionIndex + (increment > 0 ? 1 : -1) + 4) % 4; // Cambiar entre 4 tipos
    } else if (ajusteParametrosIndex == 3) { // Seleccionando Inicio Set y Calibracion
      inicioSetCalibracion = (inicioSetCalibracion == "LEFT") ? "RIGHT" : "LEFT"; // Alternar entre LEFT y RIGHT
    }
  //

  oldPosition = newPosition; // Actualiza la posición anterior
  updateMenu();  // Actualiza la pantalla para mostrar el valor ajustado
}

// Función para desplazar texto en el LCD de manera continua sin bloquear
void scrollText(int row, String message, int delayTime = 300) {

  int len = message.length();
  static unsigned long initialPauseTime = 0; // Tiempo de pausa inicial
  static bool initialPauseDone = false; // Estado de la pausa inicial

  // Si el mensaje es menor o igual al ancho de la pantalla, mostrarlo directamente
    if (len <= 16) {
      lcd.setCursor(0, row);
      lcd.print("> " + message); // Mostrar con el símbolo ">"
      isScrolling = false; // No se requiere desplazamiento
      initialPauseDone = false; // Resetear el estado de pausa inicial
      return; // Salir de la función
    }
  //

  // Establecer el texto actual que se está desplazando
  currentScrollingText = message;
  isScrolling = true;
```

```
 // Pausa inicial antes de comenzar el desplazamiento
  if (!initialPauseDone) {
    if (millis() - initialPauseTime < 1000) { // Pausa inicial de 1000 ms (1 segundo)
      lcd.setCursor(0, row);
      lcd.print("> " + message.substring(0, 16)); // Mostrar la primera parte del texto
      return; // Salir de la función hasta que la pausa inicial termine
    } else {
      initialPauseDone = true; // Pausa inicial completada
      initialPauseTime = millis(); // Reiniciar el tiempo para el desplazamiento normal
    }
  }
 //

 // Si ha pasado el tiempo de delayTime, actualizar el desplazamiento
 if (millis() - lastScrollTime >= delayTime) {
  lastScrollTime = millis(); // Actualizar el tiempo de la última actualización

  // Si el desplazamiento ha llegado al final del texto, reiniciar
    if (scrollIndex > len - 15) { // Desplazamiento hasta el final menos el ancho de la pantalla
      scrollIndex = 0;
      initialPauseDone = false; // Reiniciar el estado de la pausa inicial para la próxima vez
      initialPauseTime = millis(); // Establecer el tiempo de inicio de la pausa inicial
      return; // Pausa antes de reiniciar
    }
  //

  // Mostrar la parte del mensaje correspondiente
  lcd.setCursor(0, row);
  lcd.print("> " + currentScrollingText.substring(scrollIndex, scrollIndex + 16));
  scrollIndex++; // Incrementar el índice de desplazamiento
 }
}

// Función para actualizar el desplazamiento de texto
void updateScroll() {
 if (isScrolling) {
  scrollText(1, currentScrollingText); // Actualiza el texto desplazándose
 }
```

```
}

// Función para actualizar el menú en el display
void updateMenu() {

 lcd.clear();
 lcd.setCursor(0, 0);

 //MENU PRINCIPAL
  if (currentMenuState == MAIN_MENU) {
    lcd.setCursor(0, 0);
    lcd.print("Menu:");
    scrollIndex = 0; // Reiniciar el índice de desplazamiento
    scrollText(1, mainMenu[menuIndex]); // Iniciar desplazamiento si es necesario
 //

 //SUBMENU HOME
  } else if (currentMenuState == SUB_MENU_HOME) {
    scrollText(0, "HOME:");
    scrollText(1, homeMenu[homeMenuIndex]);
 //

 //SUBMENU PUNTO TO GO
  } else if (currentMenuState == SUB_MENU_PUNTOTOGO) {
    scrollText(0, "PuntoToGo:");
    scrollIndex = 0; // Reiniciar el índice de desplazamiento

    if (!editing) {
     scrollText(1, puntoToGoMenu[puntoToGoIndex]); // Iniciar desplazamiento si es necesario
    } else {
     switch (puntoToGoIndex) {
       case 0:
         scrollText(1, "X = " + String(X));
         break;
       case 1:
         scrollText(1, "Y = " + String(Y));
         break;
       case 2:
         scrollText(1, "Z = " + String(Z));
```

```
        break;
      }
    }
//

// SUBMENU MODELO MOVIMIENTO
   } else if (currentMenuState == SUB_MENU_MODELO) {
     scrollText(0, "Modelo de Mov:");
     scrollIndex = 0; // Reiniciar el índice de desplazamiento
     scrollText(1, modeloMovimientoMenu[modeloMovimientoIndex]); // Iniciar desplazamiento si es necesario



   // SUB-SUBMENU CONFIRMAR MODELO
     } else if (currentMenuState == CONFIRM_MODELO) {
       scrollText(0, "Cambiar a:");
       scrollText(1, confirmChange ? "SI" : "NO"); // Mostrar "SI" o "NO"
   //
//

// SUBMENU SET Y CALIBRACION
  } else if (currentMenuState == SUB_MENU_SETCALIBRACION) {
     scrollText(0, "Set y Calib:");
     scrollText(1, setCalibracionMenu[setCalibracionIndex]);



