{Learn, Create, Innovate};

Open CV

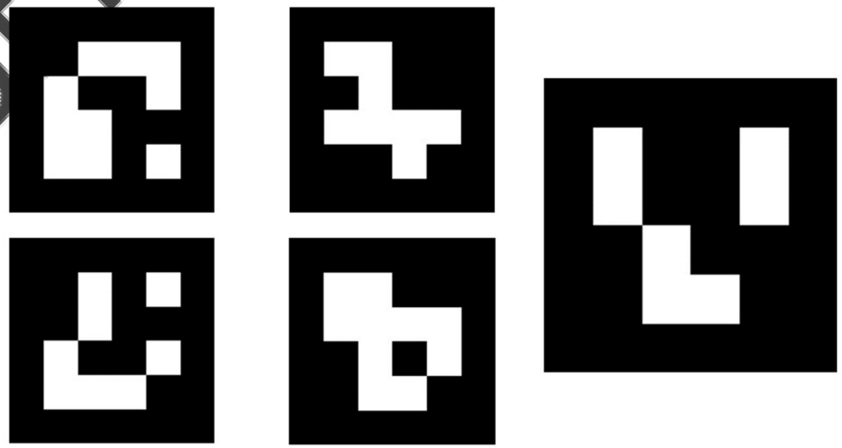
Detection of ArUco Markers

Manchester Robotics

ArUco Markers

What are ArUco Markers?

- Augmented Reality University of Cordoba (ArUco)
- An ArUco marker is a square marker composed by a wide black border and an inner binary matrix which determines its identifier (id).
- The marker resembles a QR code.
- The black border facilitates its fast detection in the image and the binary codification allows its identification and the application of error detection and correction techniques.

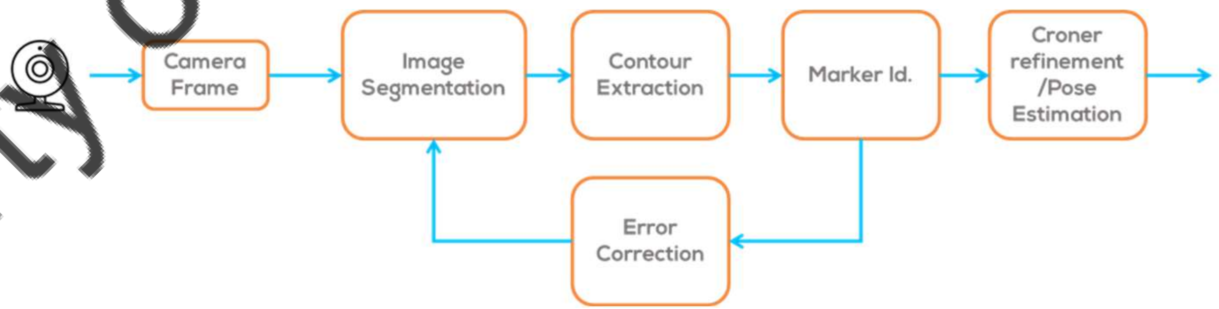


ArUco Markers

How are ArUco Markers detected?

- In order to detect the coordinates of marker corners and their respective IDs, two primary steps are involved.

1. Firstly, potential regions for ArUco markers are identified by analysing the image and identifying square shapes that could represent markers.
2. Secondly, the inner binary matrix is analysed to verify it as an actual marker and compared to a reference "dictionary".
3. Finally, if the marker is correctly identified; a corner refinement algorithm is used to obtain the corner position and pose estimation.



ArUco Markers

First step: Analysing regions.

- This process utilises adaptive threshold binarization and computer vision techniques to segment the image, extract contours, and eliminate any shapes that are not convex or do not approximate a square.
- Then, closed quadrilaterals that are not of an appropriate size are discarded, leaving only the most viable marker candidates.

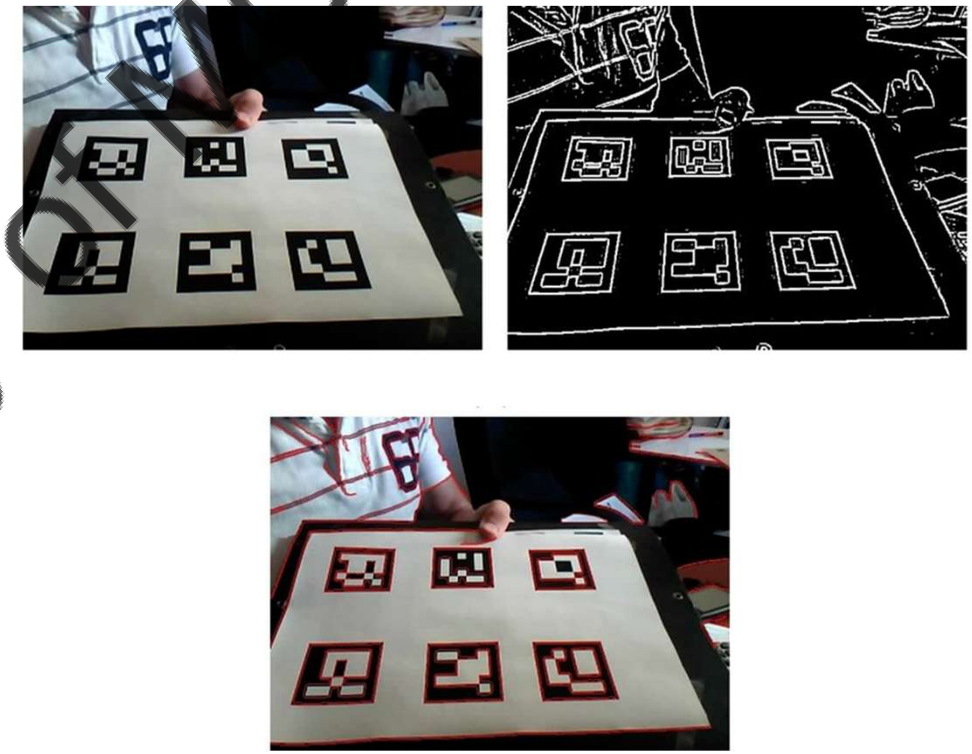


Image process for automatic marker detection

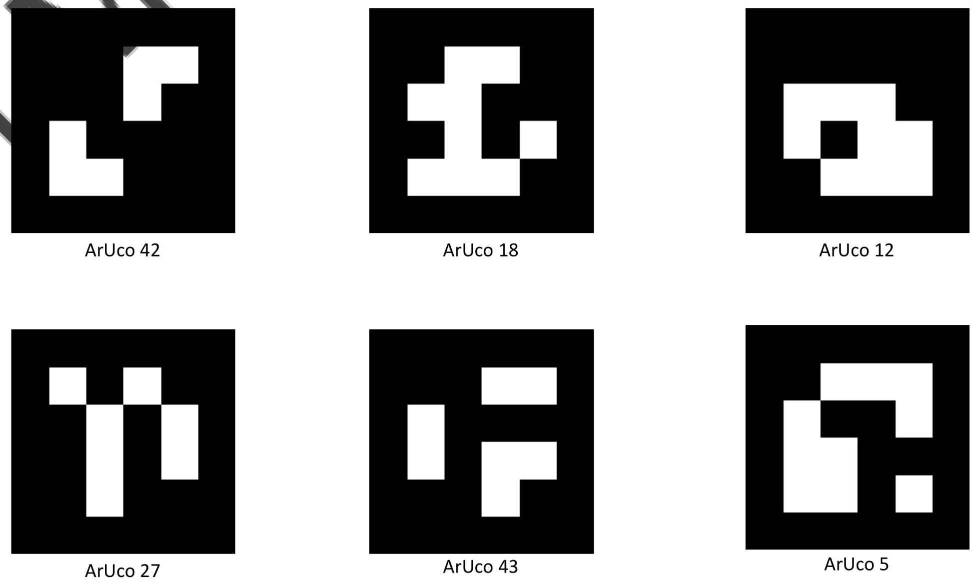
(a) Original Image, (b) Local threshold-ing. (c) Contour detection results

(Garrido-Jurado, S., Muñoz-Salinas, R., 2014)

ArUco Markers

What is a dictionary?

- A dictionary of markers a list of binary codifications of each of its markers.
- The main properties of a dictionary are the dictionary size and the marker size.
- The dictionary size is the number of markers that compose the dictionary.
- The marker size is the size of those markers (the number of bits).
- ArUco Library does not convert the code values to a decimal value, for efficiency purposes.
- Instead, it detects the marker id simply by obtaining the marker index of the dictionary it belongs to.



ArUco Markers that belong to the dictionary class 4x4_50
4x4 (Number of bits), 50 Dictionary Size.

ArUco Markers

Second step: Marker Identification.

- This process begins by applying a perspective transformation to obtain a square shape.
- The image is then thresholded using Otsu's method to separate the white and black parts.
- The marker is divided into a grid, and the value of each element is assigned depending on the values of the majority of the pixels.
- Using these values, the binary code of the candidate marker can be obtained.
- Since the markers can be rotated, four sets of binary codes can be obtained, each corresponding to a different clockwise rotation order starting point.
- Comparing these four binary codes with the marker dictionary, it is possible to identify the marker.

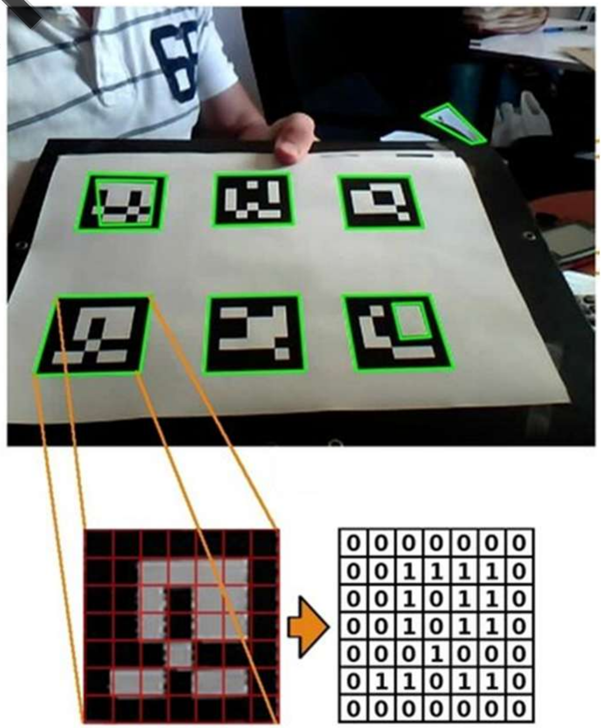


Image process for automatic marker detection

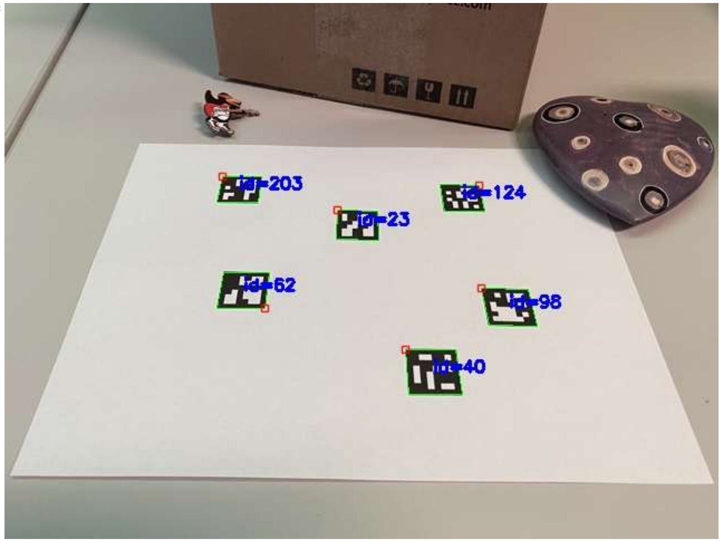
(a) Perspective transformation and threshold, (b)Grid. (c) Binary code.

