Object Detection and Camera Calibration

Sourish Merugumilli and Pratham Kamble

25/04/2024

—

Project Report

—

Guided by:

Sanjeev Sharma sir

Contents

1) Introduction

The project at the Department of Robotics and Remote Handling at Bhabha Atomic Research Centre is dedicated to developing a real-time object detection model using machine learning techniques.

Our primary focus is on harnessing the power of YOLOv8 for object detection, with a camera strategically placed atop a conveyor belt. We've selected objects like bottles and boxes for detection.

Additionally, we're integrating camera calibration techniques to obtain real-world coordinates of detected objects, enhancing precision.

By combining YOLOv8's capabilities with camera calibration, our goal is to create a robust system capable of accurately detecting and tracking objects in real-time, ultimately improving efficiency and safety in industrial environments.

2) Project Setup

The initial stages of our project were marked by challenges arising from internet restrictions imposed for security reasons within our work environment. As a result, we encountered obstacles in directly installing Python libraries crucial for our development process.

To overcome this hurdle, we devised a workaround strategy. Initially, on a computer with internet access, we downloaded the required Python libraries using the following command:

pip download -r requirements.txt -d "./dependencies"

Subsequently, we compressed the downloaded dependencies folder into a zip file and transferred it to the machine lacking internet access, typically through a pen drive. Once transferred, we extracted the contents of the dependencies zip file on the target machine.

Navigating to the dependencies folder, we executed the following command to install the libraries within the environment devoid of internet connectivity:

*for %x in (dir *.whl) do python -m pip install %x*

It's worth noting that organizing project libraries within a virtual environment is a recommended practice for maintaining a clean and isolated development environment. To achieve this, we utilized the following commands:

python -m venv venv
venv/scripts/activate

This meticulous setup process ensured that despite the internet restrictions, we could effectively access and utilize essential Python libraries for our project development.

3)Data Collection and Preparation

To train the model effectively, data was collected from various sources, including Kaggle datasets, manual captures at the conveyor belt site, and images taken at home.

The LabelImg tool facilitated the labeling process, ensuring accurate annotations for training.

Data augmentation techniques were applied to enhance model robustness and increase dataset size.

Scripts were developed for data augmentation and label format conversion from VOC to YOLO format.

Additionally, scripts were created for partitioning data into train, test, and validation sets.

Synthetic data generation methods were explored to diversify the dataset further, with dedicated scripts for rapid generation.

Why Data Augmentation

Data Augmentation is done particularly on image datasets to increase the robustness of model and grow our dataset.

Data augmentation includes flipping, translating, scaling, rotating, changing colors etc of images.

This helps us in reducing bias of model. For example, if most of bottle dataset contains images of bottles in vertical position then the model trained on this dataset will perform poorly on images of bottle in other orientations.

To avoid this, we can rotate the dataset images randomly before training.

Inside *augmented_images_and_format_conversion* folder the following files are present:

dataAug.ipynb: Script with line-by-line explanation for Data Augmentation using albumentations library

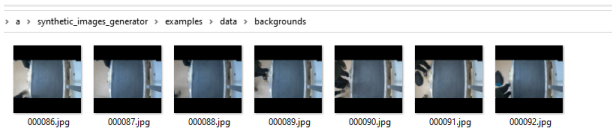
splitdata.ipynb: To split dataset into train test and val

visualize.ipynb: To visualize the bounding boxes and labels

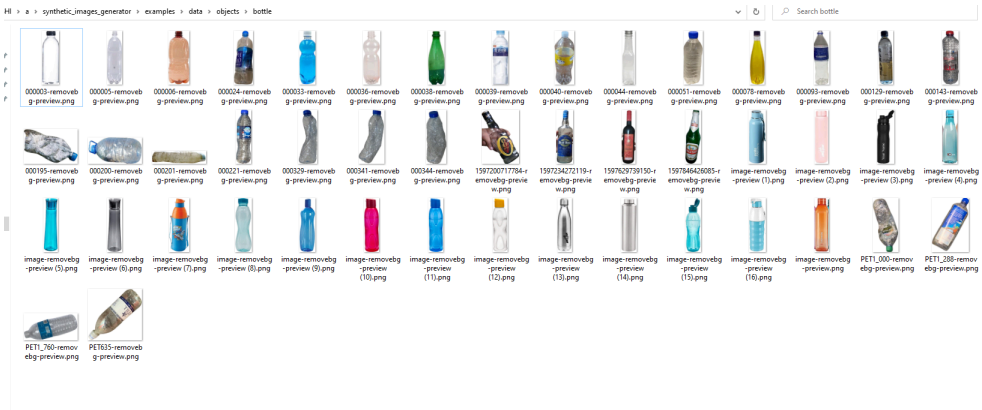
Synthetic Data Generation

This is a method of quickly generating a dataset using background images and images of object without background (stored in .png format because it can save transparency information).

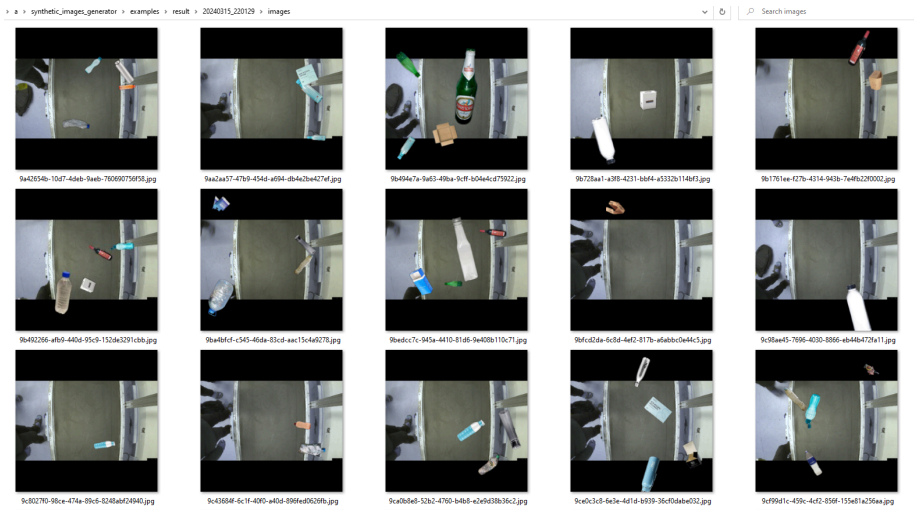
To illustrate:



Background Images



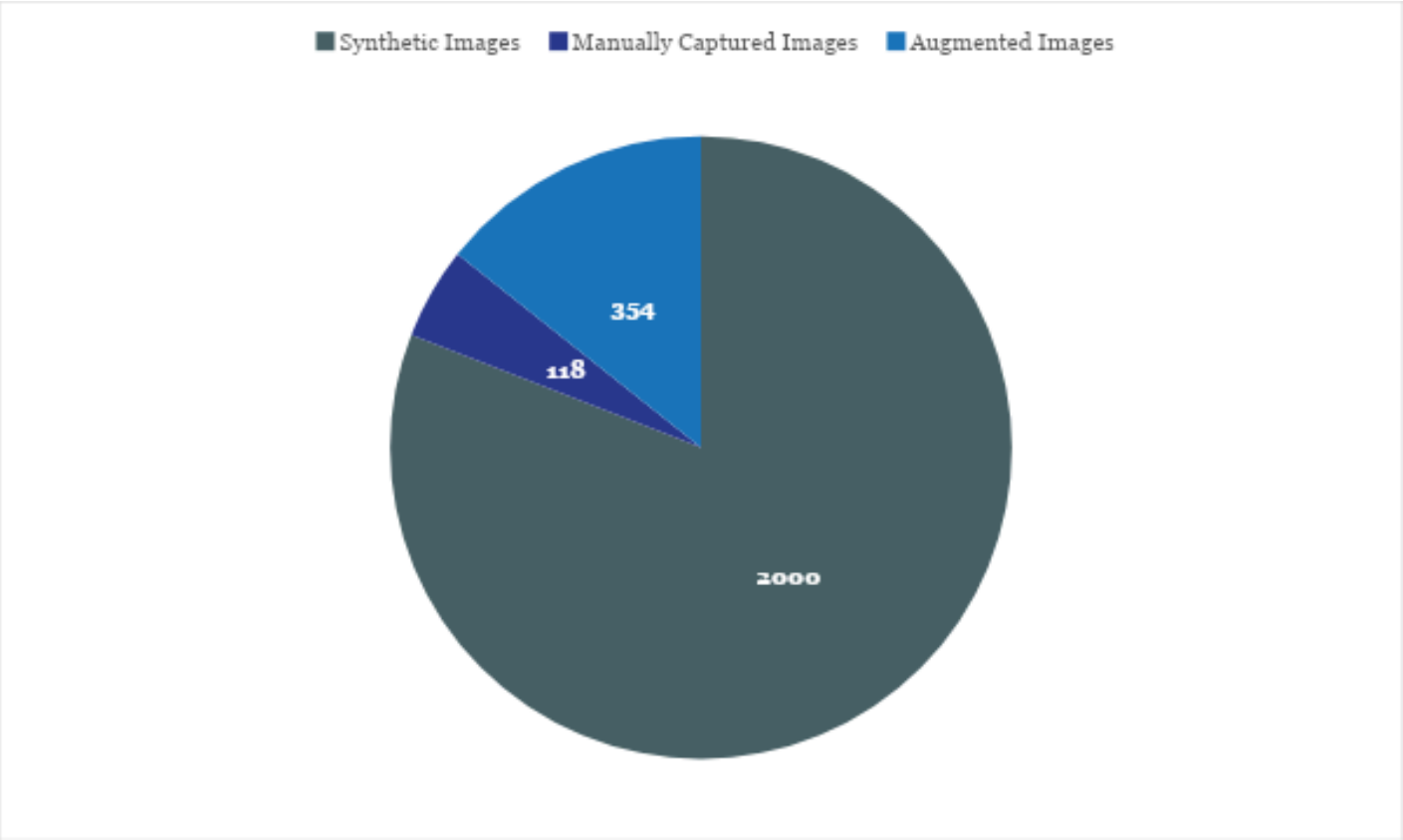
Object Images



Synthetic Dataset

Final Dataset Composition

The dataset used to train the final model consisted of the following images:



Dataset breakdown

Synthetic Images

These Images were generated by technique mentioned in the report above

Manually Captured Images

These images were captured and labelled individually at BARC on the conveyor belt with objects placed at random positions

Augmented Images

These images were created by augmenting the manually captured images using the technique mentioned before.

4) Model Training and Evaluation

Our model training and evaluation process involved several essential steps to ensure optimal performance and accuracy.

Data Organization

We meticulously arranged our dataset, comprising images and corresponding label files, in a specific folder structure.

-images/
-image000001.jpg
-image000002.jpg
-labels/
-image000001.txt
-image000002.txt

Data Import and Configuration in Google Colab

To leverage the computational capabilities of Google Colab for model training, we zipped the dataset folder and imported it into the Colab environment. Additionally, we created a "data.yaml" file within the Colab directory to specify essential configuration parameters. This file included class names, the number of classes (nc), and the location of the training and validation datasets. For example:

```
names:
 - bottle
 - box
nc: 2
train: //location of train folder
val: // location of val folder
```

Model Training

We utilized a pre-trained YOLOv8 model, "yolov8n.pt," as the base model for transfer learning. This model choice balanced performance with computational requirements, ensuring compatibility with our hardware constraints. The model training process commenced within the Google Colab environment, where we executed the following code:

```
%pip install ultralytics
import ultralytics
ultralytics.checks()
!yolo predict model=yolov8n.pt data=data.yaml epochs=15
```

