UAEM | Universidad Autónoma del Estado de México

# Reporte

## Calibración de cámara web Logitech C920 y seguimiento de trayectoria utilizando marcadores ArUco para el desarrollo de un sistema de visión por computadora

*Cuerpo Académico de Dinámica de Sistemas y Control*

*Facultad de Ingeniería, Otoño 2022*

PLBIM. Daniel Enrique Fernández García

Dr. Juan Manuel Jacinto Villegas

Toluca de Lerdo, México

Noviembre, 2022

## INTRODUCCIÓN

El proyecto está basado en la identificación y seguimiento de trayectoria de marcadores *ArUco* utilizando visión por computadora implementada en *Python* con la cámara web Logitech C920. Se utilizó la librería *OpenCV* con módulos extra, de esta forma es posible utilizar el módulo de detección de marcadores *ArUco*.

La calibración de la cámara también es realizada con la misma librería, utilizando un *checkerboard*. Los parámetros de la cámara son almacenados en un archivo *JSON* para su posterior utilización en la identificación de los marcadores. La configuración actual del software permite identificar tres marcadores, estableciendo uno de ellos como fijo, permitiendo calcular la distancia en píxeles y rotación de los otros dos marcadores que pueden moverse libremente dentro del área de trabajo con relación al fijo.

El código fuente del proyecto puede consultarse a través de su repositorio en GitHub**.**

## DESARROLLO Y PRUEBAS

### Entorno de trabajo

El proyecto fue desarrollado utilizando un entorno de trabajo virtual a través de *Pipenv*, no dentro de la instalación global de *Python*. Dentro del archivo *Pipfile*, es posible consultar las dependencias del proyecto y otros requerimientos para su correcta ejecución. Considerando una previa instalación de Python en el sistema operativo, es posible instalar *Pipenv* a través del gestor de paquetes *pip*. La documentación de *Pipenv* está disponible en el siguiente link.

```
pip install pipenv
```

Una vez instalado *Pipenv*, es posible la sincronización del entorno de desarrollo virtual. Esto podemos realizarlo estando dentro del a raíz del directorio del proyecto, utilizando el siguiente comando.

```
pipenv sync
```

Antes de realizar la calibración de la cámara y otras tareas, es necesario definir dentro de *camera.env* los parámetros de captura de video y dimensiones del *checkerboard.*

```
# Video soruce
SRC=0
# Video width
WIDTH=960
# Video height
HEIGHT=540
# Frames per second
FPS=30
# Camera  model
MODEL=C920
# Calibration chessboard settings
```

```
BOARD_ROWS=7
BOARD_COLUMNS=6
```

El proyecto fue implementado utilizando las configuraciones que se muestran en el *snippet* anterior. Esta implementación fue realizada utilizando como referencia la documentación de OpenCV y MathWorks relacionada a la calibración de la cámara. El *checkerboard* que se utiliza para la calibración de la cámara corresponde a 3 cm por cuadro, 6 filas y 7 columnas, que es el ejemplo utilizado en ambas referencias. Este patrón puede generarse para su impresión en el sitio de Ninox 360.

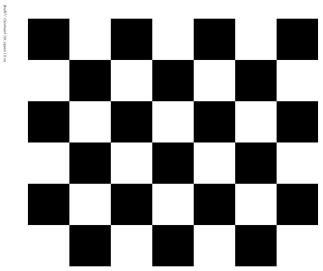

*Figura 1. Checkerboard utilizado para calibración de la cámara.*

La relación de aspecto 16:9 y resolución de 960x540 fue utilizada. El área de trabajo para la realización de tareas fue delimitada en 140x75 cm sobre una mesa de laboratorio. Utilizando un tripié, se colocó la cámara a 110 cm de la superficie de la mesa. Estas condiciones dentro del laboratorio permiten la identificación de los marcadores ArUco y la realización de movimientos de abducción y aducción horizontal del brazo sobre la superficie plana.

