Universidad Nacional de Ingeniería

Facultad de Ciencias

Proyecto de Investigación Grupal

Clasificación automatizada de productos con brazo robótico en un entorno simulado de Retail

Curso: Arquitectura de Computadoras

Docente: Cesar Martín Cruz Salazar

Integrantes:

- La Torre Urbina Brian Jair (20240398C)
- Cumpa Rodríguez Sofía Milagros (20240591H)
- Pérez Villegas Cristhyan Manuel (20231066A)

Periodo: 2025-I

Lima, Perú
3 de julio de 2025

1. INTRODUCCIÓN

Los centros de distribución minorista procesan miles de unidades cada hora; cualquier desviación manual, ya sea girar, re-etiquetar o re-ubicar una caja, ralentiza la línea y expone a los operarios a fatiga y lesiones musculoesqueléticas. En el Reino Unido, la autoridad de seguridad (HSE) calcula que las lesiones por manipulación manual suponen unos £ 7,1 millardos al año en productividad perdida y costes sanitarios [1]. En Estados Unidos, el subsector de almacenamiento y logística registra una tasa de 4,7 casos de lesión por cada 100 trabajadores/año, casi el doble de la media de todas las industrias privadas [2].

Ante esta realidad, la automatización se consolida como respuesta estratégica. Un estudio de Zebra Technologies indica que el 61 % de las empresas prevé habilitar algún grado de automatización o robotización en sus almacenes antes de 2024 para aliviar la falta de mano de obra y agilizar procesos como la clasificación de paquetes [3]. Análisis de McKinsey muestran que las organizaciones que invierten en robótica obtienen mejoras del 20–30 % en productividad y hasta un 30 % en exactitud de inventario durante el primer año de implantación [4].

En paralelo, la adopción de códigos QR simplifica el rastreo de mercancías: estos códigos bidimensionales almacenan gran cantidad de datos, se leen con cualquier cámara y reducen errores al proporcionar actualizaciones en tiempo real sobre la ubicación y el estado del producto [5].

No obstante, muchas soluciones comerciales basadas en robots colaborativos de seis grados de libertad y sistemas de visión tridimensional resultan inasequibles para pequeñas y medianas plataformas logísticas, especialmente en etapas iniciales de automatización. En respuesta a esta limitación, el presente proyecto desarrolla un prototipo funcional de bajo costo a pequeña escala, pensado para demostrar la viabilidad de automatizar la orientación y clasificación de cajas mediante tecnologías accesibles. El sistema se compone de un brazo robótico compacto de seis grados de libertad, accionado por servomotores y controlado mediante un Arduino UNO Mini, junto con una cámara fija que utiliza visión por computadora 2D con OpenCV para reconocer figuras geométricas en las caras de la caja y detectar códigos QR. A través de un protocolo simple de comunicación USB-serial, se envían al microcontrolador las instrucciones necesarias para rotar la caja o clasificarla en función de la categoría extraída del código. Aunque se trata de una implementación a escala reducida, la arquitectura modular permite su adaptación futura a sistemas más robustos dentro del ámbito logístico.

2. OBJETIVOS

2.1. Objetivo general

Diseñar e implementar un sistema económico que oriente y clasifique automáticamente cajas en un entorno de almacén utilizando visión por computadora 2-D y un brazo robótico de 6 DOF controlado con Arduino UNO Mini.

2.2. Objetivos específicos

- Realizar desplazamiento de ubicaciones con el cubo.
- Detectar si es necesario hacer rotaciones al cubo.
- Realizar rotaciones al cubo.
- Leer el QR, extraer los cinco campos de texto (categoría, producto, precio, peso, fecha).
- Con ayuda del brazo, desplazar el cubo dependiendo de la información contenida en el código QR.
- Documentar arquitectura, lógica de rotación, configuración de servos y protocolo serial.

3. FUNDAMENTO TEÓRICO

3.1. Arduino UNO Mini

El Arduino UNO Mini es una versión compacta del popular Arduino UNO Rev3, desarrollada por la plataforma de hardware libre Arduino.cc. A pesar de su reducido tamaño, mantiene las características fundamentales del modelo original, permitiendo su uso en proyectos de electrónica, robótica y automatización donde el espacio es limitado.

Este se encarga de ejecutar el programa cargado en la placa y controlar los pines de entrada/salida digitales y analógicos.

3.2. Servomotores y señales PWM

Los servomotores funcionan como los músculos del sistema, proporcionando el movimiento necesario para que el brazo robótico cumpla sus funciones. Estos motores permiten rotar las articulaciones del brazo a distintos ángulos, los cuales son controlados por el Arduino mediante señales PWM (Pulse Width Modulation).

La posición del eje del servomotor se determina por el ancho del pulso: típicamente, pulsos de entre 1 y 2 milisegundos, repetidos cada 20 milisegundos (frecuencia de 50 Hz), corresponden a ángulos de 0° a 180°, respectivamente [6].

Dado que los servomotores pueden requerir más corriente de la que el Arduino puede proporcionar, es recomendable utilizar una fuente de alimentación externa. Para asegurar una referencia lógica común, se debe conectar la masa (GND) del Arduino con la masa de la fuente externa, una práctica recomendada en la documentación oficial del SDK [7].

La razón por la que necesitamos que el brazo tenga movilidad es para poder cumplir su función de sostener objetos, moverlos o girarlos como lo haría un brazo humano.

3.3. Cinemática en brazos robóticos de 6 DOF

La cinemática en robótica se encarga de describir el movimiento de los manipuladores sin tener en cuenta las fuerzas que lo originan. En particular, los brazos robóticos con seis grados de libertad (6 DOF) pueden posicionar y orientar su efecto final en el espacio tridimensional,

imitando con precisión la movilidad del brazo humano. Esta configuración consta de tres articulaciones dedicadas al posicionamiento (base, hombro y codo) y otras tres encargadas de la orientación (muñeca-in, muñeca-rot y pinza), lo que permite realizar tareas complejas como ensamblaje, clasificación o manipulación de objetos en entornos automatizados [8].

3.4. Códigos QR

3.4.1. Funcionamiento

Un lector QR identifica un código QR por los tres grandes cuadrados en sus esquinas, los cuales indican que todo el contenido es un código QR. Luego, analiza el interior descomponiéndolo en una cuadrícula, donde cada celda blanca o negra representa un valor binario. Estos valores se agrupan para formar patrones más grandes que contienen la información codificada.

3.4.2. Partes de un QR

Un código QR estándar se puede identificar mediante seis componentes [14]:

1. **Zona tranquila:** Es el borde blanco alrededor del código QR que lo aísla visualmente y evita interferencias externas.
2. **Patrón de búsqueda:** Son tres cuadrados grandes en las esquinas que permiten al lector identificar el código y sus límites.
3. **Patrón de alineación:** Es un pequeño cuadrado cerca de la esquina inferior derecha que corrige inclinaciones en la lectura.
4. **Patrón de sincronización:** Es una línea en forma de L que ayuda a localizar las celdas individuales y permite leer códigos dañados.
5. **Información de la versión:** Indica el tipo de codificación (numérica, alfanumérica, byte o kanji) y se encuentra cerca de la parte superior derecha.
6. **Celdas de datos:** Son el área donde se almacena la información real como URLs, mensajes o números.

3.4.3. Versiones del código QR

La versión utilizada determina cómo se pueden almacenar los datos y se denomina "modo de entrada".

- **Modo numérico:** Almacena solo dígitos del 0 al 9. Es el más eficiente, con capacidad de hasta 7089 caracteres.
- **Modo alfanumérico:** Soporta dígitos, letras mayúsculas y algunos símbolos (como $, %, *, +, etc.). Puede almacenar hasta 4296 caracteres.