(Garrido-Jurado, S., Muñoz-Salinas, R., 2014)

ArUco Markers

Final Step: Corner refinement and pose estimation.

- If a marker is detected, its pose with respect to the camera is estimated using different algorithms.
- This is known as the PnP (Perspective- n-point) problem, which is an inverse problem projection of the camera.
- For some cases, a linear regression of the pixels on the side of the marker, size and shape, is used, to calculate the line intersections, obtaining its pose.



ArUco pose detection and marker Identification *(OpenCV Detection of ArUco Markers Tutorial, 2023)*

ArUco Markers

How to install the ArUco Library (Windows/Ubuntu)

{Learn, Create, Innovate};

Installing the ArUco library

On a Windows/Ubuntu computer

Install the Open CV-contrib Python library

These instructions are for installing the Open CV library on a Windows/Ubuntu machine. Not to be followed for the Jetson Nano.

- Requirements
  - Python 3.x (3.4+) or Python 2.7.x from here.
  - NumPy package (for example, using *pip install numpy* command).
  - Matplotlib (*pip install matplotlib*) (Matplotlib is optional but recommended).

1. Go to OpenCV Python
2. Follow the instructions to install the contrib-python package (Option 2)

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

3. Verify the installation on a *cmd* window and run the following commands

```
python   # python3 for ubuntu
>>>import cv2
>>>print(cv2.__version__)
>>> print(dir(cv2.aruco))
```

4. The following should appear depending on the version installed

```
>>> import cv2
>>> print(cv2.__version__)
4.5.4
>>> print(dir(cv2.aruco))
['Board_create', 'CORNER_REFINE_APRILTAG', 'CORNER_REFINE_CONTOUR', 'CORNER_REFINE_NONE', 'C
ORNER_REFINE_SUBPIX', 'CharucoBoard_create', 'DICT_4X4_100', 'DICT_4X4_1000', 'DICT_4X4_250'
, 'DICT_4X4_50', 'DICT_5X5_100', 'DICT_5X5_1000', 'DICT_5X5_250', 'DICT_5X5_50', 'DICT_6X6_1
00', 'DICT_6X6_1000', 'DICT_6X6_250', 'DICT_6X6_50', 'DICT_7X7_100', 'DICT_7X7_1000', 'DICT_
7X7_250', 'DICT_7X7_50', 'DICT_APRILTAG_16H5', 'DICT_APRILTAG_16h5', 'DICT_APRILTAG_25H9', '
DICT_APRILTAG_25h9', 'DICT_APRILTAG_36H10', 'DICT_APRILTAG_36H11', 'DICT_APRILTAG_36h10', 'D
ICT_APRILTAG_36h11', 'DICT_ARUCO_ORIGINAL', 'DetectorParameters_create', 'Dictionary_create'
, 'Dictionary_create_from', 'Dictionary_get', 'Dictionary_getBitsFromByteList', 'Dictionary_
getByteListFromBits', 'GridBoard_create', '__doc__', '__loader__', '__name__', '__package__'
, '__spec__', 'calibrateCameraAruco', 'calibrateCameraArucoExtended', 'calibrateCameraCharuc
o', 'calibrateCameraCharucoExtended', 'custom_dictionary', 'custom_dictionary_from', 'detect
CharucoDiamond', 'detectMarkers', 'drawAxis', 'drawCharucoDiamond', 'drawDetectedCornersChar
uco', 'drawDetectedDiamonds', 'drawDetectedMarkers', 'drawMarker', 'drawPlanarBoard', 'estim
atePoseBoard', 'estimatePoseCharucoBoard', 'estimatePoseSingleMarkers', 'getBoardObjectAndIm
agePoints', 'getPredefinedDictionary', 'interpolateCornersCharuco', 'refineDetectedMarkers',
 'testCharucoCornersCollinear']
```

More information can be found in the opencv repository here.

Installing the ArUco library

```
Python 3.8.10 (default, Nov 22 2023, 10:22:35)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import cv2
>>> print(cv2.__version__)
4.2.0
>>> print(dir(cv2.aruco))
['Board_create', 'CORNER_REFINE_APRILTAG', 'CORNER_REFINE_CONTOUR', 'CORNER_REFINE_NONE', 'CORNER_REFINE_SUBPIX',
 'CharucoBoard_create', 'DICT_4X4_100', 'DICT_4X4_1000', 'DICT_4X4_250', 'DICT_4X4_50', 'DICT_5X5_100', 'DICT_5X5
_1000', 'DICT_5X5_250', 'DICT_5X5_50', 'DICT_6X6_100', 'DICT_6X6_1000', 'DICT_6X6_250', 'DICT_6X6_50', 'DICT_7X7_
100', 'DICT_7X7_1000', 'DICT_7X7_250', 'DICT_7X7_50', 'DICT_APRILTAG_16H5', 'DICT_APRILTAG_16h5', 'DICT_APRILTAG_
25H9', 'DICT_APRILTAG_25h9', 'DICT_APRILTAG_36H10', 'DICT_APRILTAG_36H11', 'DICT_APRILTAG_36h10', 'DICT_APRILTAG_
36h11', 'DICT_ARUCO_ORIGINAL', 'DetectorParameters_create', 'Dictionary_create', 'Dictionary_create_from', 'Dicti
onary_get', 'Dictionary_getBitsFromByteList', 'Dictionary_getByteListFromBits', 'GridBoard_create', '__doc__', '_
_loader__', '__name__', '__package__', '__spec__', 'calibrateCameraAruco', 'calibrateCameraArucoExtended', 'calib
rateCameraCharuco', 'calibrateCameraCharucoExtended', 'custom_dictionary', 'custom_dictionary_from', 'detectCharu
coDiamond', 'detectMarkers', 'drawAxis', 'drawDetectedCornersCharuco', 'drawDetectedDiamonds', 'drawDetectedMarke
rs', 'drawMarker', 'drawPlanarBoard', 'estimatePoseBoard', 'estimatePoseCharucoBoard', 'estimatePoseSingleMarkers
', 'getBoardObjectAndImagePoints', 'getPredefinedDictionary', 'interpolateCornersCharuco', 'refineDetectedMarkers
']
```

On the Jetson Nano

The Jetson Nano already has the OpenCV-contrib python library pre-installed

- The installed version of OpenCV is 4.2.0
- The version is not optimised for GPU

ArUco Markers

OpenCV (No ROS)

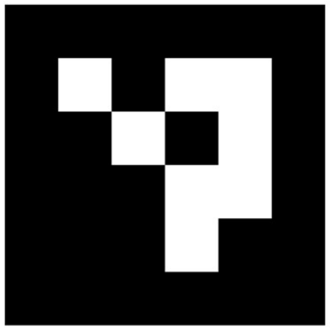
{Learn, Create, Innovate};

ArUco Markers in OpenCV (No ROS)

ArUco Markers in OpenCV

- To work with ArUco Markers, in OpenCV the user must perform several steps.
  - For this section, make sure the ArUco markers package is installed (previous slide)

1. Generate ArUco Markers
2. Recognise ArUco Markers
3. Calibrate the Camera
4. Pose Estimation of ArUco Markers



ArUco Marker
ID(0), 4x4

Generating ArUco Markers

- ArUco markers are generated using the OpenCV library, which provides a range of predefined dictionaries for different marker sizes and numbers of markers.
- The choice of dictionary depends on the application's specific requirements, such as the number of unique markers needed and the resolution of the input images.
- To generate markers, the following module can be used (some websites can also generate markers)

```
dict_aruco = aruco.Dictionary_get(aruco.DICT_4X4_50)
```

  - This command generates Markers from a dictionary 4x4 with 50 Unique ID's.

ArUco Markers in OpenCV (No ROS)

