ITESO, Universidad Jesuita de Guadalajara

PRÁCTICA 1: CALIBRACIÓN CÁMARA

Implementación de detección, decisión y evasión con control cinemático en tiempo real

Luis Eduardo Diaz Macias, Pablo Perez Sanchez, Miguel de Jesus Flores Gonzalez & Jesus Alejandro Ocegueda Melin

Título de la práctica	Práctica 1
Autor o Responsable	Luis Eduardo Diaz Macias, Pablo Perez Sanchez, Miguel de Jesus Flores Gonzalez & Jesus Alejandro Ocegueda Melin
Asignatura	PAP
Fecha de entrega	24 de octubre de 2025
Objetivo	Desarrollar un sistema autónomo reactivo para la evitación de obstáculos mediante la integración del sensor LiDAR C1 con el control cinemático del robot. Validar la detección, decisión y actuación del sistema bajo baja latencia.
Materiales y Recursos	• Cámara web con cable USB • Trípode • Tablero de ajedrez impreso (B/N) • Códigos ArUco (impresos) • Carrito del laboratorio • Perfil BOSCH de 20 • Piezas impresas en 3D • Tornillos M5 × 7, rondanas M5 × 7 • Tuercas para ranura tipo T × 7 • Tornillo M6, tuercas M6 × 2 • Extensión USB A-A (hembra–macho) de al menos 3
Duración Estimada	8 horas
Lugar o Modalidad	Laboratorio de Mecatrónica

Resumen— Esta práctica guía al estudiante en el proceso de calibración intrínseca de una cámara web mediante un patrón de ajedrez y sienta las bases para su integración posterior con marcadores ArUco. Se cubre la instalación del entorno, la captura de imágenes, la estimación de parámetros intrínsecos y recomendaciones de calidad para garantizar resultados reproducibles en un contexto educativo.

---

1. Introducción

La implementación de sistemas de visión artificial en tareas de automatización y robótica requiere una comprensión precisa de la relación geométrica entre el sensor de la cámara y el entorno físico. Este documento técnico detalla el proceso completo para calibrar una cámara web y establecer un sistema de coordenadas tridimensionales (3D) en un área de trabajo definida, sentando las bases para la localización y seguimiento de un vehículo a escala.

El núcleo de este trabajo radica en la calibración de la cámara, que se divide en dos componentes fundamentales:

1. Calibración Intrínseca: Se enfoca en los parámetros internos de la cámara (distancia focal, punto principal y coeficientes de distorsión del lente). Su objetivo es corregir las distorsiones ópticas para que las líneas rectas en el mundo real se proyecten como líneas rectas en la imagen, permitiendo que las mediciones en píxeles se conviertan de manera precisa a unidades métricas.

2. Calibración Extrínseca: Se encarga de definir la pose de la cámara (su posición y orientación) respecto a un sistema de coordenadas de referencia fijo en el mundo real. Esto es crucial para ubicar objetos y el vehículo en un sistema de coordenadas global.

Para lograr esta localización espacial, emplearemos los marcadores fiduciales ArUco. Estos patrones bidimensionales son una herramienta estándar en visión por computadora, ya que permiten la detección rápida y precisa, así como la estimación de pose de objetos, facilitando el establecimiento del origen y los ejes del sistema de coordenadas del área de trabajo.

Los capítulos siguientes guiarán al lector a través de cada etapa de la implementación en Python, utilizando la librería OpenCV: desde la captura de imágenes para la obtención de la Matriz Intrínseca y los Coeficientes de Distorsión, la generación y detección de los marcadores ArUco, hasta el cálculo final de los Parámetros Extrínsecos (R y t).

Al finalizar este documento, el lector comprenderá la metodología necesaria para transformar coordenadas de píxeles en coordenadas 3D del mundo real, un requisito indispensable para el desarrollo de cualquier aplicación de seguimiento o navegación basada en visión.

2. Capítulo 0: Instalación de Python y librerías

Para este proyecto se utilizará Python como lenguaje de programación. Primero, necesitas descargar Python desde su página oficial.

Python e IDE

Instala Python de su sitio oficial de Python y selecciona la versión correspondiente a tu sistema operativo.

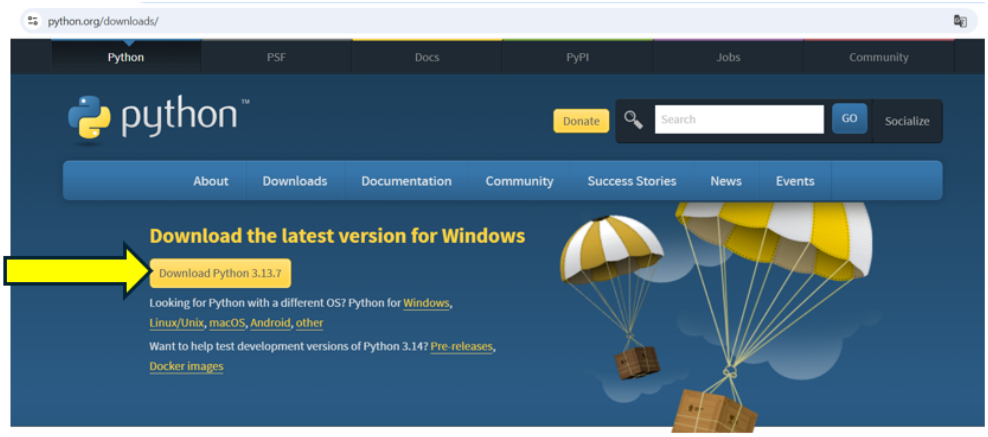

Figura 1: Sitio oficial de python.

En este proyecto utilizamos Visual Studio Code (VS Code) como editor de código, pero puedes elegir cualquier otro editor que prefieras.

Si decides usar VS Code, descárgalo desde el siguiente enlace: Descargar Visual Studio Code.

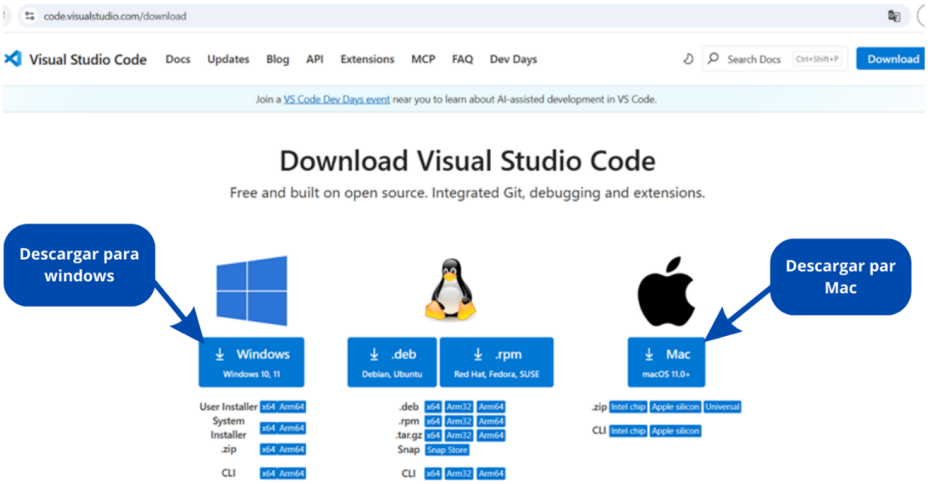

Figura 2: Descarga de Visual Studio Code.