 // SUB-SUBMENU CAMBIAR DIRECCION
   } else if (currentMenuState == SUB_MENU_CAMBIAR_DIRECCION) {
       scrollText(0, "Cambiar Direcc:");
       scrollText(1, cambiarDireccionMenu[cambiarDireccionIndex]);
 //
//

//SUBMENU AJUSTE DE PARAMETROS
  } else if (currentMenuState == SUB_MENU_AJUSTE_PARAMETROS) {
    scrollText(0, "Ajuste Param:");
    scrollIndex = 0; // Reiniciar el índice de desplazamiento
    if (!editingParametros) {
      scrollText(1, ajusteParametrosMenu[ajusteParametrosIndex]); // Muestra las opciones
    } else {
```

```
    // Mostrar los valores editables según la selección
    switch (ajusteParametrosIndex) {
     case 0:
      scrollText(1, "Velocidad: " + String(velocidad));
      break;
     case 1:
      scrollText(1, "Aceleracion: " + String(aceleracion));
      break;
     case 2:
      scrollText(1, "Curva: " + curvasAceleracion[curvaAceleracionIndex]);
      break;
     case 3:
      scrollText(1, "Inicio: " + inicioSetCalibracion);
      break;
    }
   }

//

//SUBMENU SET ORIGEN
 } else if (currentMenuState == SUB_MENU_SET_ORIGEN) {
  scrollText(0, "SetOrigen:");
  scrollText(1, setOrigenMenu[setOrigenIndex]);
  scrollIndex = 0; // Reiniciar el índice de desplazamiento
//

//SUBMENU MODO MANUAL
 } else if (currentMenuState == SUB_MENU_MODO_MANUAL) {
 scrollText(0, "Modo Manual:");
 scrollText(1, modoManualMenu[modoManualIndex]);
 scrollIndex = 0; // Reiniciar el índice de desplazamiento

  // SUB-SUBMENU ESTADO MOTORES
   } else if (currentMenuState == SUB_MENU_MODO_MANUAL) {
   scrollText(0, "Estado Motores:");
   scrollText(1, estadoMotoresMenu[estadoMotoresIndex]);
   scrollIndex = 0; // Reiniciar el índice de desplazamiento
  //
```

```
// SUB-SUBMENU MOVIMIENTO INDIVIDUAL
 } else if (currentMenuState == SUB_MENU_MOV_INDIVIDUAL) {
 scrollText(0, "Mov Ind:");
 scrollText(1, movIndividualMenu[movIndividualIndex]);
 scrollIndex = 0; // Reiniciar el índice de desplazamiento
//

 // SUB-SUB-SUBMENU MOTOR 1
  } else if (currentMenuState == SUB_MENU_M1) {
  scrollText(0, "M1:");
   if (!editingM1) {
   scrollText(1, motor1Menu[motor1Index]); // Iniciar desplazamiento si es necesario
   } else {
    switch (motor1Index) {
      case 0:
       scrollText(1, "ANG = " + String(ang_m1));
       break;
      case 1:
       scrollText(1, "PAS = " + String(pasos_m1));
       break;
    }
   }

 //

 // SUB-SUB-SUBMENU MOTOR 2
  } else if (currentMenuState == SUB_MENU_M2) {
  scrollText(0, "M2:");
   if (!editingM2) {
   scrollText(1, motor2Menu[motor2Index]); // Iniciar desplazamiento si es necesario
   } else {
    switch (motor2Index) {
      case 0:
       scrollText(1, "ANG = " + String(ang_m2));
       break;
      case 1:
       scrollText(1, "PAS = " + String(pasos_m2));
       break;
```

```
      }
     }

    // SUB-SUB-SUBMENU MOTOR 3
     } else if (currentMenuState == SUB_MENU_M3) {
     scrollText(0, "M3:");
      if (!editingM3) {
      scrollText(1, motor3Menu[motor3Index]); // Iniciar desplazamiento si es necesario
      } else {
        switch (motor3Index) {
          case 0:
            scrollText(1, "ANG = " + String(ang_m3));
            break;
          case 1:
            scrollText(1, "PAS = " + String(pasos_m3));
            break;
        }
      }

     // SUB-SUB-SUBMENU MOTOR 4
     } else if (currentMenuState == SUB_MENU_M4) {
     scrollText(0, "M4:");
      if (!editingM4) {
      scrollText(1, motor4Menu[motor4Index]); // Iniciar desplazamiento si es necesario
      } else {
        switch (motor4Index) {
          case 0:
            scrollText(1, "ANG = " + String(ang_m4));
            break;
          case 1:
            scrollText(1, "PAS = " + String(pasos_m4));
            break;
        }
      }
     }

//

//SUBMENU ESTABLECER MOV PARALELO
```

```
  //

  //SUBMENU ESTABLECER RUTAS




  //

}


// FUNCIONES HANDLE PARA MANEJO DE LOS MENUS / SUBMENUS / SUB-SUB.....