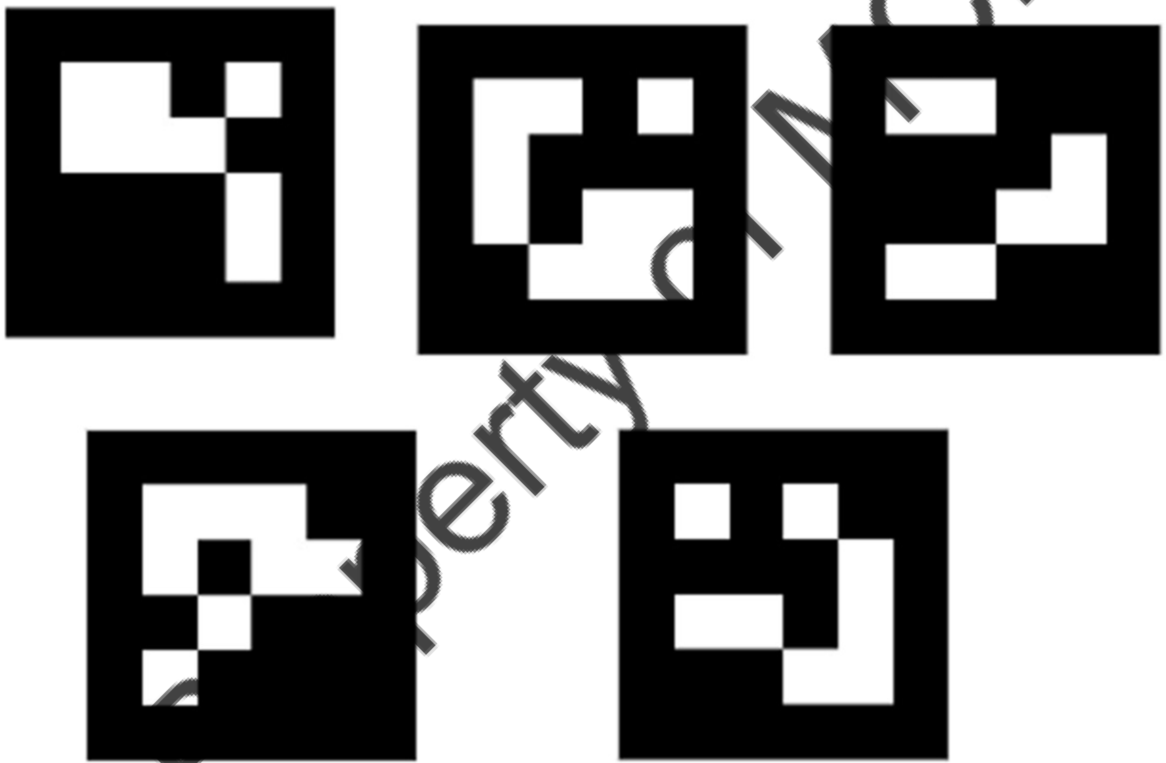
Generating ArUco Markers

- The following Code Generate ArUCo Markers inside the folder Aruco_Markers.
  - Replace the address with your own.
- The dictionary of ArUco Markes used is 4x4 with 250 unique ID's.
- The number of markers to generate is 10 and the size in pixels is 250.
- Other dictionaries are [available]

```
ARUCO_DICT = {
    "DICT_4X4_50": cv.aruco.DICT_4X4_50,
    "DICT_4X4_100": cv.aruco.DICT_4X4_100,
    "DICT_4X4_250": cv.aruco.DICT_4X4_250,
    "DICT_4X4_1000": cv.aruco.DICT_4X4_1000,
    "DICT_5X5_50": cv.aruco.DICT_5X5_50,
    "DICT_5X5_100": cv.aruco.DICT_5X5_100,
    "DICT_5X5_250": cv.aruco.DICT_5X5_250,
    "DICT_5X5_1000": cv.aruco.DICT_5X5_1000,
    "DICT_6X6_50": cv.aruco.DICT_6X6_50,
    "DICT_6X6_100": cv.aruco.DICT_6X6_100,
    "DICT_6X6_250": cv.aruco.DICT_6X6_250,
    "DICT_6X6_1000": cv.aruco.DICT_6X6_1000,
    "DICT_7X7_50": cv.aruco.DICT_7X7_50,
    "DICT_7X7_100": cv.aruco.DICT_7X7_100,
    "DICT_7X7_250": cv.aruco.DICT_7X7_250,
    "DICT_7X7_1000": cv.aruco.DICT_7X7_1000,
    "DICT_ARUCO_ORIGINAL": cv.aruco.DICT_ARUCO_ORIGINAL,
    "DICT_APRILTAG_16h5": cv.aruco.DICT_APRILTAG_16h5,
    "DICT_APRILTAG_25h9": cv.aruco.DICT_APRILTAG_25h9,
    "DICT_APRILTAG_36h10": cv.aruco.DICT_APRILTAG_36h10,
    "DICT_APRILTAG_36h11": cv.aruco.DICT_APRILTAG_36h11
}
```

```
import cv2
from cv2 import aruco
import os

# Save location
dir_mark = r'C:\Users\Test\Aruco_Markers'

# Parameter
num_mark = 10 #Number of markers
size_mark = 250 #Size of markers in pixels

### --- marker images are generated and saved --- ###
# Call marker type
dict_aruco = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_4X4_50)
print(dict_aruco)

for count in range(num_mark) :
    id_mark = count
    img_mark = cv2.aruco.generateImageMarker(dict_aruco, id_mark, size_mark)

    if count < 10 :
        img_name_mark = 'mark_id_0' + str(count) + '.jpg'
    else :
        img_name_mark = 'mark_id_' + str(count) + '.jpg'

    path_mark = os.path.join(dir_mark, img_name_mark)
    cv2.imwrite(path_mark, img_mark)
```

ArUco Markers in OpenCV (No ROS)



{Learn, Create, Innovate};

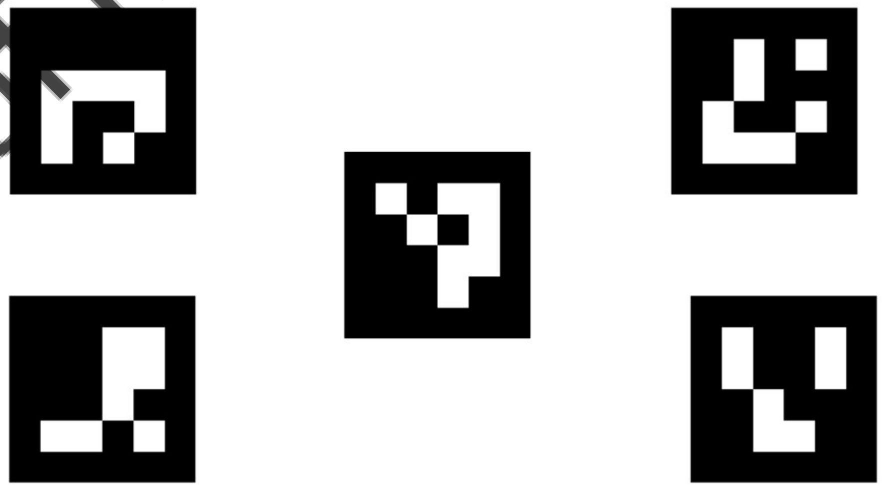
Activity 1

Simple ARUCO detection

Manchester Robotics

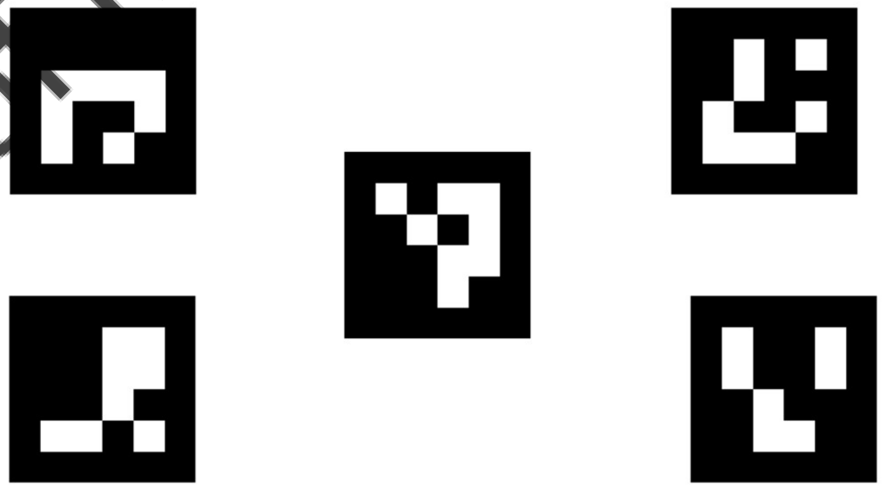
Simple ARUCO detection

- ArUco Marker detection consists of detecting markers by filtering them from other candidates, analysing their characteristics and comparing them to a predefined dictionary to ID the marker.
- This involves several steps:
  - Loading the image
  - Converting to grayscale
  - Marker detection



Simple ARUCO detection

- For this activity, a simple ArUco Detection will be performed, with the previously generated ArUco Markers
- Using PowerPoint/Paint or any other software you prefer, generate the following image by importing the previously generated ArUcos into the image (do not modify their size).
  - The image is not required to be perfect.
  - Once generated, save it with the name "Arucos.png"
- Make a new Python script to detect ArUcos called "ArucoDetect.py".



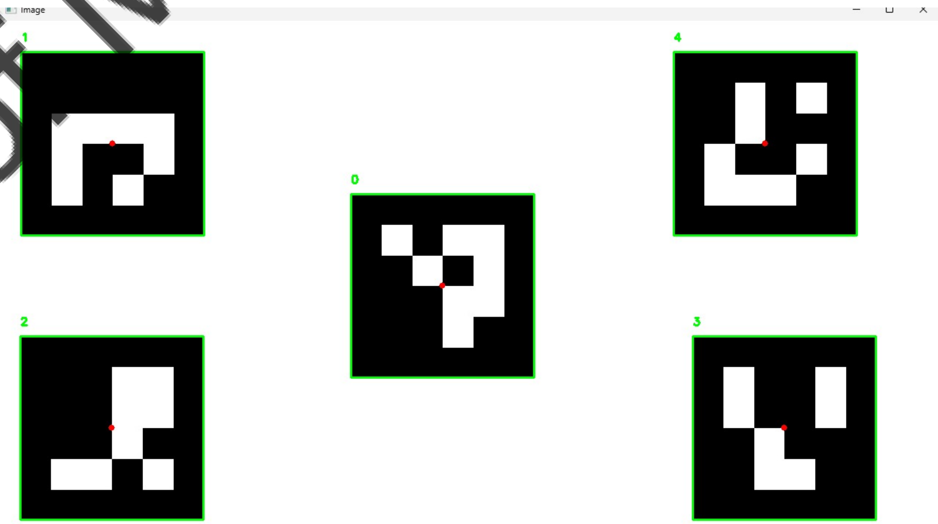
```python
import cv2 as cv
from cv2 import aruco
import os

img = cv.imread('Aruco_Markers/Arucos.png')

cv.imshow('img', img)

dictionary = cv.aruco.getPredefinedDictionary(cv.aruco.DICT_4X4_250)
parameters =  cv.aruco.DetectorParameters()
detector = cv.aruco.ArucoDetector(dictionary, parameters)

markerCorners, markerIds, rejectedCandidates = detector.detectMarkers(img)

# verify *at least* one ArUco marker was detected
if len(markerCorners) > 0:
    # flatten the ArUco IDs list
    ids = markerIds.flatten()
    # loop over the detected ArUCo corners
    for (markerCorner, markerID) in zip(markerCorners, ids):
        # extract the marker corners (which are always returned in
        # top-left, top-right, bottom-right, and bottom-left order)
        corners = markerCorner.reshape((4, 2))
        print(corners)
        (topLeft, topRight, bottomRight, bottomLeft) = corners
        # convert each of the (x, y)-coordinate pairs to integers
        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 the bounding box of the ArUCo detection
        cv.line(img, topLeft, topRight, (0, 255, 0), 2)
        cv.line(img, topRight, bottomRight, (0, 255, 0), 2)
        cv.line(img, bottomRight, bottomLeft, (0, 255, 0), 2)
        cv.line(img, bottomLeft, topLeft, (0, 255, 0), 2)
        # compute and draw the center (x, y)-coordinates of the ArUco
        # marker
        cX = int((topLeft[0] + bottomRight[0]) / 2.0)
        cY = int((topLeft[1] + bottomRight[1]) / 2.0)
        cv.circle(img, (cX, cY), 4, (0, 0, 255), -1)
        # draw the ArUco marker ID on the image
        cv.putText(img, str(markerID),
            (topLeft[0], topLeft[1] - 15), cv.FONT_HERSHEY_SIMPLEX,
            0.5, (0, 255, 0), 2)
        print("[INFO] ArUco marker ID: {}".format(markerID))
        # show the output image
        cv.imshow("Image", img)
        cv.waitKey(0)

cv.destroyAllWindows()
```

Results

- Run the code by typing

```
python ArucoDetect.py
#In Ubuntu
python3 ArucoDetect.py
```

- The following results should be visible.

