HCMC University of Technology and Education

Faculty of Mechanical Engineering

Department of Mechatronics

Final term project

Pothole detection to support vehicle shock absorption system

Lecturer: Ph.d Nguyen Van Thai

Members

Order	Name	ID
1	Nguyen Trinh Tra Giang	22134002
2	Pham Thanh Tri	22134015
3	Nguyen Quoc Trung	22134016

ThuDuc, 29th June 2024

HCMC University of Technology and Education

Faculty of Mechanical Engineering

Department of Mechatronics

HCMUTE

Final term project

Pothole detection to support vehicle shock absorption system

Lecturer: Ph.d Nguyen Van Thai

Members

Order	Name	ID
1	Nguyen Trinh Tra Giang	22134002
2	Pham Thanh Tri	22134015
3	Nguyen Quoc Trung	22134016

ThuDuc, 29th June 2024

Table of Contents

1. Abstract

This project focuses on detecting and classifying potholes on roads using the YOLO (You Only Look Once) algorithm, a real-time object detection system using convolutional neural networks (CNNs). Potholes are a major cause of traffic accidents, especially during rainy seasons, and early detection and repair are crucial for accident prevention. Currently, pothole detection methods heavily rely on manual image processing, which is labor-intensive and costly.

This project proposes a non-invasive approach using YOLOv8 to detect and classify potholes in images, aiming to provide real-time detection with highlighted visual cues. The YOLO algorithm employs CNNs to simultaneously predict object classes and bounding boxes, enhancing responsiveness and accuracy.

A dataset comprising diverse pothole scenarios in natural road conditions was collected for training, testing, and validating the model. This new method not only reduces costs but also enhances accuracy and effectiveness in pothole detection, crucially supporting road maintenance and improving traffic safety.

The project promises significant benefits in road monitoring and maintenance, particularly in the context of autonomous vehicles, by enabling real-time pothole detection and precise visual cues.

2. Introduction

2.1 Project selection reason

We chose this project to alert vehicles about potholes, aiming to prevent accidents and avoid getting stuck in potholes. This system detects potholes to provide information to authorities for road improvement.

2.2 Project goals

- Detect potholes on roads
- Collec pothole information such as length, width and distance from vehicles

2.3 Challenges

For our pothole detection project, we face several challenges, primarily related to data. If road segments are too dark, our pothole detection model may fail. Similarly, shadows or road markings can interfere with detection. Additionally, low-resolution video affects certain parameters during training. Determining pothole characteristics after detection has also been challenging, requiring extensive research and experimentation.

3. Methodology

3.1 Project solution

To address these challenges, enriching our dataset was crucial. After extensive search, we found ROBOFLOW, a reliable source for data collection. We enhanced our dataset on ROBOFLOW by adjusting training parameters to prevent false detections due to shadows. This process was detailed in our presentation video.

3.2 Study approach

Initially, understanding our project's scope was essential. We researched online platforms to determine project requirements. Our model's focus was on developing AI for pothole detection and enhancing it for effective detection and classification of potholes and general road anomalies.

- Data Collection: Training data was gathered through windshield-mounted cameras capturing videos or images of roads and from online sources.
- Model Selection: Various object detection algorithms were compared, including SSD-TensorFlow, YOLOv3-Darknet53, and YOLOv4-Darknet53. YOLOv4 demonstrated superior performance due to high accuracy and rapid processing.
- Training Process: Deep learning models were trained on labeled datasets containing pothole images, adjusting model weights to ensure accurate pothole detection and classification.
- Evaluation: Models were evaluated based on metrics such as precision, recall, and mean Average Precision (mAP). Results from different models were compared to select the optimal one.
- Real-world Deployment: The best-performing model was deployed in real-world environments to detect potholes in real-time, enhancing traffic safety and improving autonomous vehicle systems by providing early warnings about road potholes.

These steps constituted a comprehensive process from data collection to real-world deployment, aiming to efficiently detect and report potholes, thereby contributing to improved road quality and traffic safety.

3.3 YOLOv8 for pothole detection

3.3.1 Model selection

Overview of YOLOv8: YOLOv8 is a computer vision algorithm used for object detection, building on previous YOLO versions known for real-time object detection using a neural network.

Advantages of YOLOv8: YOLOv8 offers high speed and performance, capable of real-time object detection while maintaining high accuracy. Its improved neural network architecture enhances learning and detection of complex features in images.

This version integrates advanced deep learning techniques such as Mish activation function and CSP (Cross Stage Partial connections) to enhance performance and reduce computational costs, ensuring high accuracy in detecting and classifying objects.

3.3.2 Training data preparation

Training data including:

- Images containing potholes
- Background images
- Images without potholes and specific points of interest

3.3.3 Training process

Code was written to integrate YOLOv8 for training AI models, supporting the training process effectively

- Import the library and create the “parse_list” function convert a string argument into a list of integers, raising an error if the conversion fails.

```
from ultralytics import YOLO
import argparse
import sys
import ast

def parse_list(arg):
    try:
        return ast.literal_eval(arg)
    except ValueError:
        raise argparse.ArgumentTypeError("Argument must be a list of
integers.")

parser = argparse.ArgumentParser()
parser.add_argument('-directory', type=str, help='Working
Directory')
parser.add_argument('-project', type=str, help='Project Name')
parser.add_argument('-batch', type=int, help="Number of simultaneous
training images")
parser.add_argument('-epoch', type=int, help="Number of epochs")
parser.add_argument('-name', type=str, help="Name of checkpoint")
parser.add_argument('-freeze', type=parse_list, help="Freezing
range")
args = parser.parse_args()
```

– Identify and freeze certain layers in the model to prevent them from being retrained.

```
global num_freeze
num_freeze = args.freeze

def freeze_layer(trainer):
    model = trainer.model
    print(f"Freezing {num_freeze[1] - num_freeze[0]} layers")
    freeze = [f'model.{x}.' for x in range(*num_freeze)]
    for k, v in model.named_parameters():
        v.requires_grad = True
        if any(x in k for x in freeze):
            print(f'freezing {k}')
            v.requires_grad = False
    print(f"{num_freeze[1] - num_freeze[0]} layers are freezed.")
```

– Initialize a YOLO model for segmentation task and add a callback to freeze model layers when training starts.

```
model = YOLO(model = args.directory, task = 'segment')
model.add_callback("on_train_start", freeze_layer)
```

– The "train" code within the "try"lock is designed such that when a common out-of-memory error occurs, it outputs 1; conversely, upon successful completion of the training process, it outputs 0.

```
try:
   .....
except RuntimeError as e:
    print(f"An error occurred: {e}", file=sys.stderr)
    sys.exit(1)
sys.exit(0)
```

– In the train function, adjust parameters to control the training process and its performance.

```
model.train()
```

– Configure data tasks such as dataset retrieval, segmentation, number of epochs, and input image size.

```
data = r"Resources/PotholeDataset/data.yaml",
task = 'segment',
epochs = args.epoch,
imgsz = 640,
```

– Configure training parameters to adjust learning rate and prevent overfitting.

```
batch = args.batch,
patience = 20,
verbose = True,
dropout = 0.35,
optimizer = 'auto',
cos_lr = True,
label_smoothing = 0.15,
```

– Adjusting the weights of the loss function and determining bounding boxes.

```
cls = 0.75,
box = 12.5,
overlap_mask = True,
nms = True,
iou = .65,
conf = .12,
```

– The hardware for running and storing data of the model.

```
device = [0],
workers = 12,
cache = 'ram',
```

– Drawing and displaying training metrics.

```
plots = True,
show = True,
visualize = True,
save = True,
save_txt = True,
save_period = -1,
show_boxes = True,
```