  // FUNCION MANEJO DEL MENU PRINCIPAL
  void handleMainMenuSelection() {
    playBuzzer();
     // Entra al SUBMENU "HOME"
      if (menuIndex == 0) {
        currentMenuState = SUB_MENU_HOME;
        homeMenuIndex = 0;
        updateMenu();
     //

     //Entra al SUBMENU "PuntoToGo"
      } else if (menuIndex == 1) {
        currentMenuState = SUB_MENU_PUNTOTOGO;
        puntoToGoIndex = 0;
        updateMenu();
     //

     // Entra al SUBMENU "Set y Calibracion"
      } else if (menuIndex == 2) { // Añadido para "Set y Calibracion"
        currentMenuState = SUB_MENU_SETCALIBRACION;
        setCalibracionIndex = 0;
        updateMenu();
     //

     //Entra al SUBMENU "Ajuste de Parametros"
```

```
  } else if (menuIndex == 3) {
    currentMenuState = SUB_MENU_AJUSTE_PARAMETROS;
    ajusteParametrosIndex = 0;
    updateMenu();
  //

  //Entra al SUBMENU "Modelo de Movimiento"
  } else if (menuIndex == 4) {
    currentMenuState = SUB_MENU_MODELO;
    modeloMovimientoIndex = 0;
   updateMenu();
  //

  //Entra al SUBMENU "MODO MANUAL"
  } else if (menuIndex == 5) {
    currentMenuState = SUB_MENU_MODO_MANUAL;
    modoManualIndex = 0;
   updateMenu();
  //

  //Entra al SUBMENU "SET ORIGEN"
  } else if (menuIndex == 6) {
    currentMenuState = SUB_MENU_SET_ORIGEN;
    setOrigenIndex = 0;
   updateMenu();
  //

  //Entra al SUBMENU "MOV PARALELO"
  } else if (menuIndex == 7) {
    currentMenuState = SUB_MENU_MOV_PARALELO;
    movParaleloIndex = 0;
   updateMenu();
  //

  //Entra al SUBMENU "ESTABLECER RUTA"
  } else if (menuIndex == 8) {
    currentMenuState = SUB_MENU_ESTABLECER_RUTA;
    rutaMenuIndex = 0;
   updateMenu();
```

```
    //


  }
}




// Función para manejar la selección en el SUBMENU "HOME"
void handleHomeSelection() {
  playBuzzer(); // Reproduce sonido al seleccionar
  if (homeMenuIndex == 0) { // SI seleccionado
    HOMMING(); // Llamar a la función HOMMING
    currentMenuState = MAIN_MENU; // Vuelve al menú principal después de HOMMING
  } else if (homeMenuIndex == 2) { // VOLVER ATRAS seleccionado
    currentMenuState = MAIN_MENU;
  }
  updateMenu();
}

// Función para manejar la selección en el SUBMENU "PuntoToGo"
void handlePuntoToGoSelection() {
  playBuzzer(); // Reproduce sonido al seleccionar
  if (puntoToGoIndex == 3) {
    // Si se selecciona "Confirmar", se ejecuta PUNTOTOGO
      iniciarPTG = true;
      PTGCompleta = false;  // Reinicia para comenzar el proceso
  } else if (puntoToGoIndex == 4) {
    // Si se selecciona "Volver Atras", vuelve al menú principal
    currentMenuState = MAIN_MENU;
    updateMenu();
  } else {
    // Entra en modo de edición para la variable seleccionada
    editing = !editing; // Cambia el estado de edición
    updateMenu();
  }
}

// Función para manejar la selección en el SUBMENU "Modelo de Movimiento"
void handleModeloSelection() {
```

```
  playBuzzer(); // Reproduce sonido al seleccionar
  if (modeloMovimientoIndex == 2) {
    // Si se selecciona "Volver Atras", vuelve al menú principal
    currentMenuState = MAIN_MENU;
    updateMenu();
  } else {
    // Confirma el cambio de modelo
    currentMenuState = CONFIRM_MODELO;
    confirmChange = false; // Empieza con la opción en "NO"
    updateMenu();
  }
}

// Función para manejar la seleccion en el SUB-SUBMENU "Cambio de modelo"
void handleConfirmModeloSelection() {
  if (confirmChange) {
    // Cambia la variable MODELO
    MODELO = (modeloMovimientoIndex == 0) ? 0 : 1; // 0 = Cinematica Directa, 1 = Cinematica Inversa
  }
  // Regresa al menú "Modelo de Movimiento"
  currentMenuState = SUB_MENU_MODELO;
  updateMenu();
}

// Función para manejar la selección en el SUBMENU "Set y Calibracion"
  void handleSetCalibracionSelection() {
      playBuzzer(); // Reproduce sonido al seleccionar
      if (setCalibracionIndex == 0) { // CAMBIAR DIRECCION INICIO
          // Cambiar la dirección de inicio
          currentMenuState = SUB_MENU_CAMBIAR_DIRECCION;
          updateMenu();
      } else if (setCalibracionIndex == 1) { // EJECUTAR
          // Ejecuta la función VyC
          stopBuzzer();
          digitalWrite(BUZZER, HIGH);  // Apagar el buzzer
          iniciarVerificacion = true;
          currentMenuState = MAIN_MENU; // Vuelve al menú principal después de ejecutar
          updateMenu();
      } else if (setCalibracionIndex == 2) { // VOLVER ATRAS
```

```
        // Vuelve al menú principal
        currentMenuState = MAIN_MENU;
        updateMenu();
      }
    }