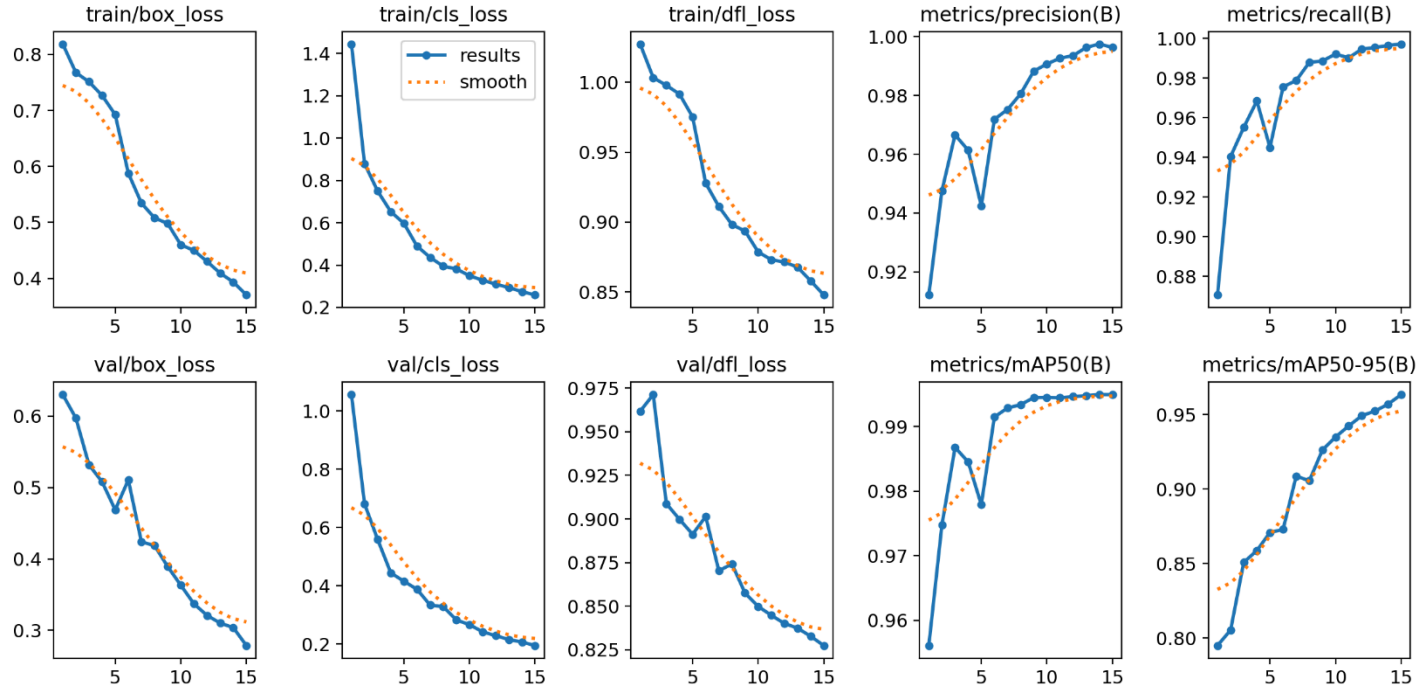
This command initiated the training process, leveraging the specified dataset and configuration parameters. We aimed to achieve convergence over 15 epochs, optimizing the model's performance for accurate object detection.

Model Evaluation Results

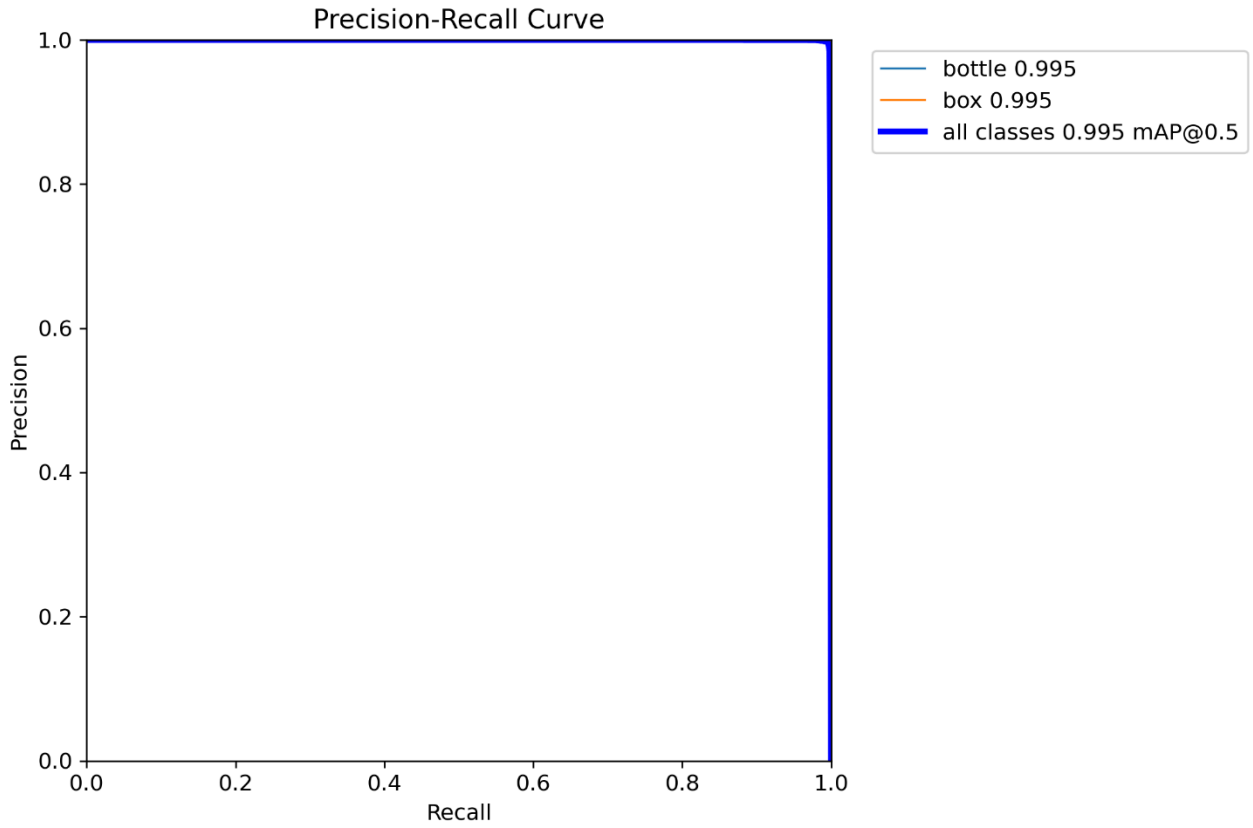
The Model training and evaluation results are as follows:

epoch	train/box_loss	train/cls_loss	train/dfl_loss	metrics/precision(B)	metrics/recall(B)	metrics/mAP50(B)	metrics/mAP50-95(B)
1	0.81854	1.4452	1.0274	0.91218	0.87064	0.95609	0.79504
2	0.76755	0.87959	1.0033	0.94763	0.94044	0.97477	0.80543
3	0.75104	0.74967	0.99782	0.96658	0.95514	0.98677	0.85085
4	0.72666	0.65255	0.99157	0.96137	0.96846	0.98453	0.85879
5	0.69244	0.59744	0.97521	0.9424	0.94498	0.9779	0.8709
6	0.58781	0.48974	0.92774	0.97189	0.97559	0.99149	0.87299
7	0.53434	0.43635	0.91112	0.97521	0.97879	0.99287	0.90877
8	0.508	0.39467	0.89814	0.98053	0.98792	0.99339	0.90562
9	0.49794	0.38212	0.89352	0.98825	0.98866	0.99449	0.92629
10	0.46004	0.34962	0.87856	0.99068	0.99215	0.99452	0.93501
11	0.44985	0.32875	0.87299	0.99265	0.99022	0.99446	0.9424
12	0.43022	0.30987	0.87134	0.99352	0.99464	0.99463	0.94928
13	0.4094	0.29403	0.86772	0.99631	0.99543	0.9948	0.9524
14	0.3935	0.27448	0.85787	0.99747	0.99649	0.99495	0.957
15	0.37048	0.25909	0.84781	0.99632	0.9972	0.99495	0.96346