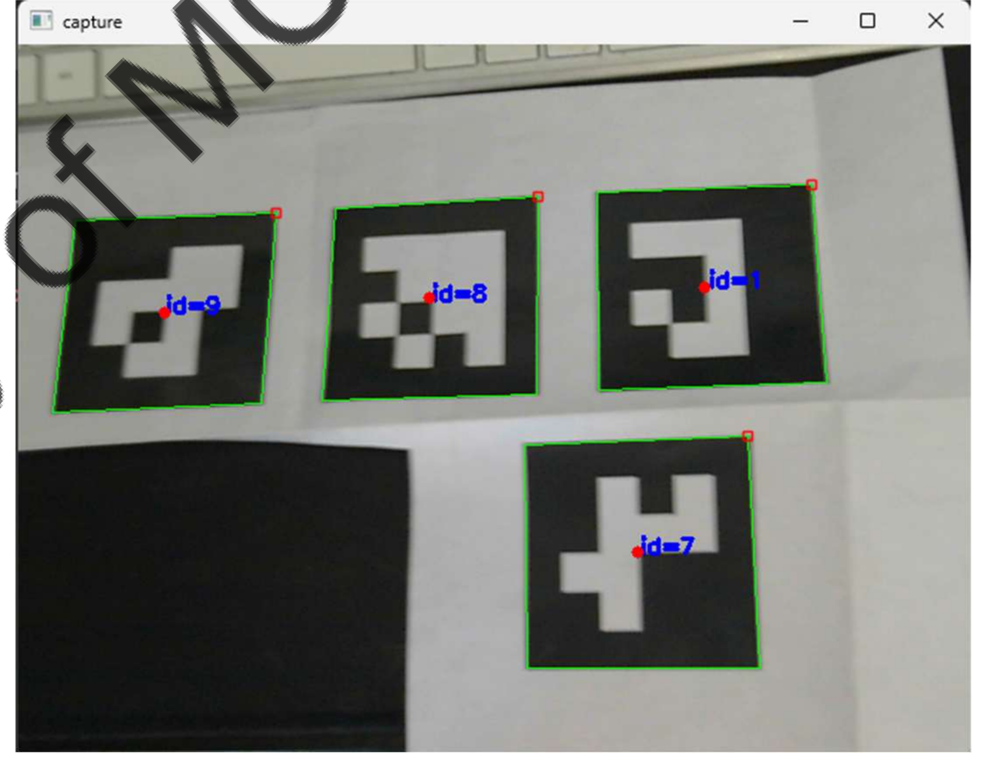


Simple ARUCO detection V1.2

- The following code (next slide) is the same as the previous one, just modified to detect ArUcos with the camera.
- Make a new script and change the name to "ArucoDetectvideo.py"
- Copy the code
- Run the code by typing

```
python ArucoDetectvideo.py
#In Ubuntu
python3 ArucoDetectvideo.py
```

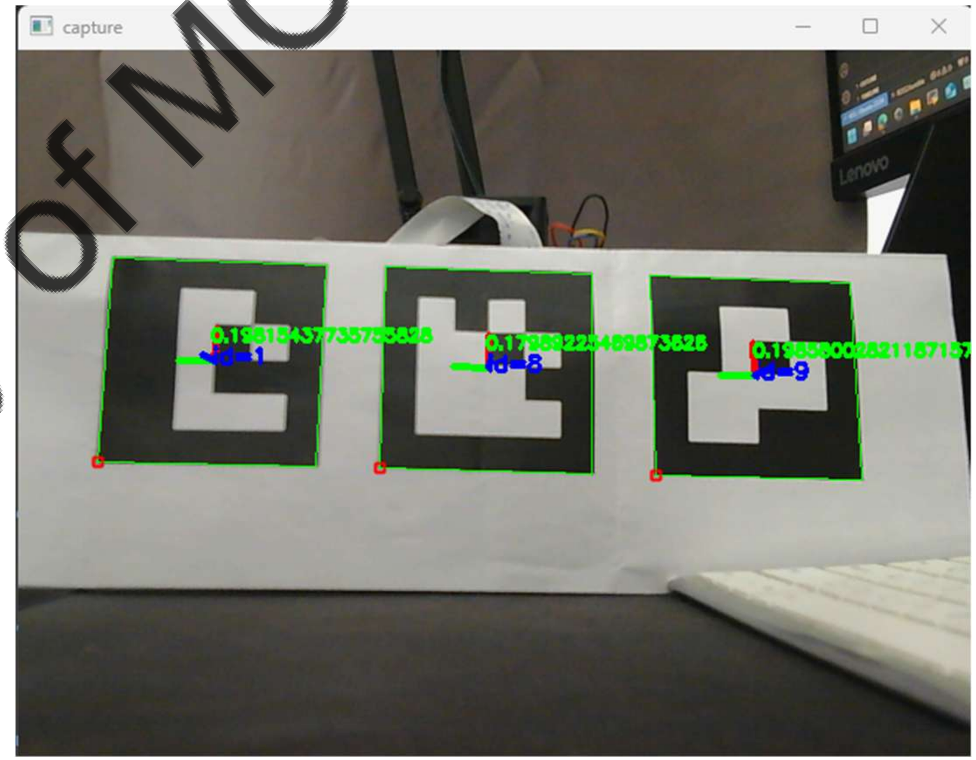
- The following results should be visible.



```python
import cv2 as cv
from cv2 import aruco

cam = cv.VideoCapture(0)

markerLength = 0.09

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

while True:

    ret, img = cam.read()
    if not ret:
        print("failed to grab frame")
        break

    markerCorners, markerIds, rejectedCandidates = detector.detectMarkers(img)

    # verify *at least* one ArUco marker was detected
    if len(markerCorners) > 0:
        cv.aruco.drawDetectedMarkers(img,markerCorners,markerIds)

        # flatten the ArUco IDs list
        ids = markerIds.flatten()
        # loop over the detected ArUCo corners
        for (markerCorner, markerID) in zip(markerCorners, ids):
            # extract the marker corners (which are always returned in
            # top-left, top-right, bottom-right, and bottom-left order)
            corners = markerCorner.reshape((4, 2))

            (topLeft, topRight, bottomRight, bottomLeft) = corners
            # convert each of the (x, y)-coordinate pairs to integers
            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]))

            # compute and draw the center (x, y)-coordinates of the ArUco
            # marker
            cX = int((topLeft[0] + bottomRight[0]) / 2.0)
            cY = int((topLeft[1] + bottomRight[1]) / 2.0)
            cv.circle(img, (cX, cY), 4, (0, 0, 255), -1)


    cv.imshow("capture", img)

    k = cv.waitKey(1)
    if k%256 == 27:
        # ESC pressed
        print("Escape hit, closing...")
        break

cam.release()
cv.destroyAllWindows()
```

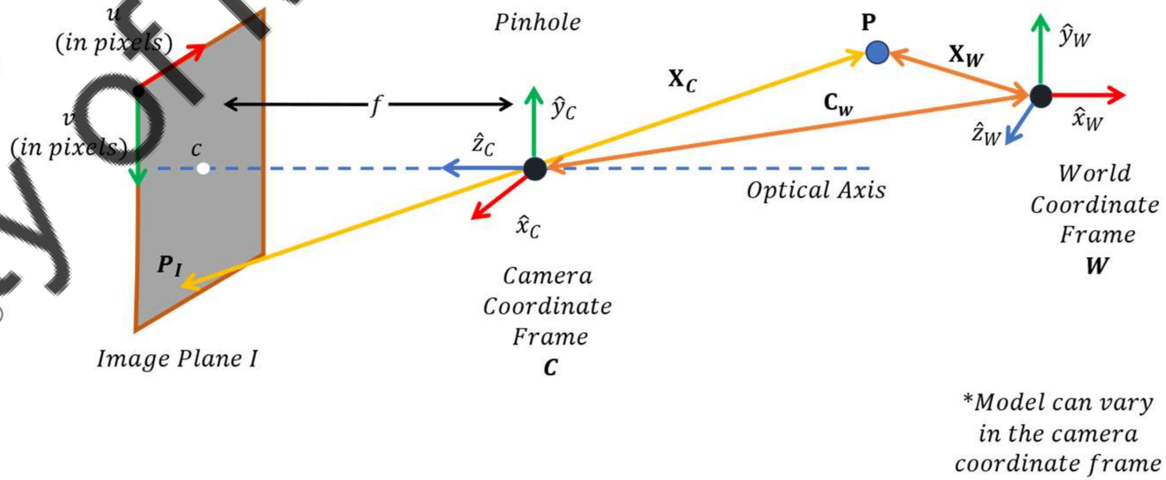
ArUco Markers Detection

- ArUco markers can be used to detect the position and orientation of an object.
- In other words, determine the position and orientation of a marker relative to the camera frame (or vice versa).
- This is very useful in robotics to determine the position of the robot relative to a fixed marker, or in augmented reality, to determine the position of the camera relative to a marker.
- To this end, a calibration process must be performed to obtain the "intrinsic parameters of the camera"

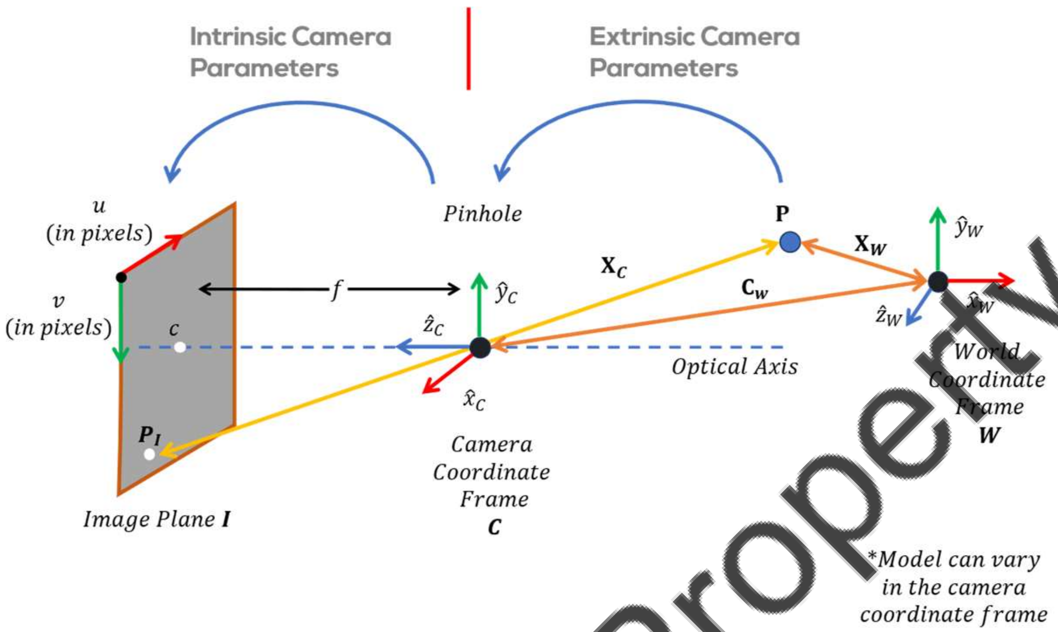


Camera Calibration

- Camera calibration refers to obtaining all the parameters regarding the camera to determine an accurate relationship between a 3D point in the real world and its corresponding 2D projection.
- The camera's parameters can be classified into external or extrinsic and internal or intrinsic parameters.