- **Modo de byte:** Permite caracteres del conjunto ISO-8859-1 (como letras minúsculas y símbolos adicionales). Almacena hasta 2953 caracteres.
- **Modo Kanji:** Diseñado originalmente por Denso Wave para caracteres japoneses en el conjunto Shift JIS. Tiene una capacidad de hasta 1817 caracteres, siendo de los menos efectivos actualmente.
- **Modo ECI (Kanji extendido):** Usa UTF-8 para más compatibilidad, pero no todos los lectores lo reconocen.

Hay dos modos adicionales que son modificaciones de los otros tipos:

- **Modo de apéndice estructurado:** Codifica los datos a través de múltiples códigos QR, permitiendo la lectura simultánea de hasta 16 códigos QR.
- **Modo FNC1:** Permite que un código QR funcione como un código de barras GS1.

3.4.4. Implementación del código QR

En este proyecto se utilizan códigos QR estándar debido a su alta compatibilidad con diversos programas y bibliotecas de visión por computadora, lo que facilita su integración en entornos automatizados. Estos códigos cuentan con patrones de búsqueda claramente definidos en tres de sus esquinas, lo cual permite que el sistema identifique rápidamente su posición y orientación, incluso si la caja está ligeramente rotada. Además, disponen de mecanismos de corrección de errores, lo que permite leer el contenido aun cuando el código se encuentra parcialmente dañado o sucio, una característica especialmente útil en entornos logísticos reales donde las condiciones pueden ser adversas.

Para generar estos códigos, se desarrolló un script en `Python` que emplea la biblioteca `qrcode`. El programa toma como entrada archivos `.json` predefinidos y produce como salida imágenes `.png` listas para imprimir. Este enfoque garantiza que cada caja esté correctamente identificada desde su origen, y que la información esté organizada y sea legible automáticamente por el sistema de visión. Adicionalmente, se utilizó la biblioteca `os` de Python para automatizar la creación de carpetas, gestionar rutas de archivo y almacenar las imágenes generadas en directorios específicos, facilitando así la organización y reutilización de los datos durante la fase de pruebas.

Además de generar los códigos QR, es necesario implementar un sistema que permita su decodificación automática utilizando una cámara convencional conectada al sistema. Para ello, se desarrolló un programa en `Python` basado en la biblioteca `OpenCV`, la cual permite capturar imágenes en tiempo real, procesarlas (por ejemplo, convertirlas a escala de grises, mejorar el contraste y eliminar ruido) y detectar regiones de interés que contienen el código QR. Posteriormente, se aplica una función de decodificación integrada en OpenCV para extraer el contenido del código.

Durante la etapa de diseño se evaluaron otras bibliotecas como `pyzbar` (una interfaz en Python que utiliza `ZBar`, una biblioteca ligera escrita en C) y `ZXing` (una solución robusta originalmente desarrollada en Java), debido a su eficiencia y compatibilidad con múltiples formatos. Sin embargo, se optó por `OpenCV` por su integración directa con el resto del sistema de visión, su amplia documentación y su versatilidad para combinar el reconocimiento de figuras

geométricas y la lectura de códigos QR en un solo flujo de trabajo. Esta decisión permitió mantener una solución funcional, accesible y multiplataforma, adecuada para automatizar la identificación y clasificación de productos en el proyecto.

3.5. Comunicación USB-Serial

La comunicación USB-Serial es una técnica ampliamente utilizada para el intercambio de datos entre dispositivos electrónicos, especialmente entre microcontroladores y computadoras personales. Aunque el estándar Universal Serial Bus (USB) se ha convertido en el principal medio físico de conexión, muchos microcontroladores — como el Arduino UNO Mini— utilizan internamente una comunicación serial UART (Universal Asynchronous Receiver-Transmitter), por lo que es necesario convertir las señales entre ambos protocolos para establecer una comunicación efectiva. [17]

En proyectos de robótica, esta forma de comunicación resulta esencial para enviar comandos desde una interfaz gráfica o consola hacia el microcontrolador, o bien para recibir datos de sensores o mensajes de estado. En particular, para nuestro proyecto en un brazo robótico controlado mediante un Arduino UNO Mini, la comunicación USB-Serial permite enviar instrucciones desde una computadora para mover motores, ajustar posiciones o iniciar rutinas automatizadas.

4. Materiales Y Herramientas

El desarrollo del prototipo requirió la selección de componentes que equilibraran coste, disponibilidad local y prestaciones suficientes para la tarea de orientación y clasificación de cajas. En la Tabla 4-1 se resume el hardware empleado y en la Tabla 4- 2 detalla el entorno software.

Cuadro 1: Hardware empleado en el prototipo

Componente	**Función en el proyecto**
Arduino UNO Mini	Microcontrolador que genera PWM y recibe comandos por USB-Serial
Brazo robótico 6 DOF	Estructura mecánica que manipula las cajas
6 servomotores	Actuadores que posicionan cada articulación del brazo
Fuente conmutada 5V / 5A	Alimenta los servos
Cámara de Celular	Captura la cara visible de la caja para visión por computadora
Borneras	Interconexión del Arduino con la fuente y los servomotores
Cable USB A-micro B	Comunicación y alimentación de la Arduino UNO Mini
Cajas de prueba (80 mm)	Objetos para clasificar con código QR

Cuadro 2: Entorno software

Componente	**Función en el proyecto**
Python	Lenguaje principal para visión y comunicación
OpenCV-Python	Detección de contornos y preprocesado de imagen
pyzbar	Decodificación de códigos QR
Pyserial	Envío de comandos ASCII vía USB CDC
MicroPython	Firmware en Arduino UNO Mini 2
Thonny IDE	Carga y depuración de scripts en la Pico

5. DISEÑO DEL SISTEMA

5.1. Generación del código QR

Cada caja del sistema incorpora un código QR impreso en su cara superior, codificado en versión 3 (29×29 módulos, nivel de corrección L), lo cual permite almacenar hasta 70 caracteres alfanuméricos con buena velocidad de lectura y adecuada tolerancia a errores. El contenido del QR se estructura en formato JSON, lo que facilita la organización jerárquica de los datos y permite su validación individual por parte del software.

Para generar estos códigos, se emplea un script en Python basado en la biblioteca qrcode. El programa toma como entrada archivos .json previamente definidos y produce como salida imágenes .png que pueden ser impresas en etiquetas adhesivas. Este procedimiento garantiza

que cada caja esté correctamente identificada desde su origen, permitiendo su lectura y procesamiento automático por parte de la cámara y el sistema de visión.

A continuación, se muestra un ejemplo del contenido embebido en el código QR:

Ejemplo de estructura JSON

```
{
                    "ID": "A00123",
                    "Nombre": "Jabón en barra",
                    "Categoría": "Limpieza y hogar",
                    "Destino": "Zona 1",
                    "Fecha": "2025-05-05"
}
```

El diagrama de flujo seguido es el siguiente:

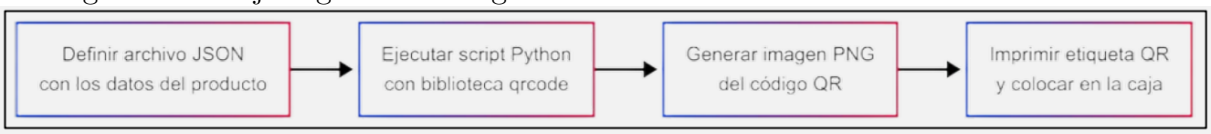

Figura 5.1.1 Diagrama de Flujo de esta etapa

Figura 5.1.2 Caja Referencial

5.2. Diseño físico de la caja

La caja utilizada en el sistema fue construida con dimensiones exteriores de 45 × 45 × 45 milímetros. Se empleó cartón micro corrugado, seleccionado por su bajo peso, suficiente rigidez para soportar múltiples manipulaciones y facilidad de reemplazo en caso de daño. Cada una de las seis caras fue pintada con un acabado blanco mate, con el objetivo de evitar reflejos que puedan interferir con el reconocimiento visual. En cinco de ellas se adhirieron figuras geométricas (estrella, equis, círculo, triángulo y cuadrado), impresas en fondo negro, a fin de garantizar un alto contraste sobre la superficie clara. La sexta cara, designada como superior, contiene un código QR versión 3 (29 × 29 módulos), centrado e impreso con 35 mm de lado, lo que asegura su legibilidad a una distancia de 10 a 20 centímetros bajo condiciones de iluminación controlada. Este diseño estandarizado permite que la caja sea fácilmente reconocida por la cámara estática y manipulada con precisión por el brazo robótico,

favoreciendo la estabilidad y reproducibilidad del sistema.