   // Función para manejar la selección en el SUB-SUBMENU "Cambiar Direccion"
    void handleCambiarDireccionSelection() {
      playBuzzer(); // Reproduce sonido al seleccionar
      if (cambiarDireccionIndex == 0) { // LEFT
        direccion = 'L';
      } else if (cambiarDireccionIndex == 1) { // RIGHT
        direccion = 'R';
      } else if (cambiarDireccionIndex == 2) { // VOLVER ATRAS
        // Volver al menú "Set y Calibracion"
        currentMenuState = SUB_MENU_SETCALIBRACION;
        updateMenu();
        return; // Salir de la función para evitar cambiar de menú al seleccionar
      }

      // Después de seleccionar una opción (LEFT o RIGHT), volver al menú "Set y Calibracion"
      currentMenuState = SUB_MENU_SETCALIBRACION;
      updateMenu();
    }
   //
  //

  // Función para manejar la selección en el SUBMENU "Ajuste de Parametros"
  void handleAjusteParametrosSelection() {
    playBuzzer(); // Reproduce sonido al seleccionar
    if (ajusteParametrosIndex == 4) {
      // Si se selecciona "Volver Atras", vuelve al menú principal
      currentMenuState = MAIN_MENU;
      updateMenu();
    } else {
      // Entra en modo de edición para la variable seleccionada
      editingParametros = !editingParametros; // Cambia el estado de edición
      updateMenu();
```

```
 }
}

//Funcion para menjar la seleccion en el SUBMENU "Modo Manual"
 void handleModoManualSelection() {
  playBuzzer(); // Reproduce sonido al seleccionar
  if (modoManualIndex == 0) {
   currentMenuState = SUB_MENU_ESTADO_MOTORES;
   updateMenu();
  } else if (modoManualIndex == 1) {
   currentMenuState = SUB_MENU_MOV_INDIVIDUAL;
   updateMenu();
  } else {
  currentMenuState = MAIN_MENU;
  updateMenu();
  }
 }

 //Funcion para manejar la seleccion en el SUB-SUBMENU "ESTADO MOTORES"
  void handleEstadoMotoresSelection() {
   playBuzzer(); // Reproduce sonido al seleccionar



  }
 //

 //Funcion para manejar la seleccion en el SUB-SUBMENU "MOV MOTORES"
  void handleMovimientoMotoresSelection() {
   playBuzzer(); // Reproduce sonido al seleccionar
  if (movIndividualIndex == 0) {
   currentMenuState = SUB_MENU_M1;
   updateMenu();
  } else if (movIndividualIndex == 1) {
   currentMenuState = SUB_MENU_M2;
   updateMenu();
   } else if (movIndividualIndex == 2) {
   currentMenuState = SUB_MENU_M3;
   updateMenu();
   } else if (movIndividualIndex == 3) {
```

```
 currentMenuState = SUB_MENU_M4;
 updateMenu();
} else {
currentMenuState = SUB_MENU_MODO_MANUAL;
updateMenu();
}

}


// Funcion para manejar la seleccion en el SUB-SUB-SUBMENU "Motor 1"
 void handleM1Selection() {
  playBuzzer(); // Reproduce sonido al seleccionar
  if (motor1Index == 2) {
   // Si se selecciona "Confirmar", se ejecuta MOVER_MOTOR_INDIVIDUAL
   MOVER_MOTOR_INDIVIDUAL(0, ang_m1, pasos_m1); // Llama a la función para el motor 1
   editingM1 = false;
  } else if (motor1Index == 3) {
   // Si se selecciona "Volver Atras", vuelve al menú principal
   currentMenuState = SUB_MENU_MOV_INDIVIDUAL;
   updateMenu();
  } else {
   // Entra en modo de edición para la variable seleccionada
   editingM1 = !editingM1; // Cambia el estado de edición
   updateMenu();
  }


 }
//

// Funcion para manejar la seleccion en el SUB-SUB-SUBMENU "Motor 2"
 void handleM2Selection() {
  playBuzzer(); // Reproduce sonido al seleccionar
   if (motor2Index == 2) {
   // Si se selecciona "Confirmar", se ejecuta MOVER_MOTOR_INDIVIDUAL
   MOVER_MOTOR_INDIVIDUAL(1, ang_m2, pasos_m2); // Llama a la función para el motor 2
   editingM2 = false;
  } else if (motor2Index == 3) {
```

```
    // Si se selecciona "Volver Atras", vuelve al menú principal
    currentMenuState = SUB_MENU_MOV_INDIVIDUAL;
    updateMenu();
   } else {
    // Entra en modo de edición para la variable seleccionada
    editingM2 = !editingM2; // Cambia el estado de edición
    updateMenu();
   }


  }
//

// Funcion para manejar la seleccion en el SUB-SUB-SUBMENU "Motor 3"
 void handleM3Selection() {
  playBuzzer(); // Reproduce sonido al seleccionar
  if (motor3Index == 2) {
   // Si se selecciona "Confirmar", se ejecuta MOVER_MOTOR_INDIVIDUAL
   MOVER_MOTOR_INDIVIDUAL(2, ang_m3, pasos_m3); // Llama a la función para el motor 3
   editingM3 = false;
  } else if (motor3Index == 3) {
   // Si se selecciona "Volver Atras", vuelve al menú principal
   currentMenuState = SUB_MENU_MOV_INDIVIDUAL;
   updateMenu();
  } else {
   // Entra en modo de edición para la variable seleccionada
   editingM3 = !editingM3; // Cambia el estado de edición
   updateMenu();
  }