Librerías necesarias

Una vez instalado Python y el editor de código, será necesario instalar las siguientes librerías para trabajar con la calibración de la cámara. Las librerías principales que vamos a utilizar son:

- OpenCV (librería y módulos "contrib")
- NumPy

Para instalarlas, abre la terminal y escribe lo siguiente:

Instalación de librerías

```
pip install opencv-contrib-python numpy
```

3. Capítulo 1: Armado de la estructura de la cámara

Demostración con pasos de como armar la estructura de la cámara y como colocarla en su lugar.

1. Base de fijación
    - Las 5 piezas de SolidWorks son las que sujetan los perfiles BOSCH de 20 mm.
    - Dos piezas se usan como soporte inferior para fijar los perfiles laterales a los trípodes.
    - Dos piezas se colocan arriba de los laterales para unirlos con el perfil horizontal.
    - La quinta pieza es la base de la cámara, que se coloca en el centro del perfil superior.

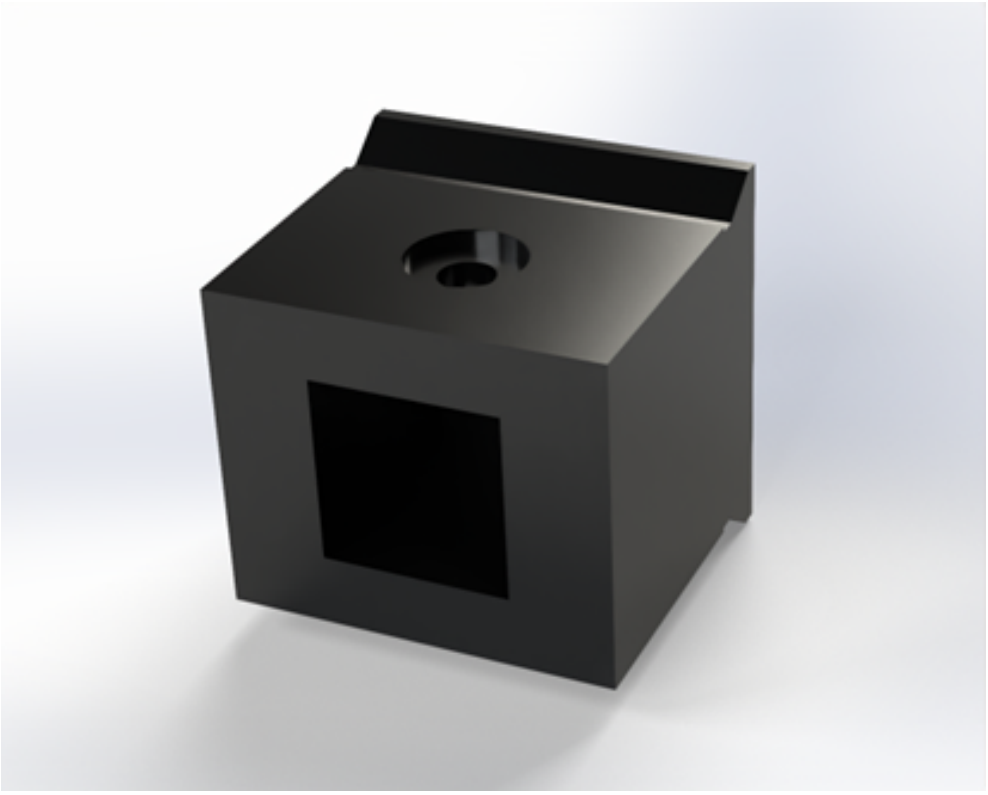

Figura 3: Soporte inferior para fijar los perfiles al trípode.

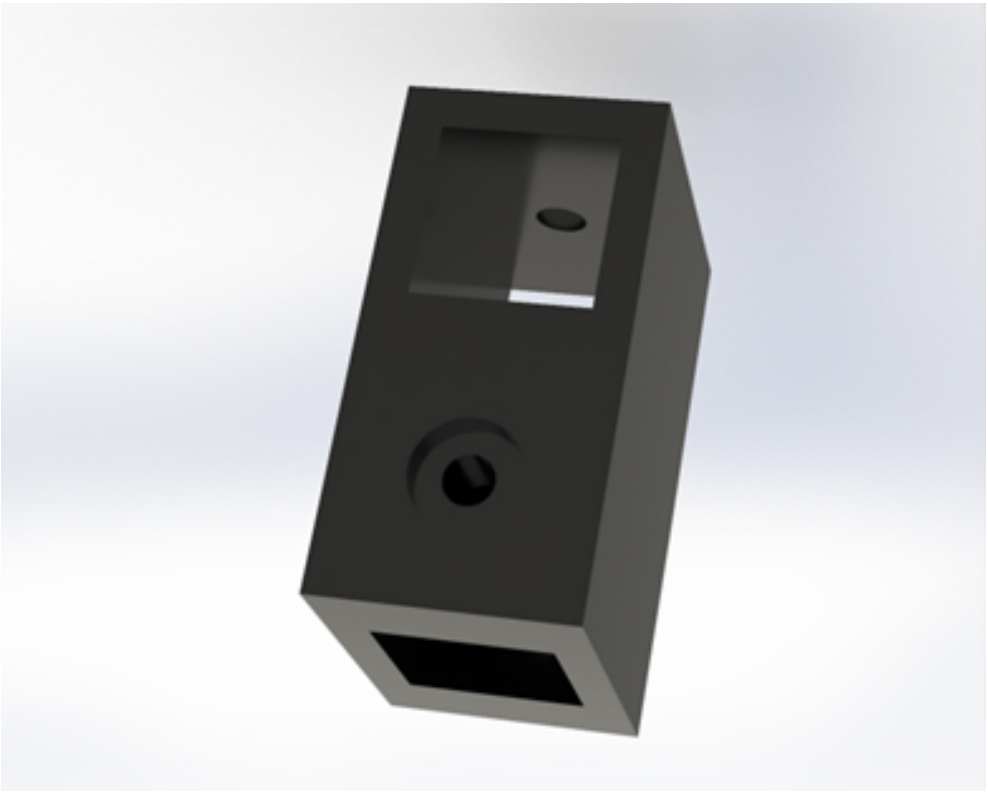

Figura 4: Unir perfiles laterales con horizontal.

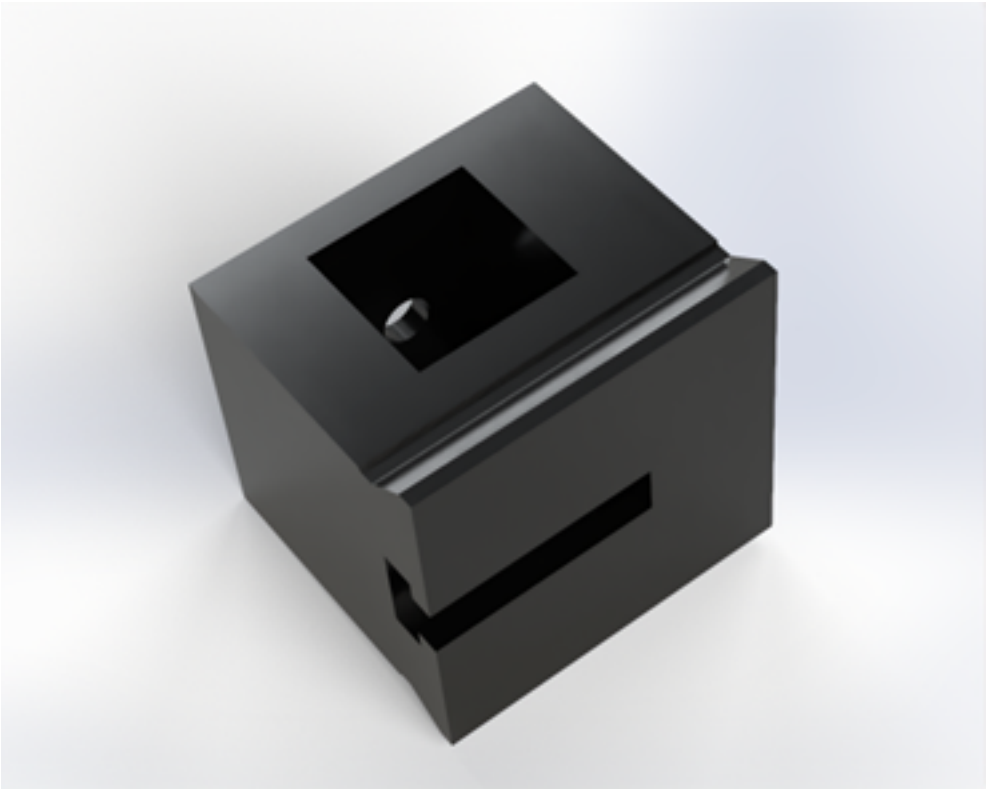

Figura 5: Base de la cámara.

