# Digital Image Processing

## (EE-333)

# DE-43 Mechatronics

### Syndicate – A

**Project Report**

**Project Title: Crack Detection in Images using Segmentation.**


**Name and CMS ID:**

1. NC Syed Muhammad Daniyal Gillani Reg# 375785
2. NC Wajieh Badar Reg# 375245


Submitted to: Dr Tahir Nawaz

# Contents

## Abstract

The first sign of structural damage is a crack. Examining the structural components is essential since cracks can compromise the resilience and security of civil infrastructures. The method is not appropriate for inspecting many structural elements since visual inspection of cracks is tedious and time-consuming. Consequently, the development of extremely dependable and effective automatic crack detecting tools is imperative. Image processing techniques are used to examine the photos of the structural components that were taken and identify any potential flaws. Apart from image processing, machine learning approaches are frequently employed in crack detection to guarantee more accurate and predictable outcomes. An overview of the various image processing and machine learning methods for spotting fractures in concrete surfaces is provided in this article. This project involves developing a digital image processing (DIP) solution with a focus on real-world applications and potential societal and environmental benefits. Solution is to be implemented using platforms such as MATLAB and YOLO v8 instance segmentation model. This project focuses on solution for crack detection using segmentation using two approaches.


## Introduction

In order to preserve the structural integrity of infrastructure, including roads, bridges, and buildings, crack detection is essential. Unnoticed fissures may result in serious structural collapses, endangering public health and possibly causing disastrous events. Conventional hand inspection techniques take a lot of time, labour, and are frequently insufficient to adequately cover big or difficult-to-reach areas. These drawbacks highlight the requirement for more sophisticated and effective inspection methods.

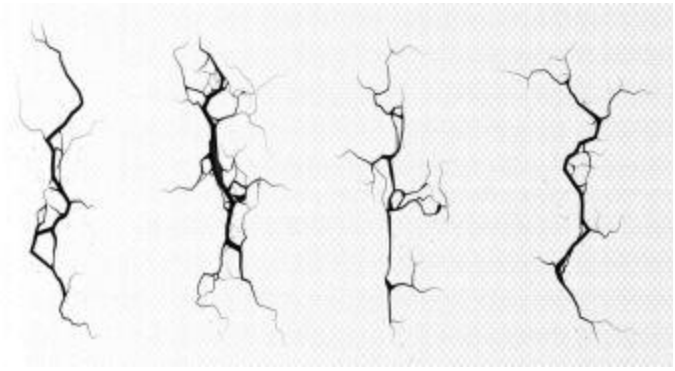

*Figure 1: Cracks illustration*

One potential way to address these issues is to implement a crack detection system based on digital image processing (DIP). By automating the detection process, such a system can notably improve the sustainability, safety, and efficiency of infrastructure management by guaranteeing more thorough and accurate inspections. Even in difficult circumstances, a DIP-based method can precisely detect and categorise cracks by utilising cutting-edge algorithms and machine learning approaches.

Furthermore, these systems may run nonstop, offering real-time monitoring and early problem detections. In addition to enhancing maintenance techniques, this proactive strategy lowers repair costs, increases infrastructure longevity, and lowers the chance of unplanned breakdowns. In the end, the use of DIP-based crack detection systems adds to the general resilience and safety of our built environment and signifies a substantial progress in smart infrastructure management.

## Methodology

Two types of methodologies were applied for the detection of cracks in images:

1. Traditional method.
2. Instance segmentation using YOLO V8.

## Traditional Method

This method aims to preprocess and analyse concrete crack images through a structured image processing workflow. The dataset includes multiple JPEG images of concrete surfaces stored in a specified input directory, with the processed results saved in a separate output directory. Each image undergoes a series of steps to enhance and identify crack features, as detailed below:

1. **Dataset Preparation**:
    - **Define Directories**: The input directory (inputDir) contains the raw concrete crack images, while the output directory (outputDir) is designated for saving processed images.
    - **Load Image Files**: A list of all JPEG files in the input directory is generated to facilitate batch processing.
2. **Reading reference image**:
    - **Load Reference Image**: A reference image ("00001.jpg") is loaded to serve as a baseline for comparison. This reference image ensures consistency in processing and provides a standard for histogram matching.
3. **For Loop for reading data**:
   For loop is used to read each image in the input directory.
4. **Reading Target Images**:
    - **Load Target Image:** The target image is read from the input directory to undergo processing.