5.3. Código generador de QRs

```
import json
import qrcode
import os
```

Estas líneas importan las librerías necesarias:

- **json:** para leer archivos .json.
- **qrcode:** para generar imágenes de códigos QR.
- **os:** para trabajar con archivos y carpetas del sistema.

```
# Carpeta donde estn todos los archivos JSON
carpeta_data = '../data'
# Carpeta donde se guardarn las imgenes PNG del QR
carpeta_qr = '../qr_codes_new'
```

Define las rutas relativas donde:

- `carpeta_data`: Contiene los archivos `.json` con la información que será codificada.
- `carpeta_qr`: Es la carpeta de salida donde se guardarán los códigos QR generados en formato `.png`.

```
# Crear la carpeta de salida si no existe
os.makedirs(carpeta_qr, exist_ok=True)
```

Crea la carpeta `qr_codes_new` si no existe aún, evitando errores al guardar los archivos.

```
# Obtener lista de todos los archivos JSON en la carpeta data
archivos_json = [f for f in os.listdir(carpeta_data) if f.endswith('.json')]
```

Obtiene una lista de nombres de archivos .json que están en la carpeta data (Incluyendo solo los que terminan en .json).

```
# Recorrer cada archivo JSON
for archivo in archivos_json:
    ruta_completa = os.path.join(carpeta_data, archivo)
```

Inicia un bucle que recorre cada archivo .json encontrado y construye la ruta completa del archivo usando su nombre y la carpeta de origen.

```
    # Abrir y leer el contenido del archivo JSON
    with open(ruta_completa, 'r', encoding='utf-8') as f:
        datos = json.load(f)
```

Abre el archivo .json y lee su contenido como un diccionario de Python llamado datos.

```
    # Convertir el diccionario a texto JSON
    contenido_qr = json.dumps(datos, ensure_ascii=False)
```

`json.dumps` convierte ese diccionario datos de nuevo a texto JSON plano (str) que se usará como contenido del QR y `ensure_ascii=False` permite que se guarden tildes, ñ y otros caracteres especiales correctamente.

```
# Generar el cdigo QR
qr = qrcode.make(contenido_qr)
```

Genera un código QR que contiene el texto convertido del archivo .json.

```
# Obtener el nombre del archivo de salida (ej. A00100.png)
nombre_archivo_salida = f"{datos['ID']}.png"
```

Usa el valor de ID del JSON (por ejemplo, .ª00100") como nombre del archivo de imagen QR para luego añadir la extensión .png.

```
ruta_salida = os.path.join(carpeta_qr, nombre_archivo_salida)
```

Construye la ruta completa donde se va a guardar la imagen del QR.

```
# Guardar la imagen del QR
qr.save(ruta_salida)
```

Guarda la imagen del código QR en la ruta especificada, como un .png.

```
print(f" QR generado: {ruta_salida}")
```

Imprime un mensaje para el usuario confirmando que el QR fue generado correctamente, junto a su ruta.

6. ¿Cómo replicar nuestro proyecto?

Hemos creado un repositorio en github con todos los programas, videos, guías, imágenes, renders y documentación de todo lo que utilizamos para realizar este proyecto, tales como:

- Guía de uso para el brazo robótico educativo.
- Videos explicativos de la rotación del cubo.
- Imágenes referenciales de los grados de libertad.
- Renders de la base para imprimir en 3D y corte láser.
- Código actualizado en Arduino y Python.

7. Código en Arduino IDE

```
#include <Arduino.h>
#include <Servo.h>

// Articulaciones del brazo
enum Articulacion {
  BASE,
  HOMBRO,
  CODO,
  MUNHECA_ROT,
  MUNHECA_IN,
  PINZA
};
```

Incluimos la librería Arduino.h para las funciones básicas del entorno de Arduino como setup() y loop().
También incluimos la librería Servo.h para el control de los servomotores con las señales PWM y funciones como .attach(), .write() y .read().
Además, creamos un conjunto de constantes con nombres legibles para el uso en el resto del código.

```
String mensaje;
Servo servos[6];
const int home[6] = {
  105,  // BASE
   35,  // HOMBRO
    5,  // CODO
   80,  // MUNHECA_ROT
  160,  // MUNHECA_IN
   70   // PINZA
};
```

Inicializamos el string mensaje, el cual nos servirá para enviar mensajes por el Serial Monitor.
También creamos un arreglo de 6 servos para cada articulación.
home[6] establece las posiciones que tiene cada articulación como posición inicial.

```
const int posicion_1[6] = {80, 70, 7, 80, 160, 0};

void mover_servos_posicion(const int destino[6]);
void mover(Articulacion articulacion, int destino);
bool parsear_comando(const String& mensaje,
String &nombre, int &angulo);
bool nombre_a_articulacion(const String& nombre,
Articulacion& resultado);
void agarrar_caja();
void zona_vencido();
void zona_no_vencido();
void suave(Articulacion articulacion, int inicio, int final);
void circuito1();
void circuito2();
void HOME();

void subir();
```

Declaramos las funciones utilizadas

```
void setup() {
  servos[BASE].attach(2);
  servos[HOMBRO].attach(3);
  servos[CODO].attach(4);
  servos[MUNHECA_ROT].attach(5);
  servos[MUNHECA_IN].attach(6);
  servos[PINZA].attach(10);

  Serial.begin(9600);

  // Posici n inicial
  mover_servos_posicion(home);
  delay(2000);

  Serial.println("READY");
}
```

Establecemos la función setup() que se ejecuta una sola vez al iniciar el programa, conectamos los servos a sus pines e iniciamos el monitor serial y le decimos cuál es la posición inicial.

```
void loop() {
  if (Serial.available() > 0) {
    mensaje = Serial.readStringUntil('\n');
    mensaje.trim();

    String nombre;
    int angulo;

    if (parsear_comando(mensaje, nombre, angulo)) {
      Articulacion articulacion;
      if (nombre_a_articulacion(nombre, articulacion)) {
        mover(articulacion, angulo);
        Serial.println("OK");
      } else {
        Serial.println("ERROR: Articulaci n no v lida");
      }
      return;
    }

    if (mensaje == "POS1") mover_servos_posicion(posicion_1);
    else if (mensaje == "CAJA") agarrar_caja();
    else if (mensaje == "VENCIDO") zona_vencido();
    else if (mensaje == "NO_VENCIDO") zona_no_vencido();
    else if(mensaje == "SUBIR") subir();
    else if(mensaje == "CIRCUITO1") circuito1();
    else if(mensaje == "CIRCUITO2") circuito2();
    else if(mensaje=="HOME") HOME();
    else {
      Serial.println("ERROR: Mensaje no reconocido");
      return;
    }

    Serial.println("OK");
  }
}
```

loop() hace diferentes funciones como revisar si un mensaje fue enviado, lee los mensajes y los limpia. Intenta identificar si los mensajes son mensajes especiales para mover el brazo. En caso contrario, buscará el mensaje entre los ya establecidos para llamar funciones.

```
void mover(Articulacion articulacion, int destino) {
  destino = constrain(destino, 0, 180);
  int actual = servos[articulacion].read();
  if (actual == destino) return;

  int paso = (destino > actual) ? 1 : -1;

  while (actual != destino) {
    actual += paso;
    servos[articulacion].write(actual);
  }
}
```