2. Perfiles de aluminio (BOSCH 20 mm)
    - Coloca los dos perfiles laterales en vertical, cada uno montado sobre un trípode.
    - Une ambos laterales con el perfil horizontal superior (más largo), formando un “puente”.

Figura 6: Perfil de aluminio 20mm.

3. Montaje de la cámara
    - En la pieza superior central (impresa en 3D) se fija la cámara web, apuntando hacia abajo.
    - Asegúrate de que la cámara quede a unos 3 m de altura para maximizar el campo de visión.

Figura 7: Trípode.

Figura 8: Ensamble cámara con pieza impresa en 3D.

4. Ajustes finales
    - Usa la tornillería M5 y M6 (tornillos, tuercas T y rondanas) para fijar los perfiles dentro de las piezas de 3D.
    - Conecta la cámara a la extensión USB de 3 m para poder trabajar cómodamente desde la computadora.
    - Ajusta la altura y nivelación con los trípodes antes de iniciar la calibración.

4. Capítulo 2: Toma de fotos del tablero de ajedrez

Una vez instalado los programas necesarios y ensamblada la estructura para la cámara, podemos iniciar la primera etapa de la calibración de la cámara web.

Imprime el patrón de ajedrez en blanco y negro (puede ser en cualquier tamaño de hoja, como se muestra en la figura 9). Lo crucial es el alto contraste entre los cuados. Se recomiendo que el tablero sea minimo de 8x7 cuadros (es decir, 7 x 6 esquinas internas).

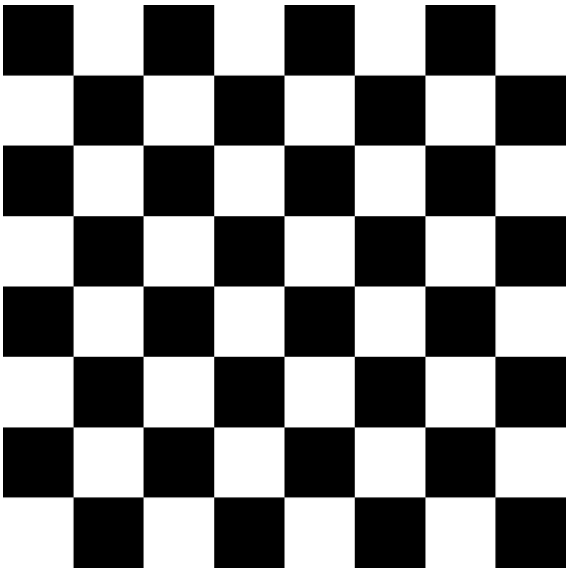

Figura 9: Ejemplo de patrón.

Del patrón, cuenta las esquinas internas de cada lado (los puntos de intersección). Además, mide longitud del lado de un cuadro del patrón en metros. Como se muestra en la siguiente figura 10:

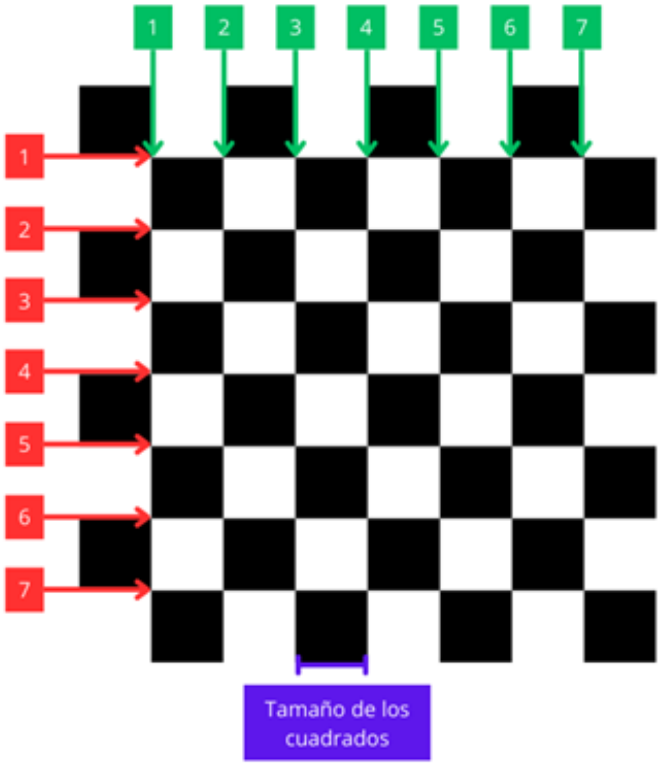

Figura 10: Información que se necesita sacar del patrón.

Con el patrón de ajedrez impreso y los datos dimensionales listos, se procede al código para la captura de las imágenes. Estas fotos se utilizarán para calcular la matriz intrínseca de la cámara.

El código permite capturar imágenes desde la cámara web presionando el botón de espacio. Cada vez que se toma una foto, esta se guarda en una carpeta llamada `Fotos_Calibracion`, ubicada en la misma dirección que el script.

Código de captura

Listing 1: Captura de imagenes para calibracion.

```
import cv2
import Funciones_Codigos as fc

# Resolucion recomendada
resolucion_width  = 1280
resolucion_height = 720
fc.guardar_Resolucion_Camara(resolucion_width, resolucion_height)

# Crear carpeta para fotos (por defecto 'Foto_Calibracion')
fc.crear_Carpeta_Fotos()

# Abrir camara y fijar resolucion
camara_web = cv2.VideoCapture(1)
camara_web.set(cv2.CAP_PROP_FRAME_WIDTH,  resolucion_width)
camara_web.set(cv2.CAP_PROP_FRAME_HEIGHT, resolucion_height)

if not camara_web.isOpened():
print("No se puede abrir la camara")
raise SystemExit

contador_fotos = 0
print("Presiona 'SPACE' para tomar una foto, 'ESC' para salir.")
```

```
23
24          while True:
25          ret, imagen = camara_web.read()
26          if not ret:
27          print("No se puede recibir imagen. Saliendo ...")
28          break
29
30          cv2.imshow('Presiona SPACE para tomar foto', imagen)
31          key = cv2.waitKey(1)
32
33          if key == 27:       # ESC
34          break
35          elif key == 32:     # SPACE
36          fc.guardar_Foto_Calibracion(imagen, contador_fotos)
37          contador_fotos += 1
38
39          camara_web.release()
40          cv2.destroyAllWindows()
41          print(f"Se guardaron {contador_fotos} fotos en: {fc.obtener_Nombre_Carpeta_Fotos()} "
42          f"({resolucion_width}x{resolucion_height}).")
```

Explicación del código

Importamos la librería de Open CV y también el módulo de funciones con funciones secundarias que ayudan al código.

```
        import cv2
        import Funciones_Codigos as fc
```

Se define la resolución de la cámara, esto es importante dado que la calibración depende de estos valore.