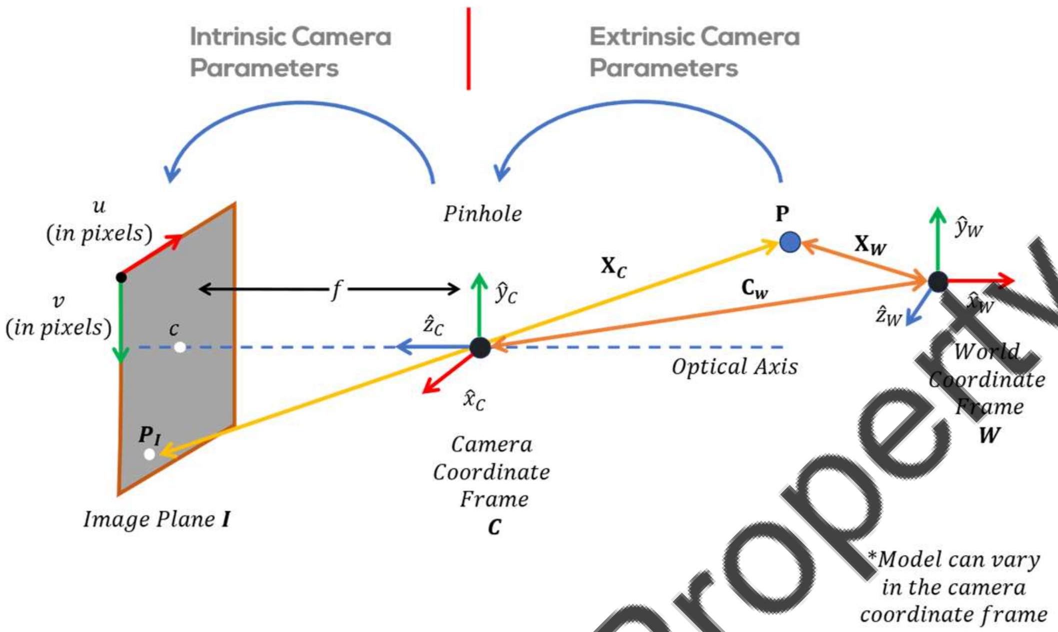


Camera Calibration



- Extrinsic parameters: Parameters required to transform a Point P in a world coordinate system W to a point in the camera coordinate system C (3D point to 3D point). In other words, refers to the pose (rotation and translation) of the camera with respect to a world coordinate system.
- Intrinsic parameters: Parameters regarding the internal lens system (focal length, optical centre/principal point, distortion coefficients, etc.). In other words, the parameters that transform a Point P in the camera coordinate system W to the Image Plane I.

Camera Calibration

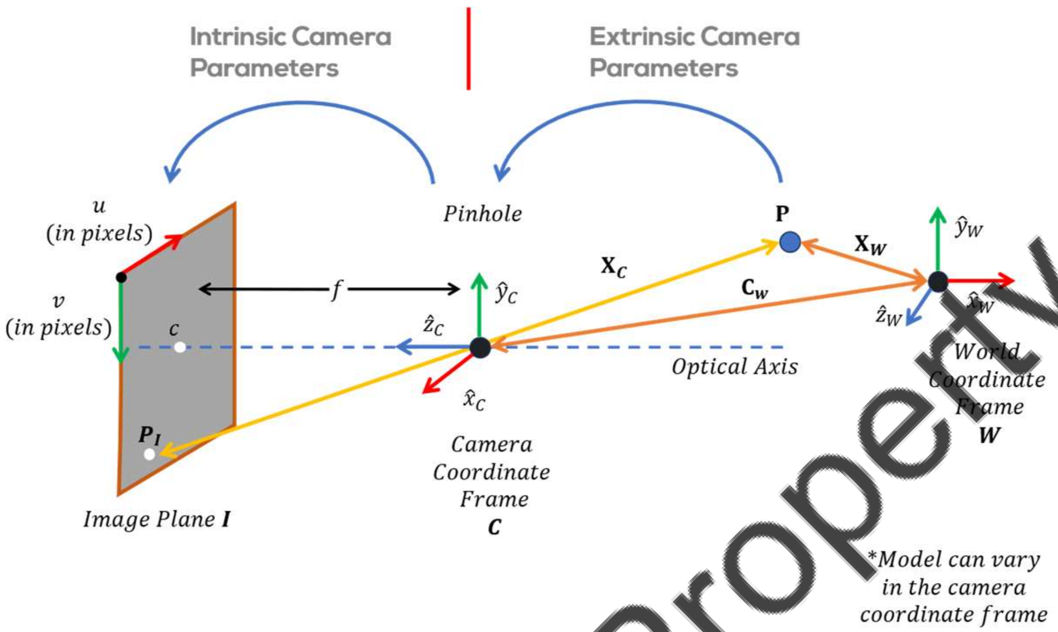


- The transformation can then be defined as follows:

$$\mathbf{x_I} = \mathbf{K[R|t]X_W}$$

Where, $\mathbf{x_I}$ is any point $\boldsymbol{P}_\mathbf{I}$ on the Image plane $\boldsymbol{I}$, $\mathbf{X_W}$ is any point $\boldsymbol{P}$ in the world frame, $\boldsymbol{R|t}$ is the rotation matrix and translation vector between the World frame $\boldsymbol{W}$ to the camera frame $\boldsymbol{C}$, $\boldsymbol{K}$ intrinsic parameter matrix, to transform the point in the camera frame $\boldsymbol{C}$ to the Image frame $\boldsymbol{I}$.

Camera Calibration

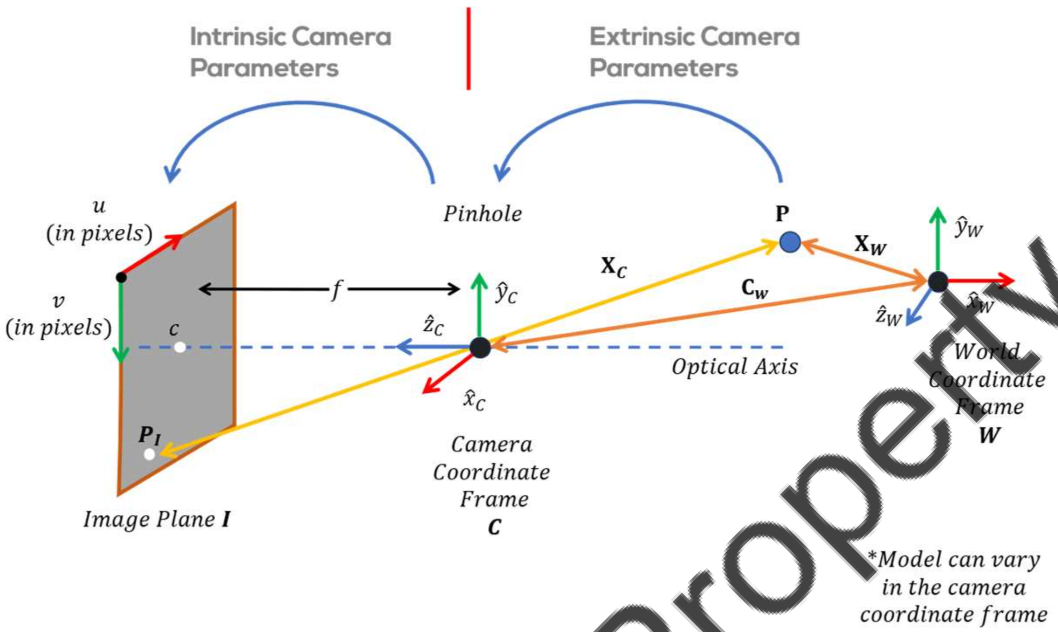


- The Extrinsic parameter matrix is defined as follows (please note that it is typically depicted using homogeneous coordinates)

$$[\boldsymbol{R}|\boldsymbol{t}] = \left[\begin{array}{ccc|c} r_1 & r_2 & r_3 & t_1 \\ r_4 & r_5 & r_6 & t_2 \\ r_7 & r_8 & r_9 & t_3 \end{array}\right]$$

Where, $r_{\mathrm{n}}, t_{\mathrm{n}}$ are the translation and rotation parameters of the of the camera coordinate frame $\boldsymbol{C}$ with respect to the world frame $\boldsymbol{W}$.

Camera Calibration



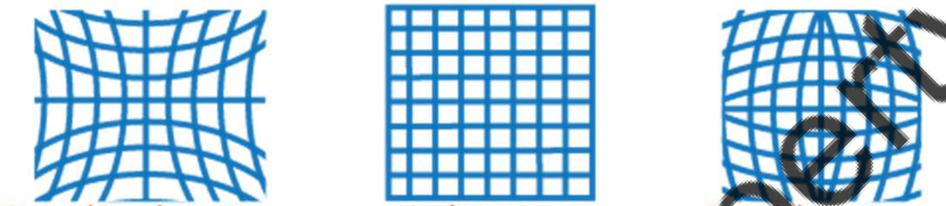
- The Intrinsic parameter matrix is defined as follows (please note that it is typically depicted using homogeneous coordinates)

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

Where, f_x, f_y are the focal lengths with respect to x and y respectively c_x and c_y are the x, y coordinates of the optical centre (typically image centre).

Camera Calibration

The intrinsic parameters must also consider the different distortions due to the lenses and the camera sensors.



Pincushion distortion
Positive radial displacement

No distortion

Barrel distortion
Negative radial displacement

*Images courtesy of [MATLAB]

- The radial distortion (barrel/pincushion) occurs because light rays bend more near the edges of a lens than they do at its optical center.

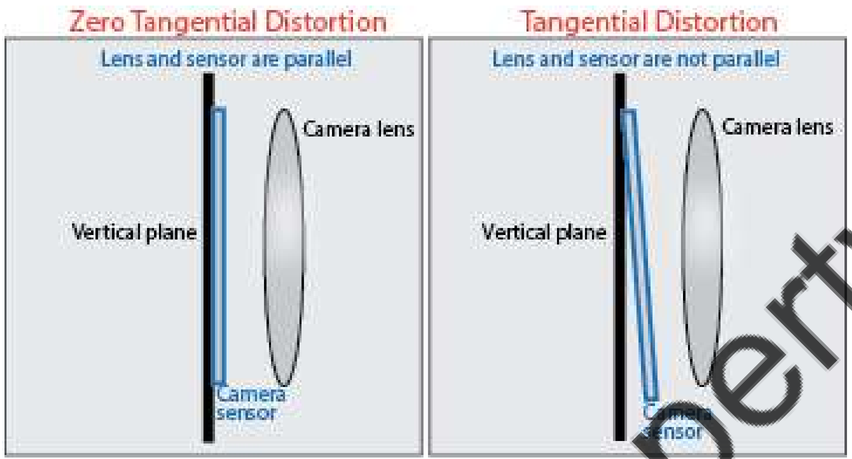
- The smaller the lens, the greater the distortion. This can be modelled as:

$$x_{distorted} = x(1 + k1 \cdot r2 + k2 \cdot r4 + k3 \cdot r6)$$

$$y_{distorted} = y(1 + k1 \cdot r2 + k2 \cdot r4 + k3 \cdot r6)$$

Where x, y are the pixel location normalised (measured from the centre of the image), k_1, k_2 *and* k_3 are the radial distortion coefficients and r is the radius i.e., $r^2 = x^2 + y^2$

Camera Calibration



- The tangential distortion occurs when the lens and the image plane are not parallel.

$$x_{distorted} = x + [2\, p_1\, x\, y + p_2\, (r^2 + 2x^2)]$$
$$y_{distorted} = y + [p_1\, (r^2 + 2y^2) + 2\, p_2\, x\, y]$$

- As in the previous model, x, y are the pixel location normalised, p_1, p_2 are the tangential distortion coefficients. Of the lens, and r is the radius from the centre.