Función especial para realizar todos los movimientos, establece los límites de los valores que puedes ingresar, verificar la posición actual y mover si es necesario.

```
bool parsear_comando(const String& mensaje,
String &nombre, int &angulo) {
  int espacio = mensaje.indexOf(' ');
  if(espacio == -1) return false;

  nombre= mensaje.substring(0, espacio);
  nombre.trim();
  nombre.toUpperCase();

  String angulo_str=mensaje.substring(espacio + 1);
  angulo_str.trim();
  angulo= angulo_str.toInt();
  angulo= constrain(angulo, 0, 180);
  return true;
}
```

Esta función interpreta el mensaje enviado mediante el monitor serial y acomoda las mayúsculas para que sea validado. Termina mandando true si es un mensaje del tipo [ARTICULACION] [ANGULO].

```
void mover_servos_posicion(const int destino[6]) {
  for (int i = 0; i < 6; i++) {
    int actual = servos[i].read();
    if (actual == destino[i]) continue;

    int paso = (destino[i] > actual) ? 1 : -1;
    while (actual != destino[i]) {
      actual += paso;
      servos[i].write(actual);
    }
  }
}
```

Vuelve una cadena de números como "home[6]"a la posición actual del brazo.

```
void HOME(){
  mover(BASE, 105);
  mover(HOMBRO, 35);
  mover(CODO, 5);
  mover(MUNHECA_IN, 160);
  mover(MUNHECA_ROT, 80);
  mover(PINZA, 70);
}
```

Función para tener una posición segura llamada "HOME"que se puede enviar por la terminal y ser llamada.

```
void suave(Articulacion articulacion, int inicio, int final){
  do{
    if(inicio<final){
      inicio= inicio + 5;
    }else{
      inicio= inicio - 5;
    }
    mover(articulacion, inicio);
    delay(500);
  }while(inicio!=final && inicio>0 &&
  inicio<180 && final>0 && final<180);
}
```

Función especial para realizar el movimiento de una articulación desde un angulo inicial a

uno final con un retraso de 0.5s y un incremento o decremento de 5°.

```
void caida_munheca(){
  mover(HOMBRO, 20);
  delay(1000);
  mover(MUNHECA_IN, 170);
  delay(4000);
  mover(HOMBRO,50);
  delay(1000);
}
```

Función útil para acomodar el servomotor "Muñeca-In", ya que no logra llegar a 180° mediante el servomotor, debe hacer una caida por peso para que sea perpendicular a la base.

```
void agarrar_caja() {
  mover(BASE, 105);
  delay(1000);
  mover(HOMBRO, 20);
  delay(1000);
  mover(MUNHECA_IN, 170);
  delay(4000);
  suave(HOMBRO, 20, 80);
  mover(PINZA, 130);
  delay(1000);
  mover(HOMBRO, 70);
  delay(1000);
  suave(BASE, 105, 80);
  suave(HOMBRO, 70, 80);
  mover(PINZA, 70);
  delay(1000);
  suave(HOMBRO, 80, 30);
}
```

Función para agarrar la caja en una posición inicial específica. Procederá a trasladarlo a la sección de la cámara.

```
void subir() {
  mover(BASE, 80);
  suave(HOMBRO, 40, 80);
  mover(PINZA, 130);       delay(1000);
  suave(HOMBRO, 80, 35);
  suave(MUNHECA_IN, 170, 150);
  suave(BASE, 80, 30);
  mover(HOMBRO, 40);delay(1000);
  mover(PINZA, 70);delay(100);
  mover(HOMBRO, 35);delay(1000);
  mover(MUNHECA_IN, 165);delay(1000);
  mover(PINZA, 130);delay(1000);
  mover(HOMBRO, 30);delay(1000);
  mover(MUNHECA_IN,150);delay(1000);
  mover(HOMBRO, 40);delay(1000);
  mover(PINZA, 70);delay(1000);
  mover(HOMBRO, 35);delay(1000);
  mover(MUNHECA_IN, 165);delay(1000);
  mover(PINZA, 130);delay(1000);
  mover(HOMBRO, 30);
  delay(1000);
  mover(MUNHECA_IN,150);
  delay(1000);
  suave(HOMBRO, 30, 20);
  mover(MUNHECA_IN, 170);
  delay(4000);
  suave(BASE, 30, 80);
  mover(MUNHECA_ROT, 150);
  suave(HOMBRO, 20, 80);
  mover(PINZA, 70);
  delay(1000);
  suave(HOMBRO, 80, 30);
  mover(MUNHECA_ROT, 80);
  delay(1000);
}
```

Esta función hace las rotaciones necesarias para poder ver las otras 3 caras en caso de que el QR no haya sido detectado. Es necesaria una base con altura y medidas especificas para poder generar las rotaciones ya que el brazo por si solo no puede hacerlas.

```
void zona_vencido() {
  mover(BASE, 80);
  suave(HOMBRO,30 ,80);
  mover(PINZA, 130);
  delay(1500);
  suave(HOMBRO, 80, 30);
  suave(BASE, 80, 140);
  suave(HOMBRO, 30, 80);
  mover(PINZA, 70);
  delay(1500);
  suave(HOMBRO, 80, 30);
  delay(1500);
}
```

Segunda opción para el contenido de un QR. Si la fecha de vencimiento no es válida, el cubo pasará a una zona específica.

```
void zona_no_vencido() {
  mover(BASE, 80);
  suave(HOMBRO,30 ,80);
  mover(PINZA, 130);
  delay(1500);
  suave(HOMBRO, 80, 30);
  suave(BASE, 80, 125);
  suave(HOMBRO, 30, 80);
  mover(PINZA, 70);
  delay(1500);
  suave(HOMBRO, 80, 30);
  delay(1500);
}
```

Una vez que un objeto sea clasificado como no vencido, pasa a la sección para objetos no vencidos específica. Esta función se llama dependiendo del contenido del QR.

```
void circuito1(){
  agarrar_caja();
  subir();
  zona_no_vencido();
}
```

Función de prueba para imitar los movimientos en caso de que un cubo tenga contenido

válido en el QR.

```
void circuito2(){
  agarrar_caja();
  subir();
  zona_vencido();
}
```

Función de prueba para imitar los movimientos en caso de que un cubo tenga contenido no válido en el QR.

8. Código de Python: GUI y Detección de QR

Este programa hace uso de diversas librerías fundamentales para su funcionamiento. En primer lugar, la librería `serial` permite establecer y gestionar la comunicación con el Arduino a través del puerto serial. Por otro lado, `time` se emplea para introducir pausas temporizadas en la ejecución, y `atexit` asegura que los recursos utilizados, como la cámara o el puerto de comunicación, se liberen correctamente al cerrar el programa.

Además, la librería `json` es utilizada para decodificar los datos extraídos del código QR, que se encuentran en formato JSON, mientras que `datetime` (importada como `dt`) permite comparar fechas, lo cual es esencial para verificar si un producto está vencido. Para mantener la fluidez de la interfaz gráfica durante operaciones que pueden tardar varios segundos, se utiliza `threading`, que permite ejecutar tareas en paralelo sin bloquear la interacción con el usuario.

En cuanto al procesamiento de imágenes, se emplea la biblioteca OpenCV (`cv2`) para capturar video en tiempo real desde una cámara IP, detectar códigos QR y dibujar sobre las imágenes. Esta funcionalidad es complementada por `numpy` (`np`), que facilita el manejo eficiente de matrices de coordenadas e imágenes. Para la creación de la interfaz gráfica, se hace uso de `tkinter` (`tk`), junto con `tkinter.font` (`tkFont`), que permiten diseñar ventanas, botones, paneles y personalizar tipografías.