  }
//

// Funcion para manejar la seleccion en el SUB-SUB-SUBMENU "Motor 4"
 void handleM4Selection() {
  playBuzzer(); // Reproduce sonido al seleccionar
  if (motor4Index == 2) {
   // Si se selecciona "Confirmar", se ejecuta MOVER_MOTOR_INDIVIDUAL
   MOVER_MOTOR_INDIVIDUAL(3, ang_m4, pasos_m4); // Llama a la función para el motor 4
```

```
        editingM4 = false;
      } else if (motor4Index == 3) {
        // Si se selecciona "Volver Atras", vuelve al menú principal
        currentMenuState = SUB_MENU_MOV_INDIVIDUAL;
        updateMenu();
      } else {
        // Entra en modo de edición para la variable seleccionada
        editingM4 = !editingM4; // Cambia el estado de edición
        updateMenu();
      }

    }
   //
  //

//

void handleSetOrigenSelection() {
  playBuzzer(); // Reproduce sonido al seleccionar
  if (setOrigenIndex == 0) {
    //Si se selecciona "Ir al Origen"
    stopBuzzer();
    digitalWrite(BUZZER, HIGH);  // Apagar el buzzer
    iniciarIrAlOrigen = true;
    currentMenuState = MAIN_MENU; // Vuelve al menú principal después de ejecutar
    updateMenu();

  } else if (setOrigenIndex == 3) {
    // Si se selecciona "Volver Atras", vuelve al menú principal
    currentMenuState = MAIN_MENU;
    updateMenu();

    } else if (setOrigenIndex == 2) {
    stopBuzzer();
    digitalWrite(BUZZER, HIGH);  // Apagar el buzzer
    iniciarRecalibracionAbsoluta = true;
    currentMenuState = MAIN_MENU;
    updateMenu();
```

```
  } else {
    // Si se selecciona "Establecer nuevo origen"
    iniciarEstablecerOrigen = true; // Cambia el estado de la flag para iniciar
    currentMenuState = MAIN_MENU; // Vuelve al menú principal después de ejecutar
    updateMenu();
  }
 }

 void handleMovParaleloSelection() {
  playBuzzer(); // Reproduce sonido al seleccionar

 }

 void handleEstablecerRutaSelection() {
  playBuzzer(); // Reproduce sonido al seleccionar

 }

//


// FUNCIONES PARA TOCAR/BLOQUEAR UN ZUMBIDO BREVE SIN BLOQUEAR EL CÓDIGO
 unsigned long lastBuzzerTime = 0;
 const unsigned long buzzerCooldown = 500;  // 500 ms de espera antes de permitir otro sonido

 void playBuzzer() {
    if (!isBuzzerOn && millis() - lastBuzzerTime > buzzerCooldown) {
        digitalWrite(BUZZER, LOW);  // Encender el buzzer
        buzzerStartTime = millis();
        isBuzzerOn = true;
        lastBuzzerTime = millis();  // Actualiza el tiempo del último sonido
    }
 }

 void stopBuzzer() {
    if (isBuzzerOn && millis() - buzzerStartTime >= buzzerDuration) {
        digitalWrite(BUZZER, HIGH);  // Apagar el buzzer
        isBuzzerOn = false;
    }
```

```
  }
//

// Definición de la función HOMMING
void HOMMING() {
  // Aquí puedes agregar la lógica real para la función HOMMING
  lcd.clear();
  scrollText(0, "HOMMING:");
  scrollText(1, "Ejecutando...");
  delay(2000);  // Simula el tiempo que tarda la ejecución (2 segundos)
}









////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

bool verificarLimites(float X, float Y, float Z) {
  return (X >= minValXPTG && X <= maxValXPTG) &&
         (Y >= minValYPTG && Y <= maxValYPTG) &&
         (Z >= minValZPTG && Z <= maxValZPTG);
}

bool cinematicaInversa(float X, float Y, float Z, float &q1, float &q2, float &q3) {

  // Calcular q
  q2 = acos((X*X + Y*Y - L1*L1 - L2*L2) / (2 * L1 * L2));
  q1 = atan2(Y, X) - atan2(L2 * sin(q2), L1 + L2 * cos(q2));

  // Calcular q3 basado en Z (DESPLAZAMIENTO VERTICAL)
  q3 = Z - d1;

  return true;
}
```

```
// Configuración inicial de movimiento
void PUNTOTOGO() {
 float q1, q2, q3;

 if (!cinematicaInversa(X, Y, Z, q1, q2, q3)) {
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Punto Invalido");
     mostrarMensajeStartTime = millis();
     mostrarMensaje = true;
     enMovimiento = false;
     return;
 }

 // Convertir los ángulos a pasos
 long pasosMotor1 = q1 * relacionMOTOR1;
 long pasosMotor2 = q2 * relacionMOTOR2;
 long pasosMotor3 = q3 * relacionMOTOR3;

 // Calcular el máximo de pasos para determinar las velocidades proporcionales
 long maxPasos = max(abs(pasosMotor1), max(abs(pasosMotor2), abs(pasosMotor3)));

 // Configurar velocidades para que los motores terminen al mismo tiempo
 MOTORS[0].setMaxSpeed(500.0 * (float(abs(pasosMotor1)) / maxPasos));
 MOTORS[1].setMaxSpeed(500.0 * (float(abs(pasosMotor2)) / maxPasos));
 MOTORS[2].setMaxSpeed(500.0 * (float(abs(pasosMotor3)) / maxPasos));