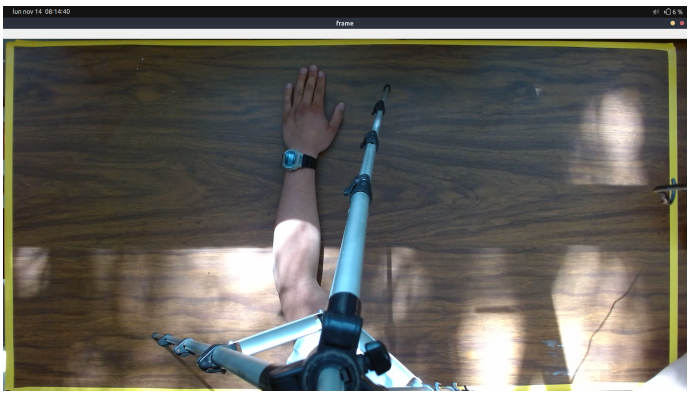

*Figura 2. Área de trabajo para la realización de tareas.*

### Calibración de la cámara

La calibración de una cámara permite la determinación de parámetros que explican la proyección de un objeto tridimensional al plano de la cámara. Estos parámetros pueden determinarse mediante escenarios conocidos llamados patrones de calibración. Cualquier objeto debidamente caracterizado podría ser utilizado como un objeto de calibración, sin embargo, la opción más práctica es el de un patrón regular, como un *checkerboard*.

A través de *calibrate_camera.py* es posible realizar una nueva calibración de la cámara.

```
python calibrate_camera.py --new true
```

Este *script* es dependiente de take_photos.py el cúal inicializa la captura de video y permite obtener fotografías para su utilización en la calibración de la cámara. Las fotografías son almacenadas dentro de un folder que es creado en la ejecución del *script.*

```
# take_photos.py
folder = "calibration_photos"
```

El script *take_photos.py* depende de los siguientes paquetes.

```
# take_photos.py
# Dependencies
import cv2 as cv
from datetime import datetime
import os
import time
from dotenv import load_dotenv
```

Se define un *entrypoint* para su ejecución independiente a través de CLI, lo que permite leer las variables del entorno de trabajo que fueron establecidas en el archivo *camera.env*.

```
# take_photos.py
def main():

    # Load .env file camera variables
    load_dotenv("./camera.env")
    SRC = int(os.getenv("SRC"))
    WIDTH = int(os.getenv("WIDTH"))
    HEIGHT = int(os.getenv("HEIGHT"))
    FPS = int(os.getenv("FPS"))
    capture(SRC, WIDTH, HEIGHT, FPS)

# entrypoint
if __name__ == "__main__":

    main()
```

A través de la función *capture* se crea una nueva ventana dentro de la pantalla para visualizar la captura de vídeo, permitiendo almacenar en el folder creado las imágenes en formato *PNG* con una estampa de tiempo. De acuerdo con la documentación de OpenCV, se recomienda obtener al menos 10 imágenes del patrón de calibración.

```
# take_photos.py
# Capture video to save photos for camera calibration procedure
def capture(src, width, height, fps):

    # Create folder if it does not exist
    if not os.path.exists(folder):
        os.makedirs(folder)

    # Initialize video source capture
    vs = cv.VideoCapture(src, apiPreference=cv.CAP_ANY,
                         params=[
                             cv.CAP_PROP_FRAME_WIDTH, width,
```

```
                                cv.CAP_PROP_FRAME_HEIGHT,height,
                                cv.CAP_PROP_FPS, fps])
    time.sleep(2.0)

    while(True):

        _ , frame = vs.read()
        cv.imshow('frame', frame)

        key = cv.waitKey(1) & 0xFF

        # Quit capture if 'q' key is pressed
        if key == ord('q'):
            vs.release()
            cv.destroyAllWindows()
            break
        # Capture photo if 'c' key is pressed
        if key == ord('c'):
            # Save current frame in the desired repository
                                                cv.imwrite(os.path.join(folder,
f"raw_{datetime.now().strftime('%Y%m%d_%H%M%S')}.png"), frame)

    # Return folder directory where calibration fotos where saved
    return folder
```

Esta función es importada posteriormente dentro de *calibrate_camera.py* por lo que es creado como un objeto que depende de los parámetros necesarios para la inicialización de la captura de video.

```
# calibrate_camera.py
# Dependencies
import numpy as np
import cv2 as cv
from datetime import datetime
import argparse
import glob
import json
import os
from dotenv import load_dotenv
import take_photos
```