Finalmente, la biblioteca `PIL` (específicamente los módulos `Image` y `ImageTk`) se encarga de convertir las imágenes capturadas por OpenCV en un formato compatible con los elementos gráficos de Tkinter. En conjunto, todas estas herramientas permiten desarrollar una aplicación robusta capaz de controlar un brazo robótico mediante una interfaz visual interactiva, y comunicación en tiempo real con el hardware.

```
1 import serial, time, json, datetime as dt
2 import serial.tools.list_ports as list_ports
3 import threading
4 import cv2
5 from PIL import Image, ImageTk
6 import tkinter as tk
```

```
import tkinter.font as tkFont
import numpy as np
```

Ahora, definimos la función `connect_arduino()`, la cual se encarga de detectar automáticamente el puerto en el que está conectado un dispositivo Arduino. Para ello, recorre todos los puertos seriales disponibles mediante e intenta establecer una conexión con cada uno utilizando la velocidad de transmisión (`baud`) y un tiempo de espera (`timeout`) especificados. Una vez conectado, espera 2 segundos para dar tiempo a que el Arduino inicie, y luego lee una línea de texto desde el puerto. Si esa línea coincide con el mensaje esperado de confirmación (`READY"`), se considera una conexión exitosa, se imprime el nombre del puerto y se retorna el objeto serial. En caso contrario, se cierra el puerto y continúa con el siguiente. Si no se encuentra ningún Arduino conectado, la función imprime un mensaje de error.

La llamada `arduino = connect_arduino()` guarda el resultado, que puede ser usado más adelante para enviar comandos al dispositivo.

```
def connect_arduino(baud=9600, handshake=b"READY", timeout=2):
    for port in list_ports.comports():
        try:
            ser = serial.Serial(port.device,
            baudrate=baud, timeout=timeout),
            time.sleep(2)
            line = ser.readline().strip()
            print(f">>> Escuchando {port.device} -> {line}")
            if line == handshake:
                print(f">>> Arduino OK -> {port.device}")
                return ser
            ser.close()
        except (OSError, serial.SerialException):
            pass
    print(">>> Arduino NO encontrado")
    return None

arduino = connect_arduino()
```

Por otro lado, definimos la función `send_cmd`, la cual permite enviar un comando en forma de texto al Arduino a través del puerto serial. Primero, verifica si el dispositivo está correctamente conectado comprobando la existencia del objeto `arduino`. Si no hay conexión, muestra un mensaje de advertencia en consola y termina la función. En caso contrario, convierte el comando en una secuencia de bytes, le agrega un salto de línea al final y lo envía mediante el método `write()`. Luego, utiliza `flush()` para asegurar que los datos se transmitan inmediatamente. Finalmente, imprime en consola el comando enviado para fines de depuración.

Esta función es esencial para la comunicación segura y controlada con el Arduino desde la interfaz en Python.

```python
def send_cmd(cmd: str):
    """Envia un comando al Arduino y lo muestra por consola."""
    if not arduino:
        print(f">>> Arduino no conectado (no se envio {cmd})")
        return
    arduino.write(cmd.encode() + b'\n')
    arduino.flush()
    print(f">>> {cmd} enviado al Arduino")
```

Ahora, el siguiente bloque de código se configura un detector de códigos QR utilizando la biblioteca OpenCV. Primero, se crea un objeto `qr_detector` que permite detectar y decodificar códigos QR en imágenes. Luego, se define la función `decode_qr(frame)`, que toma como entrada un cuadro de video o imagen. Esta función aplica el método `detectAndDecode(frame)` para extraer el contenido del código QR. Si no se detecta ningún código válido, se devuelve `None`. Si se obtiene información, se intenta convertir la cadena de texto en un objeto tipo diccionario usando `json.loads(data)`, bajo el supuesto de que el contenido está en formato JSON. Si la conversión falla, se captura la excepción `JSONDecodeError` y también se devuelve `None`. De este modo, la función proporciona tanto los datos como las coordenadas del código QR en caso de éxito.

```python
qr_detector = cv2.QRCodeDetector()
def decode_qr(frame):
    data, pts, _ = qr_detector.detectAndDecode(frame)
    if not data:
        return None, None
    try:
        return json.loads(data), pts
    except json.JSONDecodeError:
        return None, None
```

En el siguiente bloque se definen los colores principales de la interfaz, las dimensiones de visualización de la cámara y la URL desde donde se obtiene el video. La URL la obtenemos de la aplicación de celular 'IPWebcam' y para su correcto funcionamiento, el celular tiene que estar en la misma red wifi que la computadora.

Posteriormente, se configura la ventana principal de la aplicación gráfica, estableciendo su título, tamaño inicial, color de fondo, tamaño mínimo permitido y la estructura general en columnas y filas. Esto permite que la interfaz sea visualmente coherente, adaptable y con una distribución organizada entre la zona de control y la visualización de la cámara.

```python
BG_MAIN, BG_PANEL = "#181818", "#181818"
ACCENT, COLOR_TXT = "#3dbc95", "#ffffff"
DISPLAY_W, DISPLAY_H = 400, 300
INFO_W = 320
CAM_URL = "http://XX.X.XX.XXX:XXXX/video"

root = tk.Tk()
```

```
root.title("Control Brazo QR (debug)")
root.configure(bg=BG_MAIN)
root.geometry("930x520")
root.minsize(850, 480)
root.columnconfigure(0, weight=1, minsize=INFO_W + 40)
root.columnconfigure(1, weight=3)
root.rowconfigure(0, weight=1)
```

En este segmento se crea el panel lateral principal que alberga los datos del QR. Primero, `panel = tk.Frame(...)` genera un `Frame` hijo de `root`, con color de fondo `BG_MAIN`. Se coloca en la posición `row=0, column=0` mediante `grid()`, se expande en las cuatro direcciones y recibe márgenes internos (`padx`, `pady`). Luego se ajusta su capacidad de expansión vertical y horizontal con `rowconfigure` y `columnconfigure`. Dentro del panel se crea `info_frame`, otro `Frame` con ancho fijo (`INFO_W`), color de fondo `BG_PANEL` y borde blanco (`highlightbackground`, `highlightthickness`). Se ubica en la fila 1 del panel, ocupa todo el ancho (`sticky="ew"`) y su columna interna se hace flexible con `columnconfigure`. Finalmente, se añade un título "INFORMACIÓN OBTENIDA" con `tk.Label()`, usando la fuente `Segoe UI` en tamaño 12 y color de texto `ACCENT`; se empaqueta con un pequeño margen interno para separarlo de los bordes.

```
panel = tk.Frame(root, bg=BG_MAIN)
panel.grid(row=0, column=0, sticky="nsew", padx=40, pady=40)
panel.rowconfigure((0, 3), weight=1)
panel.columnconfigure(0, weight=1)

info_frame = tk.Frame(
    panel, width=INFO_W, bg=BG_PANEL,
    highlightbackground="#ffffff", highlightthickness=2
)
info_frame.grid(row=1, column=0, sticky="ew", padx=5, pady=10)
info_frame.columnconfigure(0, weight=1)

tk.Label(
    info_frame, text="INFORMACIN OBTENIDA", bg=BG_PANEL, fg=ACCENT,
    font=("Segoe UI", 12, "bold"), anchor="w"
).pack(padx=(10, 0), pady=(8, 10))
```

A continuación, se define el diccionario `raw`, que contiene las etiquetas base para los campos que serán mostrados al usuario: ID, Nombre, Categoría, Destino y Fecha. Estas cadenas incluyen tabulaciones para lograr una alineación más uniforme al momento de ser combinadas con los valores reales del QR.