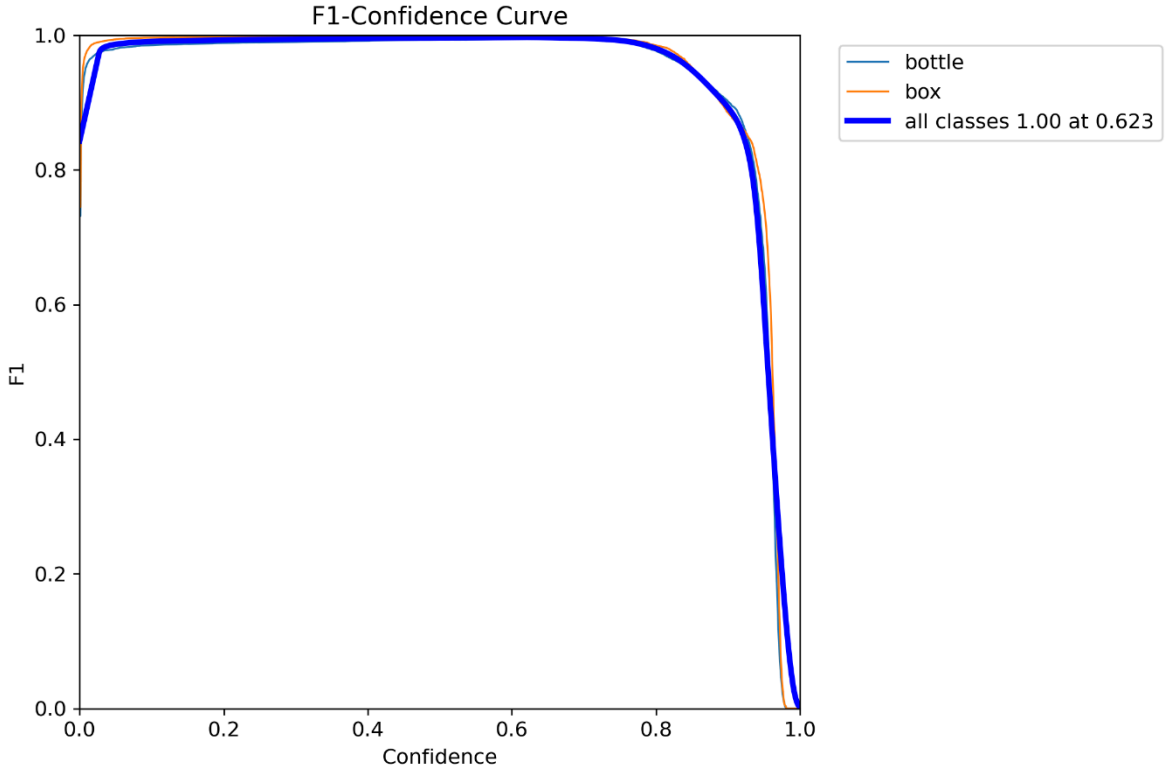
Results Table



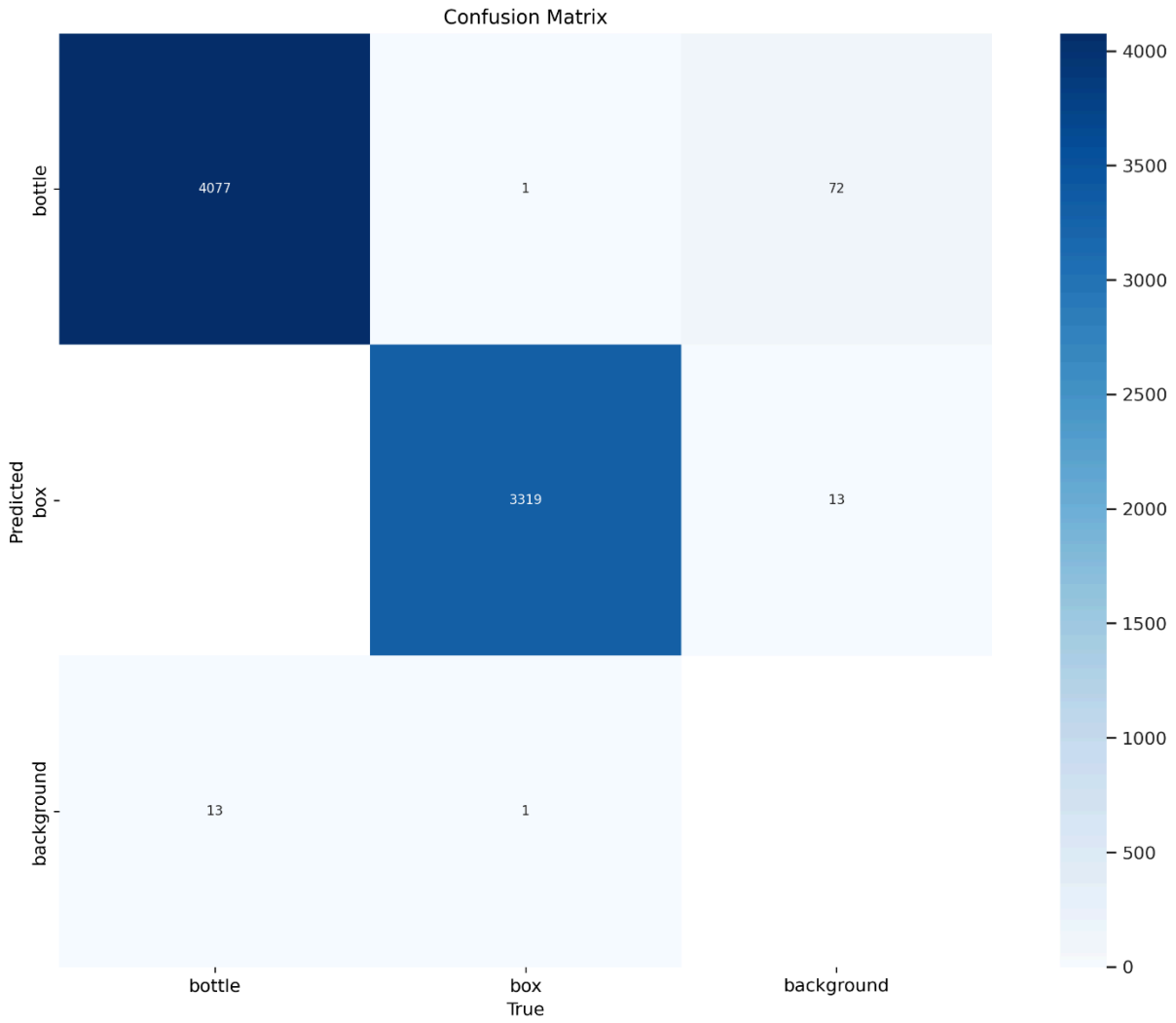
Results Graphs



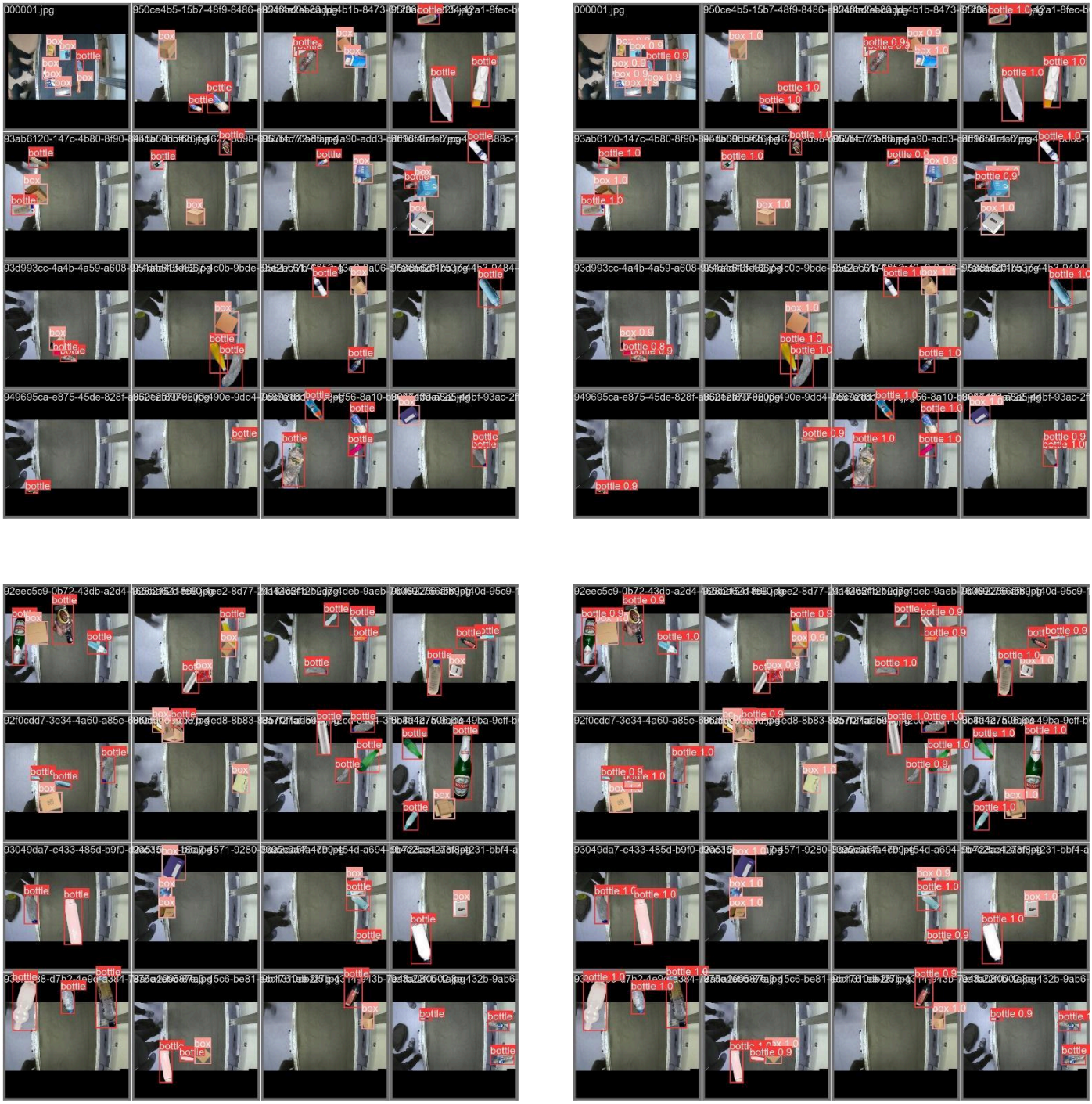
Precision-Recall Curve



F1 – Confidence curve



Confusion Matrix



Validation Labels vs Validation Predictions

5) Real Time Object Detection GUI

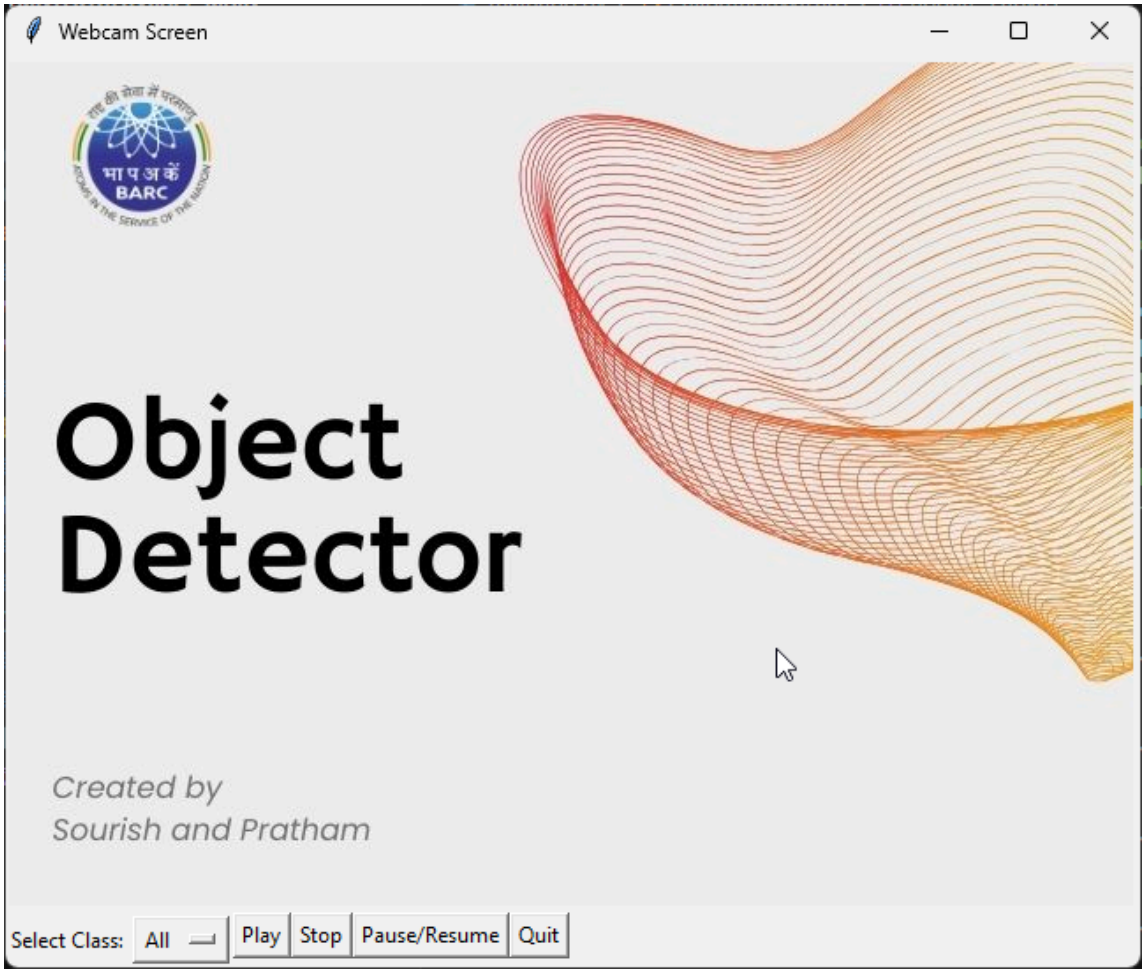
A Graphical User Interface (GUI) was developed using Tkinter to facilitate real-time object detection. The GUI seamlessly integrated the custom model, providing visualizations of bounding boxes and prediction confidence in real-time.

In order to run the real-time object detection GUI developed for our project, follow these steps:

Replace "*barc_model.pt*" with your custom-trained model. Additionally, ensure that the class names corresponding to the detected objects are correctly listed in the "*classes.txt*" file. This step is crucial for the accurate detection and labeling of objects by the GUI.

Execute the Python script "*app.py*" within your development environment.

Prior to running the GUI, ensure that the webcam connected to the system is not being utilized by any other application. This step is essential to prevent conflicts and ensure smooth operation of the real-time object detection functionality.



6) Camera Calibration

What is camera calibration?

The process of estimating the parameters of a camera is called camera calibration.

This means we have all the information (parameters or coefficients) about the camera required to determine an accurate relationship between a 3D point in the real world and its corresponding 2D projection (pixel) in the image captured by that calibrated camera.

Typically, this means recovering two kinds of parameters:

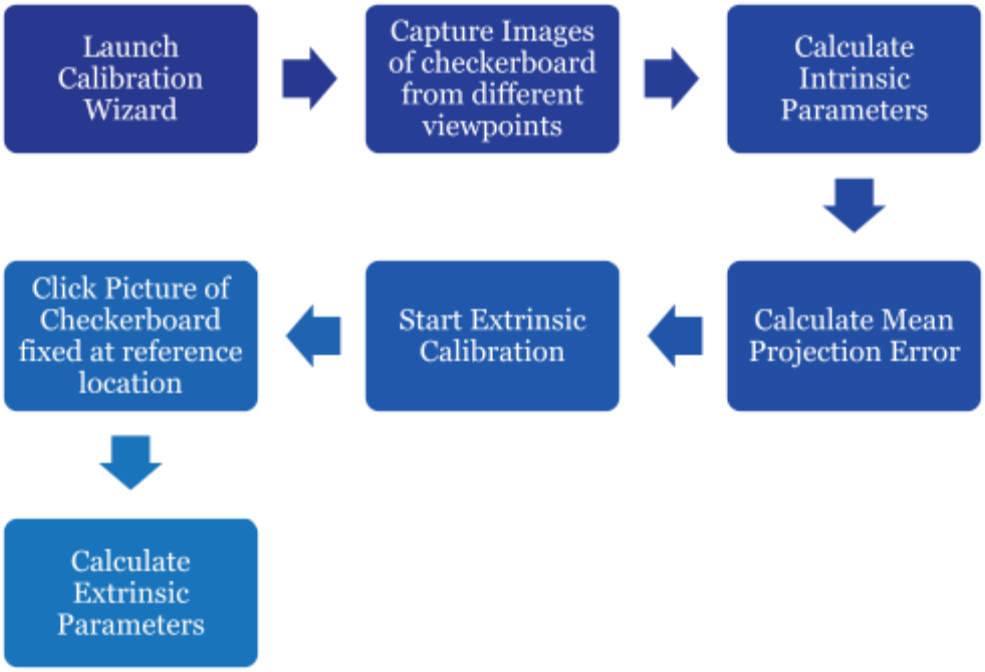
- Internal parameters of the camera/lens system. E.g. focal length, optical center, and radial distortion coefficients of the lens.
- External parameters: This refers to the orientation (rotation and translation) of the camera with respect to some world coordinate system.

The project expanded to include camera calibration to obtain real-world coordinates of detected objects.

A calibration script was developed, leveraging the principles of intrinsic and extrinsic camera parameters.

A user-friendly calibration wizard GUI was created to simplify the calibration process.

Integration of the calibration wizard into the real-time object detection app enabled the display of real-world coordinates alongside object detection results.



Intrinsic Calibration Steps

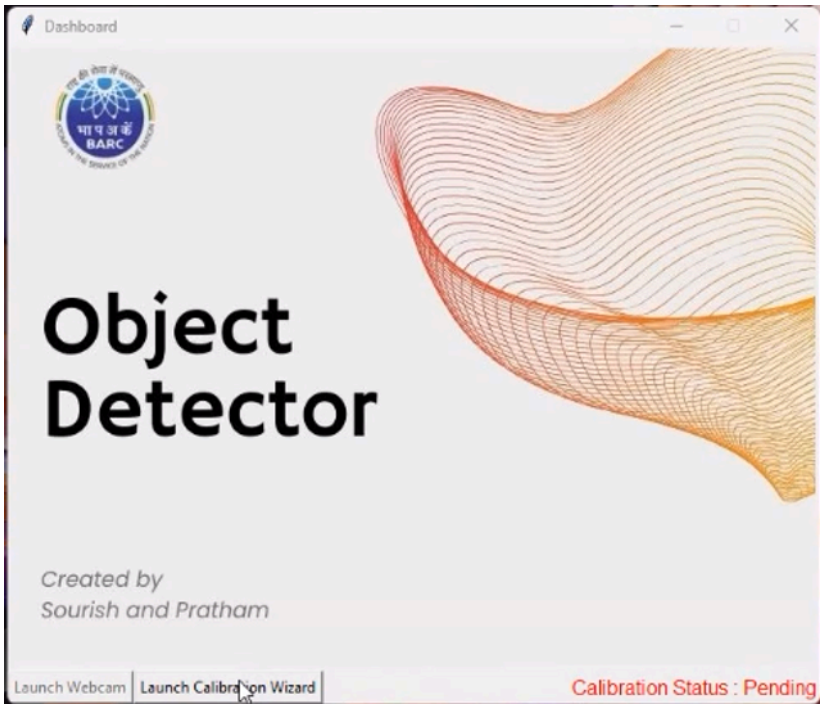
- A checkerboard is needed to calibrate the camera, preferably of rectangular shape.
- Print a checkerboard pattern with known number of squares and size of square.
- In this demonstration, we are using a checkerboard of size 600mm x 500mm with each square being 50mm.
- Capture multiple images of the printed checkerboard pattern at various orientations and angles to camera
- We made a calibration Wizard to allow easy navigation through the process
- Firstly, edit the *config.json* file (specify numbers of squares and size of squares)
- *python mainapp.py*
- Launch Calibration Wizard Button opens a new window with live webcam.
- This window detects the checkerboard pattern and overlays it on the footage in real time
- Additionally, there is a capture button to click a picture.
- Capture button automatically disables when no pattern is detected
- Capture 20 images while holding the checkerboard in different positions.
- After 20 images are captured, the program will calculate and store camera matrix and distortion matrix which are intrinsic parameters of a camera.
- Additionally, it will calculate error rate. If the error rate meets your desired use case threshold, then you can click accept and continue with extrinsic calibration or click decline and redo the intrinsic calibration

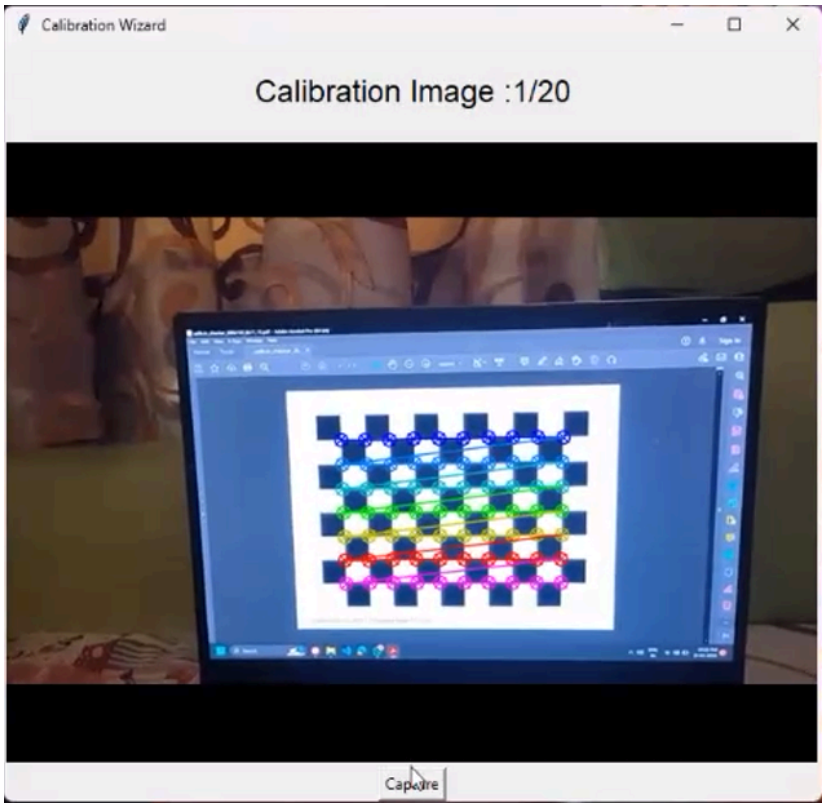
Extrinsic Calibration Steps

- After intrinsic calculation, one more image is to be taken for the extrinsic calibration. Fix the checkerboard pattern on ground plane. The first detection point where two squares meet will become the origin of real-world coordinate system.
- Click Capture. Rotational and Translation matrices will be calculated and stored.
- DO NOT SHIFT THE CAMERA AFTER EXTRINSIC CALIBRATION. IF IT DOES THEN YOU HAVE TO REDO EXTRINSIC CALIBRATION.

Get Real World Coordinates

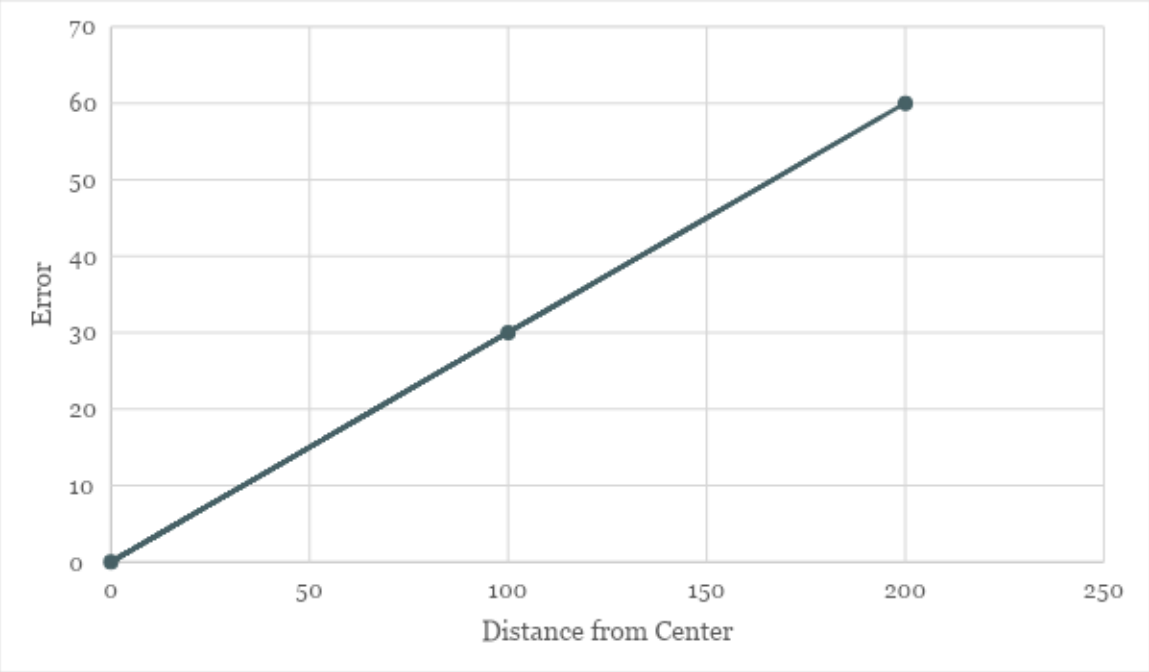
- You can now click on launch webcam button to open a new window with webcam and object detection using custom model.
- Here you the program will display real world coordinates of the center of bounding box of the detected object.




Calibration Wizard
Calibration Image :1/20
Capture

7) Error Analysis and Solutions

During calibration, errors in real-world coordinates were observed, particularly as objects moved away from the conveyor belt's center. A graphical representation illustrated the error pattern.



8) Conclusion

In summary, our project at the Department of Robotics and Remote Handling at BARC has made significant strides in the development of real-time object detection and camera calibration systems.

Through the utilization of YOLOv8 for object detection and the creation of a user-friendly GUI, we have successfully demonstrated the practical application of machine learning in industrial settings.

While we have achieved success in model training and GUI development, there remains room for improvement, particularly in refining real-world coordinate accuracy through camera calibration.

Future efforts will be dedicated to resolving existing errors and enhancing calibration techniques to ensure greater precision in object localization.

Overall, our project represents a promising advancement in industrial automation, with the potential to streamline processes and improve operational efficiency.

By addressing challenges and leveraging emerging technologies, we are poised to make further contributions to the field of robotics and remote handling, ultimately enhancing safety and productivity in industrial environments.