– Project name and model preserving initial training parameters, overwritten after each training iteration.

```
project = f"{args.project}",
name = args.name,
seed = 0,
exist_ok = True,
```

3.3.4 Automating the training process.

- This process begins with initializing the model name, setting the batch size (number of data samples trained simultaneously), and defining the number of epochs (training iterations). It allows for multiple training adjustments, including freezing specific layers. If a memory error occurs during training, it automatically resolves by reducing the batch size to ensure continuous training.

```
usage() {
    echo "Usage: $0 <directory_path> <batch_size> <epochs> -m
<model_name> [-f <freeze_layers>]"
    echo "   -m <model_name>                YOLO model name"
    echo "   -f <freeze_layers>             Layers to freeze,
e.g., \"[firstLayerIndex, lastLayerIndex]\""
    exit 1
}
def_freeze="[0, 0]"
model=""
freeze=""
POSITIONAL_ARGS=()
while [[ $# -gt 0 ]]; do
    case $1 in
        -f|--freeze )
                freeze="$2"
                shift
            shift
            ;;
        -m|--model )
                model="$2"
            shift
            shift
                ;;
    -*|--*)
        echo "Unknown option $1"
        usage
        exit 1
        ;;
        : )
                echo "Invalid option: -$OPTARG requires an
argument" 1>&2
                usage
                ;;
    * )
        POSITIONAL_ARGS+=("$1")
        shift
        ;;
    esac
done
set -- "${POSITIONAL_ARGS[@]}"
if [ $# -ne 3 ]; then
    usage
fi
if [ -z "$freeze" ]; then
        echo "No freezing layer"
        freeze=$def_freeze
else
        if [[ ! $freeze =~ ^\[\ *[0-9]+\ *(,\ *[0-9]+\ *)*\]$ ]];
then
            echo "Error: Input must be in python format [int, int,
...]"
            exit 1
        else
            echo "Freezing from $freeze"
    fi
fi
transferLearn=false
if [ -z "$model" ]; then
    echo "Transfer learning enabled"
    transferLearn=true
else
    echo "Using model: $model"
fi
echo ""
directory_path=$1
batchsize=$2
epochs_str=$3

if [[ ! $epochs_str =~ ^\(\ *[0-9]+\ *(,\ *[0-9]+\ *)*\)$ ]]; then
    echo "Error: Input must be in the format (int, int, ...)"
    exit 1
fi
epochs_str=${epochs_str//[\(\)]/}
epochs_str=${epochs_str// /}
IFS=',' read -r -a epochs <<< "$epochs_str"
default_name="Cook_Station_"
default_name2="/weights/best.pt"
head_length=$((${#directory_path}+${#default_name}))
tail_length=${#default_name2}
#tensorboard --logdir $directory_path/ &
```

```
idx=1
for epoch in "${epochs[@]}"
do
    echo "Running iteration $idx..."
    echo ""
    matching_directories=($(find $directory_path -type f -path
'*/Cook_Station_*/weights/best.pt' | sort -rV))

    if [ -z "$matching_directories" ] && [ "$transferLearn" ==
true ]; then
        echo "No model specified. Exiting..."
        break
    fi
    choosen_dir=${matching_directories[0]}
    tail_pos=$((${#choosen_dir}-tail_length))
    iterationID=${choosen_dir:$head_length:$(($tail_pos-
$head_length))}
    name="$default_name$(($iterationID+1))"
    if [ -z "$matching_directories" ]; then
        choosen_dir=$model
    fi
    ./PotholeV8.py -directory "$choosen_dir" -project
"$directory_path" -batch "$batchsize" -epoch "$epoch" -name
"$name" -freeze "$freeze"
    exit_status=$?
    if [ $exit_status -eq 1 ]; then
        batchsize=$((batchsize - 1))
        echo "BATCH SIZE REDUCED BY ONE: $batchsize"
    else
        ./export.py -directory
"$directory_path$name$default_name2"
    fi
        echo "Iteration $idx complete."
    idx=$((idx+1));
done
```

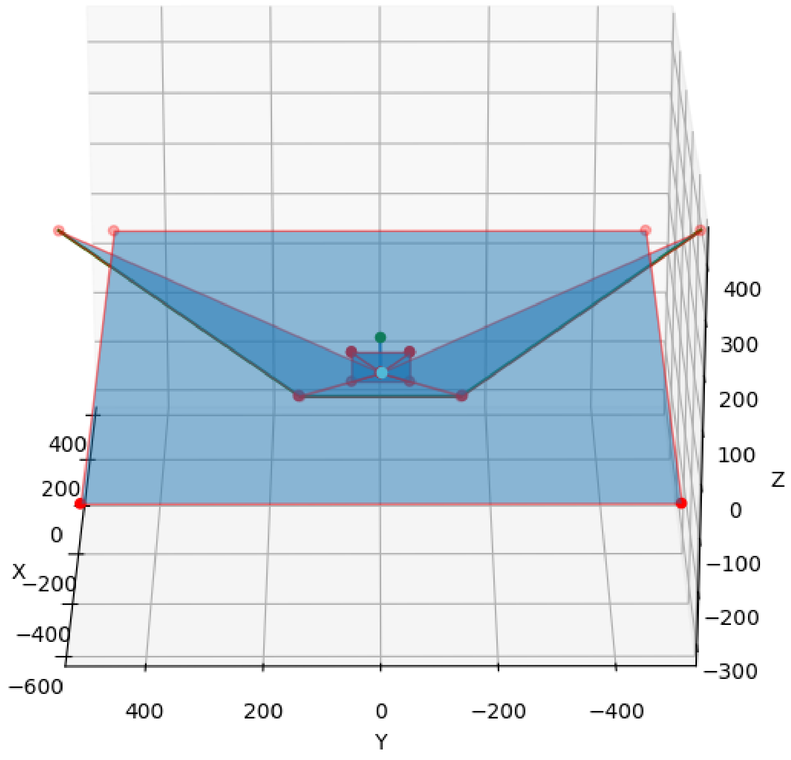
3.4 Classical Machine vision technique for Pothole's dimension measurement.

3.4.1 Method

- After finding bounding box coordinates and mask for each pothole, we can determine pothole's dimension using mapping 2D to 3D technique, the idea is that if we have image dimensions, camera specifications and location in space, we can determine height and width of pothole in SI unit.

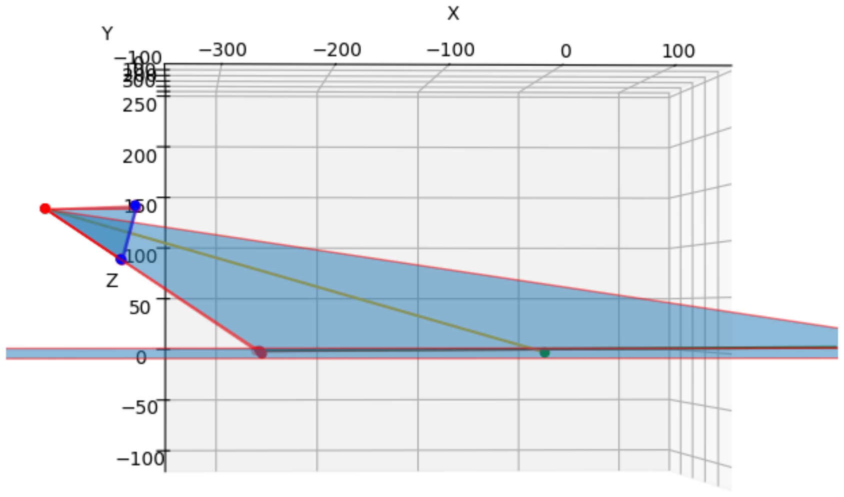
3.4.2 Visualization

- Using *matplolib* library, we can make a simple camera simulation in realword:



Picture 3.4.2.1: Camera simulation using matplotlib library

- In *Picture 3.4.2.1*, the middle aqua dot is the center of projection, the small rectangular in the middle is the image in pixel scale and the largest plane is the road (assuming road is z=0 plane).