Luego, se construye el diccionario `labels`, donde a cada clave de `raw` se le asocia un objeto `tk.Label`. Estos elementos se crean con el texto inicial de cada campo, una fuente `Roboto` de tamaño 10, y se configuran con el color de fondo y texto previamente definidos. Las etiquetas se alinean a la izquierda mediante el parámetro `anchor="w"`.

Posteriormente, se recorre el diccionario `labels` y cada etiqueta se empaqueta verticalmente dentro del `info_frame`, ocupando todo el ancho disponible y con márgenes internos para separar visualmente cada línea.

Finalmente, se define la función `set_labels(data)`, que actualiza dinámicamente el contenido de las etiquetas según la información recibida desde el código QR. Para cada campo, se concatena la cabecera original del diccionario `raw` con el valor correspondiente del diccionario `data`, y se actualiza el texto visible en la interfaz.

```
raw = {
    "ID": "ID:\t\t",
    "Nombre": "Nombre:\t\t",
    "Categoria":"Categora:\t",
    "Destino": "Destino:\t\t",
    "Fecha": "Fecha:\t\t"
}

labels = {
    k: tk.Label(
        info_frame, text=v, font=("Roboto", 10),
        bg=BG_PANEL, fg=COLOR_TXT, anchor="w"
    )
    for k, v in raw.items()
}

for lbl in labels.values():
    lbl.pack(fill="x", padx=10, pady=(2, 4))

def set_labels(data):
    for k, lbl in labels.items():
        lbl.config(text=f"{raw[k]}{data.get(k, '')}")
```

Este bloque crea dos secciones importantes de la interfaz gráfica: el área de botones de control y el marco donde se visualizará la cámara.

En primer lugar, se construye el contenedor `frame_btns`, que se ubica en la fila 2 de la columna izquierda (`panel`). Este `Frame` tiene el mismo color de fondo que el panel principal y un margen superior de 30 píxeles. Se configura para que ambas columnas internas (índices 0 y 1) tengan el mismo peso, lo cual permite que los botones se distribuyan de manera uniforme. Además, se define la fuente de los botones mediante `tkFont.Font()`, usando el tipo de letra `Segoe UI`, tamaño 11 y estilo en negrita.

A continuación, se construye el contenedor `frame_cam`, que está destinado a mostrar la imagen proveniente de la cámara IP. Este marco se ubica en la columna derecha de la interfaz (`column=1, row=0`), con márgenes horizontales y verticales, un borde blanco y dimensiones fijas definidas por `DISPLAY_W` y `DISPLAY_H`. Se desactiva la expansión automática de su contenido mediante `grid_propagate(False)` para mantener su tamaño constante. Dentro de él se coloca `lbl_cam`, un `Label` sin bordes que se expande para ocupar todo el espacio

disponible; allí se mostrará el video capturado en tiempo real.

```
frame_btns = tk.Frame(panel, bg=BG_MAIN)
frame_btns.grid(row=2, column=0, pady=(30, 0))
frame_btns.columnconfigure((0, 1), weight=1)
btn_font = tkFont.Font(family="Segoe UI", size=11, weight="bold")

frame_cam = tk.Frame(
    root, width=DISPLAY_W, height=DISPLAY_H,
    bg=BG_PANEL, highlightbackground="#ffffff",
    highlightthickness=2
)
frame_cam.grid(row=0, column=1, padx=40, pady=40)
frame_cam.grid_propagate(False)

lbl_cam = tk.Label(frame_cam, bg=BG_PANEL, bd=0)
lbl_cam.pack(fill='both', expand=True)
```

En este fragmento se inicializa la captura de video desde la cámara IP configurada previamente. Para ello, se utiliza `cv2.VideoCapture(CAM_URL)`, que intenta abrir el flujo de video en tiempo real desde la dirección proporcionada. A continuación, se verifica si la cámara se abrió correctamente mediante el método `isOpened()`; en caso contrario, se muestra un mensaje de error por consola informando que no se pudo establecer conexión con la cámara.

Seguidamente, se declaran tres variables globales que serán utilizadas a lo largo del programa. La variable `last_frame` almacenará el último fotograma capturado por la cámara, lo cual es necesario para detectar y procesar códigos QR. La variable `qr_bbox` guardará las coordenadas del contorno del QR detectado, si existe. Por último, la variable `scanning` actuará como bandera lógica para controlar si el sistema está actualmente en modo de búsqueda de un código QR.

```
cap = cv2.VideoCapture(CAM_URL)
if not cap.isOpened():
    print(">>> ERROR: No se pudo abrir la cmara en", CAM_URL)

last_frame = None
qr_bbox = None
scanning = False
```

Esta función, llamada `update_camera()`, se encarga de capturar continuamente imágenes desde la cámara IP y mostrarlas en tiempo real dentro del widget gráfico `lbl_cam`.
Primero, se declara como global la variable `last_frame`, que almacenará la última imagen capturada. Luego, se intenta leer un nuevo fotograma desde la cámara mediante `cap.read()`. Si la lectura es exitosa, el fotograma se redimensiona a las dimensiones establecidas en `DISPLAY_W` y `DISPLAY_H`, y se guarda una copia del mismo.

Posteriormente, se convierte el fotograma de formato BGR (por defecto en OpenCV) a

formato RGB, necesario para que sea compatible con la visualización en Tkinter. Si existe una variable `qr_bbox` definida (es decir, si ya se ha detectado un código QR), se importan las herramientas de `numpy` para transformar las coordenadas del contorno y se dibuja un polígono verde sobre el código QR detectado mediante `cv2.polylines`.

A continuación, se convierte la imagen procesada en un objeto `PhotoImage` compatible con Tkinter usando `ImageTk.PhotoImage`, y se actualiza el contenido del widget `lbl_cam` con la nueva imagen. Finalmente, se utiliza `after(20, update_camera)` para que la función se vuelva a ejecutar automáticamente después de 20 milisegundos, generando así un bucle de actualización continua del video.

```
def update_camera():
    global last_frame
    ret, frame = cap.read()
    if ret:
        frame_r = cv2.resize(frame, (DISPLAY_W, DISPLAY_H))
        last_frame = frame_r.copy()
        frame_rgb = cv2.cvtColor(frame_r, cv2.COLOR_BGR2RGB)
        if qr_bbox is not None:
            import numpy as np
            pts = qr_bbox.astype(int).reshape(-1, 1, 2)
            cv2.polylines(frame_rgb, [pts], True, (0, 255, 0), 2)
        img = ImageTk.PhotoImage(Image.fromarray(frame_rgb))
        lbl_cam.imgtk = img
        lbl_cam.config(image=img)
    lbl_cam.after(20, update_camera)
```

La función `evaluate_qr(info)` se encarga de analizar la fecha contenida en la información obtenida del código QR y decidir si el producto ha vencido.

Primero, se extrae el valor asociado a la clave `"Fecha"` desde el diccionario `info`. Luego, se intenta convertir esa fecha desde una cadena de texto con el formato `"%Y-%m-%d"` a un objeto de tipo `date` mediante `strptime()`. En caso de que la conversión falle (por ejemplo, si el formato no es válido), se imprime un mensaje de advertencia y se envían al Arduino dos comandos: uno indicando que el producto está vencido, y otro para devolver el brazo robótico a su posición estable mediante `HOME`.