*Images courtesy of MATLAB

Camera Calibration

- The goal of the calibration process is to find the parameters of the matrix K, the rotation matrix R, the translation vector t and the distortion coefficients $\mathrm{k}_1, k_2, p_1, p_2, k_3$.
- This is done by using a set of known 3D points ($\boldsymbol{X}_{\mathbf{W}}$) and their corresponding image coordinates $\mathbf{x}_{\mathbf{I}}$ (u, v).
- Obtaining the intrinsic and extrinsic parameters of the camera, is said to be “calibrated”.
- When using a mono camera with no reference to a world frame, the calibration amounts to only obtaining the intrinsic parameters, reference only to the camera frame $\mathbf{C}$.

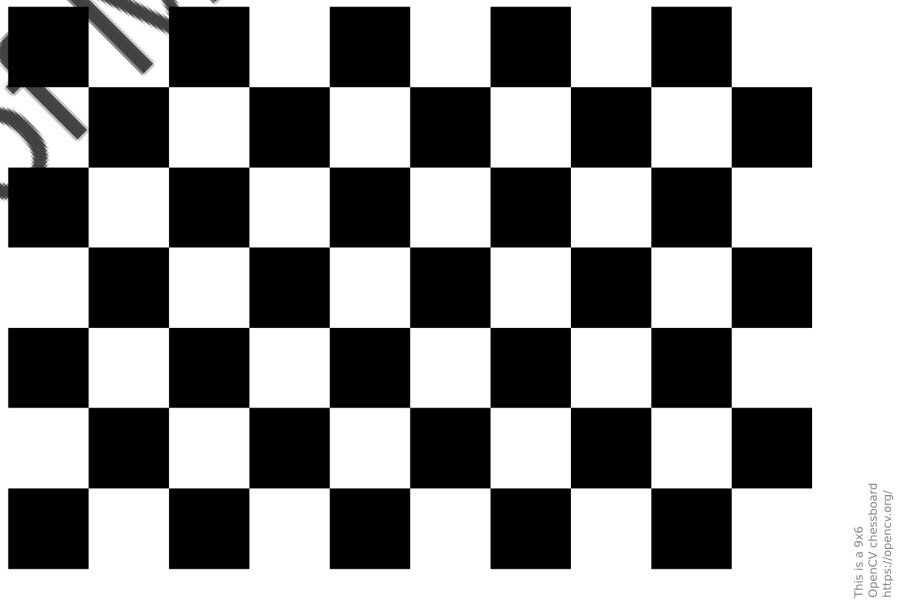
$$\mathbf{K}' = \mathbf{K}[\mathbf{I}|\mathbf{0}] \Rightarrow \mathbf{x}_{\mathbf{I}} = \mathbf{K}'\mathbf{X}_{\mathbf{W}}$$


```yaml
camera_matrix:
- - 415.153748671152
  - 0.0
  - 319.3653123397526
- - 0.0
  - 415.4888245132374
  - 232.80120575418516
- - 0.0
  - 0.0
  - 1.0
dist_coeff:
- - 0.0004099851341626534
  - 0.048343955855519344
  - -0.007544038459241634
  - 0.00021114292420816653
  - -0.08104098709508069
```

Camera Calibration

Calibration Methods

- Different calibration methodologies exists, the most common ones are:
  - **Calibration pattern:** Consists of capturing several images of an object or pattern of known dimensions from different viewpoints and using optimisation algorithms to obtain the parameters.
  - **Geometric Clues:** Using geometric clues in the scene like straight lines and vanishing points which can be used for calibration.
  - **Deep Learning Methods:** Use deep learning to approximate the parameters using simple images.



{Learn, Create, Innovate};

Activity 2

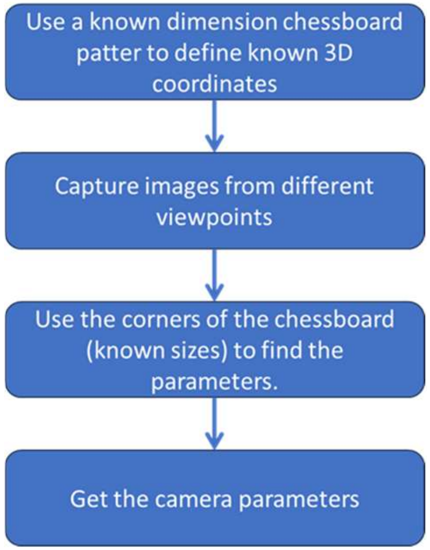
Simple Camera Calibration (OpenCV)

MCR2 Manchester Robotics

Activity 2

- In this activity, a simple camera calibration algorithm will be performed using OpenCV libraries.
- To this end, some images need to be taken, using the chessboard pattern.
- The following code allows the user to take some images using the keyboard “space” key
- Print the chessboard from [here]
- Make a Python script called “save_images.py”
- Create a folder called “images” on the same folder as your script.

Camera Calibration Flowchart



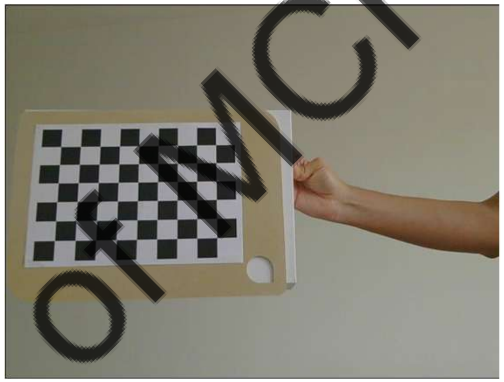
```python
import numpy as np
import cv2

# === [1] Initialize Camera ===

# Open the default camera (device index 0)
cam = cv2.VideoCapture(0)

# Create a named window to display the live camera feed
cv2.namedWindow("capture")

# Counter to keep track of saved images
img_counter = 0

# === [2] Main Loop: Capture, Display, and Save Frames ===

while True:
    # Read a frame from the camera
    ret, frame = cam.read()

    # If the frame wasn't captured correctly, exit the loop
    if not ret:
        print("⚠️ Failed to grab frame")
        break

    # Display the frame in the window
    cv2.imshow("capture", frame)

    # Wait for a key press (1ms)
    key = cv2.waitKey(1)

    if key % 256 == 27:
        # ESC key pressed: Exit the loop
        print("🔚 ESC pressed, closing...")
        break

    elif key % 256 == 32:
        # SPACE key pressed: Save the current frame as an image
        img_name = f"images/chess{img_counter}.png"
        cv2.imwrite(img_name, frame)
        print(f"📸 Saved: {img_name}")
        img_counter += 1

# === [3] Cleanup ===

# Release the camera and close all OpenCV windows
cam.release()
cv2.destroyAllWindows()
```

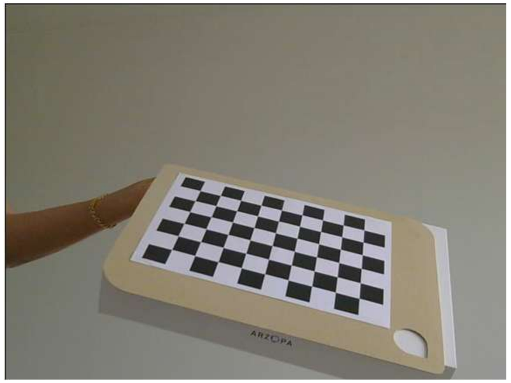
Activity 2

- Run your code

```
python saveImages.py
#In Ubuntu
python3 saveImages.py
```

- Make at least 30 images in different positions of the chessboard pattern (close, far, right left centre, rotated, etc.), covering as many rotations and translations as possible (the more captures and movements are registered, the more accurate will be the calibration)
- Your resulting images should be inside the folder "images"



Activity 2

- Make a new Python script called "CameraCal.py" to use the previous images inside the folder "images" to calibrate the camera.
- For this code the YAML library is used to dump the parameters obtained into a YAML file named "calibration_matrix.yaml"

```
pip install PyYAML
```

- Copy the following code, save it and run it using the following commands

```
python CameraCal.py
#In Ubuntu
python3 CameraCal.py
```

```python
import numpy as np
import cv2 as cv
import glob
import yaml

# === [1] Setup ===

# Size of the chessboard pattern (number of inner corners per chessboard row and column)
chessboard_size = (9, 6)

# Image resolution
frame_size = (640, 480)

# Termination criteria for cornerSubPix: max 30 iterations or epsilon < 0.001
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# Prepare known object points in 3D space (0,0,0), (1,0,0), ..., (8,5,0)
objp = np.zeros((chessboard_size[0] * chessboard_size[1], 3), np.float32)
objp[:, :2] = np.mgrid[0:chessboard_size[0], 0:chessboard_size[1]].T.reshape(-1, 2)

# Arrays to store 3D object points and 2D image points from all calibration images
objpoints = []  # 3D points in real world space
imgpoints = []  # 2D points in image plane

# === [2] Load and process all chessboard images ===

images = glob.glob('images/*.png')
```

```python
for fname in images:
    img = cv.imread(fname)
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

    # Try to find the chessboard corners
    ret, corners = cv.findChessboardCorners(gray, chessboard_size, None)
    print(fname, "  Chessboard found:", ret)

    if ret:
        # Refine corner positions and save them
        corners2 = cv.cornerSubPix(gray, corners, (11, 11), (-1, -1),
criteria)
        objpoints.append(objp)
        imgpoints.append(corners2)

        # Draw the corners on the image
        cv.drawChessboardCorners(img, chessboard_size, corners2, ret)
        cv.imshow('Chessboard', img)
        cv.waitKey(500)

cv.destroyAllWindows()

# === [3] Calibrate the camera ===

ret, camera_matrix, dist_coeffs, rvecs, tvecs = cv.calibrateCamera(
    objpoints, imgpoints, gray.shape[::-1], None, None)

print("\n=== Calibration Results ===")
print("Reprojection error:", ret)
print("\nCamera Matrix:\n", camera_matrix)
print("\nDistortion Coefficients:\n", dist_coeffs)
print("\nRotation Vectors:\n", rvecs)
print("\nTranslation Vectors:\n", tvecs)
```

```python
# === [4] Undistort one of the images ===

img = cv.imread('images/chess3.png')
h, w = img.shape[:2]

# Compute the optimal new camera matrix to minimize unwanted pixels
new_camera_matrix, roi = cv.getOptimalNewCameraMatrix(camera_matrix,
dist_coeffs, (w, h), 1, (w, h))

# Undistort the image
undistorted = cv.undistort(img, camera_matrix, dist_coeffs, None,
new_camera_matrix)

# Crop the image using the ROI
x, y, w, h = roi
undistorted_cropped = undistorted[y:y+h, x:x+w]

# Save the undistorted image
cv.imwrite('calibresult.png', undistorted_cropped)

# === [5] Save calibration data to a YAML file ===

calibration_data = {
    'camera_matrix': camera_matrix.tolist(),
    'dist_coeff': dist_coeffs.tolist(),
    'rotation_vector': [r.tolist() for r in rvecs],
    'translation_vector': [t.tolist() for t in tvecs]
}

with open("calibration_matrix.yaml", "w") as f:
    yaml.dump(calibration_data, f)
```

```
# === [6] Calculate mean reprojection error ===

mean_error = 0
for i in range(len(objpoints)):
    projected_imgpoints, _ = cv.projectPoints(objpoints[i], rvecs[i], tvecs[i],
camera_matrix, dist_coeffs)
    error = cv.norm(imgpoints[i], projected_imgpoints, cv.NORM_L2) /
len(projected_imgpoints)
    mean_error += error

print("\nMean reprojection error: {:.4f}".format(mean_error / len(objpoints)))
```