El script *calibrate_camera.py* requiere de un argumento el cuál indica si el usuario desea realizar una nueva calibración. Esto se pretende que el usuario lo realice cuando aún no se tiene alguna captura de fotografías para la calibración o cuando el entorno de trabajo y posición de la cámara ha cambiado. Si el nombre del folder para almacenar las imágenes de calibración fue cambiado, necesita modificarse también dentro de la función *main*.

```
# calibrate_camera.py
# Set required script arguments
def arguments():

        parser  =  argparse.ArgumentParser(description='Generate  .json  camera
correction settings after camera calibration.')
                parser.add_argument('--new',    type=str,    choices=['true',
'false'],required=True, help='Select if is required to take new photos for
calibration')
```

```
    return parser.parse_args()

def main():

    # Set required arguments
    args = arguments()

    # Load .env file camera variables
    load_dotenv("./camera.env")
    MODEL = os.getenv("MODEL")
    BOARD_ROWS = int(os.getenv("BOARD_ROWS"))
    BOARD_COLUMNS = int(os.getenv("BOARD_COLUMNS"))

    # Take new photos
    if args.new == 'true':
        SRC = int(os.getenv("SRC"))
        WIDTH = int(os.getenv("WIDTH"))
        HEIGHT = int(os.getenv("HEIGHT"))
        FPS = int(os.getenv("FPS"))
        folder = take_photos.capture(SRC, WIDTH, HEIGHT, FPS)
        calibration(folder, MODEL, BOARD_ROWS, BOARD_COLUMNS)

    # Photos already exist
    else:
        folder = "calibration_photos"
        if os.path.exists(folder):
            calibration(folder, MODEL, BOARD_ROWS, BOARD_COLUMNS)
        else:
            print("There are no calibration photos to use. Run a new calibration
procedure using the flag '--new true'.")

# entrypoint
if __name__ == "__main__":

    main()
```

El proceso de calibración se realiza a través de la función *calibration*, utilizando el método de detección de las esquinas del *checkerboard* en cada imágen que fue capturada previamente. Las imágenes son transformadas a grises, se localizan sus esquinas, y se incrementa la precisión de detección utilizando el método de subpixeles. Posteriormente, todos los puntos del mundo real en 3D y sus correspondientes puntos de imagen en 2D son utilizados para obtener la matriz de la cámara y coeficientes de distorsión. El error promedio de reproyección es calculado y mostrado en CLI, este da una buena estimación de cuán exactos son los parámetros encontrados, este debe ser lo más cercano a cero. Finalmente, los parámetros son estructurados en un *payload* para almacenarse como archivo JSON.

```
# calibrate_camera.py
# Camera calibration procedure
def calibration(folder, model, rows, columns):

    # Termination criteria for the cornerSubPix() algorithm
    criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)

    # Ensure the number of rows and columns is one less than the actual rows and
columns
     # on your chessboard. The reason for doing this is that the algorithm will
```

```
be looking
    # for internal corners on the chessboard

    rows = rows - 1;
    columns = columns - 1;

     # Create some arrays to store the object points and image points from all
the images of the chessboard
    objp = np.zeros((rows*columns,3), np.float32)
    objp[:,:2] = np.mgrid[0:columns,0:rows].T.reshape(-1,2)

    objpoints = []
    imgpoints = []

    # Get the filenames of all the raw calibration images
    images = glob.glob(os.path.join(folder,'raw*.png'))
    print(len(images), "raw images found")

    for fname in images:

        # The image is converted to grayscale
        img = cv.imread(fname)
        gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

        # Find the chessboard corners
         chessboard_flags = cv.CALIB_CB_ADAPTIVE_THRESH + cv.CALIB_CB_FAST_CHECK
+ cv.CALIB_CB_NORMALIZE_IMAGE
               ret,  corners  =  cv.findChessboardCorners(gray,  (columns,rows),
chessboard_flags)

        if ret == True:
            objpoints.append(objp)
            # Increase the accuracy using cornerSubPix()
            corners2 = cv.cornerSubPix(gray,corners, (11,11), (-1,-1), criteria)
            imgpoints.append(corners)

              # Uncomment if is desired to save Chessboard Corners images in the
calibration photos directory
            #cv.drawChessboardCorners(img, (columns,rows), corners2, ret)
            # folder = "calibration_photos"

#cv.imwrite(os.path.join(folder,f"corners_{datetime.now().strftime('%Y%m%d_%H%M%
S')}.png"), img)
            #cv.imshow('img', img)
            #cv.waitKey(1500)

    # Calibrate the camera using the corners that have been found
    # ret = RMS re-projection error
    # mtx =  Camera matrix
    # dist  =  Distortion coefficients
    # rvecs = Rotation vectors
    # tvecs = Translation vectors
      ret, mtx, dist, rvecs, tvecs = cv.calibrateCamera(objpoints, imgpoints,
gray.shape[::-1], None, None)
    cv.destroyAllWindows()

      #   Compute the arithmetical mean of the errors calculated for all the
calibration images
```

```
    mean_error = 0
    for i in range(len(objpoints)):
         imgpoints2, _ = cv.projectPoints(objpoints[i], rvecs[i], tvecs[i], mtx,
dist)
        error = cv.norm(imgpoints[i], imgpoints2, cv.NORM_L2)/len(imgpoints2)
        mean_error += error
    print("Average re-projection: {}".format(mean_error/len(objpoints)) )

    # Create .json output file for the obtained calibration parameters
    jsonify(model, ret, mtx, dist, rvecs, tvecs)
```