 // Configurar aceleraciones
 MOTORS[0].setAcceleration(100);
 MOTORS[1].setAcceleration(100);
 MOTORS[2].setAcceleration(100);

 // Establecer los targets para cada motor
 MOTORS[0].moveTo(pasosMotor1);
 MOTORS[1].moveTo(pasosMotor2);
 MOTORS[2].moveTo(pasosMotor3);

 // Inicia el movimiento sincronizado
```

```
    enMovimiento = true;
    mostrarMensaje = false;

}

void moverMotoresSincronizados() {
  if (enMovimiento) {
    if (MOTORS[0].distanceToGo() == 0 && MOTORS[1].distanceToGo() == 0 && MOTORS[2].distanceToGo() == 0) {
      enMovimiento = false;
      mostrarMensaje = true;
      mostrarMensajeStartTime = millis();
      lcd.clear();
      lcd.setCursor(0, 0);
      lcd.print("Punto Alcanzado");
    } else {
      MOTORS[0].run();
      MOTORS[1].run();
      MOTORS[2].run();
    }
  } else if (mostrarMensaje && millis() - mostrarMensajeStartTime >= 2000) {
    mostrarMensaje = false;
    lcd.clear();
    currentMenuState = MAIN_MENU; // Vuelve al menú principal después del mensaje
    updateMenu(); // Vuelve al menú principal después del mensaje
  }
}



//
void MOVER_MOTOR_INDIVIDUAL(int motorIndex, int ang, int pasos) {
  AccelStepper &motor = MOTORS[motorIndex]; // Selecciona el motor correspondiente

  // Configurar velocidad y aceleración
  motor.setMaxSpeed(400);
  motor.setAcceleration(100);

  // Mover el motor a la posición calculada basada en el ángulo y pasos
  int targetSteps = ang + pasos;
  motor.moveTo(targetSteps);
```

```
    digitalWrite(ENAPIN1 + (motorIndex * 2), LOW);

    // Mover el motor de forma no bloqueante
    while (motor.distanceToGo() != 0) {
      motor.run();
    }

    // Detener y deshabilitar el motor una vez que alcanza la posición
    motor.stop();
    digitalWrite(ENAPIN1 + (motorIndex * 2), HIGH);
  }
  //

  void RecalibrarAbsoluto() {
   static int motorIndex = 0;
   static bool procesoCompleto = false;
   static bool motorEnMovimiento = false;
   static long pasosAjusteMotor[3] = {0, -1040, -1625}; // Ajustes individuales
   static int motoresOrden[3] = {1, 2, 0}; // Orden de ejecución: motor 2, motor 3, motor 1
   const int velocidad = 500; // Velocidad común para cada motor

   // Desenergizar motor 4 siempre
   digitalWrite(ENAPIN4, HIGH);

   // Si el proceso ya se completó, mostrar mensaje y resetear variables
   if (procesoCompleto) {
    lcd.clear();
    lcd.setCursor(0, 0);
    lcd.print("Ajuste Completo");
    recalibracionAbsolutaStartTime = millis();
    recalibracionAbsolutaCompleta = true;
    procesoCompleto = false;
    motorIndex = 0; // Reiniciar para futuras ejecuciones
    return;
   }

   // Mostrar mensaje de inicio solo al comenzar
   if (motorIndex == 0 && !motorEnMovimiento) {
    lcd.clear();
```

```
  scrollText(0, "Ajustando:");
  scrollText(1, "Posicion Absoluta");
 }

 int motorActual = motoresOrden[motorIndex];
 digitalWrite(ENAPIN1 + motorActual * 2, LOW); // Energizar el motor actual

 // Configurar velocidad y aceleración del motor si no está en movimiento
 if (!motorEnMovimiento) {
  MOTORS[motorActual].setMaxSpeed(velocidad);
  MOTORS[motorActual].setAcceleration(100);
  MOTORS[motorActual].move(pasosAjusteMotor[motorIndex]);
  motorEnMovimiento = true; // Indica que el motor está en movimiento
 }

 // Ejecutar movimiento
 if (MOTORS[motorActual].distanceToGo() != 0) {
  MOTORS[motorActual].run();
 } else {
  // El motor ha llegado al destino, desenergizar y pasar al siguiente
  digitalWrite(ENAPIN1 + motorActual * 2, HIGH); // Desenergizar el motor
  motorIndex++;
  motorEnMovimiento = false; // Preparado para el siguiente motor

  // Si completó todos los motores, marca el proceso como completo
  if (motorIndex >= 3) {
   procesoCompleto = true;
  }
 }
}


void IrAlOrigen() {
 static int motorIndex = 0;
 static unsigned long lastActionTime = 0;
 static bool retrayendo = false; // Indica si el motor está en retracción
 static int pasosRetraccion[3] = {90, 30, 50}; // Pasos de retracción para cada motor
 static int motoresOrden[3] = {1, 2, 0}; // Orden de los motores
 static int switchesOrigen[3] = {LIMMIN2, LIMMIN3, LIMMAX1}; // Pines de switches ajustados
```

```
static bool procesoCompleto = false;
static int speedsOrigen[3] = {400, 400, 300}; // Velocidades individuales para motor 2, motor 3, motor 1

// Desenergiza motor 4 siempre
digitalWrite(ENAPIN4, HIGH);