Si el formato de la fecha es correcto, se compara contra la fecha actual usando `dt.date.today()`. Si la fecha del producto es posterior a la actual, se considera que el producto no está vencido y se envía el comando `NO_VENCIDO`. En caso contrario, se envía `VENCIDO`. Finalmente, sin importar el resultado, se envía nuevamente el comando `HOME` para que el sistema retorne a su estado de reposo.

```
def evaluate_qr(info):
    """Procesa la fecha y enva el comando apropiado."""
    fecha_txt = info.get("Fecha","")
    try:
```

```
        fecha_qr = dt.datetime.strptime(fecha_txt, "%Y-%m-%d").date()
    except ValueError:
        print(">>> Formato de fecha invlido:", fecha_txt)
        send_cmd("VENCIDO") # cuidador: tratar como vencido
        send_cmd("HOME")
        return
    hoy = dt.date.today()
    if fecha_qr > hoy:
        print(">>> Producto NO vencido (", fecha_txt, ")")
        send_cmd("NO_VENCIDO")
    else:
        print(">>> Producto VENCIDO (", fecha_txt, ")")
        send_cmd("VENCIDO")
    send_cmd("HOME")
```

La función `search_qr_loop()` ejecuta el proceso principal de búsqueda del código QR dentro del flujo de trabajo. Su propósito es intentar detectar un código QR en la imagen actual capturada por la cámara, y responder adecuadamente según los resultados.

Inicialmente, verifica si la variable global `scanning` está activada. Si no lo está, se detiene la ejecución. Si la imagen más reciente (`last_frame`) aún no ha sido capturada, se programa una reintento después de 100 milisegundos. Una vez disponible el fotograma, se llama a `decode_qr()` para intentar detectar un código QR. Si se detecta correctamente, se almacenan sus coordenadas en `qr_bbox`, se actualiza la interfaz mediante `set_labels(info)`, se detiene la búsqueda y se evalúa la información con `evaluate_qr()`.

Si no se detecta un QR, el sistema evalúa si ya se hizo un primer intento. Si no es así, espera 4 segundos antes de intentar nuevamente (con `attempt=1`). En caso de que ambos intentos fallen y el `stage` sea 0, se ejecuta la orden `SUBIR` mediante un hilo (`threading.Thread`), se espera 45 segundos y luego se realiza una nueva búsqueda (ahora con `stage=1`).

Si también falla la detección en el segundo `stage`, se concluye que no hay un código QR legible y se envía el comando `HOME`, deteniendo finalmente la búsqueda. Esta función permite una recuperación ordenada del sistema ante fallos de escaneo.

```
def search_qr_loop(stage=0, attempt=0):
    global scanning, qr_bbox, last_frame
    if not scanning:
        return
    if last_frame is None:
        root.after(100, lambda: search_qr_loop(stage, attempt))
        return

    info, pts = decode_qr(last_frame)
    if info:
        qr_bbox = pts
        set_labels(info)
        scanning = False
        print(">>> QR ENCONTRADO Informacin:", info)
```

```
        evaluate_qr(info)
        return

    print(">>> QR no encontrado (stage", stage, "attempt", attempt, ")")
    if attempt == 0:
        root.after(4000, lambda: search_qr_loop(stage, 1))
        return

    # fall el segundo intento
    if stage == 0:
        # --- pasar a SUBIR ---
        print(">>> Dos intentos fallidos  ejecutando SUBIR")
        def subir_worker():
            send_cmd("SUBIR")
            print(">>> Esperando 45 s tras SUBIR ")
            time.sleep(45)
            root.after(0, lambda: search_qr_loop(1, 0))
        threading.Thread(target=subir_worker, daemon=True).start()
    else:
        # stage 1 y tambin fall  HOME y terminar
        print(">>> No se detect QR tras SUBIR  HOME y fin")
        send_cmd("HOME")
        scanning = False
```

La función `task_iniciar()` representa el punto de inicio de la secuencia operativa para procesar una nueva caja. Su objetivo es preparar el sistema para la lectura de un nuevo código QR.

Primero, activa la variable global `scanning` y reinicia `qr_bbox` a `None` para eliminar posibles datos previos. También se limpia la información mostrada en pantalla usando `set_labels()`, estableciendo los campos en blanco.

Luego, se define una función interna llamada `worker()`, la cual es ejecutada en un hilo independiente usando `threading.Thread`. Esta función envía al Arduino el comando `CAJA` a través de `send_cmd()`, lo que indica que una nueva caja ha sido colocada para escaneo. A continuación, el sistema espera 30 segundos, permitiendo que el brazo robótico posicione correctamente la caja. Finalmente, se lanza la búsqueda del código QR mediante `search_qr_loop(0, 0)` usando `root.after()`, lo que permite mantener la interfaz reactiva.

Esta separación en hilos permite ejecutar tareas que consumen tiempo sin congelar la interfaz gráfica del usuario.

```
def task_iniciar():
    """Enva CAJA, espera 30 s y lanza la bsqueda de QR (stage 0)."""
    global scanning, qr_bbox
    scanning = True
    qr_bbox = None
    set_labels({k:"" for k in labels})

    def worker():
```

```
        send_cmd("CAJA")
        print(">>> Esperando 30 s tras CAJA")
        time.sleep(30)
        print(">>> Iniciando bsqueda de QR (stage 0)")
        root.after(0, lambda: search_qr_loop(0, 0))
    threading.Thread(target=worker, daemon=True).start()
```

La función `task_stop()` detiene inmediatamente cualquier proceso activo de escaneo y devuelve el brazo robótico a su posición inicial (comando `HOME`).

Para lograr esto, desactiva la variable global `scanning`, evitando que continúe la búsqueda del código QR, y reinicia `qr_bbox` a `None`, eliminando cualquier información previa sobre la ubicación del código QR.

Luego, envía el comando `"HOME"` al Arduino utilizando la función `send_cmd()`, lo cual hace que el brazo robótico regrese a su posición de reposo. Finalmente, limpia el panel de información llamando a `set_labels()` con campos vacíos.

Esta función es esencial para abortar procesos en curso y asegurar que el sistema vuelva a un estado estable.

```
def task_stop():
    """Detiene todo y lleva el brazo a HOME."""
    global scanning, qr_bbox
    scanning = False
    qr_bbox = None
    send_cmd("HOME")
    set_labels({k:"" for k in labels})
```

En este bloque se crean dos botones para la interfaz gráfica, utilizando `tk.Button`, cada uno con una funcionalidad clave: iniciar el proceso de detección de QR o detenerlo.

El primer botón, etiquetado como `ÏNICIAR"`, se configura con la fuente previamente definida (`btn_font`), un fondo con el color de énfasis definido en `ACCENT`, y texto blanco. Se ajustan los márgenes internos mediante `padx` y `pady`, se elimina el borde (`bd=0`) y se configura un color de fondo activo cuando el botón es presionado. Además, el cursor cambia a una mano al pasar el mouse, mejorando la experiencia de usuario. Al hacer clic, se ejecuta la función `task_iniciar`. Este botón se ubica en la primera fila, primera columna del contenedor `frame_btns`.

De manera similar, el segundo botón, titulado `"STOP"`, detiene el proceso en curso llamando a la función `task_stop`. Su estilo visual utiliza un fondo rojo (`#f53b4b`) y un color de fondo activo más oscuro para indicar una acción de cancelación. Se sitúa junto al botón anterior, en la primera fila pero en la segunda columna.

Ambos botones están diseñados para integrarse visualmente con el resto de la interfaz y responder inmediatamente a las acciones del usuario.

```
tk.Button(frame_btns, text="INICIAR", font=btn_font, bg=ACCENT, fg="white",
          padx=20, pady=10, bd=0, activebackground="#246f58", cursor="hand2",
          command=task_iniciar).grid(row=0, column=0, sticky="e", padx=(0,20))
tk.Button(frame_btns, text="STOP", font=btn_font, bg="#f53b4b", fg="white",
          padx=20, pady=10, bd=0, activebackground="#a82833", cursor="hand2",
          command=task_stop).grid(row=0, column=1, sticky="w", padx=(20,0))
```

Finalmente, este segmento representa el punto de entrada principal del programa gráfico. La función `update_camera()` se invoca primero para iniciar el ciclo continuo de captura de video desde la cámara IP y su visualización en la interfaz. Esta función se llama una única vez al comienzo, pero internamente se reinvoca a sí misma cada 20 milisegundos, lo que permite un flujo constante de imágenes.