Activity 2: Results

calibration_matrix.yaml


```yaml
camera_matrix:
- - 757.0352999532336
  - 0.0
  - 260.4335709291612
- - 0.0
  - 757.479721764773
  - 257.4309714872177
- - 0.0
  - 0.0
  - 1.0
dist_coeff:
- - 0.01159436944733824
  - -0.268202281884559
  - 0.0012289250365545724
  - -0.026537555718567696
  - 0.818296721394548
```

{Learn, Create, Innovate};

Activity 3

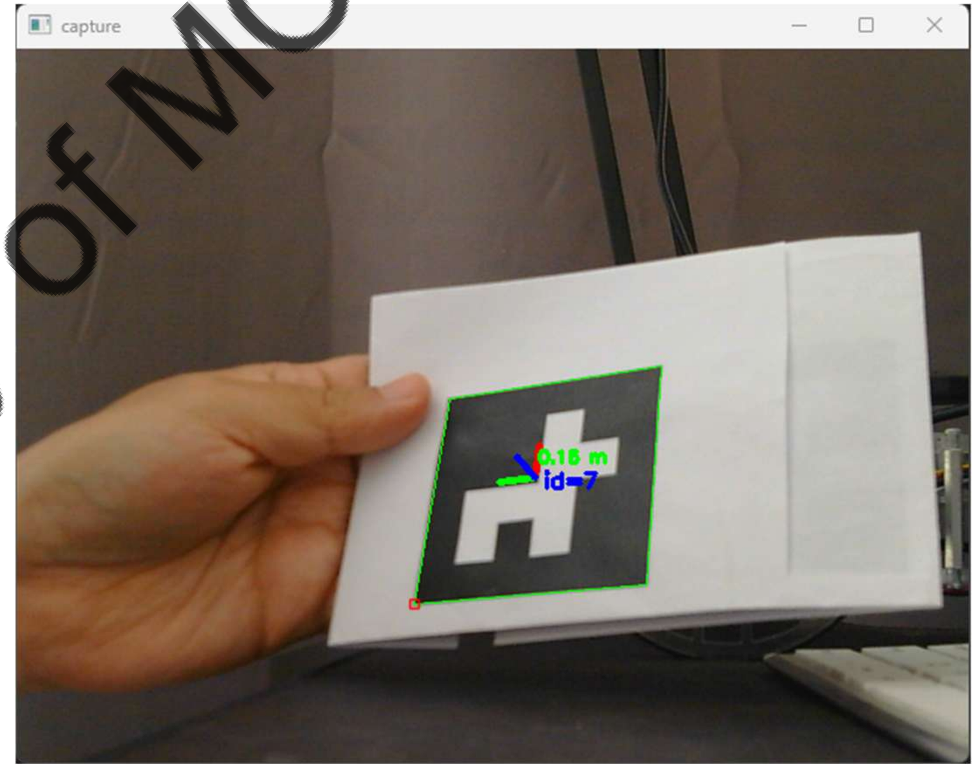
Simple ArUco pose detection (OpenCV)

MCR2 Manchester Robotics

Aruco Pose detection

- Using the previous camera calibration, we can now use ArUco Markers to detect the position of a marker relative to the camera.
- This is done by using the function "`aruco.estimatePoseSingleMarkers`".
- Make a Python script called "ArucoPose.py"
- Copy the following code
- Run the file

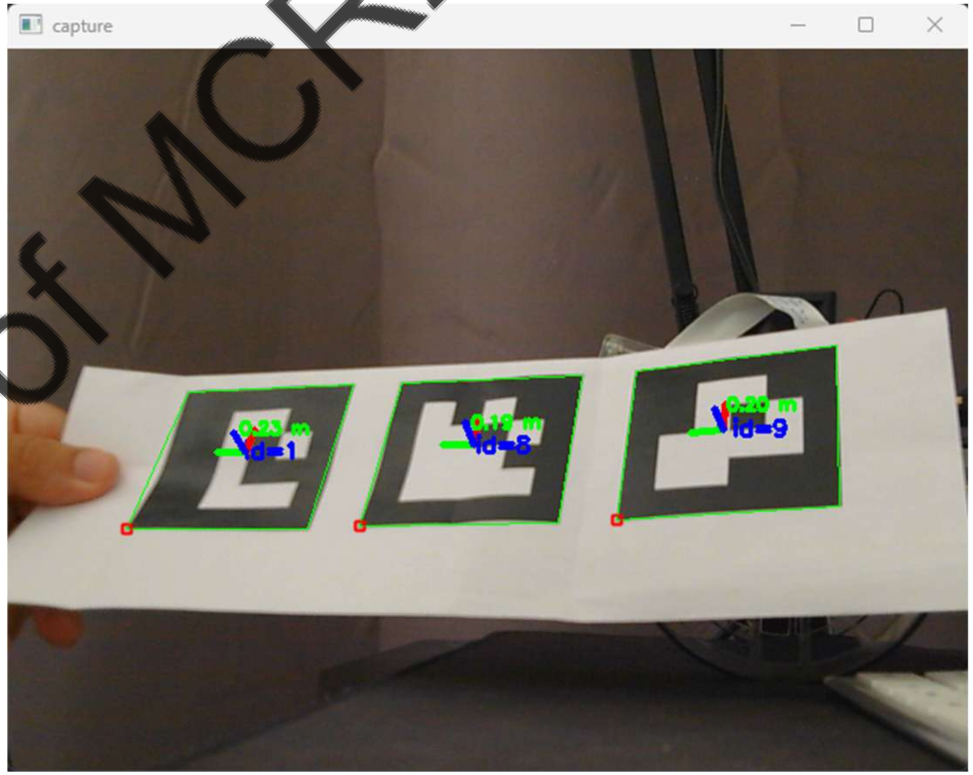
```
python ArucoPose.py
#In Ubuntu
python3 ArucoPose.py
```



```python
import numpy as np
import cv2 as cv
from cv2 import aruco
import yaml
from yaml.loader import SafeLoader

# === [1] Setup and Configuration ===

# Open camera device (adjust index if needed)
cam = cv.VideoCapture(2)

# Load camera calibration data from YAML file
with open('calibration_matrix.yaml') as f:
    data = yaml.load(f, Loader=SafeLoader)
    print("Loaded Camera Matrix:\n", np.array(data['camera_matrix']))

# Calibration parameters
K = np.array(data['camera_matrix'])  # Camera matrix
D = np.array(data['dist_coeff'])     # Distortion coefficients

# Define the real-world length of the marker’s side (in meters)
markerLength = 0.06  # 6 cm

# Set up ArUco dictionary and detector
dictionary = aruco.getPredefinedDictionary(aruco.DICT_4X4_50)
parameters = aruco.DetectorParameters()
detector = aruco.ArucoDetector(dictionary, parameters)
```

```python
while True:
    # Capture frame from the camera
    ret, img = cam.read()
    if not ret:
        print("❌ Failed to grab frame")
        break

    # Convert image to grayscale
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

    # Detect ArUco markers
    markerCorners, markerIds, rejectedCandidates = detector.detectMarkers(gray)

    if len(markerCorners) > 0:
        for i in range(len(markerIds)):
            # Estimate pose of each marker
            rvec, tvec = aruco.estimatePoseSingleMarkers(markerCorners[i],
markerLength, K, D)[:2]

            # Compute distance from camera to marker
            distance = np.linalg.norm(tvec[0][0])
            # Calculate marker center in image
            corners = markerCorners[i].reshape((4, 2))
            topLeft, topRight, bottomRight, bottomLeft = corners
            cX = int((topLeft[0] + bottomRight[0]) / 2.0)
            cY = int((topLeft[1] + bottomRight[1]) / 2.0)
            # Draw marker and its pose axis
            aruco.drawDetectedMarkers(img, markerCorners, markerIds)
            cv.drawFrameAxes(img, K, D, rvec, tvec, 0.01)
            # Annotate distance on image
            cv.putText(img, f"{distance:.2f} m", (cX, cY - 15),
                        cv.FONT_HERSHEY_SIMPLEX, 0.4, (0, 255, 0), 2)

    # Display result
    cv.imshow("capture", img)
```

```
    # Break loop if ESC is pressed
    k = cv.waitKey(1)
    if k % 256 == 27:
        print("🔚 ESC pressed, closing...")
        break

# === [3] Cleanup ===

cam.release()
cv.destroyAllWindows()
```



ArUco Markers

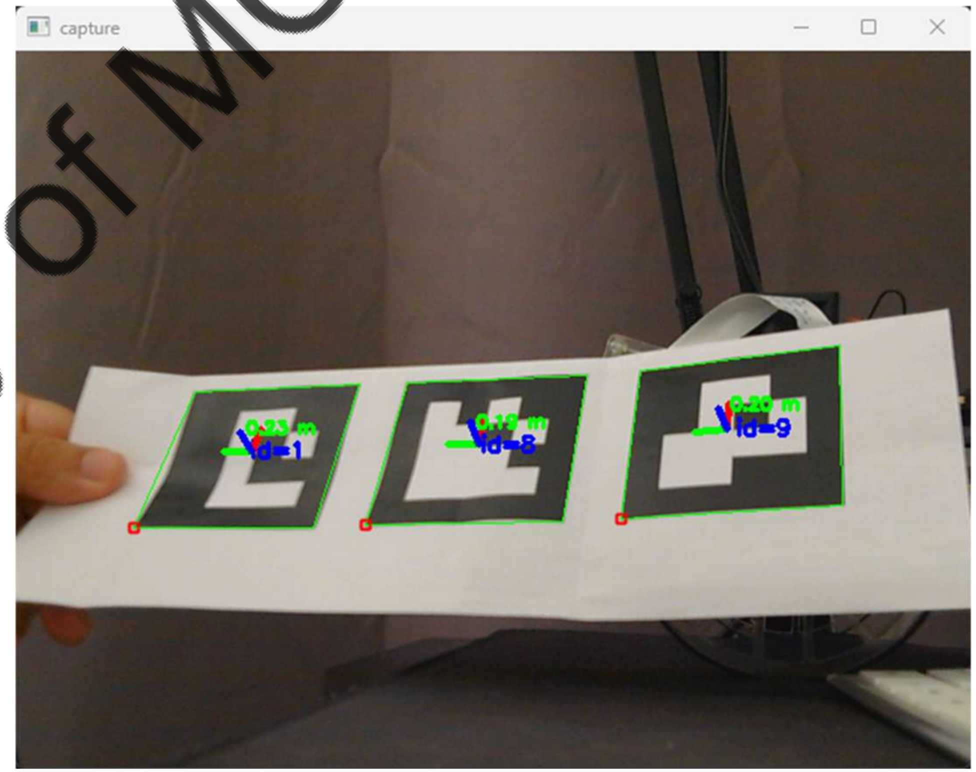
ArUco Markers in ROS

{Learn, Create, Innovate};

ArUco ROS

The following tutorial is for ROS only.