El archivo con nombre del modelo de la cámara que contiene los parámetros de calibración obtenidos es creado implementando el módulo *JSON* de *Python*. En este proyecto, el archivo obtenido es *C920.json*.

```
# Create .json file from the calibration camera results
def jsonify(model,ret, mtx, dist, rvecs, tvecs):

    # Create JSONEncoder object
    camera = {}
    class NumpyEncoder(json.JSONEncoder):
        def default(self, obj):
            if isinstance(obj, np.ndarray):
                return obj.tolist()
            return json.JSONEncoder.default(self, obj)

    for variable in ['ret', 'mtx', 'dist', 'rvecs', 'tvecs']:
        camera[variable] = eval(variable)

    # Create output .json file
    with open((model +".json"), 'w') as f:
        json.dump(camera, f, indent=4, cls=NumpyEncoder)
```

### Detección de los marcadores ArUco

Un marcador *ArUco* es un marcador cuadrado sintético compuesto por un borde negro ancho y una matriz binaria interna que determina su identificador. El borde negro facilita su rápida detección en la imagen y la codificación binaria permite su identificación. Cada marcador detectado incluye la posición de sus cuatro esquinas en la imagen y el ID del marcador.

Se utiliza un diccionario para identificar el marcador. El diccionario define un conjunto de reglas que se utilizan para calcular el identificador del marcador, realizar la validación y aplicar la corrección de errores. Entre menor el número de bits de cada marcador y menor número de marcadores dentro del diccionario, mejor es su identificación.

En este proyecto se utiliza DICT_4X4_50, este contiene 50 marcadores, de 4x4 bits. De acuerdo con la documentación de ArUco, se implementa una relación mínima del marcador de 0.03 con la dimensión máxima de la imagen. Considerando el área de trabajo de 140 cm, se utilizaron marcadores de 5 cm. Los marcadores con ID 0, 1, y 2 fueron generados utilizando esta herramienta. Sin embargo, pueden generarse de forma manual dentro de un script independiente utilizando el módulo de ArUco. A través del script aruco_detection.*py* es posible la identificación de los 3 marcadores utilizados, estableciendo uno como fijo para calcular la posición de los otros dos, con respecto a este. Las coordenadas de los marcadores son mostradas en CLI.

```
python aruco_detection.py
```

Se puede encontrar mayor información del procesamiento que realiza la librería y recomendaciones para la mejora de su rendimiento en este documento adicional del autor.

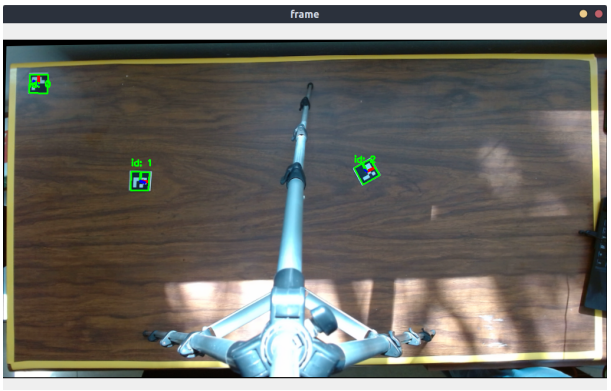


*Figura 3. Identificación de los 3 marcadores ArUco.*

El script *aruco_detection.py* depende de los siguientes módulos.

```
# aruco_detection.py
# Dependencies
import cv2 as cv
import argparse
import json
import numpy as np
import os
import time
from dotenv import load_dotenv
from scipy.spatial.transform import Rotation as R
```