// Si el proceso ya terminó, no hace nada
if (procesoCompleto) {
 lcd.clear();
 lcd.setCursor(0, 0);
 lcd.print("Origen Alcanzado");
 origenAlcanzadoStartTime = millis();
 IrAlOrigenCompleta = true;
 procesoCompleto = false;
 motorIndex = 0; // Reiniciar para futuras ejecuciones
 return;
}

// Configuración inicial del motor si es la primera vez
if (motorIndex == 0 && !retrayendo) {
 lcd.clear();
 scrollText(0, "IR AL ORIGEN:");
 scrollText(1, "Ejecutando...");
}

int motorActual = motoresOrden[motorIndex];
int switchPin = switchesOrigen[motorIndex];

// Configuración de velocidad y aceleración individual
MOTORS[motorActual].setMaxSpeed(5000);
MOTORS[motorActual].setAcceleration(50);  // Aceleración común

// Comprobar si el switch está presionado
if (!retrayendo && digitalRead(switchPin) != HIGH) {
 MOTORS[motorActual].setSpeed(speedsOrigen[motorIndex]); // Ajustado para moverse hacia el switch
 MOTORS[motorActual].runSpeed();
} else if (!retrayendo) {
 // Switch presionado, iniciar retracción
 MOTORS[motorActual].stop();
```

```
    MOTORS[motorActual].move(-pasosRetraccion[motorIndex]); // Ajustado para invertir la dirección
    retrayendo = true;
   }

   // Ejecutar la retracción si es el caso
   if (retrayendo && MOTORS[motorActual].distanceToGo() != 0) {
    MOTORS[motorActual].run();
   } else if (retrayendo) {
    // Retracción completada, desactivar motor y pasar al siguiente
    digitalWrite(ENAPIN1 + motorActual * 2, HIGH);
    retrayendo = false;
    motorIndex++;