If using the Puzzlebot with the MCR2 image, you do not need to follow this tutorial, since the ArUco Library and calibration are already done; follow the instructions on how to activate the ArUco detection on the Puzzlebot Manual.



ArUco ROS

- ROS has two different ways of interacting with ArUco Markers.
- The classic way is to use "cvbridge" to get the image from a camera and then use the OpenCV libraries to rectify the image and detect the ArUco Markers.
- Another way is to use some predefined nodes to detect ArUco Markers, usually these nodes come predefined and optimised for ArUco detection. Also, some of them do not require OpenCV to be installed.
- In this tutorial, the second manner will be used.

- For this tutorial, the following nodes must be installed in your Ubuntu environment.
  - USB cam node

```
sudo apt install ros-humble-usb-cam
```

  - Camera calibration node

```
sudo apt install ros-humble-camera-calibration
```

  - Ros Aruco opencv node

```
sudo apt install ros-humble-aruco-opencv
```

ArUco ROS – USB Camera

- Run the USB cam node as follows

```
ros2 run usb_cam usb_cam_node_exe   --ros-args  -p
pixel_format:=mjpeg2rgb  -p image_width:=640   -p
image_height:=360   -p framerate:=30.0   -p
video_device:=/dev/video0
```

- The usb camera node publish the image form an USB camera, under the “image_raw” topic and the calibration information in the topic “camera_info”.
  - More information about the camerainfo message here.

```
mario@MarioPC:~$ ros2 topic list
/camera_info
/image_raw
/image_raw/compressed
/image_raw/compressedDepth
/image_raw/theora
/parameter_events
/rosout
```

- Pixel_format: Pixel format for Video4linux device more info here.
- Image width/height: Frame width/height in pixels
- Framerate: Camera polling frequency, Hz
- Video device: Device driver's entry point in Linux kernel's virtual filesystem. Check in terminal using “`cd /dev/`” and look for the “`video0`” or “`video1`” that is the port where your camera is connected.
- To publish the “camera_info” topic, the node requires the camera parameters to be set in a YAML file.
- More information about the parameters can be found here and here.

ArUco ROS – Camera Calibration

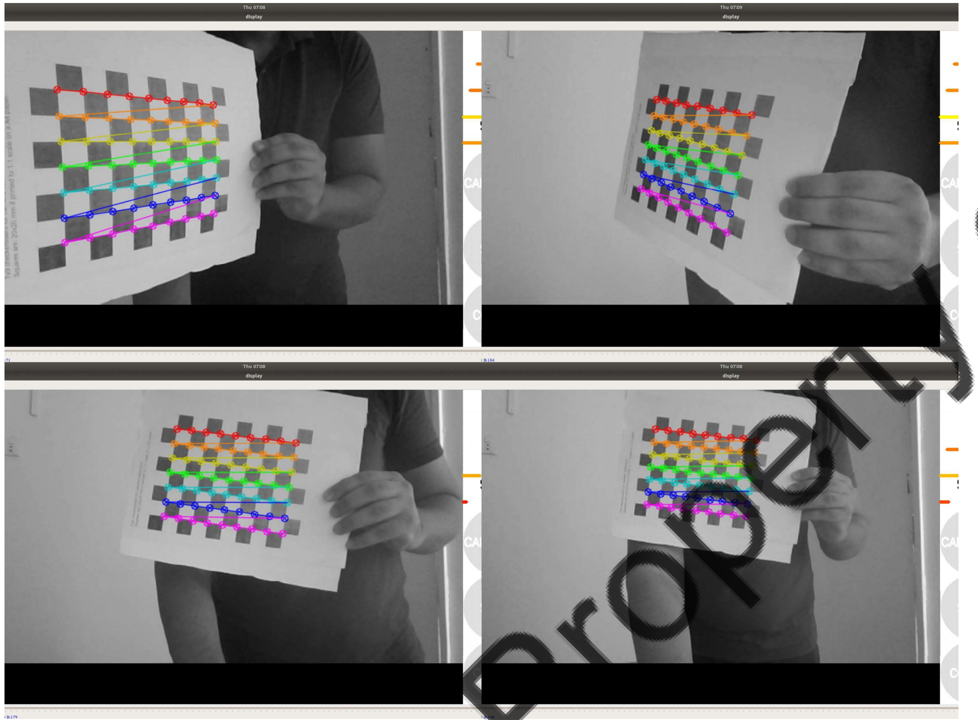
- ROS provides a simple GUI for calibrating the camera, making the process more efficient.
- Print the chessboard from here
- Run the USB camera node as in the previous slide
- Run the camera calibration node as follows

```
ros2 run camera_calibration cameracalibrator --size
6x9 --square 0.028   --ros-args -r image:=/image_raw
```

- This will open a calibration window
- **Size**: is the size of the chessboard (inner squares)
- **Square**: dimension in meters of the squares
- **Image**: image topic

- To get a good calibration, you will need to move the chessboard around in the camera frame from left to right, top to bottom, towards and away from the field of view, and you must tilt the chessboard left, right, top and bottom.
- Green bars will show you your progress: **X bar** – left/right in field of view, **Y bar** – top/bottom in field of view, **Size bar** – toward/away, **Skew** bar shows the tilting progress.

ArUco ROS – Camera Calibration



- When all 4 bars are green and enough data is available for calibration, the CALIBRATE button will light up.
- Click it to see the results. It takes a few minutes for calibration to take place.
- After the calibration is completed, press the "save" button.
- Data is saved to "/tmp/calibrationdata.tar.gz"
- To use the data, unzip the "calibration.tar.gz" and move the "ost.yaml" YAML file to a known location (usually in your package or in ".ros" hidden folder)

ArUco ROS – ArUco Detection

- Once the calibration parameters have been obtained, we can use them to publish the "/camera_info" topic from the USB camera node.

```
ros2 run usb_cam usb_cam_node_exe   --ros-args   -p
camera_name:=brio100   -p pixel_format:=mjpeg2rgb -p
frame_id:=camera_optical  -p image_width:=640   -p
image_height:=360   -p framerate:=30.0   -p
video_device:=/dev/video0   -p
camera_info_url:=file:///home/mario/.ros/camera_info/b
rio100.yaml
```

calibration file (YAML).

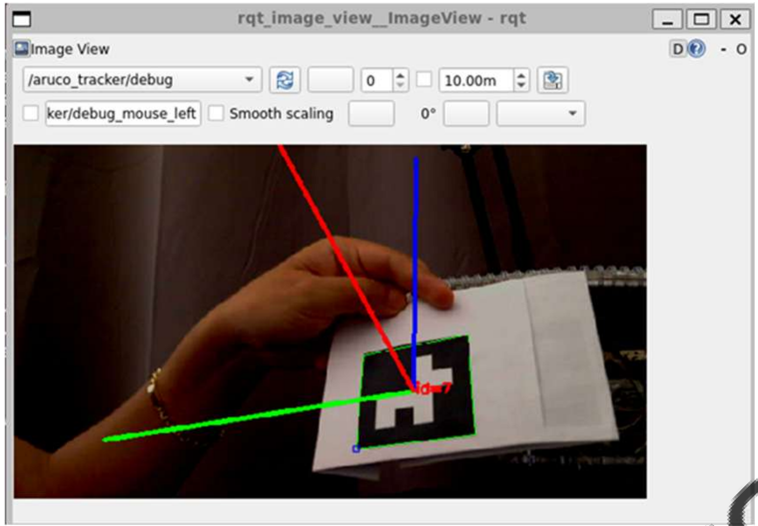
- Set a camera frame for the TF.

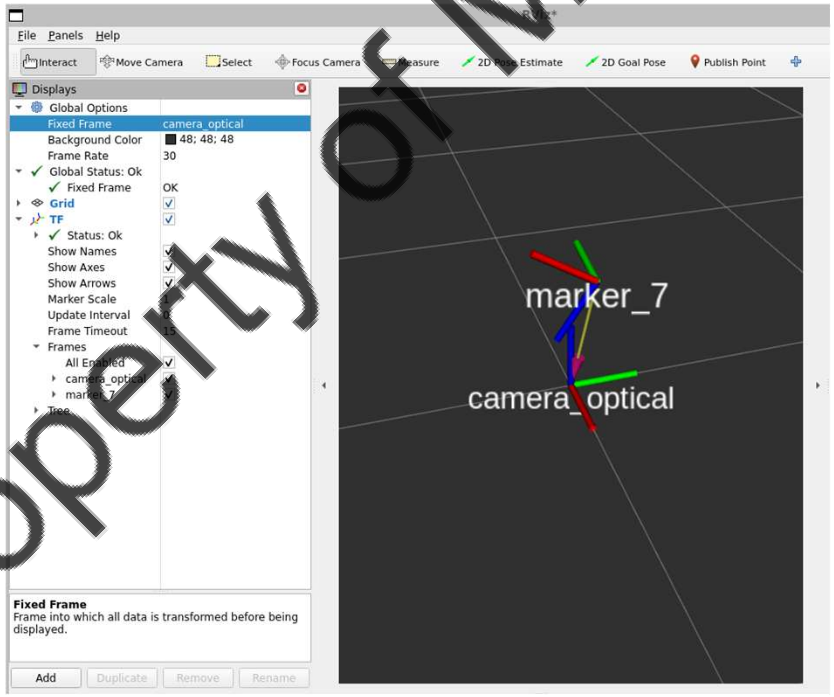
- Launch the ArUco detection node as follows.

```
ros2 run aruco_opencv aruco_tracker_autostart --ros-
args -p cam_base_topic:=image_raw -p marker_size:=0.06
```

- Launch the "rqt_image_view" node to observe the results on the "/aruco_tracker/debug" topic.
- For each received image, aruco_tracker will publish a message on aruco_detections topic
- Put the marker in front of the camera. If the marker is detected, the markers array should contain the marker poses.
- The marker poses are also published on TF. You can visualize the data in RViz by setting fixed frame to the frame_id of the camera and adding the TF display.

ArUco ROS – ArUco Detection





```
header:
  stamp:
    sec: 1744985994
    nanosec: 569603000
  frame_id: camera_optical
markers:
- marker_id: 7
  pose:
    position:
      x: 0.015349765352222931
      y: 0.0548152271727785
      z: 0.31459540920231605
    orientation:
      x: -0.6716829641065714
      y: 0.7196651122397428
      z: -0.1549226084636425
      w: -0.08320521222128949
boards: []
```

Thank you

{Learn, Create, Innovate};

T&C

Terms and conditions

{Learn, Create, Innovate};

Manchester Robotics

Terms and conditions