Se establecen tres variables globales las cuales permiten seleccionar el diccionario de marcadores que se utilizará para la detección, el ID del marcador que se estableceŕa como fijo para calcular la posición de los otros respecto a este y una lista para almacenar la pose actual de todos los marcadores tras realizar las operaciones correspondientes.

```
# aruco_detection.py
# ArUco markers dictionary to use
DICT = cv.aruco.DICT_4X4_50
# Grounded marker to be set as reference
REF_MARKER = 0
# List to save identified markers location
markersPose =  [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Utilizando los parámetros establecidos en el archivo *camera.env* se realiza una inicialización de la fuente de video para comenzar la detección de la cámara.

```
# aruco_detection.py
# Main
def main():
```

```
    # Load .env file camera variables
    load_dotenv("./camera.env")
    SRC = int(os.getenv("SRC"))
    WIDTH = int(os.getenv("WIDTH"))
    HEIGHT = int(os.getenv("HEIGHT"))
    FPS = int(os.getenv("FPS"))
    MODEL = os.getenv("MODEL")

    capture(MODEL, SRC, WIDTH, HEIGHT, FPS, DICT)

# Entrypoint
if __name__ == "__main__":

    main()
```

Antes de comenzar la captura de video, se leen los valores de la cámara que fueron obtenidos tras la calibración de la misma. Así obtenemos la matriz de la cámara y parámetros de distorsión.

```
# aruco_detection.py
# Read existing camera calibration parameters
def read_calibration(calibration_file):

    # Load .json file
    with open(calibration_file, 'r') as json_file:
        camera_data = json.load(json_file)
        dist = np.array(camera_data["dist"])
        mtx = np.array(camera_data["mtx"])
    return dist, mtx
```

Para la identificación de marcadores, se inicializa la captura de video, se corrige el *frame* capturado y se realiza la identificación de todos los marcadores en el área de trabajo. Obteniendo sus IDs, esquinas y las detecciones rechazadas. Posteriormente, estos datos son procesados a través de *draw_frame_makers* para mostrar dentro de la imagen el contorno del marcador y el número de ID.

```
# aruco_detection.py
def capture(model, src, width, height, fps, dict):

    # Read existing camera calibration parameters
    # mtx =  Camera matrix
    # dist  =  Distortion coefficients
    dist, mtx = read_calibration((model+".json"))
    newcameramtx, _ = cv.getOptimalNewCameraMatrix(mtx, dist, (width,height), 1,
(width,height))

    # Initialize video source capture
    vs = cv.VideoCapture(src, apiPreference=cv.CAP_ANY,
                         params=[
                             cv.CAP_PROP_FRAME_WIDTH, width,
```

```
                                cv.CAP_PROP_FRAME_HEIGHT,height,
                                cv.CAP_PROP_FPS, fps,
                                cv.CAP_PROP_AUTOFOCUS, 0 ])
    time.sleep(2.0)

    # Set ArUco dictionary and initalize parameters
    arucoDict = cv.aruco.Dictionary_get(dict)
    arucoParams = cv.aruco.DetectorParameters_create()

    while(True):

        _ , frame = vs.read()
        frame = cv.undistort(frame, mtx, dist, None, newcameramtx)

        # Convert current frame to grayscale
        gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)

        # Detect ArUco markers
        corners, ids, rejected = cv.aruco.detectMarkers(gray, arucoDict,
parameters=arucoParams)

        # Draw current coordinates and makers data
        frame = draw_frame_markers(corners, ids, rejected, frame, mtx, dist)

        # Display current frame with it's drawing
        cv.imshow('frame', frame)

        key = cv.waitKey(1) & 0xFF
        # Quit capture if 'q' key is pressed
        if key == ord('q'):
            vs.release()
            cv.destroyAllWindows()
            break
```

Finalmente, se realizan las operaciones para obtener los valores de los ángulos de Euler (roll- pitch-yaw) para cada marcador, además de sus coordenadas con respecto al que ha sido fijado como referencia. En este proyecto se utilizó el marcador 0 para fijarse en la parte superior izquierda del área de trabajo, así que los marcadores 1 y 2 son movidos libremente y sus coordenadas son mostradas en CLI.