    // Verificar si todos los motores han terminado
    if (motorIndex >= 3) {
     procesoCompleto = true; // Termina el proceso si es el último motor
    }
   }
  }
```

8.2 Bloques Principales

El código está organizado en varios bloques principales que se encargan de gestionar aspectos específicos del funcionamiento del robot. Estos bloques incluyen:

- **Inicialización y Configuración**: Este bloque configura los pines de entrada/salida, inicializa la pantalla LCD y el codificador rotativo, y define los parámetros iniciales de los motores paso a paso. También se configuran las variables para gestionar la interfaz HMI y los modos de operación del robot.
- **Interfaz de Usuario (HMI)**: Este bloque gestiona el menú y las interacciones con el usuario a través del codificador rotativo y la pantalla LCD. Incluye funciones para navegar por el menú, seleccionar opciones y ajustar parámetros de operación. También se utilizan temporizadores y desplazamiento de texto para facilitar la lectura de mensajes largos.
- **Control de Motores**: En este bloque se utilizan funciones de la biblioteca **AccelStepper** para controlar el movimiento de los motores paso a paso. Cada motor tiene un control de velocidad y aceleración, lo que permite movimientos suaves y precisos. Las funciones implementadas permiten mover los motores de manera sincronizada o individual, dependiendo del modo de operación.
- **Manejo de Parada de Emergencia**: Este bloque se encarga de gestionar el botón de parada de emergencia, que desactiva los motores y muestra un mensaje de advertencia en la pantalla LCD. Esta función se ejecuta de manera no bloqueante mediante interrupciones, asegurando una respuesta rápida.

- **Funciones de Movimiento y Calibración**: Incluye las funciones **IrAlOrigen**, **RecalibrarAbsoluto** y **PUNTOTOGO**, las cuales se encargan de mover el robot a su posición de origen, realizar ajustes absolutos y llevar el robot a un punto específico en el espacio, respectivamente. Estas funciones utilizan cálculos de cinemática inversa y sincronización de motores para asegurar que los movimientos se realicen de manera controlada.

9. Mejoras a implementar a futuro

El proyecto SCARA actual es funcional y cumple con los objetivos de control y precisión establecidos. Sin embargo, hay varias áreas donde se pueden realizar mejoras para optimizar el rendimiento, la facilidad de uso y la versatilidad del sistema. A continuación, se detallan las mejoras propuestas en cuatro categorías: físicas, de control, electrónicas y de software.

9.1 Mejoras físicas

- **Optimización del Diseño Mecánico**: Se pueden revisar los eslabones y la estructura del robot para reducir el peso sin comprometer la resistencia. La implementación de piezas con perfiles más delgados, pero reforzadas en puntos estratégicos, podría mejorar la eficiencia energética y reducir el esfuerzo de los motores.
- **Uso de Materiales más Resistentes**: Para componentes críticos, como los soportes de los motores y el brazo, se podrían emplear materiales más duraderos y resistentes, como el aluminio o compuestos plásticos reforzados. Esto aumentaría la vida útil del robot y reduciría el desgaste en condiciones de uso intensivo.
- **Integración de un Sistema de Contrapeso**: Agregar contrapesos en el brazo del robot podría reducir el esfuerzo requerido por los motores, especialmente durante movimientos rápidos o de carga pesada. Esta mejora también contribuiría a reducir las vibraciones y mejorar la estabilidad general del sistema.

9.2 Mejoras de control

- **Control PID para Movimiento Suave**: La implementación de un sistema de control proporcional-integral-derivativo (PID) podría mejorar significativamente la precisión y la suavidad del movimiento de los motores. Este ajuste fino permitiría minimizar los errores de posicionamiento y reducir el tiempo de ajuste en los desplazamientos largos.
- **Sincronización Avanzada de Motores**: Mejorar la interpolación actual para que los motores mantengan velocidades proporcionales incluso en trayectorias complejas podría hacer los movimientos más fluidos y precisos. Se podrían

utilizar algoritmos de sincronización avanzados que ajusten dinámicamente la velocidad y aceleración de cada motor.

- **Implementación de Límites Virtuales Ajustables**: Los límites de movimiento actualmente están definidos físicamente. Implementar límites virtuales en el software permitiría al usuario ajustar los rangos de movimiento del robot sin necesidad de realizar cambios físicos. Esta flexibilidad sería útil para adaptarse a diferentes entornos de trabajo.

9.3 Mejoras de electronica

- **Incorporación de Sensores de Retroalimentación**: Agregar encoders en los motores permitiría obtener retroalimentación de posición en tiempo real, lo que contribuiría a mejorar la precisión y a detectar desviaciones o errores durante el movimiento. Con esta información, el robot podría corregir automáticamente su posición.
- **Optimización del Cableado y Uso de PCB Personalizadas**: Para simplificar el ensamblaje y mejorar la estética del proyecto, se podrían diseñar y utilizar placas de circuito impreso (PCB) personalizadas. Estas placas facilitarían la integración de los componentes electrónicos y reducirían el número de cables sueltos, disminuyendo el riesgo de desconexiones y mejorando la confiabilidad del sistema.
- **Mejora de la Fuente de Alimentación**: Una fuente de alimentación con capacidad de entregar mayor corriente o con distribución regulada a diferentes voltajes sería útil para optimizar el rendimiento de los motores y proteger los componentes electrónicos de fluctuaciones de energía. Además, se podrían incluir módulos de protección contra sobrecarga y cortocircuitos.

9.4 Mejoras de Software

- **Optimización de Código para Mayor Eficiencia**: La implementación de algoritmos de programación más eficientes y el uso de funciones no bloqueantes en todo el código podrían mejorar la capacidad de respuesta del robot. Por ejemplo, la optimización del manejo de interrupciones y el ajuste dinámico de parámetros permitirían un control más fluido.
- **Desarrollo de una Interfaz Gráfica de Usuario (GUI)**: Para una experiencia de usuario más intuitiva, se podría desarrollar una interfaz gráfica en una computadora o dispositivo móvil. Esto permitiría controlar el robot de forma remota, visualizar el estado en tiempo real y realizar ajustes a los parámetros de manera sencilla.
- **Implementación de un Sistema de Monitoreo en Tiempo Real**: Al integrar el robot con un sistema IoT, el usuario podría monitorear el rendimiento y estado del robot en tiempo real, incluso desde ubicaciones remotas. Esto sería útil para realizar mantenimiento preventivo y detectar anomalías en tiempo real.
- **Automatización de la Calibración y Ajustes Iniciales**: Automatizar los procesos de calibración y ajuste iniciales mediante rutinas programadas simplificaría el

uso del robot y reduciría el tiempo de configuración. La posibilidad de almacenar los ajustes en memoria no volátil permitiría que el robot mantuviera su configuración incluso después de ser apagado.

10. Conclusión

El proyecto del robot SCARA ha sido un proceso integral que abarca desde el diseño mecánico y electrónico hasta la programación y la interfaz de usuario. A través de este trabajo, se ha logrado desarrollar un sistema de manipulación automatizado, capaz de realizar movimientos precisos en el espacio y controlar sus parámetros en tiempo real. Este proyecto demuestra la importancia de la intersección entre diversas áreas de la ingeniería, incluyendo la mecatrónica, la electrónica y la programación.

Durante el desarrollo, se enfrentaron diversos desafíos, como la implementación de la cinemática inversa para el posicionamiento exacto del brazo, la sincronización de motores y la creación de una interfaz de usuario intuitiva. La integración de estos elementos ha permitido que el robot SCARA no solo cumpla con las especificaciones técnicas iniciales, sino que también ofrezca flexibilidad y facilidad de uso al operador.

El diseño actual permite al usuario interactuar con el robot a través de un menú estructurado, ajustar parámetros de operación, y llevar a cabo funciones avanzadas como la calibración, el establecimiento de límites de movimiento y la configuración de rutas de trabajo. Sin embargo, se identificaron áreas de mejora que podrían optimizar aún más el rendimiento del sistema, como la inclusión de sensores de retroalimentación, el uso de control PID y la incorporación de una interfaz gráfica de usuario (GUI).

En términos de aprendizaje, este proyecto ha brindado una experiencia enriquecedora en el manejo de componentes de hardware y software, y ha resaltado la importancia de la programación estructurada y del diseño modular en la construcción de sistemas complejos. Las mejoras propuestas en el futuro no solo incrementarán la capacidad operativa del robot, sino que también abrirán nuevas oportunidades de aplicación en áreas de automatización industrial y robótica de precisión.

En resumen, el robot SCARA es una plataforma prometedora que, con futuras mejoras, tiene el potencial de convertirse en una herramienta versátil y efectiva para una amplia gama de tareas de manipulación y ensamblaje. Este proyecto no solo ha cumplido con sus objetivos iniciales, sino que también ha establecido una base sólida para futuras exploraciones y desarrollos en el campo de la robótica.