> ¡Advertencia!
>
> Si se modifica la resolución, la calibración existente dejará de ser válida y se deberá repetir el proceso.

```
        # Se especifica la resolucion de la camara (se recomineda el valor de 1280x720)
        resolucion_width = 1280
        resolucion_height = 720
        fc.guardar_Resolucion_Camara(resolucion_width, resolucion_height)
```

En estos puntos se crea la carpeta donde se van a guardar las fotos que se tomen y esta carpeta se va a guardar en la dirección donde esta guardado el código.

```
        # (Por default el nombre de la carpeta es: 'Foto_Calibracion').
        fc.crear_Carpeta_Fotos()
```

Se inicializa la cámara y se ajusta la resolución para que este en los valores establecidos por ustedes. Después se evalúa si se puede abrir y si no se puede se sale del programa.

```
        # Iniciamos la camara y ajustamos la resolucion.
        camara_web = cv2.VideoCapture(1)    camara_web.set(cv2.CAP_PROP_FRAME_WIDTH,
            resolucion_width)
        camara_web.set(cv2.CAP_PROP_FRAME_HEIGHT, resolucion_height)

        # Se verifica si se abrio la camara, si no se sale del programa.
        if not camara_web.isOpened():
        print("No se puede abrir la camara")
        exit()
```

Esta la parte principal del código la cual hace que se muestre a pantalla la imagen que está viendo la cámara para observar donde poner el tablero de ajedrez. Una vez presionado el botón de espacio se guardará la foto en la carpeta designada, podrás tomar las fotos que creas necesario y una vez que ya no quieras tomar más fotos se debe de presionar el botón de ESC.

```
print("Presiona 'SPACE' para tomar una foto, 'ESC' para salir.")
while True:
ret, imagen = camara_web.read()

if not ret:
print("No se puede recibir imagen. Saliendo ...")
break

# Mostrar el video en una ventana
cv2.imshow('Presiona SPACE para tomar foto', imagen)
key = cv2.waitKey(1)

if key == 27:  # ESC para salir
break
elif key == 32:  # SPACE para tomar foto
fc.guardar_Foto_Calibracion(imagen, contador_fotos)
contador_fotos += 1
```

Una vez dado el botón de ESC se liberará la cámara y se cerrara la ventana, lo siguiente es una impresión a consola sobre la cantidad de fotos tomadas, en donde se guardaron y la resolución con la cual se tomaron las fotos.

```
# Liberar la camara y cerrar ventanas
camara_web.release()
cv2.destroyAllWindows()

print(f"Se guardaron {contador_fotos} fotos en la carpeta:
    {fc.obtener_Nombre_Carpeta_Fotos()} con una resolucion de
    {resolucion_width}x{resolucion_height}.")
```

Recomendaciones

- Mover el tablero por **todo** el rango de visión de la cámara: centro: esquinas, bordes. Es **fundamental variar su posición y orientación**.
- Toma fotos desde distintos **ángulos de inclinación** (no solo de frente), así como **acerca y aleja** el patrón para obtener diversas perspectivas.
- Asegura que **todas las esquinas internas** del tablero sean **visibles y nítidas** en la imagen, ya que son los puntos clave que utiliza el algoritmo de calibración.
- Utilizar un patrón **bien impreso**, con alto contraste, montado sobre una **superficie plana y rígida** para evitar curvaturas o arrugas.
- Las fotos deben de estar **bien enfocadas**. Evitar sombras, reflejos o brillos que puedan oscurecer u ocultar las esquinas.
- Se recomienda tomar **al menos 15 fotos**. Cuantas más imágenes diversas y de alta calidad se capturen, más precisa resultara la calibración.

5. Capítulo 3: Calibración intrínseca (matriz y distorsión)

Ya con las fotos en la carpeta se puede pasar a la calibración intrínseca de la cámara, para esto se utiliza el siguiente código de Python.

Código de calibración

Listing 2: Estimacion de parametros intrinsecos con OpenCV.

```
1 import cv2
```

```
        import numpy as np
        import Funciones_Codigos as fc

        tamano_patron   = (7, 7)      # esquinas internas (no cuadros)
        long_cuadro_m   = 0.024       # lado del cuadro en metros

        # Puntos 3D del patron en el mundo
        objp = np.zeros((tamano_patron[0]*tamano_patron[1], 3), np.float32)
        objp[:, :2] = np.mgrid[0:tamano_patron[0], 0:tamano_patron[1]].T.reshape(-1, 2)
        objp *= long_cuadro_m

        objpoints = []   # puntos 3D
        imgpoints = []   # puntos 2D

        imagenes = fc.filtrar_Archivos_JPG()
        for fname in imagenes:
        img  = cv2.imread(fname)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        ok, esquinas = cv2.findChessboardCorners(gray, tamano_patron, None)
        if ok:
        objpoints.append(objp)
        imgpoints.append(esquinas)

        cv2.drawChessboardCorners(img, tamano_patron, esquinas, ok)
        cv2.imshow('Deteccion', img)
        cv2.waitKey(100)

        cv2.destroyAllWindows()

        # Calibracion
        ret, K, dist, rvecs, tvecs = cv2.calibrateCamera(
        objpoints, imgpoints, gray.shape[::-1], None, None
        )

        print("Matriz intrinseca K:\\n", K)
        print("Coeficientes de distorsion:\\n", dist.ravel())
        print("Error de reproyeccion:", ret)

        fc.guardar_Parametros_Camara(K, dist)
        print("Parametros guardados en JSON.")
```

Explicación del código

Importamos las librerías necesarias las cuales son: open cv, numpy y la Funciones_Codigo para partes secundarías del código.

```
        import cv2
        import numpy as np
        import Funciones_Codigos as fc
```

En esta sección se tienen que establecer los parámetros necesarios para la calibración. Estos parámetros son el número de esquinas internas del patrón impreso y la longitud de un lado de un cuadro.

```
        tamano_patron = (7, 7) # Dimensiones del patron (esquinas internas horizontales,
            esquinas internas verticales)
        longitud_cuadro = 0.024  # Longitud del lado de cada cuadro (en metros)
```

Esta sección es para preparar los puntos del patrón de ajedrez, para esto se crea un array de ceros del tamaño del múltiplo de las esquinas internas (Por ejemplo, el tablero antes creado sería una multiplicación de 7x7). De este array, creamos una malla con todas las coordenadas de las esquinas. Aunque se crea con 3 dimensiones (x, y, z), el componente z

se mantiene a cero, ya que asumimos que el patrón se encuentra en el plano z = 0. Al final es escalar las coordenadas por el tamaño real de los cuadros, creando así un array con los puntos físicos de las esquinas internas del patrón de ajedrez.

```
# Preparar coordenadas del patron (puntos 3D en el mundo real)
objp = np.zeros((tamano_patron[0] * tamano_patron[1], 3), np.float32)
objp[:, :2] = np.mgrid[0:tamano_patron[0], 0:tamano_patron[1]].T.reshape(-1, 2)
objp *= longitud_cuadro # Escalar segun el tamano real de los cuadros
```

