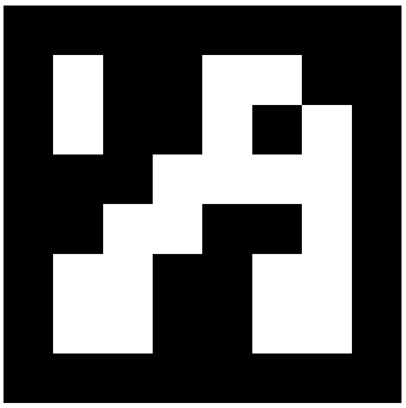
Lab #5: Visual Perception and Control using ArUco Markers

In this experiment, you will utilize the monocular camera on your TurtleBot3 to detect ArUco markers and estimate their 3D pose relative to the robot. Finally, you will implement a visual servoing controller that allows the robot to search for, track, and follow a marker autonomously.

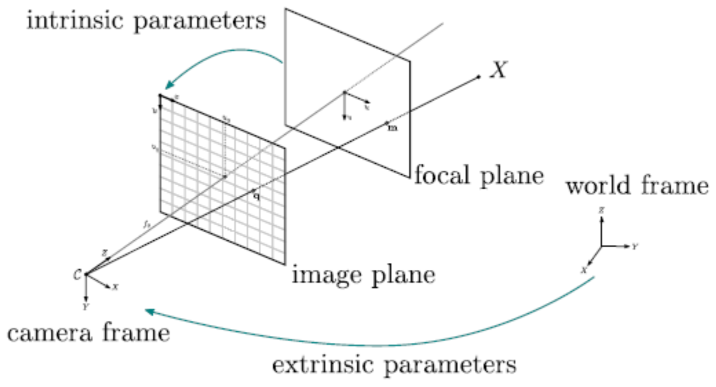
ArUco markers are binary square fiducial markers composed of a wide black border and an inner binary matrix that determines their identifier (ID). To accurately estimate the position of these markers in 3D space, the camera must be calibrated. You will be provided with a `camera_parameters.mat` file containing the intrinsic parameters.
The relationship between a 3D point in the world (X, Y, Z) and its 2D projection (u, v) on the image plane is defined by the pinhole camera model:

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{1}$$

where $\lambda = Z_c$ is the Z coordinate of the world point in the camera frame. The first matrix contains the *Intrinsic Parameters* (focal lengths f_x, f_y and principal point c_x, c_y), which you will load from the provided file. The second matrix contains the *Extrinsic Parameters*, consisting of a rotation matrix $\mathbf{R}$ and a translation vector $\mathbf{t}$, which define the position and orientation of the camera relative to the world coordinate frame.



(a) Example of ArUco Marker.



(b) Pinhole Camera Model.

Figure 1: Visual Perception Fundamentals.

You will implement algorithms to detect these markers and use the position error to close the control loop, allowing the robot to "chase" the marker.

Visual Servoing Control Logic

To implement the "Chase" behavior (Task 3), you must derive control inputs based on the marker's position in the camera frame. The camera calibration returns a translation vector $\mathbf{t} = [t_x, t_y, t_z]^T$.

In the standard optical frame, z represents depth and x represents the lateral axis. However, for the robot's coordinate system (where x_R is forward and y_R is left), the coordinates are transformed as follows:

$$x_R = t_z \quad \text{(Forward Distance)} \tag{2}$$

$$y_R = -t_x \quad \text{(Lateral Deviation)} \tag{3}$$

The heading error e_θ (the angle the robot needs to turn to face the marker) and the distance error e_d are calculated as:

$$e_\theta = \text{atan2}(y_R, x_R) \tag{4}$$

$$e_d = x_R - d_{desired} \tag{5}$$

where $d_{desired}$ is the target stopping distance (e.g., 0.2m).
The robot is controlled using a Proportional (P) controller with saturation limits:

$$v = k_{lin} \cdot e_d \tag{6}$$

$$\omega = k_{ang} \cdot \theta_{err} \tag{7}$$

Note: The logic should prevent the robot from reversing; if $e_d < 0$, set $v = 0$.

Environment Set-up

Refer to the Lab #0 document to connect to the **TurtleBot3** and prepare the MATLAB environment in order to start the algorithm implementation.

Important Note

Download the provided `camera_parameters.mat` file and place it in your working directory. This file contains the calibration data required for accurate pose estimation. You can load it then by using: `load(camera_parameters.mat);`

Things to do:

1. Write an **ArucoDetect.m** script. With the robot stationary, capture the real-time video feed from the robot's camera. Detect any ArUco marker within the Field of View (FOV). Overlay the detected marker with a square outlining the marker boundaries.