*Figure 2: Reference image (left) vs Target image (right)*

5. **Preprocessing**:
    - **Contrast Stretching:** Both the reference and target images are enhanced using contrast stretching (**imadjust with stretchlim**). This step improves the visibility of features by expanding the dynamic range of pixel intensities.

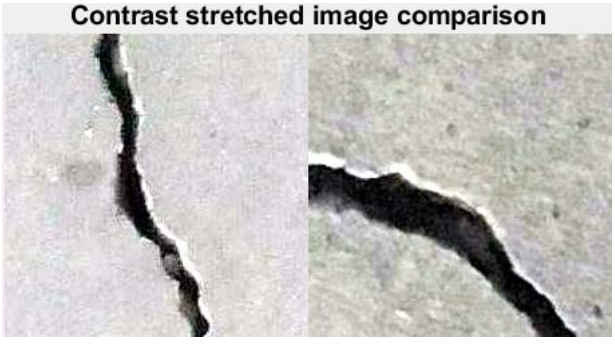


*Figure 3: Contrast stretching*

- **Grayscale Conversion**: The enhanced images are converted to grayscale (**rgb2gray**) to simplify the analysis, as crack detection relies primarily on intensity variations rather than color information.


*Figure 4: GrayScale conversion*

- **Histogram Matching:** The grayscale target image is histogram-matched to the reference image (**imhistmatch**). This standardizes the intensity distributions across images, ensuring that variations in lighting conditions do not affect the analysis.

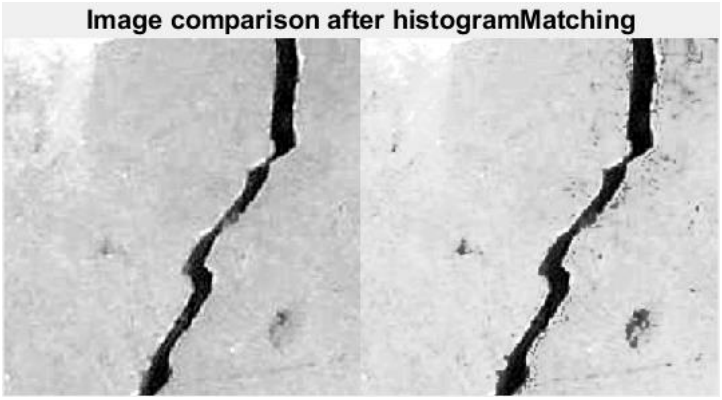


*Figure 5: Histogram matching of target image before (left) and after (right)*

6. **Post Processing**:
   - **Binary Conversion**: The processed target image is converted to a binary image (**imbinarize**). This segmentation step isolates the crack regions by distinguishing between the crack and non-crack areas based on intensity thresholds.

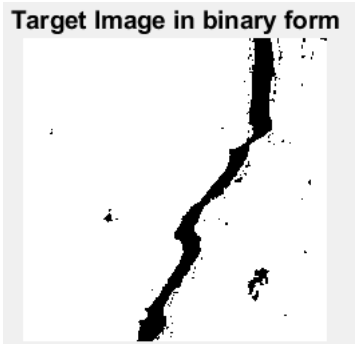


*Figure 6: Conversion of image to binary.*

   - **Noise Removal**: A morphological opening operation with a spherical structuring element (**strel('sphere', 5)**) is performed to remove noise. This step cleans the binary image by eliminating small artifacts that are not part of the actual cracks.