Se crea dos variables para los puntos en 3D (mundo real) y uno para 2D (la imagen). También se llama a la función filtrar_Archivos_JPG para guardar en la variable imágenes de todas las fotos tomadas del tablero ajedrez visto en el capítulo anterior.

```
objpoints = [] # Puntos 3D en el mundo real
imgpoints = [] # Puntos 2D detectados en la imagen

# Filtrar archivos .jpg dentro de esa carpeta
imagenes = fc.filtrar_Archivos_JPG()
```

En este for es donde se procesan las imágenes y hacemos que Open CV detecte las esquinas internas del patrón y las guarde en la variable esquinas, esto se logra gracias a la función findChessboardCorners. Esta función requiere de dos parámetros, la primera es la imagen, el segundo es el tamaño del patrón (número de esquinas internas). Lo que retorna es una booleano siendo True si se identificaron y ordenaron de manera correcta las esquinas. Lo segundo que retorna es una matriz con las coordenadas 2D en pixeles de las esquinas.

```
for fname in imagenes:
img = cv2.imread(fname)
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Encontrar esquinas del tablero
ret, esquinas = cv2.findChessboardCorners(gray, tamano_patron, None)
```

Si la función logra identificar las esquinas, se guarda la matriz de coordenadas 2D en la variable imgpoints y en la variable de objpoints se guarda las coordenadas de los puntos reales. Luego se muestra a pantalla las esquinas que detecto Open CV para revisión del usuario y confirmar que se haya hecho de manera correcta.

```
if ret:
objpoints.append(objp)
imgpoints.append(esquinas)

# Dibujar las esquinas encontradas
cv2.drawChessboardCorners(img, tamano_patron, esquinas, ret)
cv2.imshow('Deteccion', img)
cv2.waitKey(100)

cv2.destroyAllWindows()
```

Una vez obtenidas todos los puntos 3D y 2D de la imagen pasamos a la calibración de la cámara, lo cual se hace con la función calibrateCamera. Este requiere de la matriz de puntos 3D, matriz de puntos 2D, y el tamaño de la imagen en pixeles (se usa gray.shape[::-1] para obtener las dimensiones (ancho, alto), que es el formato requerido por la función). Lo que nos devuelve son: el error de re-proyeccion, la matriz intrínseca (K), los coeficientes de distorsión y los vectores de rotación (rvecs) y traslación (tvecs) para cada imagen. Solo usaremos los tres primeros parámetros para el siguiente paso.

```
# Calibrar la camara
ret, matriz_camara, dist_coeffs, rvecs, tvecs = cv2.calibrateCamera(
objpoints, imgpoints, gray.shape[::-1], None, None )
```

Se manda la imprimir la matriz, los coeficientes y el error. Para después, guardar estos parámetros en un archivo tipo .json, todo esto se hace en la función guardar_Parametros_Camara lo cual pide dos variables, la matriz intrínseca y los coeficientes de distorsión.

```
print(f"Matriz de camara (intrinseca): \n {matriz_camara} \n")
print(f"Coeficientes de distorsion:\n {dist_coeffs}")
print(f"Reporjection error: {ret}")
```

```
fc.guardar_Parametros_Camara(matriz_camara, dist_coeffs)
print("Se guardaron los parametros en el archivo json")
```

Explicación de la matriz intrínseca, coeficientes de distorsión y re-proyección.

A. Matriz Intrínseca de la Cámara (K)

Esta matriz, a menudo llamada Matriz de la Cámara, contiene los parámetros geométricos internos de la cámara que son fijos una vez que la cámara está fabricada y no dependen de la posición del mundo.

Propósito: Proyecta puntos 3D del mundo real en coordenadas 2D de la imagen (píxeles).

Elementos clave (y su estructura):

$$K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

- f_x y f_y: Son las distancias focales de la cámara, medidas en unidades de píxeles (no en milímetros). Representan qué tan "cercada" o "alejada" se ve la escena.
- c_x y c_y: Son las coordenadas del punto principal (o centro óptico), también en píxeles. Representan el punto exacto en la imagen donde el eje óptico de la cámara la intersecta. Idealmente, este punto está en el centro geométrico del sensor.

B. Coeficientes de Distorsión

La distorsión es un efecto óptico que hace que las líneas rectas en el mundo real aparezcan curvas o dobladas en la imagen, especialmente cerca de los bordes.

Propósito: Modelar la distorsión del lente de la cámara.

Tipos de Distorsión modelados:

- Distorsión Radial (k_1, k_2, k_3): Hace que las líneas se curven hacia adentro (distorsión de barril) o hacia afuera (distorsión de cojín).
- Distorsión Tangencial (p_1, p_2): Ocurre porque el lente y el sensor no están perfectamente paralelos, lo que resulta en una inclinación.

Estructura del Vector de Coeficientes: Generalmente un vector de 5 (o más) valores: [k_1, k_2, p_1, p_2, k_3]. Los valores más cercanos a cero indican menor distorsión.

C. Error de Re-proyección

Este valor es la métrica más importante para evaluar la precisión de la calibración.

Concepto: Después de calcular la Matriz Intrínseca y los Coeficientes de Distorsión, el algoritmo de calibración usa estos parámetros para re-proyectar los puntos 3D conocidos del patrón de ajedrez (mundo real) en las imágenes 2D.

Cálculo: El error de re-proyección es la diferencia promedio (en píxeles) entre:

1. La posición real de la esquina en la foto (el imgpoints que detectó Open CV).
2. La posición donde el algoritmo predice que debería estar esa esquina usando los parámetros calculados.

Interpretación: Un menor error de re-proyección implica una mejor calibración. Un valor generalmente menor a 1.0 píxel se considera una calibración muy buena.

6. Capítulo 4: Generación y detección de códgios ArUco

Para la detección precisa del carro y de la zona de trabajo se emplearán marcadores ArUco. Estos son códigos fiduciales bidimensionales (2D) con patrones binarios en blanco y negro, ampliamente utilizados en visión de computadora para tareas de seguimiento, localización y detección de pose de objetos. Open CV integra una librearía específica para la generación y detección eficiente de marcadores ArUco. A continuación, se presentan los códigos Python para su generación y posterior detección. Posteriormente, se detallará el funcionamiento de ambos códigos.

Código para genera los códigos ArUco

Listing 3: Generacion marcadores ArUco

```
import cv2.aruco as aruco
import matplotlib.pyplot as plt
import Funciones_Codigos as fc

# Seleccionar el diccionario de marcadores de ArUco
# DICT_4x4_50, DICT_5x5_100, DICT_6x6_250, DICT_7x7_1000, DICT_ARUCO_ORIGINAL -> 1024
aruco_dict = aruco.getPredefinedDictionary(aruco.DICT_4X4_50)

id_codigos = [1, 2, 3, 4, 5]   # ID de los codigos ArUco
tamano_codigo = 300             # tamano en pixeles

for x in id_codigos:
imagen_codigo = aruco.generateImageMarker(aruco_dict, x, tamano_codigo)

# Muestra el marcador con matplotlib
plt.imshow(imagen_codigo, cmap='gray')
plt.axis('off')
plt.title(f'ArUco ID {x}')
plt.show()

# Guardar el marcador como imagen PNG
fc.guardar_Codigo_ArUco(imagen_codigo, x)
```

Explicación del código

Este bloque selecciona el conjunto o diccionario de marcadores ArUco que se utilizará. El diccionario aruco.DICT_4X4_50 especifica que los marcadores tendrán una matriz interna de 4×4 bits y que el diccionario contiene un total de 50 identificadores únicos. Esta elección es crucial porque define la estructura y el número máximo de marcadores que se pueden generar.

```
# Seleccionar el diccionario de marcadores de ArUco
aruco_dict = aruco.getPredefinedDictionary(aruco.DICT_4X4_50)
```