2. Write a **SearchAndStop.m** script. Implement a logic where the robot rotates in place (yaw motion) to search for a marker.
   - If no marker is detected: The robot rotates.
   - If a marker is detected: The robot stops immediately.
   - If the marker is removed from the FOV: The robot resumes rotation until a marker is found again.
   - Display the real-time feed with the detection overlay during operation.

3. Write a **VisualServoing.m** script. Implement a "Chase" behavior using a Proportional Controller.
   - **Search:** If no marker is detected, the robot must rotate at a constant speed to scan the environment.
   - **Control Loop:** Once detected, calculate the linear (v) and angular (ω) velocities based on the distance and heading errors.
   - **Constraints:** Apply saturation limits to velocities. Ensure the robot **does not reverse** (i.e., if the robot is too close, $v = 0$).
   - **Stop Condition:** The robot should stop automatically if the distance error and heading error are within small thresholds (e.g., $<$ 5cm and $< 10°$).
   - **Output:** Display the real-time feed with the marker overlay. Additionally, plot the **distance & angle errors** over time.

- **Record a video** demonstration for **application 3 only**, upload the video to a cloud platform (e.g., Google Drive, YouTube), and include the shared link in your report.
- Submit your report **via SUCourse** until the report submission deadline.

Post-Lab Report Deadline: 24 December 2025, 23:59 via **SUCourse**

- Your report must include:
  - **Introduction**
  - **Procedure**
  - **Results**

- **Conclusion**
- **Discussion**
- **Appendix**
- Provide your **MATLAB** codes in Appendix section appropriately.

Answer the following questions in the Discussion section of your Post-lab report:

1. Explain the significance of the Camera Intrinsic Parameters in distance estimation. What happens if these parameters are incorrect?
2. How did the estimated distance vary as you moved the marker physically closer or further? Was the measurement noisy?
3. Describe the robot's behavior in Task 3 when the marker was moved laterally quickly. Did it overshoot or lose track?
4. How did lighting conditions affect the ArUco detection? Did you experience any false negatives or detection jitter?

Important Note

Don't forget to add the link of the video for Task 3!

Appendix

A. MATLAB ArUco Interface

```
% I: Input image (grayscale or RGB)
% 'DICT_5X5_250': The specific dictionary used in our lab
[ids, locs] = readArucoMarker(I, "DICT_5X5_250");

% ids: Vector of detected IDs
% locs: 4x2 array of (x,y) pixel coordinates for the corners
```

B. Coordinate Systems: Camera vs. Robot

A critical challenge in Visual Servoing (Task 3) is the difference between the **Optical Frame** and the **Robot Body Frame**.

- **Optical Frame:** By standard convention, the Z-axis points forward (depth), X points right, and Y points down.
- **Robot Frame:** As defined in Lab #1, the Robot's X-axis points forward, and Y points left.

When calculating errors for your controller, you must map the camera translation $\mathbf{t} = [t_x, t_y, t_z]$ to the robot frame variables:
Note: Failing to invert the lateral axis will cause the robot to turn in the opposite direction of the marker.

C. Visual Servoing Logic (Task 3)

Unlike the continuous controllers in Lab #2, Task 3 requires a state-based logic loop. Below is the recommended structure:

```
while true
    % Get Image & Detect Marker
    if isempty(ids)
        % STATE: SEARCHING
    else
        % STATE: TRACKING
        % Apply Control
    end
    send(velPub, velMsg);
    drawnow limitrate;
end
```

D. Visualization Technique

To prevent execution lag, we avoid clearing and redrawing figures in every loop iteration. Instead, we initialize plot handles once and update their data properties dynamically.

```
% SETUP (Before Loop)
hIm = imshow(zeros(480, 640, 'uint8')); hold on;
hPlot2D = plot(0, 0, 'g-', 'LineWidth', 3);
receive(imgSub, 10);

% UPDATE (Inside Loop)
if ~isempty(ids)
    set(hIm, 'CData', I);
    corners = locs(:,:,1);
    boxX = [corners(:,1); corners(1,1)];
    boxY = [corners(:,2); corners(1,2)];
    set(hPlot2D, 'XData', boxX, 'YData', boxY, 'Visible', 'on');
else
    set(hPlot, 'Visible', 'off');
end
```