```
# aruco_detection.py
def draw_frame_markers(corners, ids, rejected, frame, mtx, dist):

    if len(corners) > 0:s
        ids = ids.flatten()
        # If the reference marker is not detected, no other operations are
performed
        if REF_MARKER in ids:
            for (markerCorner, markerID) in zip(corners, ids):
                if markerID < len(markersPose):
                    # Estimate individual pose for single markers according to
```

```
the detected corners
                    rvec, tvec, _ =
cv.aruco.estimatePoseSingleMarkers(markerCorner, 0.023, mtx, dist)
                    # Draw xyz frames to identify rotation
                    cv.drawFrameAxes(frame, mtx, dist, rvec, tvec, 0.01)
                    # RPY angles computing
                    rotM = np.zeros(shape=(3,3))
                    cv.Rodrigues(rvec[0], rotM, jacobian = 0)
                    r = R.from_matrix(rotM)
                    angles = r.as_euler('zxy', degrees=True)
                    yaw = round(angles[0], 1)
                    roll = round(angles[1], 1)
                    pitch = round(angles[2], 1)

                    # Corners coordinates
                    corners = markerCorner.reshape((4, 2))
                    (topLeft, topRight, bottomRight, bottomLeft) = corners

                    topRight = (int(topRight[0]), int(topRight[1]))
                    bottomRight = (int(bottomRight[0]), int(bottomRight[1]))
                    bottomLeft = (int(bottomLeft[0]), int(bottomLeft[1]))
                    topLeft = (int(topLeft[0]), int(topLeft[1]))

                    # Draw each marker corners
                    cv.line(frame, topLeft, topRight, (0, 255, 0), 2)
                    cv.line(frame, topRight, bottomRight, (0, 255, 0), 2)
                    cv.line(frame, bottomRight, bottomLeft, (0, 255, 0), 2)
                    cv.line(frame, bottomLeft, topLeft, (0, 255, 0), 2)

                    # Compute marker center coordinates and display it's ID
                    cX = int((topLeft[0] + bottomRight[0]) / 2.0)
                    cY = int((topLeft[1] + bottomRight[1]) / 2.0)
                    cv.putText(frame,
                    ('id: ' + str(markerID)),
                    (topLeft[0], topLeft[1] - 10),
                    cv.FONT_HERSHEY_SIMPLEX,
                    0.5,
                    (0, 255, 0),
                    2)

                    # Save current marker location and pose in a list to be
operated
                    markersPose[markerID] = [cX, cY, yaw]

            # If the three markers are detected, operations are performed
            if len(ids) == len(markersPose):

                # Reference markers coordinates are subtracted from the other
marker coordinates
                markersPose[1][0] -= markersPose[0][0]
                markersPose[1][1] -= markersPose[0][1]
```

```
                markersPose[2][0] -= markersPose[0][0]
                markersPose[2][1] -= markersPose[0][1]

                markersPose[0][0] = 0
                markersPose[0][1] = 0

                # Corrected markers pose is printed in CLI
                print(markersPose)

    return frame
```

## CONCLUSIONES

1. Los autores de la librería de ArUco recomiendan la utilización de la librería *ARUCO_MIP_36h12* para la identificación más robusta de marcadores. Sin embargo, es necesario cambiar de entorno de desarrollo hacia C++ y compilar el código fuente de la librería en conjunto con OpenCV.

2. Para mejorar la detección de las esquinas de los marcadores, puede implementarse el uso de *enclosed markers*. Esto permite mejorar la estimación de la esquina con precisión subpixel. Cada marcador utilizado a través de este método, debe de ser generado de forma manual.

3. Durante el desarrollo del proyecto, se experimentó utilizando la máxima resolución disponible de la cámara (1920x1080). Sin embargo, esta no fue implementada debido a un retraso en el procesamiento y visualización de los cuadros del video. Se utilizó la resolución de 960x540 la cuál mejoró el rendimiento de ejecución de las tareas, permite la visualización de toda el área de trabajo y la correcta identificación de los marcadores.

4. En general, la biblioteca ArUco es muy robusta y puede detectar marcadores en una amplia gama de situaciones. Normalmente, no es necesario cambiar los valores predeterminados empleados para la detección, excepto para el diccionario. Sin embargo, en algunas situaciones, es posible que se desee ajustar los parámetros de acuerdo con el escenario que se tenga.

### Recursos adicionales

1. ArUco: One library to rule them all
2. ArUco tag guided drawing robot