Aquí se definen los parámetros esenciales para la generación. La lista id_codigos especifica los identificadores numéricos únicos (ID) que se asignarán a cada marcador que se desee crear. La variable tamaño_codigo establece el tamaño deseado para la imagen de salida del marcador, medida en píxeles (en este caso, 300x300 píxeles).

```
id_codigos = [1, 2, 3, 4, 5]   # ID de los codigos ArUco
tamano_codigo = 300             # tamano en pixeles
```

Este bucle for itera sobre cada ID definido para generar el marcador correspondiente. La función aruco.generateImageMarker crea el marcador utilizando el diccionario y el tamaño especificados. Luego, se utiliza matplotlib para mostrar una vista previa de cada marcador generado, incluyendo su ID en el título. Finalmente, la función fc.guardar_Codigo_ArUco se encarga de guardar cada marcador como un archivo de imagen (PNG) en el sistema, permitiendo su impresión y uso posterior.

```
for x in id_codigos:
imagen_codigo = aruco.generateImageMarker(aruco_dict, x, tamano_codigo)

# Muestra el marcador con matplotlib
plt.imshow(imagen_codigo, cmap='gray')
plt.axis('off')
plt.title(f'ArUco ID {x}')
plt.show()

# Guardar el marcador como imagen PNG
fc.guardar_Codigo_ArUco(imagen_codigo, x)
```

Código para detectar los códigos ArUco

Listing 4: Detección de los marcadores ArUco

```
import cv2
import cv2.aruco as aruco
import numpy as np
import Funciones_Codigos as fc

parametros_camara = fc.obtener_Parametros_Camara()
matriz_camara = np.array(parametros_camara["matriz_camara"])
coef_dist = np.array(parametros_camara["coef_dist"])

aruco_dict = aruco.getPredefinedDictionary(aruco.DICT_4X4_50)
parameters = aruco.DetectorParameters()
detector = aruco.ArucoDetector(aruco_dict, parameters)

longitud_marcador = 0.143

camara = cv2.VideoCapture(1)

camara.set(cv2.CAP_PROP_FRAME_WIDTH, parametros_camara["resolucion_width"])
camara.set(cv2.CAP_PROP_FRAME_HEIGHT, parametros_camara["resolucion_height"])

while True:
ret, imagen = camara.read()
if not ret:
break

gray = cv2.cvtColor(imagen, cv2.COLOR_BGR2GRAY)
esquinas, ids, _ = detector.detectMarkers(gray)

if ids is not None:
aruco.drawDetectedMarkers(imagen, esquinas, ids)

rvecs, tvecs, _ = aruco.estimatePoseSingleMarkers(
esquinas, longitud_marcador, matriz_camara, coef_dist
)

for i, (rvec, tvec) in enumerate(zip(rvecs, tvecs)):
cv2.drawFrameAxes(
imagen, matriz_camara, coef_dist,
rvec, tvec, 0.05
)

print(f"ID detectado: {ids}")

cv2.imshow("Deteccion ArUco", imagen)

if cv2.waitKey(1) & 0xFF == ord('q'):
break

camara.release()
cv2.destroyAllWindows()
```

Explicación del código

Esta sección inicia el proceso cargando los resultados de la calibración anterior: la matriz intrínseca y los coeficientes de distorsión. Estos son esenciales ya que corrigen las distorsiones de la lente y permiten la conversión de píxeles a unidades del mundo real. Además, se inicializa el detector ArUco, especificando el diccionario (con 50 IDs) y sus parámetros de detección.

```
parametros_camara = fc.obtener_Parametros_Camara()
matriz_camara = np.array(parametros_camara["matriz_camara"])
coef_dist = np.array(parametros_camara["coef_dist"])

aruco_dict = aruco.getPredefinedDictionary(aruco.DICT_4X4_50)
parameters = aruco.DetectorParameters()
detector = aruco.ArucoDetector(aruco_dict, parameters)
```

Se define la longitud real del lado del marcador ArUco en metros (longitud_marcador). Este valor es crucial para calcular la pose. Luego, se inicializa la cámara y se configura su resolución para que coincida exactamente con la resolución utilizada durante la calibración.

```
longitud_marcador = 0.143

camara = cv2.VideoCapture(1)

camara.set(cv2.CAP_PROP_FRAME_WIDTH, parametros_camara["resolucion_width"])
camara.set(cv2.CAP_PROP_FRAME_HEIGHT, parametros_camara["resolucion_height"])
```

Este es el bucle de procesamiento en tiempo real. Después de leer el frame y detectar los marcadores (guardando sus esquinas y IDs), se invoca la función clave: aruco.estimatePoseSingleMarkers. Esta función utiliza las esquinas detectadas, la longitud real del marcador y los parámetros de calibración (matriz y coeficientes) para calcular los vectores de rotación (R) y traslación (t). Estos vectores definen la pose 3D del marcador respecto a la cámara.

```
while True:
ret, imagen = camara.read()
if not ret:
break

gray = cv2.cvtColor(imagen, cv2.COLOR_BGR2GRAY)
esquinas, ids, _ = detector.detectMarkers(gray)

if ids is not None:
aruco.drawDetectedMarkers(imagen, esquinas, ids)

rvecs, tvecs, _ = aruco.estimatePoseSingleMarkers(
esquinas, longitud_marcador, matriz_camara, coef_dist
)
```

Dentro del bucle, se visualiza el resultado de la pose para cada marcador detectado. La función cv2.drawFrameAxes dibuja un sistema de ejes 3D sobre el marcador, utilizando los rvecs y tvecs calculados, lo que permite al usuario confirmar visualmente la posición y orientación del objeto. La tvecs (traslación) representa la posición (x y, z) del marcador en el espacio tridimensional (en metros). Finalmente, el código muestra el frame y cierra la cámara al presionar 'q'.

```
for i, (rvec, tvec) in enumerate(zip(rvecs, tvecs)):
cv2.drawFrameAxes(
imagen, matriz_camara, coef_dist,
rvec, tvec, 0.05
)

print(f"ID detectado: {ids}")

cv2.imshow("Deteccion ArUco", imagen)

if cv2.waitKey(1) & 0xFF == ord('q'):
break

camara.release()
cv2.destroyAllWindows()
```

7. Capítulo 5: Obtención de los parámetros extrínsecos: matriz de rotación y vector de traslación de mundo a cámara.

El paso final para completar la calibración de la cámara es la obtención de sus parámetros extrínsecos, los cuales definen la relación geométrica entre el sistema de coordenadas del Mundo (W) y el sistema de coordenadas de la Cámara (C). Dicha relación se expresa mediante la Matriz de Rotación (R) y el Vector de Traslación (t). Se utilizarán marcadores ArUco para definir y demarcar el área de trabajo, estableciendo así el sistema de coordenadas del Mundo (W). Es crucial que los marcadores tengan una orientación conocida y uniforme, como se muestra en la figura 11.

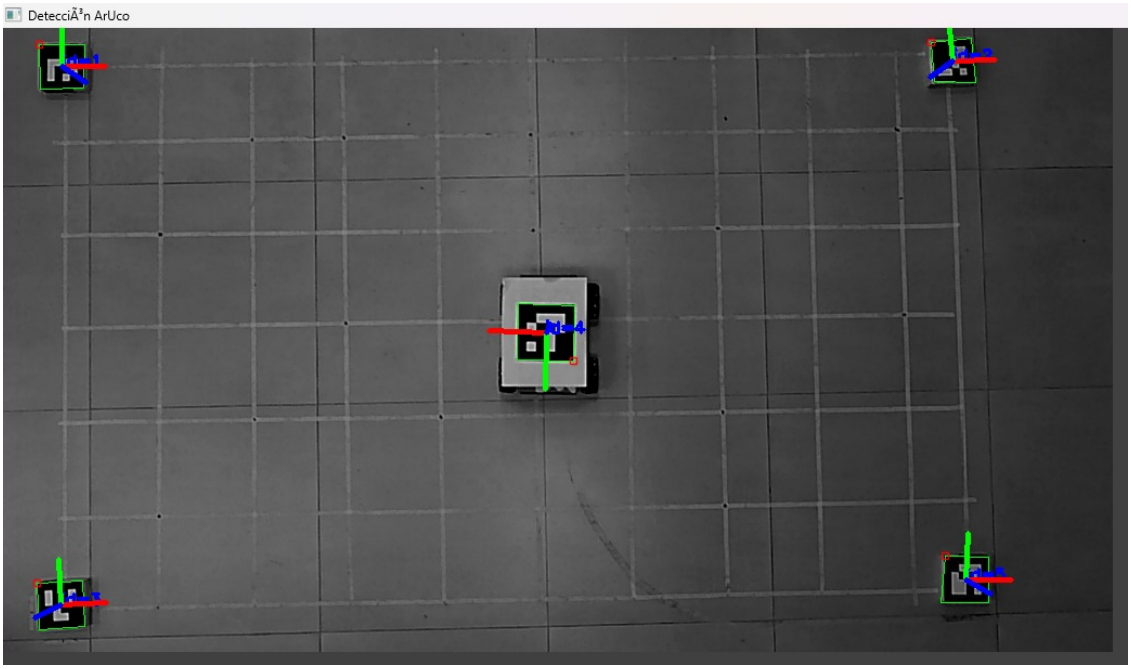

Figura 11: Ejemplo de orientación y colocación de los marcadores ArUco.

Para robustecer la referencia espacial, se emplearán cuatro marcadores ArUco. Esta redundancia minimiza el impacto en los cálculos si uno de ellos se pierde o se ocluye temporalmente. Es indispensable designar un marcador como el origen (W) y medir con precisión la posición relativa de los demás, ya que esta configuración de marca el espacio de movimiento del vehículo. Cabe recalcar que para este proceso solo se requieren los cuatro marcadores ArUco de referencia, ya que el vehículo o su marcador no intervienen en el establecimiento del sistema de coordenadas del Mundo.

Código

Listing 5: Obtención de los parámetros extrínsecos

```
import cv2
import cv2.aruco as aruco
import numpy as np
import Funciones_Codigos as fc

# ------ Parametros generales ------
tamano_aruco = 0.143  # en metros
coordenadas_world = {
    1: np.array([0.0, 0.0, 0.0]),   # Esquina inferior izq.
    2: np.array([2.554, 0.0, 0.0]), # Esquina inferior der.
    3: np.array([0.0, 1.803, 0]),   # Esquina superior izq.
    5: np.array([2.554, 1.803, 0])  # Esquina superior der.
}

# ------ Configuracion codigos ArUco ------
```

```
        aruco_dict = aruco.getPredefinedDictionary(aruco.DICT_4X4_50)
        parameters = aruco.DetectorParameters()
        detector = aruco.ArucoDetector(aruco_dict, parameters)

        # ------ Obtencion de los parametros de la camara ------
        parametros_camara = fc.obtener_Parametros_Camara()

        def obtener_imagen_camara(camara):
        ret, imagen = camara.read()
        if not ret:
        print("No se pudo tomar una imagen.")
        exit()
        return imagen

        def detectar_codigos(imagen):
        obj_points = []  # Puntos 3D conocidos
        img_points = []  # Puntos 2D detectados en la imagen

        corners, ids, _ = detector.detectMarkers(imagen)
        if ids is None:
        print("No se detectaron ArUcos en la imagen.")
        exit()

        for i, marker_id in enumerate(ids.flatten()):
        if marker_id in coordenadas_world:
        c = corners[i][0]
        centro = c.mean(axis=0)
        img_points.append(centro)
        obj_points.append(coordenadas_world[marker_id])

        obj_points = np.array(obj_points, dtype=np.float32)
        img_points = np.array(img_points, dtype=np.float32)

        return obj_points, img_points

        def obtener_rotacion_traslacion(obj_points, img_points, parametros_camara):
        _, rvec, tvec = cv2.solvePnP(
        obj_points, img_points,
        np.array(parametros_camara["matriz_camara"]),
        np.array(parametros_camara["coef_dist"])
        )

        R, _ = cv2.Rodrigues(rvec)
        print("Matriz de rotacion (R):\n", R)
        print("Vector de traslacion (t):\n", tvec)

        fc.guardar_Parametros_Extrinsecos_Camara(R, tvec)
        print("Se guardaron la R y t en el archivo json")

        return rvec, tvec

        def dibujar_ejes(imagen, rvec, tvec, img_points, parametros_camara):
        axis = np.float32([[0.1,0,0], [0,0.1,0], [0,0,-0.1]])
        imgpts, _ = cv2.projectPoints(
        axis, rvec, tvec,
        np.array(parametros_camara["matriz_camara"]),
        np.array(parametros_camara["coef_dist"])
        )

        corner = tuple(img_points[0].astype(int))  # primer punto detectado
        imagen = cv2.line(imagen, corner, tuple(imgpts[0].ravel().astype(int)), (0,0,255),
            3) # X
```

```
77          imagen = cv2.line(imagen, corner, tuple(imgpts[1].ravel().astype(int)), (0,255,0),
                3) # Y
78          imagen = cv2.line(imagen, corner, tuple(imgpts[2].ravel().astype(int)), (255,0,0),
                3) # Z
79
80          cv2.imshow("Extrinseca con 4 ArUco", imagen)
81          cv2.waitKey(0)
82          cv2.destroyAllWindows()
83
84          # --- Captura de imagen y procesamiento ---
85          camara = cv2.VideoCapture(1)
86          camara.set(cv2.CAP_PROP_FRAME_HEIGHT, parametros_camara["resolucion_height"])
87          camara.set(cv2.CAP_PROP_FRAME_WIDTH, parametros_camara["resolucion_width"])
88
89          imagen = obtener_imagen_camara(camara)
90
91          obj_points, img_points = detectar_codigos(imagen)
92
93          # --- Obtenemos la rotacion y traslacion ---
94          rvec, tvec = obtener_rotacion_traslacion(obj_points, img_points, parametros_camara)
95
96          # --- Dibujar ejes en un ArUco de referencia ---
97          dibujar_ejes(imagen, rvec, tvec, img_points, parametros_camara)
98
99          camara.release()
100         cv2.destroyAllWindows()
```

Explicación del código

Esta parte define el sistema de coordenadas del Mundo (W). La variable tamaño_aruco establece la dimensión física real de los marcadores. El diccionario coordenadas_world asigna a cada ID de marcador ArUco (1, 2, 3, 5) su posición tridimensional (X,Y,Z) exacta en metros, estableciendo el origen y las dimensiones del área de trabajo.

```
        # ------ Parametros generales ------
        tamano_aruco = 0.143  # en metros
        coordenadas_world = {
            1: np.array([0.0, 0.0, 0.0]),   # Esquina inferior izq.
            2: np.array([2.554, 0.0, 0.0]), # Esquina inferior der.
            3: np.array([0.0, 1.803, 0]),   # Esquina superior izq.
            5: np.array([2.554, 1.803, 0])  # Esquina superior der.
        }
```

Se realiza la configuración estándar del detector ArUco, definiendo el diccionario para la identificación de los marcadores. Simultáneamente, se cargan los parámetros intrínsecos de la cámara (matriz intrínseca, coeficientes de distorsión y resolución) que fueron obtenidos en el capítulo anterior, ya que son necesarios para cualquier cálculo de pose.

```
        # ------ Configuracion codigos ArUco ------
        aruco_dict = aruco.getPredefinedDictionary(aruco.DICT_4X4_50)
        parameters = aruco.DetectorParameters()
        detector = aruco.ArucoDetector(aruco_dict, parameters)

        # ------ Obtencion de los parametros de la camara ------
        parametros_camara = fc.obtener_Parametros_Camara()
```

Esta función auxiliar está diseñada para capturar un frame único de la cámara de manera segura. Verifica si la lectura fue exitosa (ret). Si la captura falla, imprime un mensaje de error y finaliza el programa; de lo contrario, devuelve la imagen capturada.

```
        def obtener_imagen_camara(camara):
        ret, imagen = camara.read()
        if not ret:
```

```
print("No se pudo tomar una imagen.")
exit()
return imagen
```

La función detectar_codigos identifica los marcadores en la imagen y asocia sus coordenadas. Por cada marcador detectado que pertenece a las coordenadas_world (los de referencia), calcula el centroide (centro) de sus esquinas en píxeles (img_points) y lo empareja con su coordenada real tridimensional previamente definida (obj_points). Este par de conjuntos de puntos es la base para calcular la extrínseca.

```
def detectar_codigos(imagen):
obj_points = []  # Puntos 3D conocidos
img_points = []  # Puntos 2D detectados en la imagen

corners, ids, _ = detector.detectMarkers(imagen)
if ids is None:
print("No se detectaron ArUcos en la imagen.")
exit()

for i, marker_id in enumerate(ids.flatten()):
if marker_id in coordenadas_world:
c = corners[i][0]
centro = c.mean(axis=0)
img_points.append(centro)
obj_points.append(coordenadas_world[marker_id])

obj_points = np.array(obj_points, dtype=np.float32)
img_points = np.array(img_points, dtype=np.float32)

return obj_points, img_points
```

Esta función realiza el cálculo fundamental de la calibración extrínseca utilizando el algoritmo cv2.solvePnP (Perspective-n-Point). Este resuelve la posición y orientación de la cámara (rvec y tvec) a partir de los pares de puntos 3D (obj_points) y 2D (img_points). Luego, el rvec se convierte a la Matriz de Rotación (R) usando cv2.Rodrigues. Finalmente, los parámetros R y tvec se imprimen y se guardan en un archivo .json para su uso posterior.

```
def obtener_rotacion_traslacion(obj_points, img_points, parametros_camara):
_, rvec, tvec = cv2.solvePnP(
obj_points, img_points,
np.array(parametros_camara["matriz_camara"]),
np.array(parametros_camara["coef_dist"])
)

R, _ = cv2.Rodrigues(rvec)
print("Matriz de rotacion (R):\n", R)
print("Vector de traslacion (t):\n", tvec)

fc.guardar_Parametros_Extrinsecos_Camara(R, tvec)
print("Se guardaron la R y t en el archivo json")

return rvec, tvec
```

La función dibujar_ejes se utiliza para validar visualmente la calibración extrínseca. Utiliza los vectores rvec y tvec (calculados en el bloque 5) junto con los parámetros intrínsecos para proyectar un sistema de ejes 3D desde el origen del mundo (definido por el primer marcador) hacia la imagen. Al dibujar los ejes X (rojo), Y (verde) y Z (azul) sobre la imagen, se confirma que la pose se ha estimado correctamente.

```
def dibujar_ejes(imagen, rvec, tvec, img_points, parametros_camara):
axis = np.float32([[0.1,0,0], [0,0.1,0], [0,0,-0.1]])
imgpts, _ = cv2.projectPoints(
axis, rvec, tvec,
np.array(parametros_camara["matriz_camara"]),
np.array(parametros_camara["coef_dist"])
```

```
)

corner = tuple(img_points[0].astype(int))  # primer punto detectado
imagen = cv2.line(imagen, corner, tuple(imgpts[0].ravel().astype(int)), (0,0,255),
    3) # X
imagen = cv2.line(imagen, corner, tuple(imgpts[1].ravel().astype(int)), (0,255,0),
    3) # Y
imagen = cv2.line(imagen, corner, tuple(imgpts[2].ravel().astype(int)), (255,0,0),
    3) # Z

cv2.imshow("Extrinseca con 4 ArUco", imagen)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Este bloque contiene la lógica de ejecución principal (main). Inicializa la cámara y configura la resolución. Luego, llama secuencialmente a las funciones definidas: captura una imagen, detecta los marcadores y asocia los puntos, calcula la rotación y traslación extrínseca, y finalmente dibuja los ejes para su inspección visual. Concluye liberando la cámara y cerrando las ventanas.

```
# --- Captura de imagen y procesamiento ---
camara = cv2.VideoCapture(1)
camara.set(cv2.CAP_PROP_FRAME_HEIGHT, parametros_camara["resolucion_height"])
camara.set(cv2.CAP_PROP_FRAME_WIDTH, parametros_camara["resolucion_width"])

imagen = obtener_imagen_camara(camara)

obj_points, img_points = detectar_codigos(imagen)

# --- Obtenemos la rotacion y traslacion ---
rvec, tvec = obtener_rotacion_traslacion(obj_points, img_points, parametros_camara)

# --- Dibujar ejes en un ArUco de referencia ---
dibujar_ejes(imagen, rvec, tvec, img_points, parametros_camara)

camara.release()
cv2.destroyAllWindows()
```

8. Conclusión

El objetivo principal de este trabajo fue establecer una metodología robusta y precisa para la calibración geométrica de una cámara web y su posterior aplicación en la determinación de pose 3D dentro de un área de trabajo definida. Los resultados se obtuvieron en tres etapas cruciales:

1. Calibración Intrínseca (Capítulos 2 y 3): Se capturaron múltiples imágenes de un patrón de ajedrez para modelar las propiedades internas de la cámara. Este proceso arrojó la Matriz Intrínseca (K) y los Coeficientes de Distorsión, parámetros esenciales que permiten corregir las deformaciones ópticas del lente y pasar de coordenadas de píxeles a coordenadas métricas. La precisión de esta etapa fue validada por un bajo Error de Re-proyección, confirmando la fiabilidad de los parámetros intrínsecos obtenidos.

2. Generación y Detección de Marcadores ArUco (Capítulo 4): Se implementó y verificó el uso de los marcadores fiduciales ArUco para tareas de seguimiento. Estos códigos binarios demostraron ser una herramienta efectiva para la detección y la posterior estimación de la pose (rvec,tvec) de objetos en tiempo real, sentando las bases para el establecimiento del sistema de coordenadas del mundo.

3. Calibración Extrínseca (Capítulo 5): Finalmente, se estableció la relación espacial entre la cámara y el área de trabajo (W→C). Utilizando cuatro marcadores ArUco de referencia, se calculó la Matriz de Rotación (R) y el Vector de Traslación (t). Estos parámetros extrínsecos transforman las coordenadas medidas por la cámara en las coordenadas exactas y reales del área de trabajo, cerrando así el proceso de calibración y creando un sistema que permite medir y localizar cualquier objeto dentro de la zona demarcada.

En resumen, la metodología secuencial de calibración intrínseca, identificación de marcadores ArUco y calibración extrínseca ha permitido crear un sistema de visión artificial confiable y métricamente preciso. Este sistema está ahora listo para ser aplicado en la localización y seguimiento del vehículo, permitiendo la automatización y navegación dentro del entorno físico.