Picture 3.4.2.2: Camera simulation from right POV

- In *Picture 3.4.2.2,* the yellow line is principal length and intersection between image and principal length is principal point. Length from center of projection and principal point is called folcal length

3.4.3 Intrinsic matric

- Each camera has different internal parameters, those parameters represented by Intrinsic matrix.

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

- Where f_x, f_y is focal length in x and y direction respectively and c_x, c_y is coordinate of principal point.
- Here we will use *Camera Calibration toolkit* from matlab and import it using *scipy* library. After importing, we will have Intrinsic matrix.

```
mat = matlab.loadmat(r"Resources/CameraIntrinsic.mat")
intrinsicMatrix = mat["IntrinsicMatrix"].T
```

- Notice that, in matlab, intrinsic matrix will be transposed so after import into python, we must transpose it.

3.4.4 Geometry calculations

- First we will extract frame dimensions and camera properties

```
# Extract frame dimensions
heightPx, widthPx, _ = frame.shape
# Extract focal lengths and principal point coordinates from the
intrinsic matrix
fx, fy, cx, cy = intrinsicMatrix[0, 0], intrinsicMatrix[1, 1],
intrinsicMatrix[0, 2], intrinsicMatrix[1, 2]
```

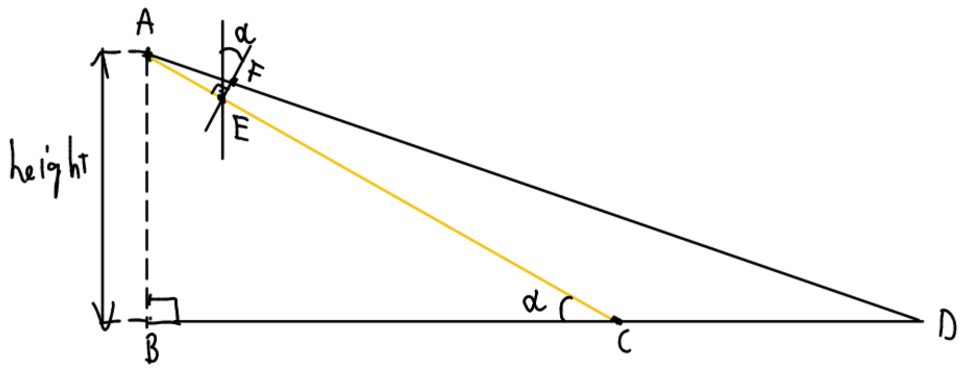
- As mention before, we used *Camera Calibration toolkit* from matlab for getting camera intrinsic matrix, in some cases, dimension of frame and image used for camera calibration may be different, so we will calculate ratio between dimension of frame used in matlab and input.

```
# Rescaling fx, fy
fx /= (cx * 2 / widthPx)
fy /= (cy * 2 / heightPx)
```

- Now we will get vector u in width and vector v in height

```
u, v = np.arange(- widthPx // 2, widthPx // 2, 1, dtype = float),
np.arange(- heightPx // 2, heightPx // 2, 1, dtype = float)
```

- Imagine we take a slice along vector v from *Picture 3.4.2.2*, we will have a geometry demo look like this:



Picture 3.4.4.1: Geometry presentation of 1 slice

  - With point A is the center of projection, point E is principal point, α is tilt angle.

- So with EF is vector v, AE is f_y (focal length in y direction)

$$\widehat{EAD} = \tan^{-1}\left(\frac{FE}{AE}\right)$$

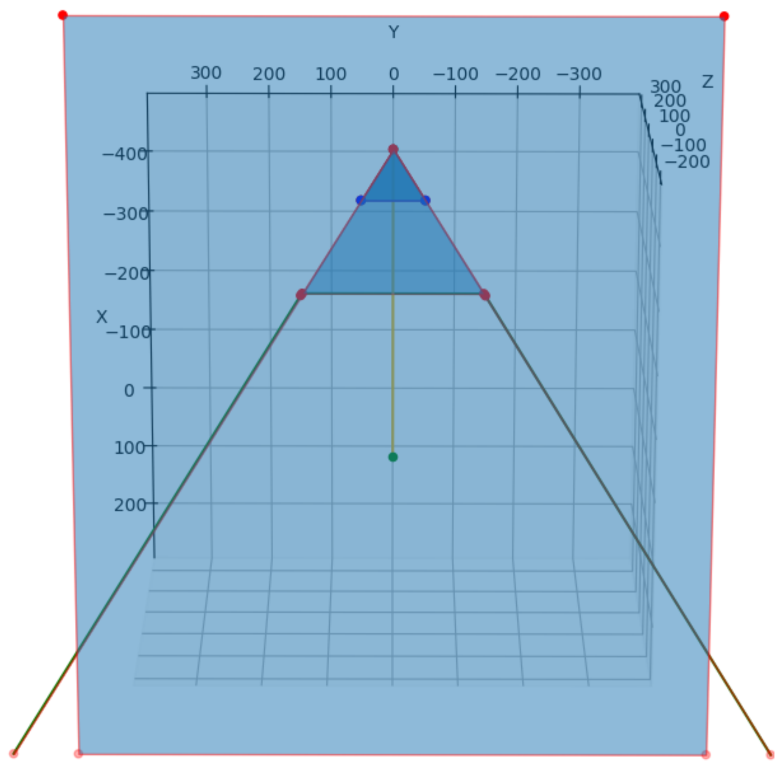
$$\rightarrow \widehat{ADC} = 180° - \left[180° - \alpha + \widehat{EAD}\right] = \alpha - \widehat{EAD}$$

$$\Rightarrow AD = \frac{AB}{\sin\left[\widehat{ADC}\right]}$$

- From those calculation we will apply it in python:

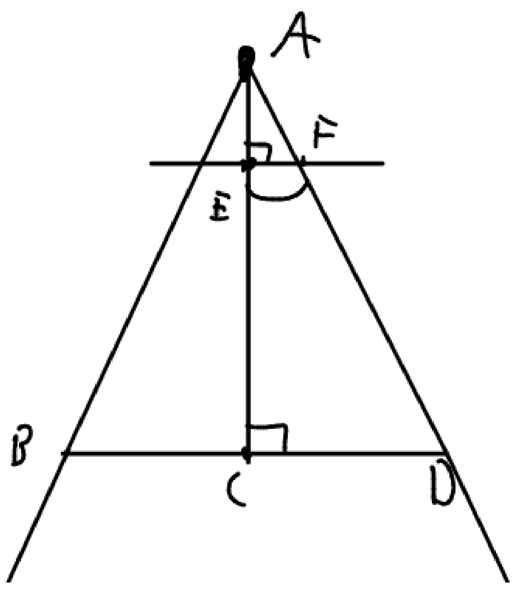
```
# Generate arrays of pixel indices for width and height
u, v = np.arange(- widthPx // 2, widthPx // 2, 1, dtype = float),
np.arange(- heightPx // 2, heightPx // 2, 1, dtype = float)
# Calculate relative tilt angles for each pixel
relativeYTilt = np.arctan2(v, fy)
# Calculate absolute tilt angles by adjusting with the camera tilt
absYTilt = tilt - relativeYTilt
# Compute the depth vector for the center line of the image
centerDepthVect = height / np.sin(absYTilt)
```

- Now we will change perspective of the simulation model:



Picture 3.4.4.2: Camera simulation from bottom perspective

- Get a slice along vector u from *Picture 3.4.4.2*, we can have a geometry model:



Picture 3.4.4.3: slice from Picture 3d.2

- From *Picture 3.4.4.3*
  - First, we will split the triangle into 2 part: ΔACD and ΔABC. From ΔACD, we have:
    - AE is focal length along x axis, EF is haft right of vector u

$$\widehat{CAD} = \tan^{-1}\left(\frac{u_{right}/2}{f_x}\right)$$

$$\rightarrow AD = \frac{AC}{\cos[\widehat{CAD}]}$$

- Apply the same logic into ΔABC, we have

$$\widehat{CAB} = \tan^{-1}\left(\frac{u_{left}/2}{f_x}\right)$$

$$\rightarrow \text{AB} = \frac{AC}{\cos[\widehat{CAB}]}$$

- Then we will apply into python:

```
# Split the horizontal pixel indices at the center to process
left and right halves separately
halfRightU = u[np.where(u == 0)[0][0] + 1:
halfLeftU = np.abs(u[: np.where(u == 0)[0][0]])
# Calculate tilt angles for the right and left halves
halfRightTilt = np.arctan2(halfRightU, fx)
halfLeftTilt = np.arctan2(halfLeftU, fx)
# Calculate depth values for the right and left halves using
cosine adjustments
halfRightDepth = centerDepthVect[::-1, np.newaxis] /
np.cos(halfRightTilt)
halfLeftDepth = centerDepthVect[::-1, np.newaxis] /
np.cos(halfLeftTilt)
```

- The reason we split it into 2 triangle because in images, width usually an even number, distance between left and right isn't equal.

- Then we will combine all 3 depth matrices into 1:

```
 # Combine the depth values from left, center, and right to form the
complete depth map
depth = np.hstack([halfLeftDepth, centerDepthVect[::-1, np.newaxis],
halfRightDepth])
```

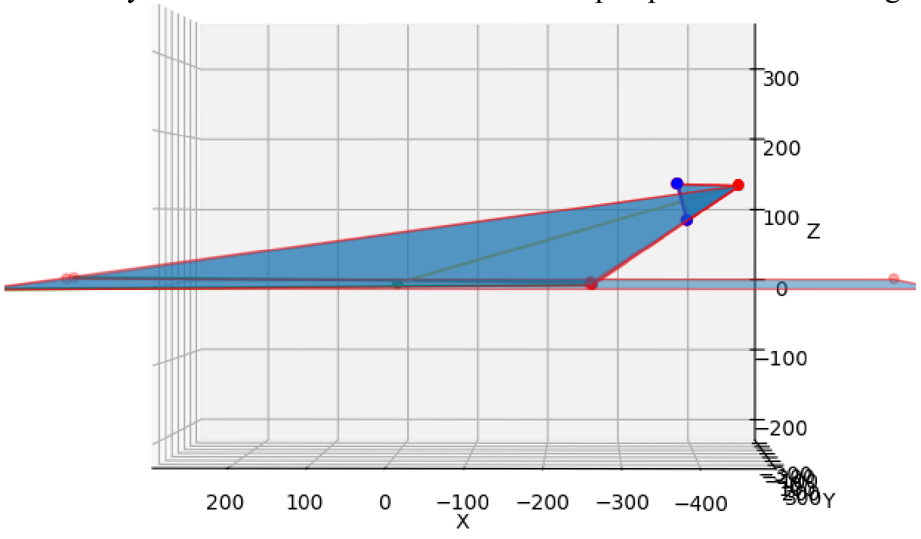
- From *Picture 3.4.4.3,* we can calculate mapping 2D to 3D values in x direction*:*
  - Apply pythagoras theorem into ΔACD, we can have:

$$CD = \sqrt{AC^2 + AD^2}$$

  - Apply into python:

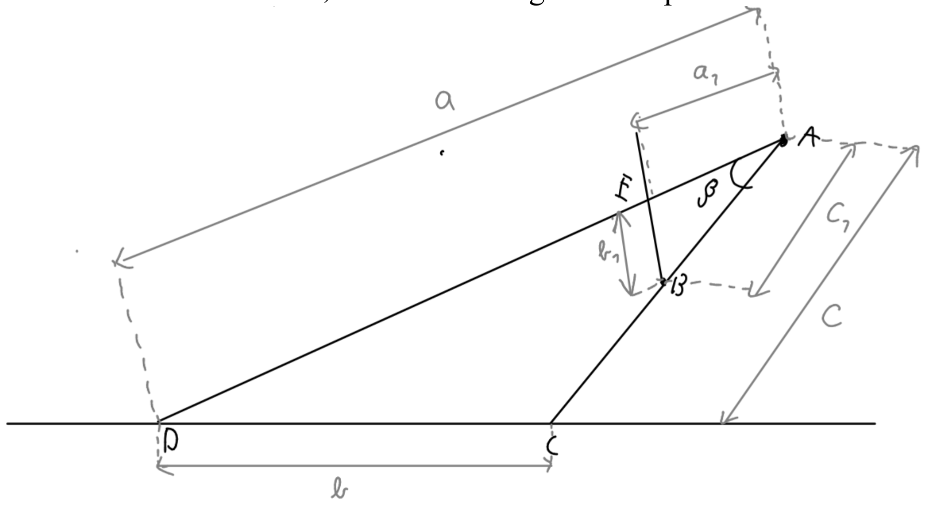
```
# Calculate the horizontal (X) mapping values from depth
Xmapper = np.sqrt(depth ** 2 - depth[:, centerX, np.newaxis] ** 2)
Xmapper[:, :centerX] = - Xmapper[:, :centerX]
# Handle invalid depth values
Xmapper = np.where(depth < 0, np.inf, Xmapper)
```

- Because of the tilt angle of camera, calculating mapping 2D to 3D values in y direction require additional steps:
    - Firstly we have a camera simulation with perspective from the right:



Picture 3.4.4.4: Simulation model with perspective from the right

    - From *Picture 3.4.4.4,* we can have a geometric presentation of a slice:



Picture 3.4.4.5: A geometric prensentation of a slice

    - From which we can also have:
        - Apply cosine theorem into both triangle ΔAEB and ΔACD:

$$\begin{cases} a_1^2 + c_1^2 - 2a_1c_1\cos(DAC) = b_1^2 \\ a^2 + c^2 - 2ac\cos(DAC) = b^2 \end{cases}$$

$$\Leftrightarrow \begin{cases} \cos(DAC) = \dfrac{a_1^2 + c_1^2 - b_1^2}{2a_1c_1} \\ \cos(DAC) = \dfrac{a^2 + c^2 - b^2}{2ac} \end{cases}$$

$$\Rightarrow b = \sqrt{\frac{c_1^2 + a_1^2 - b_1^2}{2a_1c_1}2ac - a^2 - c^2}$$

- a_1 is length from center of projection to any point of the image
- b_1 is length from bottom left of image to any point of the image
- c_1 is length from center of projection to any point along the bottom edge of image
- a is depth after combining 3 depth matrices as calculation above
- b is mapping 2D to 3D values in y axis
- c is depth like a but along bottom edge of image

  - Apply above calculation into python we have:

```
 # Create meshgrid for pixel indices
 u, v = np.meshgrid(u, v[::-1])
 # Calculate distances from image center to all pixels
 imageCenterToALLPixels = np.sqrt(u ** 2 + v ** 2)
 # Calculate distances from projection center to all pixels
 projectionCenterToImagePixels = np.sqrt(np.mean([fx, fy]) ** 2
+ imageCenterToALLPixels ** 2)
 # Calculate distances and differences for vertical (Y) mapping
 c1 = projectionCenterToImagePixels[centerY].astype(float)
 a1 = projectionCenterToImagePixels.astype(float)
 b1 = (v - v[centerY]).astype(float)
 c = depth[centerY]
 a = depth
 # Calculate the vertical (Y) mapping values from depth
 Ymapper = np.sqrt(-(((c1 ** 2 + a1 ** 2 - b1 ** 2) / (2 * a1 *
c1)) * 2 * a * c - a ** 2 - c ** 2))
 Ymapper = np.where(depth < 0, np.inf, Ymapper)
```

- After proving and calculating, we created a function that take in frame that needed to map 2D to 3D in SI unit, tilt angle of camera with respect to ground, height from camera to road and intrinsic camera and output 2 matrices in SI unit along x and y axis respectively.

```
def SIApproxMapping(frame: np.ndarray, tilt: float, height: float, intrinsicMatrix:
np.ndarray) -> np.ndarray:
   # Extract frame dimensions
   heightPx, widthPx, _ = frame.shape
   # Extract focal lengths and principal point coordinates from the intrinsic matrix
   fx, fy, cx, cy = intrinsicMatrix[0, 0], intrinsicMatrix[1, 1], intrinsicMatrix[0, 2],
intrinsicMatrix[1, 2]
   # Rescaling fx, fy
   fx /= (cx * 2 / widthPx)
   fy /= (cy * 2 / heightPx)
   # Generate arrays of pixel indices for width and height
```

```
    u, v = np.arange(- widthPx // 2, widthPx // 2, 1, dtype = float), np.arange(-
heightPx // 2, heightPx // 2, 1, dtype = float)
    # Calculate relative tilt angles for each pixel
    relativeYTilt = np.arctan2(v, fy)
    # Calculate absolute tilt angles by adjusting with the camera tilt
    absYTilt = tilt - relativeYTilt
    # Compute the depth vector for the center line of the image
    centerDepthVect = height / np.sin(absYTilt)
    # Split the horizontal pixel indices at the center to process left and right halves
separately
    halfRightU = u[np.where(u == 0)[0][0] + 1: ]
    halfLeftU = np.abs(u[: np.where(u == 0)[0][0]])
    # Calculate tilt angles for the right and left halves
    halfRightTilt = np.arctan2(halfRightU, fx)
    halfLeftTilt = np.arctan2(halfLeftU, fx)
    # Calculate depth values for the right and left halves using cosine adjustments
    halfRightDepth = centerDepthVect[::-1, np.newaxis] / np.cos(halfRightTilt)
    halfLeftDepth = centerDepthVect[::-1, np.newaxis] / np.cos(halfLeftTilt)
    # Combine the depth values from left, center, and right to form the complete
depth map
    depth = np.hstack([halfLeftDepth, centerDepthVect[::-1, np.newaxis],
halfRightDepth])
    # Define the center coordinates of the image
    centerX, centerY = widthPx // 2, heightPx - 1
    # Calculate the horizontal (X) mapping values from depth
    Xmapper = np.sqrt(depth ** 2 - depth[:, centerX, np.newaxis] ** 2)
    Xmapper[:, :centerX] = - Xmapper[:, :centerX]
    # Handle invalid depth values
    Xmapper = np.where(depth < 0, np.inf, Xmapper)
    # Create meshgrid for pixel indices
    u, v = np.meshgrid(u, v[::-1])
    # Calculate distances from image center to all pixels
    imageCenterToALLPixels = np.sqrt(u ** 2 + v ** 2)
    # Calculate distances from projection center to all pixels
    projectionCenterToImagePixels = np.sqrt(np.mean([fx, fy]) ** 2 +
imageCenterToALLPixels ** 2)
    # Calculate distances and differences for vertical (Y) mapping
    c1 = projectionCenterToImagePixels[centerY].astype(float)
    a1 = projectionCenterToImagePixels.astype(float)
    b1 = (v - v[centerY]).astype(float)
    c = depth[centerY]
    a = depth
    # Calculate the vertical (Y) mapping values from depth
    Ymapper = np.sqrt(-(((c1 ** 2 + a1 ** 2 - b1 ** 2) / (2 * a1 * c1)) * 2 * a * c - a
** 2 - c ** 2))
    Ymapper = np.where(depth < 0, np.inf, Ymapper)
    return Xmapper, Ymapper
```

- Next we will make function to get centroid of a mask using *cv2.moment* function
    - Centroid of a mask (or of a shape) is the arithmetic mean of all the points in a shape. Suppose a shape consists of n distinct points $x_1, \dots, x_n$, then the centroid is given by

$$C = \frac{1}{n}\sum_{i=1}^{n} x_i$$

    - cv2.moment is a function that take in a array of x, y coordinates and return $M_{00}, M_{01}, M_{10}, \dots$ with M_{01} is sum of y coordinates, M_{10} is sum of x coordinates, M_{00} is area of those coordinates (meaning total pixel make up that array)
    - So we can assume:

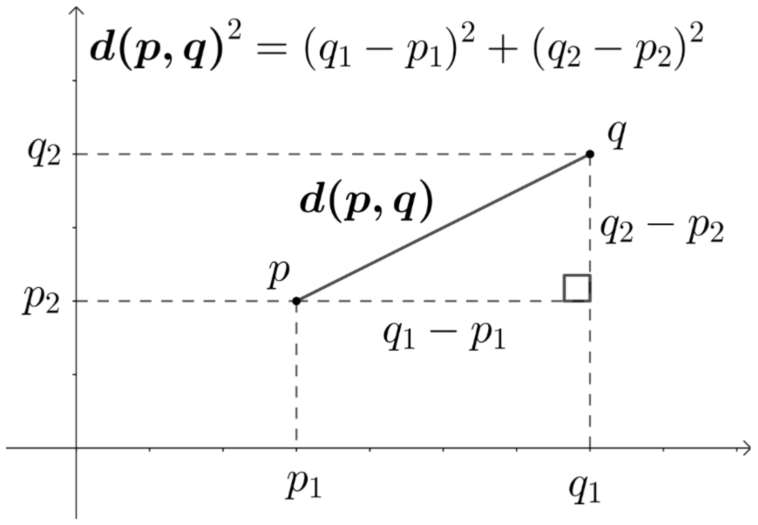
$$n = M_{00}, \sum_{i=1}^{n} x_i = M_{10}, \sum_{i=1}^{n} y_i = M_{01}$$

$$\Rightarrow C_x = \frac{M_{10}}{M_{00}}, C_y = \frac{M_{01}}{M_{00}}$$

    - Apply it into code:

```
def centerOfMass(mask: np.ndarray) -> np.ndarray:
    moments = cv2.moments(mask)
    return np.array([moments['m10'] // moments['m00'], moments['m01'] //
moments['m00']]).astype(int)
```

- And lastly we will make a function to get index of closest pothole
    - Using euclidean distance method, we can get distance value from 1 point to another point.



Picture 3.4.4.6: Euclidean distance in coordinate system

- So we can create a function that take orginal point – the point where we want to measure from and array of x, y coordinates, with output is index of the nearest coordinate

```
def nearestPoint(origin: np.ndarray, mask: np.ndarray) -> int:
    distances = np.sqrt((mask[:, 0] - origin[0]) ** 2 + (mask[:, 
1] - origin[1]) ** 2)
    minDist = distances.min()
    return np.where(distances == minDist)[0][0]
```

3.5 Video setup

3.5.1 Bounding box design

- We will create a bounding box displaying pothole dimensions and confidence score. We will use *cv2.rectangle* to create a rectangle and *cv2.putText* to display text inside the bounding box.
  - This function will take bounding box locations (x, y coordinates) after model detected, pothole's dimension after calculating using def SIApproxMapping function and frame needed to be drawn on.

```
def drawBox(self, box: Boxes, dim: tuple[float, float] = (None, 
None), frame: np.ndarray = None) -> None:
        frame = frame if frame is not None else self.frame
        # Extract the coordinates of the bounding box and 
convert them to integers
        x1, y1, x2, y2 = np.int32(box.xyxy[0].cpu())
        # Extract the confidence score and round it to 2 
decimal places
        confidence = round(float(box.conf.cpu()[0]), 2)
        # Draw the bounding box on the frame
        cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 0, 255), 
2)
        # Draw a filled rectangle on the overlay for the text 
background
        cv2.rectangle(frame, (x1, y1 - 35), (x1 + 290, y1), (0, 
0, 255), cv2.FILLED)
        # Put the confidence text on the overlay
        cv2.putText(frame, f"Pothole {confidence}", (x1 + 3, y1 
- 20), cv2.FONT_HERSHEY_DUPLEX, .5, (255, 255, 255), 1, 
cv2.LINE_AA)
        # Put the dimensions text on the overlay
        cv2.putText(frame, f"Height: {round(dim[0])} (cm), 
Width: {round(dim[1])} (cm)", (x1 + 3, y1 - 5), 
cv2.FONT_HERSHEY_DUPLEX, .5, (255, 255, 255), 1, cv2.LINE_AA)
```

3.5.2 Model’s mask transparency

- We will make model’s mask transparent and outer edge easier to recognize using *cv2.polylines* and *cv2.addWeighted:*
    - First we will draw a polygon using model’s mask coordinates over a different frame, we will this frame as overlay

```
cv2.fillPoly(overlay, [mask], (0, 255, 0), cv2.LINE_AA)
```

-
    - Then we will draw the outer edge of the mask over the orginal frame

```
cv2.polylines(frame, [mask], True, (0, 255, 0))
```

-
    - Lastly we will make 2 frame transparent with sum of transparency equal 1, then we will blend 2 frame into 1 and make a new frame. We will denote transparency value is α

```
cv2.addWeighted(overlay, alpha, frame, 1 - alpha, 0, frame)
```

-
    - So, we create a function that take model’s mask coordinates and frame needed to be drawn on and output a frame with transparent mask

```
def drawMask(self, mask: np.ndarray, frame: np.ndarray = None)
-> None:
        frame = frame if frame is not None else self.frame
        # Set the transparency level for the mask overlay
        alpha = .5
        # Create a copy of the frame to use as an overlay
        overlay = frame.copy()
        # Fill the area of the mask with green color on the
overlay
        cv2.fillPoly(overlay, [mask], (0, 255, 0), cv2.LINE_AA)
        # Draw green polylines around the mask on the original
frame
        cv2.polylines(frame, [mask], True, (0, 255, 0))
        # Blend the overlay with the original frame using the
specified transparency level
        cv2.addWeighted(overlay, alpha, frame, 1 - alpha, 0,
frame)
```

3.5.3 Display length to pothole on video

- We will draw a line from camera to pothole and add length to pothole in SI unit above the line.
    - Getting line coordinate with origin is start point of line and targetPoint is end point of line

```
deltaX = origin[0] - targetPoint[0]
deltaY = origin[1] - targetPoint[1]
```

-
    - Calculate angle to rotate length to pothole text corresponding to pothole

```
angle = - np.arctan2(deltaY, deltaX) * 180 / np.pi
```

- Make length to pothole text position above and in the middle of the line

```
textX = int((targetPoint[0] + origin[0]) // 2)
textY = int((targetPoint[1] + origin[1]) // 2 - 15)
```

- Then we will rotate the text and put it on a blank canvas. However, if potholes are being detected in the haft right of the frame, the text will be upside down since we are rotating anti-clockwise, so we will make any text associates with haft right targetPoint rotates 180°.

```
M = cv2.getRotationMatrix2D((textX, textY), angle, 1)
 canvas = cv2.putText(np.zeros_like(frame), f"{SIlength}",
(textX, textY), cv2.FONT_HERSHEY_DUPLEX, .5, (0, 0, 255), 1,
cv2.LINE_AA)
rotatedText = cv2.warpAffine(canvas, M, (frame.shape[1],
frame.shape[0]))
 # If the center point is to the right, rotate text 180 degrees
if (targetPoint[0] > origin[0]):
     M = cv2.getRotationMatrix2D((textX, textY), 180, 1)
     rotatedText = cv2.warpAffine(rotatedText, M,
(frame.shape[1], frame.shape[0]))
```

- Next, we will draw line connect origin and targetPoint

```
cv2.line(frame, (targetPoint[0], targetPoint[1]), (origin[0],
origin[1]), (0, 215, 255), 2, cv2.LINE_AA)
```

- Finally, we will check any frame has length on the blank and replace text pixels with corresponding pixels in origin frame

```
textMask = cv2.cvtColor(rotatedText, cv2.COLOR_BGR2GRAY) > 0
frame[:] = np.where(np.dstack([textMask] * 3), rotatedText,
frame)
```

- So, we created a function with inputs are origin and targetPoint of the line, length from camera to pothole value in SI unit and frame that needed to be drawn on, output a frame that has been drawn.

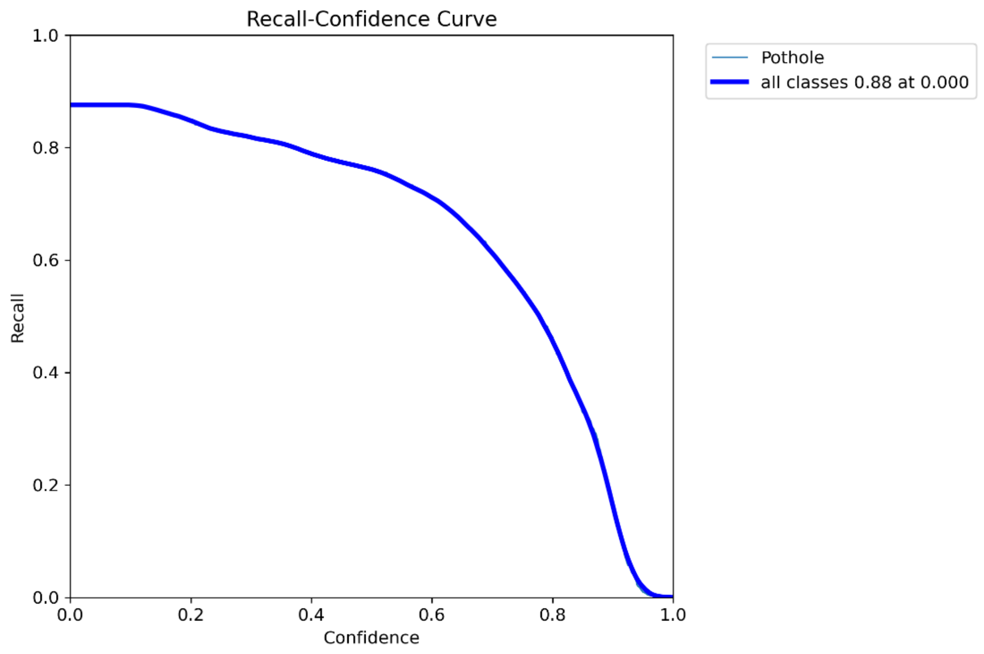
```
def drawDist(self, origin: np.ndarray, targetPoint: np.ndarray,
SIlength: float, frame: np.ndarray = None) -> None:
        frame = frame if frame is not None else self.frame
        deltaX = origin[0] - targetPoint[0]
        deltaY = origin[1] - targetPoint[1]
        angle = - np.arctan2(deltaY, deltaX) * 180 / np.pi
        textX = int((targetPoint[0] + origin[0]) // 2)
        textY = int((targetPoint[1] + origin[1]) // 2 - 15)
        M = cv2.getRotationMatrix2D((textX, textY), angle, 1)
        canvas = cv2.putText(np.zeros_like(frame),
f"{SIlength}", (textX, textY), cv2.FONT_HERSHEY_DUPLEX, .5, (0,
0, 255), 1, cv2.LINE_AA)
```

```
        rotatedText = cv2.warpAffine(canvas, M,
(frame.shape[1], frame.shape[0]))
        if (targetPoint[0] > origin[0]):
            M = cv2.getRotationMatrix2D((textX, textY), 180, 1)
            rotatedText = cv2.warpAffine(rotatedText, M,
(frame.shape[1], frame.shape[0]))
        cv2.line(frame, (targetPoint[0], targetPoint[1]),
(origin[0], origin[1]), (0, 215, 255), 2, cv2.LINE_AA)
        textMask = cv2.cvtColor(rotatedText,
cv2.COLOR_BGR2GRAY) > 0
        frame[:] = np.where(np.dstack([textMask] * 3),
rotatedText, frame)
```

4. Result

4.1 Model evaluation

4.1.1 Recall – Confidence curve:

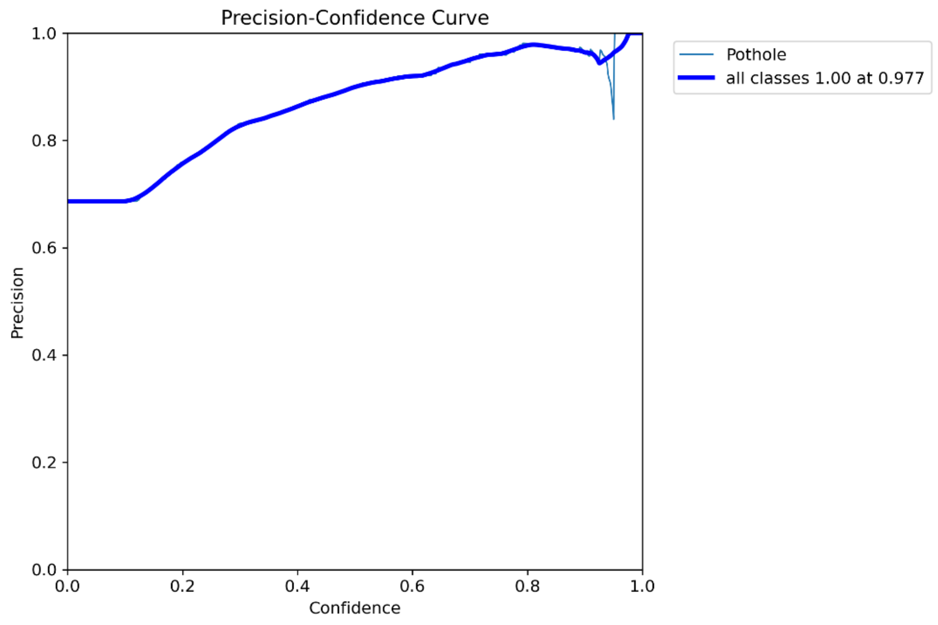


Picture 4.1.1.1: Recal – Confidence curve of final model

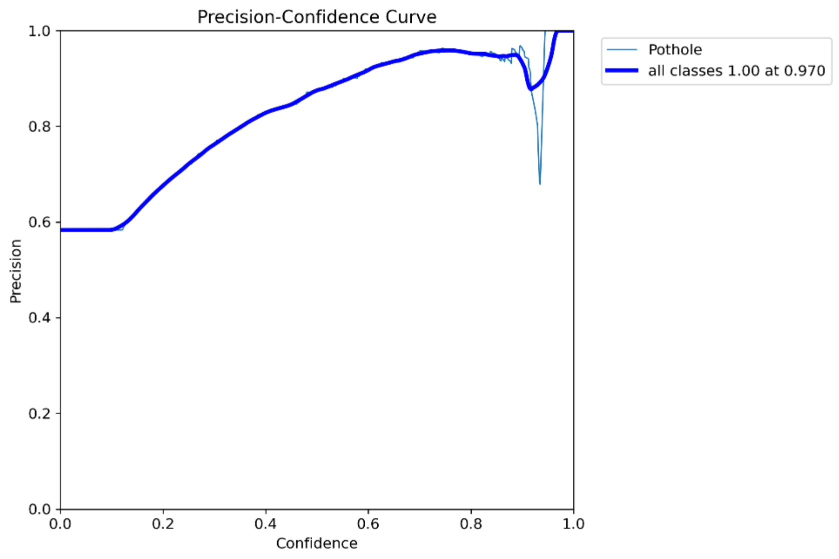
- The curve starts at high recall (close to 1) when the confidence threshold is low. This indicates that the model is able to recall almost all true positives when it is not restrictive about the confidence level.

- The recall remains high (>0.8) until the confidence threshold reaches around 0.5. This suggests that the model maintains a good recall even with a moderate confidence level.

4.1.2 Precision – Confidence curve



Picture 4.1.2.2: Precision – Confidence curve of final trained model

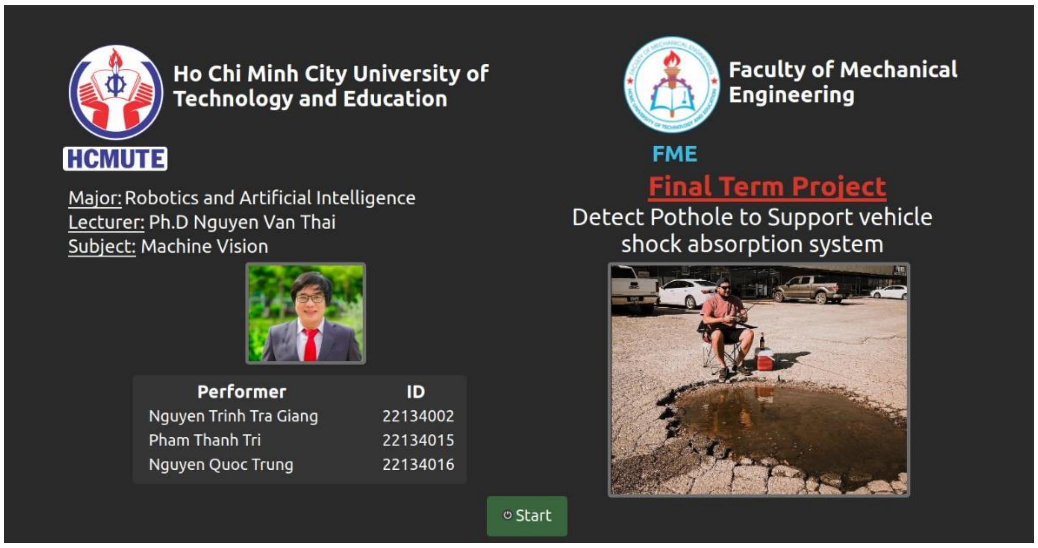


Picture 4.1.2.3: Precision – Confidence curve of first trained model

- Here we see high drop in precision at 0.8 to 1 confidence, this is expected since model detected objects's shadow as true positive at high confidence, if we compare *Picture IV.1a.2* and *Picture IV.1a.3*, the final trained model idemonstrates a smoother and more stable precision curve, has fewer fluctuations, suggesting that it is more stable and less prone to variability in its predictions compared to the first trained model.

4.2 Software application

Here is our starting UI:



Ho Chi Minh City University of Technology and Education

HCMUTE

Faculty of Mechanical Engineering

FME

Final Term Project

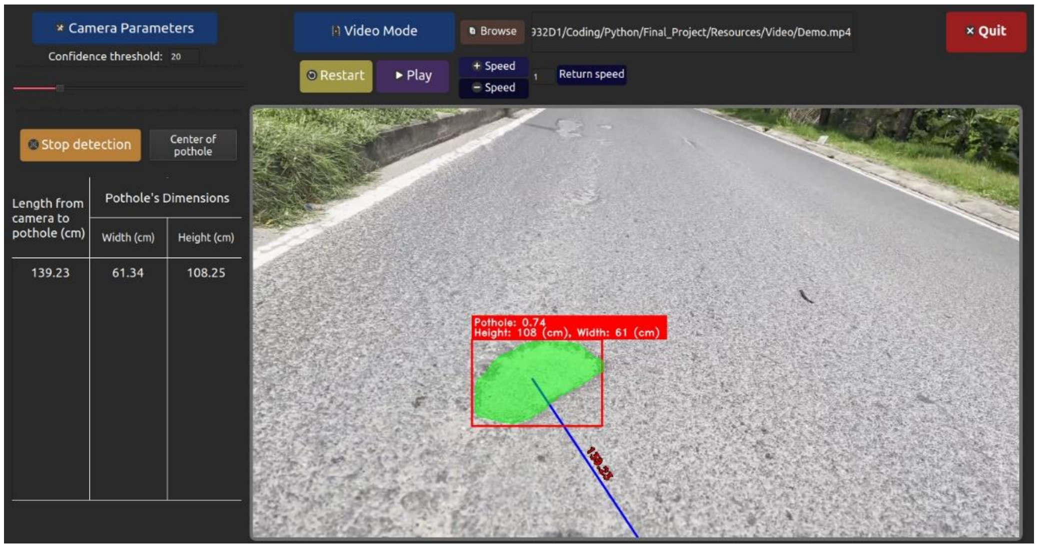
Detect Pothole to Support vehicle shock absorption system

Major: Robotics and Artificial Intelligence
Lecturer: Ph.D Nguyen Van Thai
Subject: Machine Vision

Performer	ID
Nguyen Trinh Tra Giang	22134002
Pham Thanh Tri	22134015
Nguyen Quoc Trung	22134016

Next is our controlling UI. We have 2 mode,

- First is center of pothole mode:



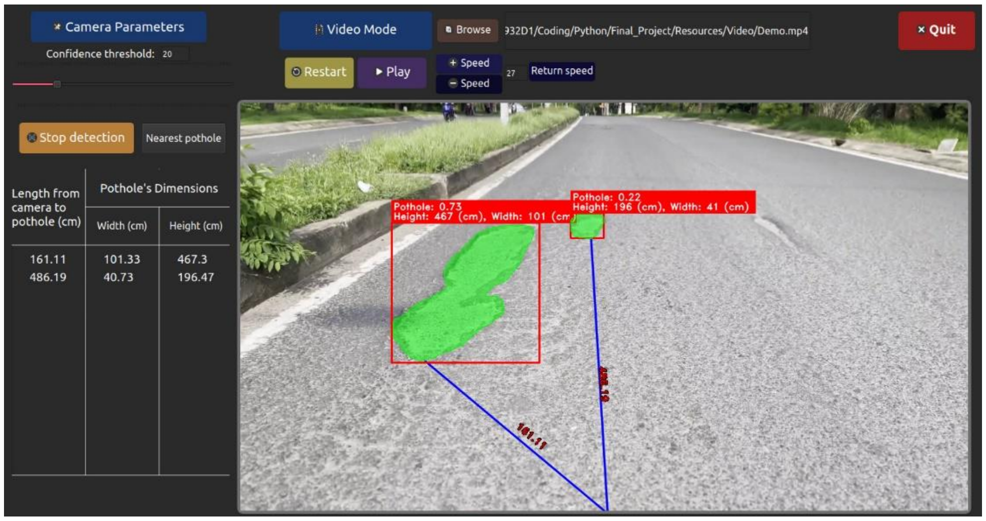
Confidence threshold: 20

932D1/Coding/Python/Final_Project/Resources/Video/Demo.mp4

Length from camera to pothole (cm)	Pothole's Dimensions	
	Width (cm)	Height (cm)
139.23	61.34	108.25

Pothole: 0.74
Height: 108 (cm), Width: 61 (cm)

- Second mode is nearest pothole mode:



Confidence threshold: 20

932D1/Coding/Python/Final_Project/Resources/Video/Demo.mp4

27

Length from camera to pothole (cm)	Pothole's Dimensions	
	Width (cm)	Height (cm)
161.11	101.33	467.3
486.19	40.73	196.47

Pothole: 0.73
Height: 467 (cm), Width: 101 (cm)

Pothole: 0.22
Height: 196 (cm), Width: 41 (cm)

4.3 Efficiency analysis

Our tests indicated that while the model performs exceptionally well on dedicated GPUs, achieving real-time analysis speeds of up to 30 frames per second, its performance drops significantly on less capable hardware, which can be a limiting factor for deployment in less controlled environments.

5. Conclusion and future work

5.1 Summary

In this study, we developed and evaluated an integrated system for pothole detection and measurement using YOLOv8 and classical machine vision techniques. The combination of these methods leverages the strengths of each approach to create a comprehensive and robust solution for road maintenance and safety management.

5.2 Limitations

While the integration of YOLOv8 and classical machine vision methods has shown promising results in pothole detection and measurement, the system does face certain limitations that can impact its performance and applicability in real-world scenarios:

Environmental Conditions:

- The performance of both YOLOv8 and classical machine vision techniques can significantly vary under different environmental conditions. Extreme lighting, such as direct sunlight, heavy shadows, or night-time conditions, can affect the visibility of potholes in the images captured. Additionally, adverse weather conditions like rain, snow, or fog can obscure pothole boundaries, leading to decreased detection accuracy. These variations in environmental conditions require robust image preprocessing and possibly adaptive algorithms that can adjust to changes in visibility and contrast.

Complex Road Surfaces:

- Highly textured or complex road surfaces present a significant challenge for pothole detection systems. Surfaces with patches, varying asphalt types, or previous repairs can create patterns that mimic or obscure potholes, leading to false positives or missed detections. The presence of road markings, such as crosswalks and painted symbols, can also complicate the detection process. These issues necessitate more sophisticated segmentation techniques and improved training datasets that include a wide variety of road conditions to enhance the model's ability to differentiate between true potholes and other similar-looking features.

Processing Speed and Real-Time Application:

- Although the YOLOv8 model and classical machine vision code are optimized to run on powerful GPUs, which significantly enhance processing speed, this setup creates limitations when applied in realistic environments. The necessity for high-performance GPUs means that running the system in real-time on standard vehicle-mounted hardware or other non-specialized equipment is challenging. In realistic environments where resources may be limited, ensuring the system's efficiency and responsiveness remains difficult.

5.3 Recommendations for future research

To address these limitations, future research and development efforts should focus on enhancing the robustness of the detection algorithms under varied environmental conditions, improving the discriminative power of the models against complex road textures, and refining the measurement accuracy for diverse pothole characteristics.

Additionally, advancements in camera technology and sensor fusion approaches could mitigate the impact of hardware limitations.

Lastly, optimizing the processing algorithms to reduce latency will be crucial for deploying these technologies in real-time applications, ensuring that road safety and maintenance can be effectively managed with minimal human intervention.

6. References

https://link.springer.com/article/10.1007/s42979-024-02887-1
https://www.baeldung.com/cs/focal-length-intrinsic-camera-parameters
https://leimao.github.io/blog/Camera-Intrinsics-Extrinsics/