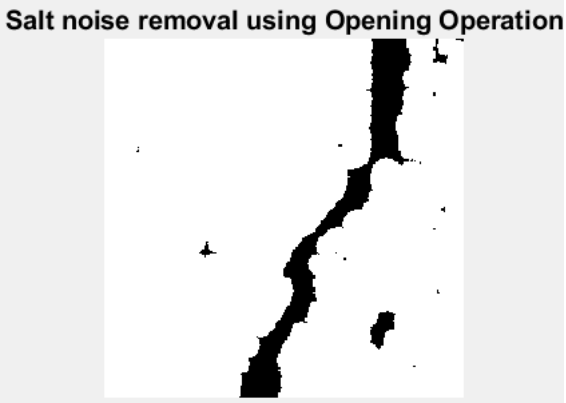


*Figure 7: Removal of Salt noise*

- **Masking**: The cleaned binary image is used to create a mask applied to the original target image. The **imoverlay()** function highlights the cracks in red, making them more visible for further analysis or visualization.

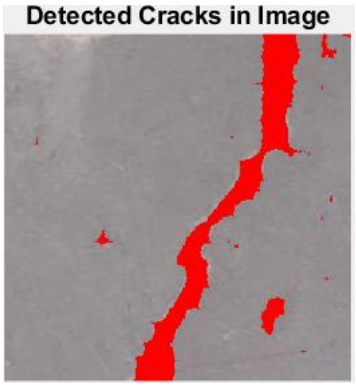


*Figure 8: Final result of cracks detected*

7. **Saving Results**:
    - **Save Processed Image**: The final masked result is saved to the output directory with a modified filename that includes a suffix (**_MaskedResult**). This indicates that the image has been processed and contains the highlighted crack regions.

This methodology ensures a consistent and systematic approach to enhancing and identifying cracks in concrete images, facilitating accurate analysis and visualization of structural defects.

## Yolov8 Instance Segmentation

YOLOv8 leverages deep learning for real-time crack detection with high accuracy and speed. Key benefits:

- **High Accuracy:** Precise crack segmentation.
- **Real-Time Processing:** Immediate identification and reporting.
- **Robust Performance:** Effective in varied conditions.
- **Scalability:** Applicable to various infrastructures.
- **Easy Integration:** Enhances existing monitoring systems.

This ensures timely, accurate detection for improved infrastructure safety and maintenance. The following are brief steps for training a model for segmentation using Yolov8 for crack detection:

- The project is carried out using Google Colab notebooks.

- A custom model training notebook from GitHub is used. The dataset is acquired from Roboflow Universe. The link is as follows: [train-yolov8-instance-segmentation-on-custom-dataset.ipynb - Colab (google.com)]

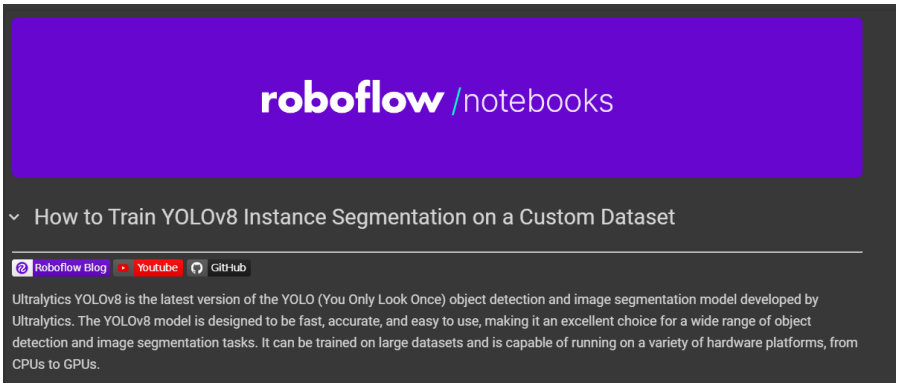


*Figure 9: Instance segmentation notebook*

- Acquire dataset from Roboflow. Use its API in the notebook above and train the model using it. Dataset link: [crack - v2 2022-09-29 2:14pm (roboflow.com)]

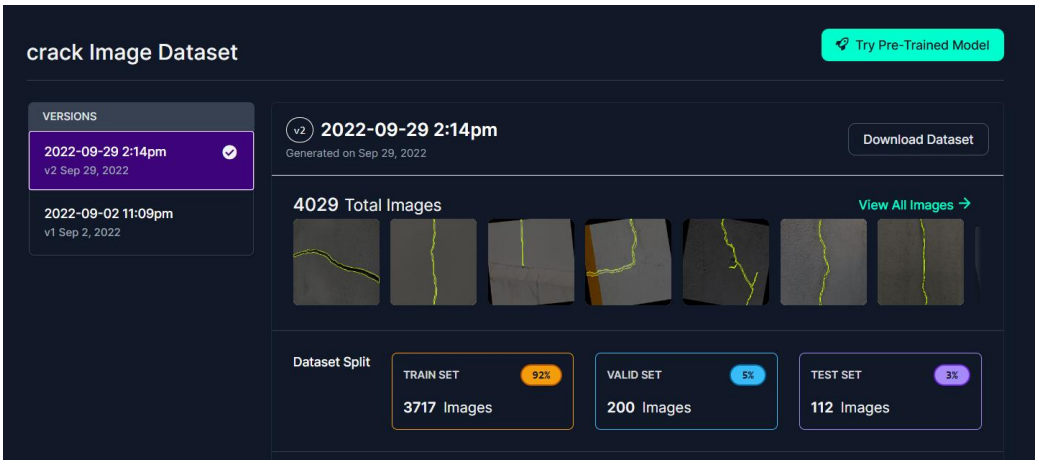


*Figure 10: Cracks dataset*

- Train the model for 10 epochs, and download the best model “best.pt”.

```
%cd {HOME}

!yolo task=segment mode=train model=yolov8s-seg.pt data={dataset.location}/data.yaml epochs=10 imgsz=640
```

| Epoch | GPU_mem | box_loss | seg_loss | cls_loss | dfl_loss | Instances | Size | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1/10 | 4.91G | 1.022 | 1.98 | 2.54 | 1.383 | 36 | 640: 100% 413/413 [03:12<00:00, 2.14it/s] | | | | | | | | |
| | Class | Images | Instances | Box(P | R | mAP50 | mAP50-95) | Mask(P | R | mAP50 | mAP50-95): 100% 59/59 [00:46<00:00, 1.28it/s] | | | | |
| | all | 1887 | 5543 | 0.682 | 0.708 | 0.715 | 0.6 | 0.679 | 0.703 | 0.708 | 0.587 | | | | |
| Epoch | GPU_mem | box_loss | seg_loss | cls_loss | dfl_loss | Instances | Size | | | | | | | | |
| 2/10 | 6.37G | 0.6079 | 1.006 | 0.8858 | 1.012 | 33 | 640: 100% 413/413 [03:01<00:00, 2.28it/s] | | | | | | | | |
| | Class | Images | Instances | Box(P | R | mAP50 | mAP50-95) | Mask(P | R | mAP50 | mAP50-95): 100% 59/59 [00:45<00:00, 1.31it/s] | | | | |
| | all | 1887 | 5543 | 0.732 | 0.724 | 0.756 | 0.643 | 0.73 | 0.716 | 0.749 | 0.629 | | | | |
| Epoch | GPU_mem | box_loss | seg_loss | cls_loss | dfl_loss | Instances | Size | | | | | | | | |
| 3/10 | 6.37G | 0.6251 | 0.9888 | 0.8112 | 1.019 | 28 | 640: 100% 413/413 [03:01<00:00, 2.28it/s] | | | | | | | | |
| | Class | Images | Instances | Box(P | R | mAP50 | mAP50-95) | Mask(P | R | mAP50 | mAP50-95): 100% 59/59 [00:47<00:00, 1.23it/s] | | | | |
| | all | 1887 | 5543 | 0.742 | 0.742 | 0.766 | 0.646 | 0.739 | 0.738 | 0.758 | 0.638 | | | | |
| Epoch | GPU_mem | box_loss | seg_loss | cls_loss | dfl_loss | Instances | Size | | | | | | | | |
| 4/10 | 6.37G | 0.6387 | 1.006 | 0.7797 | 1.027 | 26 | 640: 100% 413/413 [03:01<00:00, 2.27it/s] | | | | | | | | |
| | Class | Images | Instances | Box(P | R | mAP50 | mAP50-95) | Mask(P | R | mAP50 | mAP50-95): 100% 59/59 [00:46<00:00, 1.27it/s] | | | | |
| | all | 1887 | 5543 | 0.717 | 0.799 | 0.804 | 0.673 | 0.716 | 0.797 | 0.797 | 0.664 | | | | |

*Figure 11: Model Training*

- In a separate notebook, use this model to detect more cracks from images.
- The results can be saved to Google Drive.

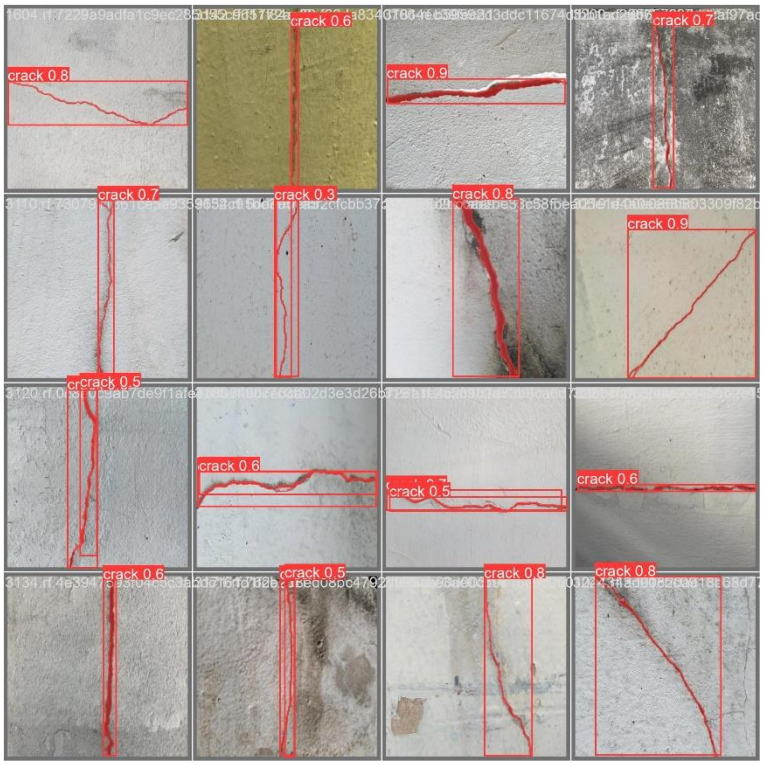


*Figure 12: Final result of cracks detected*

## Comparison of Approaches

| Feature/Aspect | Traditional Methods (No AI/ML) | YOLOv8 Instance Segmentation |
|---|---|---|
| **Approach** | Rule-based, manual tuning | Deep learning, automated |
| **Accuracy** | Moderate, quality-dependent | High, robust to variations |
| **Automation Level** | Low, manual adjustments needed | High, minimal manual intervention |
| **Adaptability** | Low, needs re-tuning for new conditions | High, retrainable for new conditions |
| **Processing Speed** | Moderate, slower on complex images | Fast, optimized for real-time |
| **Scalability** | Limited, manual scaling | High, handles large datasets |
| **Implementation Complexity** | Low to moderate | High, requires deep learning expertise |
| **Maintenance** | Moderate, periodic adjustments | Low, minimal after setup |

## Conclusion

In summary, the application of YOLOv8 and MATLAB for crack identification and segmentation has shown a great deal of promise for raising the precision and efficiency of infrastructure maintenance. Through the use of MATLAB's powerful image processing tools and the sophisticated real-time object identification capabilities of YOLOv8, this project offers a dependable and automated method for locating and evaluating cracks in a variety of constructions. This method contributes to the sustainability and lifespan of vital infrastructure in addition to increasing safety and reducing the need for manual inspections. The effective fusion of these technologies represents a significant advancement in preventive maintenance tactics and intelligent infrastructure management.

# Appendix

## Google Colab notebook

```
!pip install ultralytics==8.0.196
# Create the output folder (if it doesn't exist)
output_folder = "crack_detection_output"
import os
if not os.path.exists(output_folder):
    os.makedirs(output_folder)

# Load the model
model = YOLO("best.pt")  # pretrained YOLOv8n model

# Run batched inference on a list of images
results = model(["/content/00001.jpg",
"/content/00002.jpg","/content/00003.jpg"])  # return a list of Results objects

# Process results list
for result in results:
    boxes = result.boxes  # Boxes object for bounding box outputs
    masks = result.masks  # Masks object for segmentation masks outputs
    keypoints = result.keypoints  # Keypoints object for pose outputs
    probs = result.probs  # Probs object for classification outputs

    # Construct the output image path with filename and extension
    output_path = os.path.join(output_folder, f'{result.path.split("/")[-
1]}_Crack.jpg')  # Adjust path separator if needed

    # Save the processed image with crack detection visualization
    cv2.imwrite(output_path, result.plot())
    print(f'Image saved to: {output_path}')
```

## Matlab Code

```
%Define dataset read/write directory
%Dataset read directory
inputDir = 'D:\SystemFiles\Desktop\Digital Image Processing Project\Concrete Crack Images';
%Dataset write directory
outputDir = 'D:\SystemFiles\Desktop\Digital Image Processing Project\Results';
imageFiles = dir(fullfile(inputDir, '*.jpg'));

%Reference image
refImage = imread("00001.jpg");
%-----------------------------Step 0-------------------------------------
%---------------------------For Loop-------------------------------------
```

```
for i = 1:length(imageFiles)
%Read target images
tarImage = imread(fullfile(inputDir, imageFiles(i).name));
%figure(1)
%figure, imshowpair(refImage, tarImage, "montage")
%title("Reference Image vs Target Image");


%-------------------------Preprocessing-----------------------------------
%-----------------------------Step 1-------------------------------------
%Apply Contrast Stretching to both images
refStretch = imadjust(refImage,stretchlim(refImage));
tarStretch = imadjust(tarImage,stretchlim(tarImage));
%figure(2)
%figure, imshowpair(refStretch, tarStretch, "montage")
%title('Contrast stretched image comparison')


%-----------------------------Step 2-------------------------------------
%Convert both images to grayScale
refGray = rgb2gray(refStretch);
tarGray = rgb2gray(tarStretch);
%figure(3)
%figure, imshowpair(refGray, tarGray, "montage")
%title('RGB to Gray Conversion')


%-----------------------------Step 3-------------------------------------
%Perform Histogram Matching on target image
tarMatch = imhistmatch(tarGray, refGray);
%figure(4)
%figure, montage({tarGray, tarMatch})
%title('Image comparison after histogramMatching')


%------------------Post Processing -> Target Image-----------------------
%-----------------------------Step 4-------------------------------------
%Convert Processed target image to binary
tarBW = imbinarize(tarMatch);
%figure(5)
%figure, imshow(tarBW)


%-----------------------------Step 5-------------------------------------
%Remove noise from final Result
%define structuring element
se = strel('sphere',5);

%perform opening operation to remove Salt noise
tarOpen = imopen(tarBW,se);
%figure(6)
%figure, imshow(tarOpen,[]);


%-----------------------------Step 6-------------------------------------
%Apply Mask on original image - Final Result
MaskedResult = imoverlay(tarImage, ~tarOpen, 'red');
%figure(7)
%figure, imshow(MaskedResult)


%-----------------------------Step 7-------------------------------------
%Save the masked result to the output directory
```

```
    [~, name, ~] = fileparts(imageFiles(i).name);
    outputFileName = fullfile(outputDir, strcat(name, '_MaskedResult.jpg'));
    imwrite(MaskedResult, outputFileName);
end
```