Luego, se llama a `root.mainloop()`, que es el bucle principal de eventos de la interfaz gráfica en Tkinter. Esta instrucción mantiene la ventana abierta y en espera de eventos del usuario, como clics, teclas o acciones del sistema. Sin esta línea, la interfaz gráfica se cerraría inmediatamente después de crearse. Este bucle garantiza que la aplicación sea interactiva y se mantenga en ejecución hasta que el usuario la cierre manualmente.

```
update_camera()
root.mainloop()
```

9. Resumen de las referencias consultadas

[1] Reporte sobre fatiga y lesiones musco-esqueléticas en trabajadores que realizan carga de productos.

[2] Información acerca del subsector de almacenamiento y depósito.

[3] Automatización de almacenes para mejorar problemas laborales.

[4] Rol, historia, cambios y oportunidades que brinda la automatización.

[5] Cambio de la gestión de inventario que brindó el código QR.

[6] Teoría de servomotores, PWM, Arduino UNO, motores, software y comunicación entre piezas.

[7] Librerías, herramientas y ejemplos para el Arduino UNO.

[8] Movimiento cinemático y su espacio de trabajo de un brazo robótico.

[**9**] El modelo de Denavit-Hartenberg, utilizado para representar matemáticamente los movimientos de brazos robóticos.

[**10**] Análisis cinemático de un brazo robótico de 6 DOF con muñeca esférica.

[**11**] Información acerca de qué es OpenCV.

[**12**] Documentación de herramientas de OpenCV para procesamiento de imágenes en Python.

[**13**] Cómo detectar y clasificar figuras geométricas en imágenes usando OpenCV.

[**14**] Anatomía y riesgos en los códigos QR.

[**15**] Definición, versiones, partes, lectura y generación de códigos QR.

[**17**] Descripción de la comunicación USB-Serial para control de dispositivos.

10. Conclusiones

- La automatización mediante robótica y visión artificial representa una solución eficaz para reducir riesgos ergonómicos y aumentar la eficiencia en centros de distribución, especialmente frente a las altas tasas de lesiones laborales y los costos asociados a la manipulación manual en sectores logísticos de alto volumen.
- El uso del microcontrolador Arduino UNO Mini en el proyecto se justifica por su bajo costo, facilidad de uso y capacidad para generar señales PWM estables. Además, su comunicación USB permite integrarlo fácilmente con el software de visión por computadora, convirtiéndolo en una solución eficiente para controlar servomotores en tiempo real a partir de las instrucciones procesadas desde la cámara.
- La integración de códigos QR estructurados en formato JSON permite identificar de forma precisa y automatizada cada caja, facilitando el procesamiento de información por el sistema de visión y reduciendo errores manuales desde el inicio de la operación.
- OpenCV nos brinda una herramienta práctica para la detección de figuras y formas, lo cual abre amplias posibilidades de aplicación en nuevas tecnologías. Además, su implementación en un lenguaje accesible como Python elimina barreras para quienes inician en la programación.
- El diseño físico estandarizado de la caja, con alto contraste visual y dimensiones controladas, facilita el reconocimiento visual confiable y la manipulación precisa por parte del brazo robótico. Al emplear una vista isométrica, se permite observar simultáneamente tres caras del cubo, lo que reduce la cantidad de escaneos necesarios y mejora la eficiencia en la detección de los códigos QR, sin necesidad de visión tridimensional.
- La lógica de rotación combinada con la lectura secuencial de tres caras visibles por vista y una única rotación adicional permite escanear completamente el cubo. Esto optimiza el tiempo de ciclo del sistema, logrando una clasificación ágil y eficaz de las cajas según su destino, todo con componentes accesibles y de bajo costo.

- La ejecución del proyecto requirió la integración de conocimientos en electrónica, programación embebida, visión artificial y diseño de algoritmos, demostrando el potencial de proyectos interdisciplinarios para resolver problemas reales de forma creativa y escalable.
- La estructura modular del sistema permite extender fácilmente sus capacidades para incluir más categorías, usar diferentes tipos de sensores o incorporar redes neuronales para una clasificación más avanzada en futuras versiones.
- Todos los componentes utilizados son de bajo costo y ampliamente disponibles, lo cual permite replicar el sistema en otros entornos educativos o industriales sin necesidad de equipamiento especializado.
- El sistema puede beneficiarse en el futuro de mejoras como una base giratoria automática, mayor precisión en el control de servos, o la integración de algoritmos de aprendizaje automático para reconocer patrones más complejos.
- Aunque se diseñó inicialmente para clasificar productos según su fecha de vencimiento, la misma lógica puede adaptarse a otros procesos industriales como separación por tipo de material, destino logístico o estado del producto (nuevo/usado).

11. Referencias

1. Georgia. (2025, 17 enero). The True Cost of Manual Handling Injuries: HSE 2024 Report Insights. Master Mover.
https://www.mastermover.com/blog/hse-report-insights-2024

2. U.S. Bureau of Labor Statistics. (2022, 5 octubre). *Industries at a Glance: Warehousing and Storage: NAICS 493*.
https://www.bls.gov/iag/tgs/iag493.htm

3. Industry Week. (2019, 21 junio). *60 % of companies plan warehouse automation to ease labor woes*.
https://www.industryweek.com/talent/article/22027798/60-of-companies-plan-warehouse-automation-to-ease-labor-woes

4. psico-smart.com. (s. f.). *The Impact of Automation on Labor Productivity Management: Challenges and Opportunities*.
https://psico-smart.com/en/blogs/blog-the-impact-of-automation-on-labor-productivity-management-challenges-and-opportunities-164926

5. Johns, S. (2025, 3 abril). *How Have QR Codes Changed Inventory Tracking and Management?* Global Trade Magazine.
https://www.globaltrademag.com/how-have-qr-codes-changed-inventory-tracking-and-management/

6. Lopez, J. (2024). *A New Approach to Robot Motor Control*. FERMILAB-PUB-24-0440 STUDENT.
https://lss.fnal.gov/archive/2024/pub/fermilab-pub-24-0440-student.pdf

7. Arduino (2025). Arduino UNO Documentation.
https://docs.arduino.cc/hardware/uno-rev3/

8. Iqbal, J., Islam, R. U., & Khan, H. (2012). *Modeling and analysis of a 6 DOF robotic arm manipulator*.
https://www.researchgate.net/publication/280643085_Modeling_and_analysis_of_a_6_DOF_robotic_arm_manipulator

9. Wikipedia contributors. (n.d.). *Denavit–Hartenberg parameters*. Wikipedia.
https://en.wikipedia.org/wiki/Denavit%E2%80%93Hartenberg_parameters

10. Asif, S., & Webb, P. (2021). *Kinematics analysis of 6-DoF articulated robot with spherical wrist*. Mathematical Problems in Engineering.
https://doi.org/10.1155/2021/6647035

11. OpenCV. (2020, November 4). *About - OpenCV*.
https://opencv.org/about/

12. OpenCV documentation. *Image processing in OpenCV*.
https://docs.opencv.org/4.x/d2/d96/tutorial_py_table_of_contents_imgproc.html

13. Rosebrock, A. (2021, July 7). *OpenCV shape detection - PyImageSearch*.
https://pyimagesearch.com/2016/02/08/opencv-shape-detection/

14. Ruoti, S. (2022, 29 mayo). *Cómo funcionan los códigos QR y qué peligros suponen para tu celular*. BBC News Mundo.
https://www.bbc.com/mundo/noticias-61084121

15. Kaspersky. (2020, September 8). *¿Qué es un código QR y cómo puedo leerlo?*.
https://www.kaspersky.es/resource-center/definitions/what-is-a-qr-code-how-to-scan

16. Wikipedia contributors. (n.d.). *Universal asynchronous receiver-transmitter*.
https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter