TATA STEEL

TATA STEEL

#WeAlsoMakeTomorrow

Project Report: Automated Diagram Extraction and Bend Counting from BBS (Bar Bending Schedule) Documents

Purpose of the Project

The project aims to automate the extraction and analysis of diagrams from Bar Bending Schedule (BBS) documents, and then to automate the calculation of the number of edges in each diagram. The primary objective is to enhance efficiency, accuracy, and consistency in processing BBS documents by leveraging image processing and machine learning techniques.

Intended Audience

This project is intended for:

- Engineers and analysts at Tata Steel involved in the construction sector.

- Software developers and data scientists working on automating document processing tasks.

- Stakeholders interested in reducing manual errors and increasing productivity in handling construction project documentation.

Market Overview

The construction industry is a massive global market, valued at around \$10.7 trillion (about \$33,000 per person in the US) in 2020 (Source: Statista). It encompasses a wide range of activities, including residential, commercial, and infrastructure projects. Within this industry, the use of rebar (reinforcing steel bars) is crucial for providing structural integrity in concrete constructions. Bar Bending Schedules (BBS) documents are essential for detailing the rebar shapes and configurations required for each construction project.

Target Market

The primary target market for the Image Analytics for Construction Project BBS solution is steel manufacturing companies like Tata Steel, that provide rebar and related construction materials. These companies often receive numerous BBS documents from clients, which need to be analyzed and processed.

Market Drivers

- Increasing construction activities globally, driven by urbanization and infrastructure development.
- Growing demand for efficient and accurate rebar detailing and estimation processes.
- Rising focus on cost optimization and minimizing errors in construction projects.
- Adoption of digitalization and automation in the construction industry.

Market Opportunities

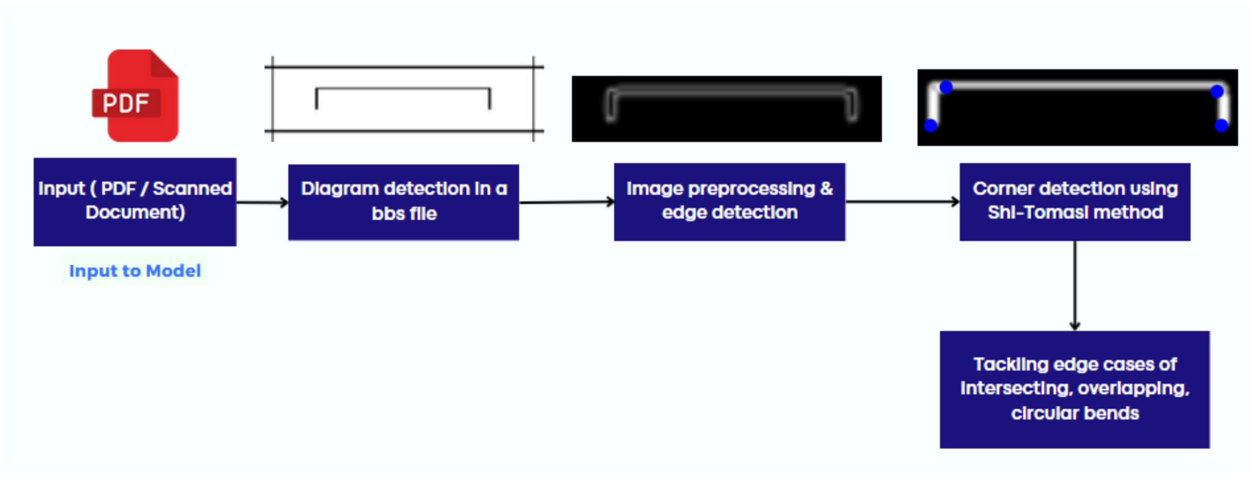
- Expanding the solution's capabilities to include additional construction document analysis, such as architectural drawings and structural plans.
- Offering the solution as a cloud-based service, enabling easier access and scalability for clients.
- Exploring partnership opportunities with engineering firms and construction companies to promote wider adoption of the solution.

Project Scope

The project scope is defined as follows:

1. Input: BBS documents in PDF format provided by Tata Steel and its clients.
2. Output: Extracted diagrams, identified rebar shapes, and accurate bend counts for each shape.
3. Processes:
    - Conversion of BBS PDF documents to high-quality images
    - Diagram extraction from the converted images
    - Number removal from the diagrams using CNN (Convolutional Neural Networks) models
    - Corner detection and bend counting for various rebar shapes
    - Handling of overlapping, intersecting, and circular bends

Final Pipeline


PDF
Input (PDF / Scanned Document)
Input to Model
Diagram detection in a bbs file
Image preprocessing & edge detection
Corner detection using Shi-Tomasi method
Tackling edge cases of intersecting, overlapping, circular bends

Problem Overview

Tata Steel faces significant challenges while processing numerous BBS documents from clients. The key issues include:

- Error Prone: Manual counting of bends is susceptible to errors, leading to data inaccuracies.

- Time Consuming: BBS documents often have more than 100 pages, making the manual counting process extremely time-consuming.

- Lack of Consistency: Different formulae are used for complex rebar shapes, leading to non-uniform bend counting across multiple users.

Key Objectives

- To develop an automated system for detecting and extracting diagrams from BBS documents.

- Bend Counting: To implement an algorithm that accurately counts bends in the extracted diagrams, handling complex scenarios like overlapping, intersecting, and circular bends.

- To minimize human errors and ensure consistent data extraction.

- To reduce the time required for processing BBS documents.

Approach

The project adopts a multi-step approach involving various image processing and machine learning techniques:

- PDF to Image Conversion: Use PyMuPDF to convert BBS PDF pages into high-quality images, ensuring clarity by increasing DPI.

- Image Preprocessing: Apply techniques like blurring, inversion, and dilation to prepare images for edge detection.

- Edge Detection: Utilize the Canny edge detection algorithm to identify boundaries of diagrams and table lines.

- Diagram Extraction: Implement smart cropping based on detected edges to isolate relevant diagrams.

- Corner Detection: Use the Shi-Tomasi method for detecting corners, overcoming challenges related to overlapping, intersecting, and circular bends.

- Bend Counting: Apply techniques like Perimeter Pixel-Cluster Identification (PPCI) and Perimeter Pixel-Cluster Angle Identification (PPCAI) for accurate bend counting.

- Circular Bends: Use the change in angles across the corner to detect them.

Challenges

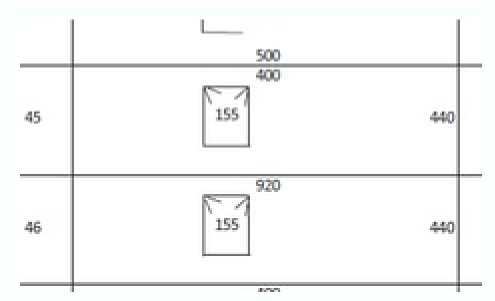
- Inconsistent Layouts: BBS files lack a standardized format for diagram placement, making it difficult to develop a one-size-fits-all solution.

- Mimicking random lines: Diagrams often visually resemble other lines within the document, making it hard to automatically distinguish them from tables.

- Interference from Non-Diagram Elements: Headers, footers and other non-diagram elements interfere with diagram detection.

- Number Removal: Presence of numbers within diagrams complicates the extraction process.

- Complex Bend Detection: Handling overlapping, intersecting, and circular bends requires sophisticated algorithms.



As we can see in this snippet of a BBS file, the numbers are remarkably close to the diagram, some are inside the diagram itself, while some are close to the table columns & rows, making it difficult to remove these numbers from the images

Potential Solutions

- Standardizing Image Preprocessing: Develop robust preprocessing pipelines to handle varied document layouts.

- Enhanced Corner Detection: Employ advanced corner detection methods and refine algorithms for better accuracy.

- Machine Learning Models: Train models using datasets augmented to mimic real-world scenarios, enhancing their ability to distinguish between relevant and irrelevant elements.

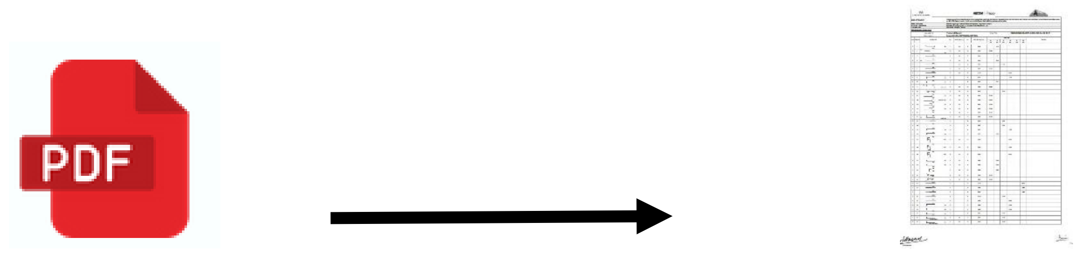
- Iterative Improvement: Continuously refine and test the system with diverse BBS documents to improve reliability and accuracy.

The solution aims to streamline the process of handling numerous BBS documents from clients, reducing the time and effort required for manual data extraction.

Detailed Approach

PDF to Image Conversion:

Extracting diagrams from Bar Bending Schedule (BBS) documents starts with converting PDF files into high-resolution images. This transformation is executed utilizing the PyMuPDF library, which facilitates the enhancement of image quality by increasing the DPI and minimizing JPEG compression artifacts. The resulting images are then transmuted into NumPy arrays, enabling flexible manipulation, and their color space is converted from RGB to BGR to augment compatibility with various image processing tools like the OpenCV library in Python.

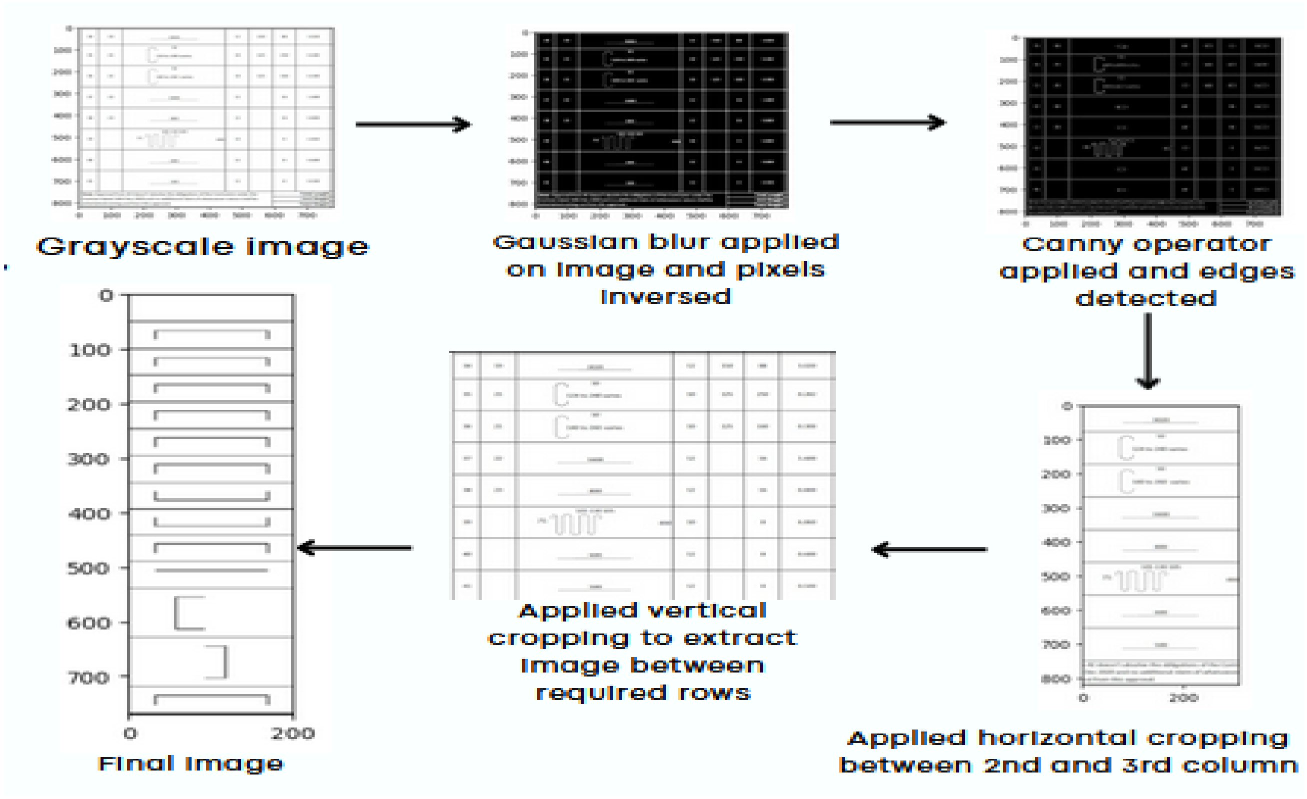


Input PDF of a BBS file

Editable image of each page of the PDF

Diagram Extraction:

The diagram extraction procedure initiates by transmuting the PDF page image to grayscale, followed by the application of a Gaussian blur to attenuate background noise and amplify clarity. Subsequently, the Canny edge detection algorithm is employed to precisely delineate the boundaries of salient elements such as table lines and diagrams. This edge information is then leveraged to implement smart cropping techniques, which automatically isolate and extract the specific regions containing the desired diagrams.

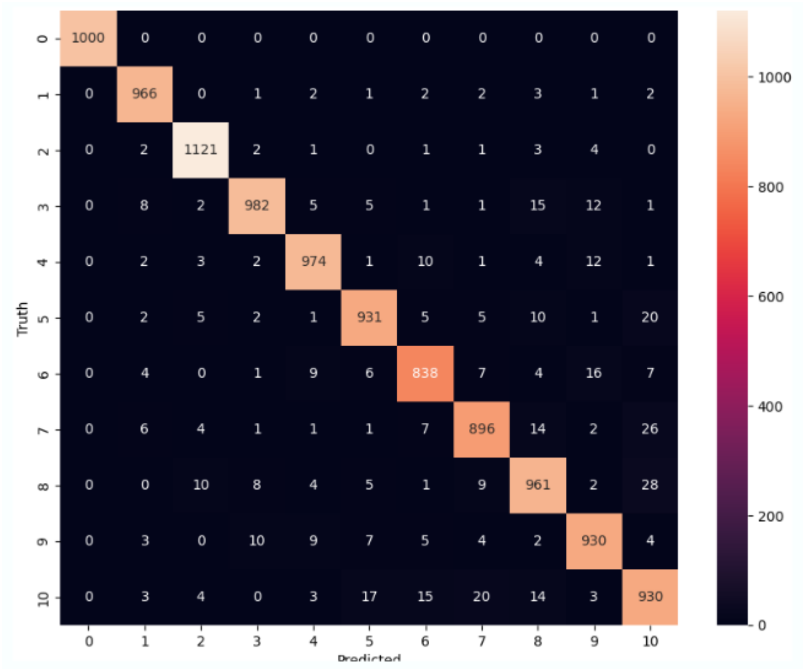

Grayscale image
Gaussian blur applied on image and pixels inversed
Canny operator applied and edges detected
Applied horizontal cropping between 2nd and 3rd column
Applied vertical cropping to extract image between required rows
Final image

Challenges in Automated Diagram Extraction:

The inconsistent layout of BBS files precludes the implementation of location-dependent algorithms, while the visual similarity between diagrams and other linear elements in the document complicates automatic distinction. To surmount these obstacles, several approaches were explored, including serial number-based identification, table detection coupled with column extraction, and the application of the Hough Transform with distance-based filtering. Unfortunately, these methods failed to yield consistent results due to variations in numbering schemes, layout inconsistencies, and the requirement for precise threshold values. The complexity of this task underscores the need for more sophisticated and robust algorithms to effectively automate the extraction and processing of diagrams from BBS documents.

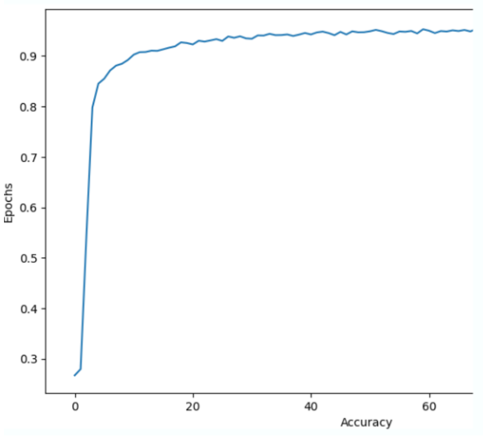
Number removal:

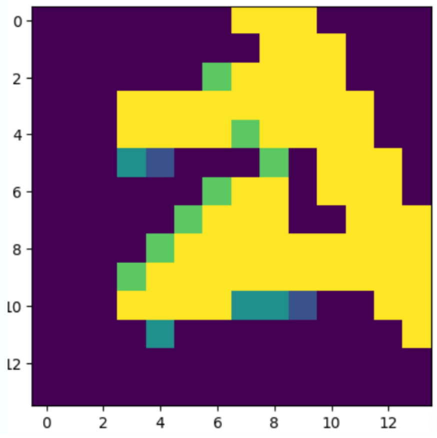
The model's training dataset was generated with the help of the MNIST dataset as a base, and using additional functions to augment, tilt, and add lines to the image; many blank images with/ without lines were added to act as a control. The purpose of the model was to

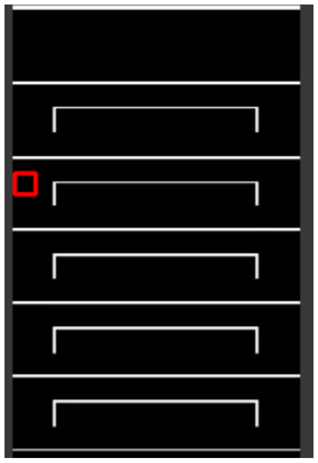


	0	1	2	3	4	5	6	7	8	9	10
0	1000	0	0	0	0	0	0	0	0	0	0
1	0	966	0	1	2	1	2	2	3	1	2
2	0	2	1121	2	1	0	1	1	3	4	0
3	0	8	2	982	5	5	1	1	15	12	1
4	0	2	3	2	974	1	10	1	4	12	1
5	0	2	5	2	1	931	5	5	10	1	20
6	0	4	0	1	9	6	838	7	4	16	7
7	0	6	4	1	1	1	7	896	14	2	26
8	0	0	10	8	4	5	1	9	961	2	28
9	0	3	0	10	9	7	5	4	2	930	4
10	0	3	4	0	3	17	15	20	14	3	930

recognize numbers and detect their presence. Using the model, we achieved ~96% accuracy on recognizing numbers, ~100% accuracy on detecting numbers.







Finally, the code was implemented using sliding window CNN to detect number patch, and cv2.line() was used to fill the line up. After the patch (number + line) was removed. The centroid points of line just outside the patch were detected and line drawn.

<u>Note</u>: Number Removal was partially implemented, currently it is being carried currently by other groups.

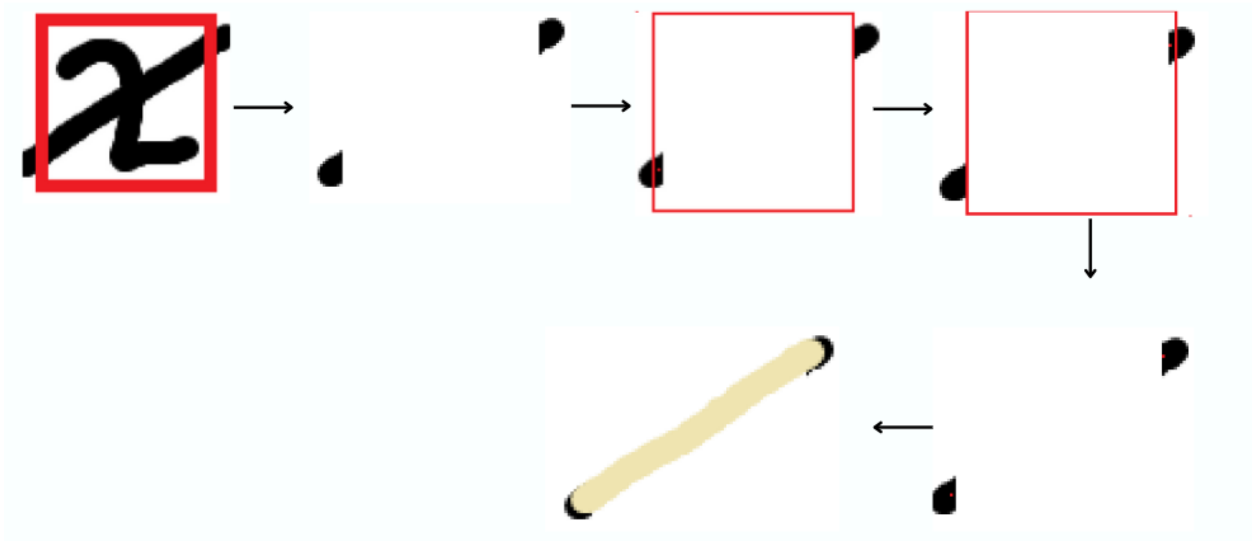
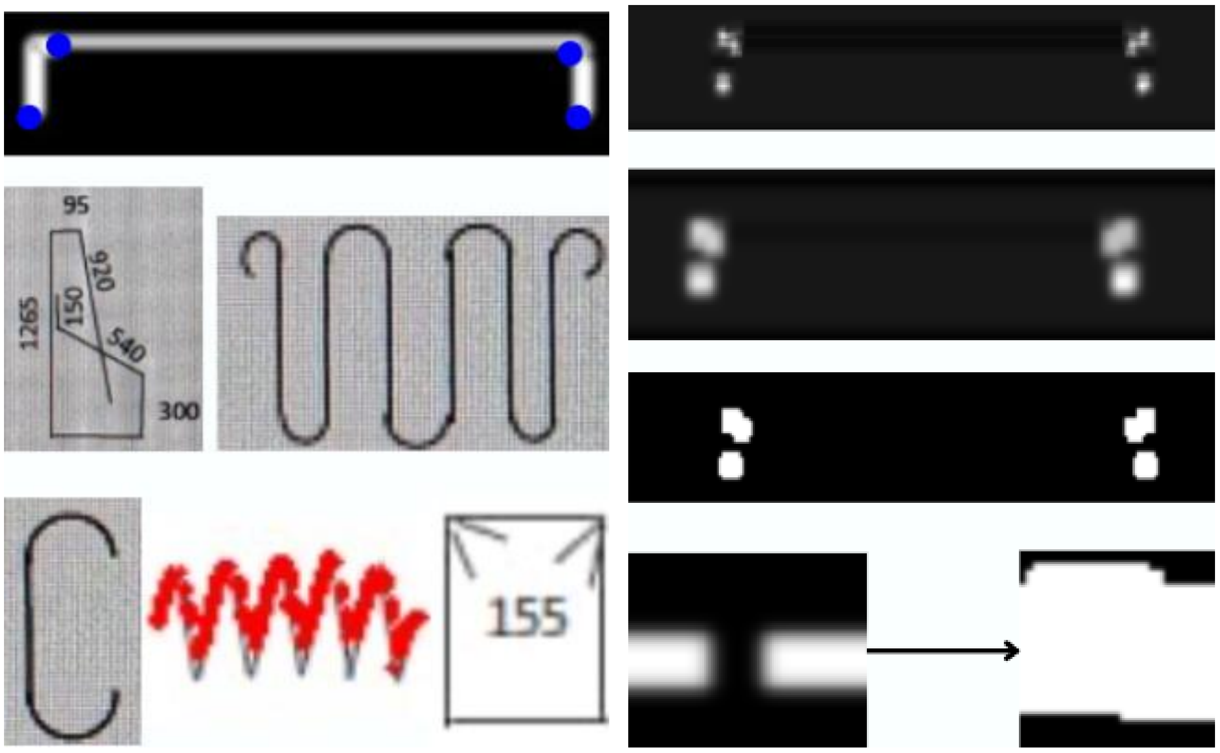


Image description of the process

Corner detection and bend counting

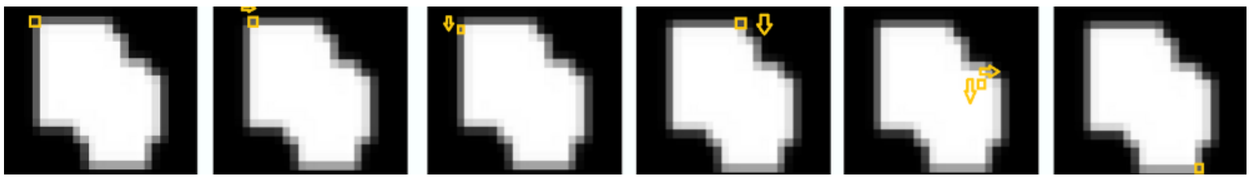
Corners are detected using the Shi-Tomasi method. There are certain edge cases remaining, for double bends in a patch, overlapping, intersecting, circular and spiral bends countered using various approaches.



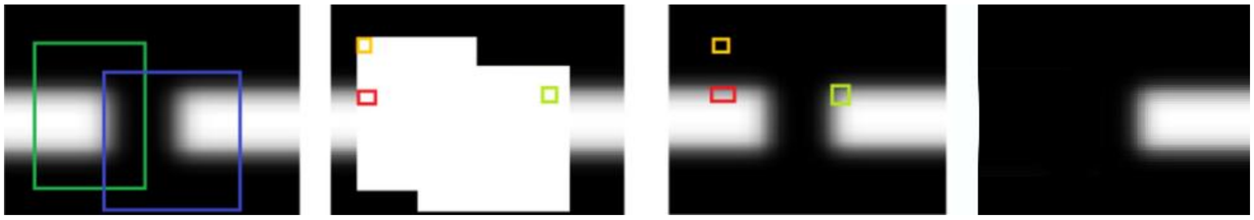
Corners were initially detected using Harrison corner detection method. The Corner Harrison detection method returns a patch based on the probability of a pixel being a corner. Challenges faced: Multiple different corners due to edges both sides. (Dilated image to tackle it). Have further thresholded it to make it binary (for easy calculations further on). Further challenge of nearby endpoints merging is faced.

Initial challenges:

The nearby endpoint challenge is solved by taking the white patch (binary) and recursing in it (shown below). We first identify the points of a patch through a recursive algorithm and then recurse again on those points, but on the main image, if we detect multiple instances of a line in the patch then that confirms that corners were merged.



Tackling Challenges:



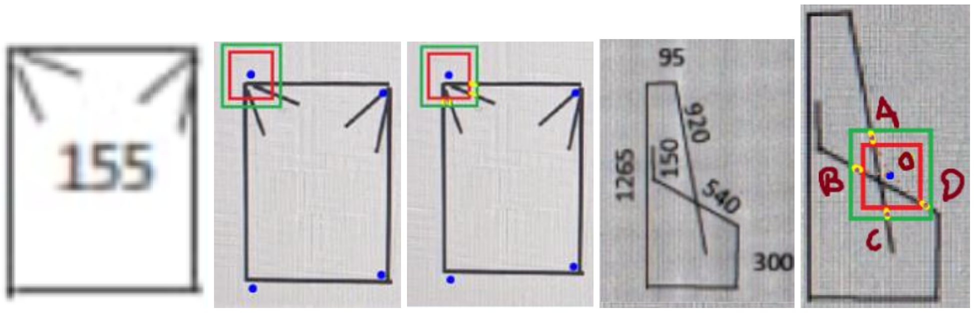
Multiple instances are detected by removing the connected line pixels (white pixels) as soon as one is detected, and recursing further. If more line pixels are detected, that means multiple line instances in a patch. The green and blue boxes merge to give the white patch (2nd), Yellow is when the first algorithm recurses on it. Red box is when the first line is detected (shown in 3), thus removing its connected pixels (4). The green box detects the second instance of the line.

Corners are detected using the Shi-Tomasi corner detection method. This is superior to Harrison corner detection method and returns the corner points right away. This further eliminates the need to check merger of patches.

Further Challenges Faced: Overlapping bend detection, Intersecting bend detection, Circular/Spiral bend detection.

Overlapping Bends

Overlapping bends are those, which are two or more behind each other but appear to be the same in a 2d space (first image). To tackle this, we devised a technique we name as "perimeter pixel-cluster identification" (PPCI). Here we draw two boxes around the detected corner and detect any pixels of the line. On detecting a pixel, we remove the connected pixels from inside the box, thus getting each to cluster separately once.



Now we have several pixel clusters (third image). To identify overlapping bends, we divide the number of pixel clusters by two. One bend is formed by two lines and thus two clusters, so clusters/two gives us number of corners. Here we got 4 lines or 4 clusters (yellow) for the top left corner, thus implying two bends on top of each other.

Intersecting Bends

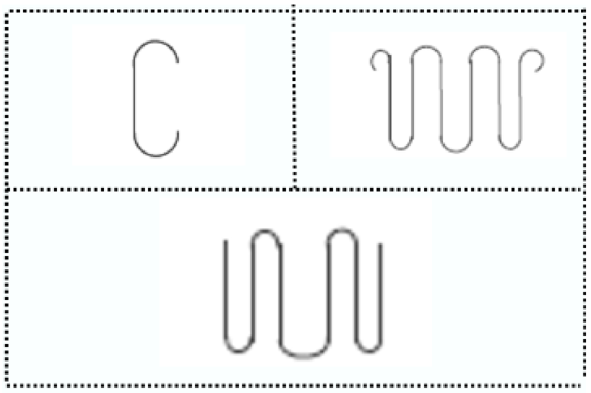
Intersecting bends (image 4): Intersecting bends are solved by a similar technique, called "Perimeter Pixel-Cluster Angle Identification" (PPCAI). Again, we draw two boxes around the detected corner and detect any pixels of the line. On detecting a pixel, we remove the connected pixels to it inside the box, but this time we also note down all the positions of the pixels in the cluster. Then we find the centroid of the clusters (fifth image).

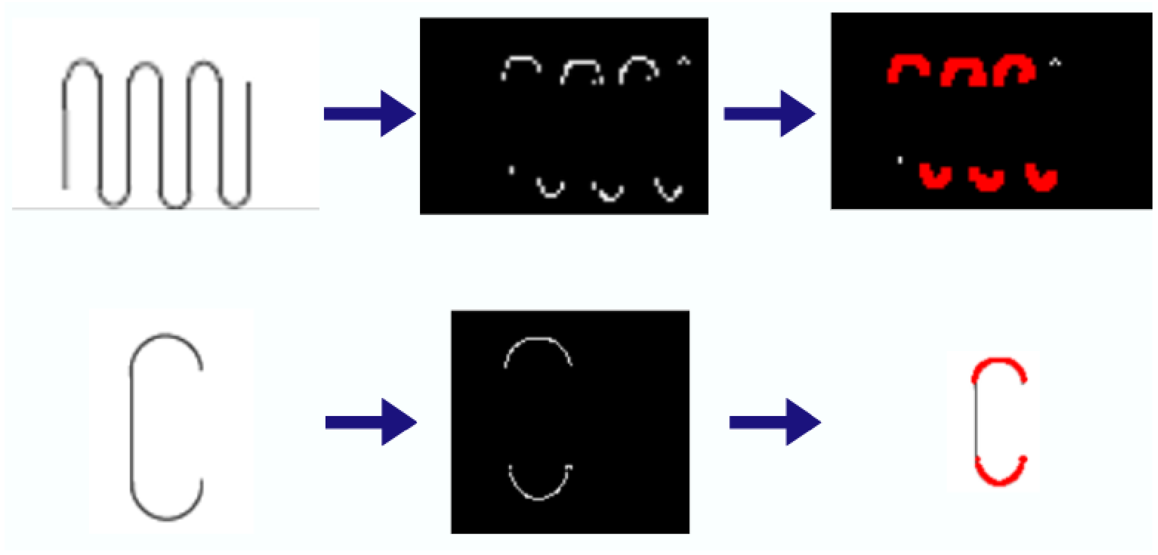
After finding the centroid we find the angles, between the corner point and centroid of cluster. For example in this case we find the clusters (yellow), their centroids (purple dots), named them (A,B,C,D) to show, and find angle between the blue dot and all 2 points, if any angle exceeds a certain threshold (180 deg ideally, around 150-160 for practical purposes) then we say that the two centroids lie on the same line and hence the corner detected is an intersecting point.

Circular Bend Counting

Circular Bends are bends which aren't sharp enough to be detected directly using the previous methods.

We have used the change in the angles across the corner to detect circular bends. We have applied angle normalisation using the Sobel Operator which gives us the following output -


Examples of Circular Bends



Benefits

- Increased Accuracy: Automated extraction reduces human errors, ensuring accurate data collection.

- Efficiency: Significantly reduces the time required to process BBS documents, improving overall productivity.

- Consistency: Provides uniform results across different users and documents.

- Scalability: The automated system can handle large volumes of BBS documents, making it scalable for extensive projects.

- Cost savings: By reducing the need for manual labor and minimizing errors, the solution can lead to significant cost savings for Tata Steel and its clients.

Unique Solution

The Image Analytics for Construction Project BBS solution offers a unique approach to addressing the challenges faced by Tata Steel in handling Bar Bending Schedules (BBS) documents. Here are some key aspects that make this solution unique:

- **Specialized Focus on BBS Document Analysis**: While there are general document analysis solutions available, this project specifically targets the unique challenges posed by BBS documents, which contain complex diagrams, rebar shapes, and construction-specific information.

- **Advanced Image Processing Techniques**: The solution employs innovative image processing techniques, such as smart cropping, edge detection, and corner detection methods like Shi-Tomasi, to accurately extract diagrams and identify rebar shapes from BBS documents.
- **Handling of Complex Rebar Shapes**: Unlike traditional solutions that may struggle with overlapping, intersecting, and circular rebar shapes, this project incorporates specialized algorithms and techniques to accurately detect and count bends in these complex shapes. This includes methodologies like "Perimeter Pixel-Cluster Identification" and "Perimeter Pixel-Cluster Angle Identification."
- **Integration of Machine Learning**: The solution leverages ML models, such as Convolutional Neural Networks (CNNs), to tackle challenges like number removal from diagrams. This AI-powered approach enhances the solution's accuracy and adaptability.
- **End-to-End Automation**: From PDF conversion to diagram extraction, corner detection, and bend counting, the solution provides a fully automated workflow, minimizing the need for manual intervention and increasing efficiency.
- **Scalability and Flexibility**: The solution is designed to handle large volumes of BBS documents, making it suitable for projects of varying scales. Additionally, its modular architecture allows for future expansion and integration with other construction management systems.
- **Tailored for Construction Industry**: While many document analysis solutions are generic, this project is specifically tailored to meet the unique requirements of the construction industry, including adherence to industry standards, terminology, and best practices.

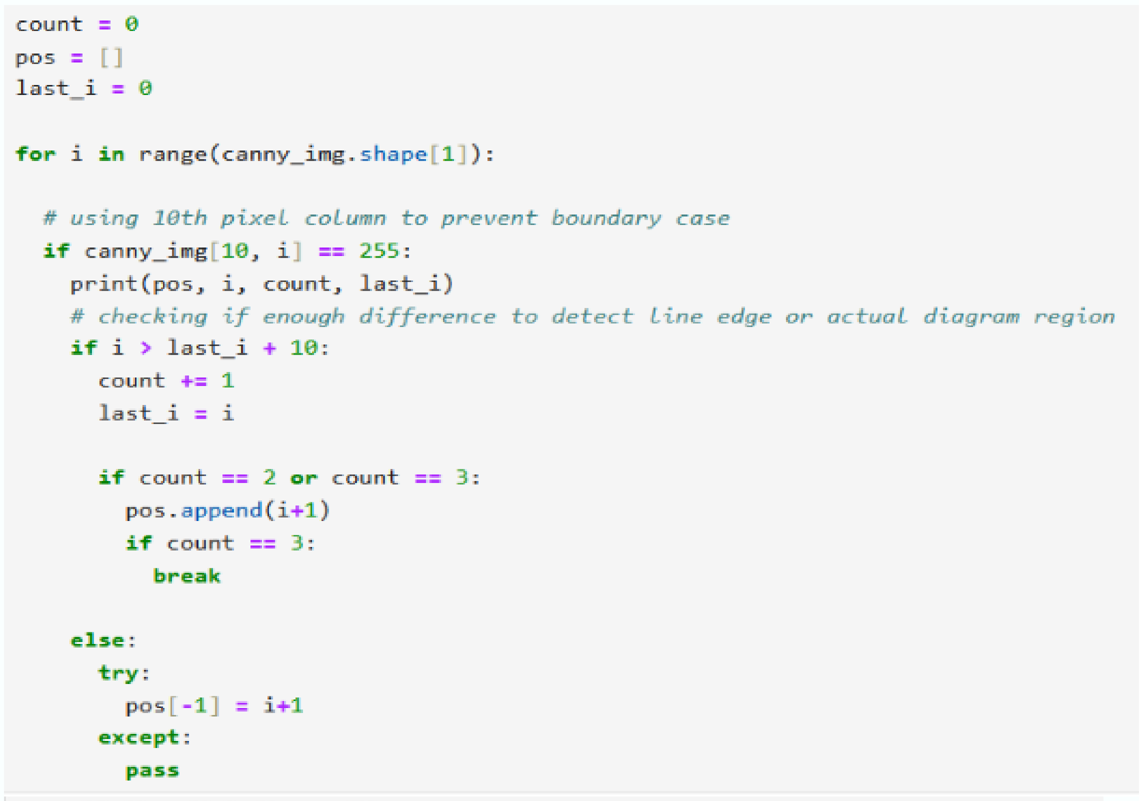
By combining advanced image processing techniques, machine learning models, and specialized algorithms for complex rebar shapes, the Image Analytics for Construction Project BBS solution offers a unique and comprehensive approach to streamlining the BBS document analysis process. This sets it apart from traditional solutions and positions it as an asset for Tata Steel, seeking improved efficiency, accuracy, and cost savings.

Conclusion

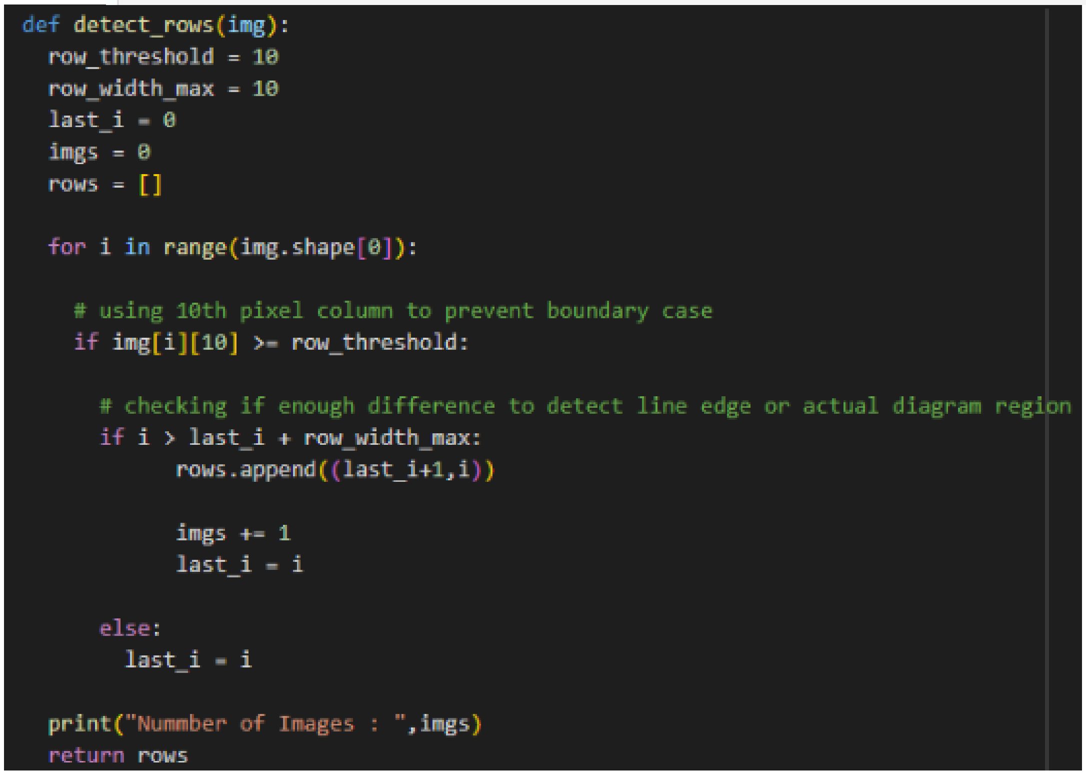
The automated diagram extraction and analysis project for BBS documents is a significant step towards modernizing construction project management at Tata Steel. By leveraging advanced image processing and machine learning techniques, the project addresses critical challenges, enhances accuracy, and improves efficiency. The solution's scalability and adaptability make it an asset for Tata Steel and its clients in the construction industry. Continuous improvements and refinements will further solidify its effectiveness, making it an invaluable tool for the construction industry.

Code Snippets

- The following code crops out the column containing all the diagrams, from a particular page of the BBS documents, achieved by first detecting the rows in an image of the BBS table

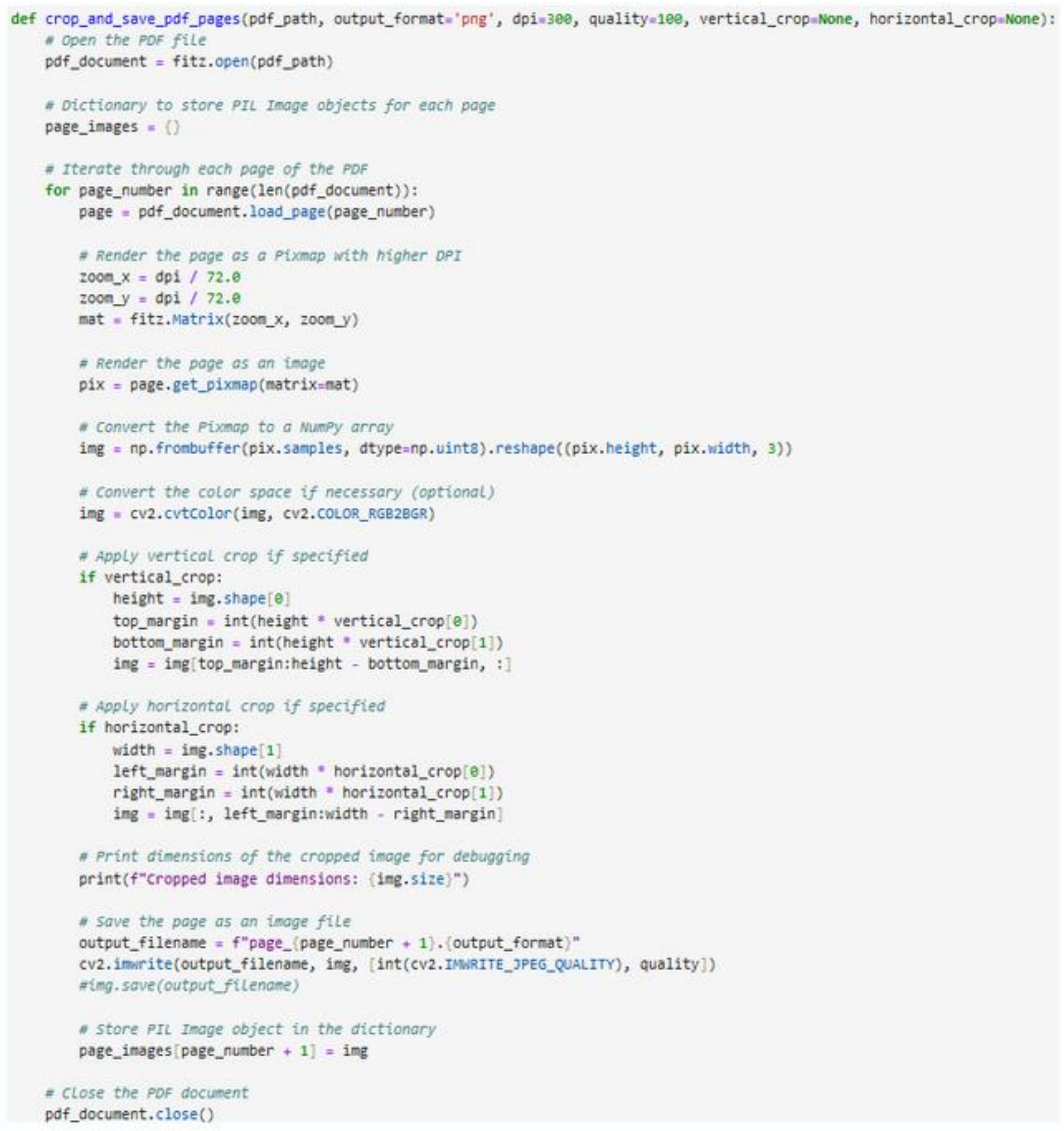


```
count = 0
pos = []
last_i = 0

for i in range(canny_img.shape[1]):

  # using 10th pixel column to prevent boundary case
  if canny_img[10, i] == 255:
    print(pos, i, count, last_i)
    # checking if enough difference to detect line edge or actual diagram region
    if i > last_i + 10:
      count += 1
      last_i = i

      if count == 2 or count == 3:
        pos.append(i+1)
        if count == 3:
          break

    else:
      try:
        pos[-1] = i+1
      except:
        pass
```



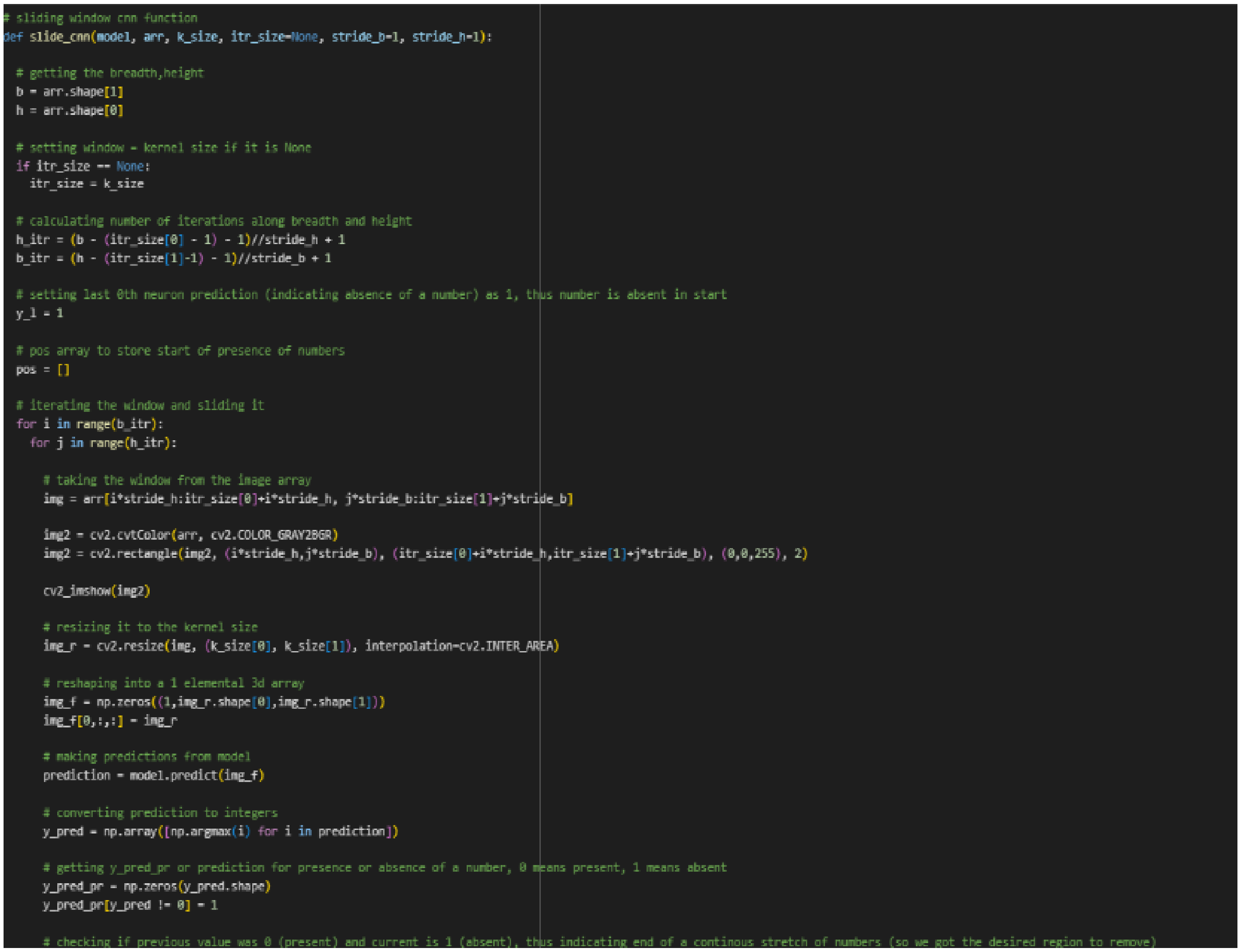
```
def detect_rows(img):
  row_threshold = 10
  row_width_max = 10
  last_i = 0
  imgs = 0
  rows = []

  for i in range(img.shape[0]):

    # using 10th pixel column to prevent boundary case
    if img[i][10] >= row_threshold:

      # checking if enough difference to detect line edge or actual diagram region
      if i > last_i + row_width_max:
            rows.append((last_i+1,i))

            imgs += 1
            last_i = i

      else:
        last_i = i

  print("Nummber of Images : ",imgs)
  return rows
```

➢ This code converts the BBS document into individual pages, saves them in .png format, and in this image processing pipeline, the quality of the image is enhanced using various parameters and functions of PyMuPDF library

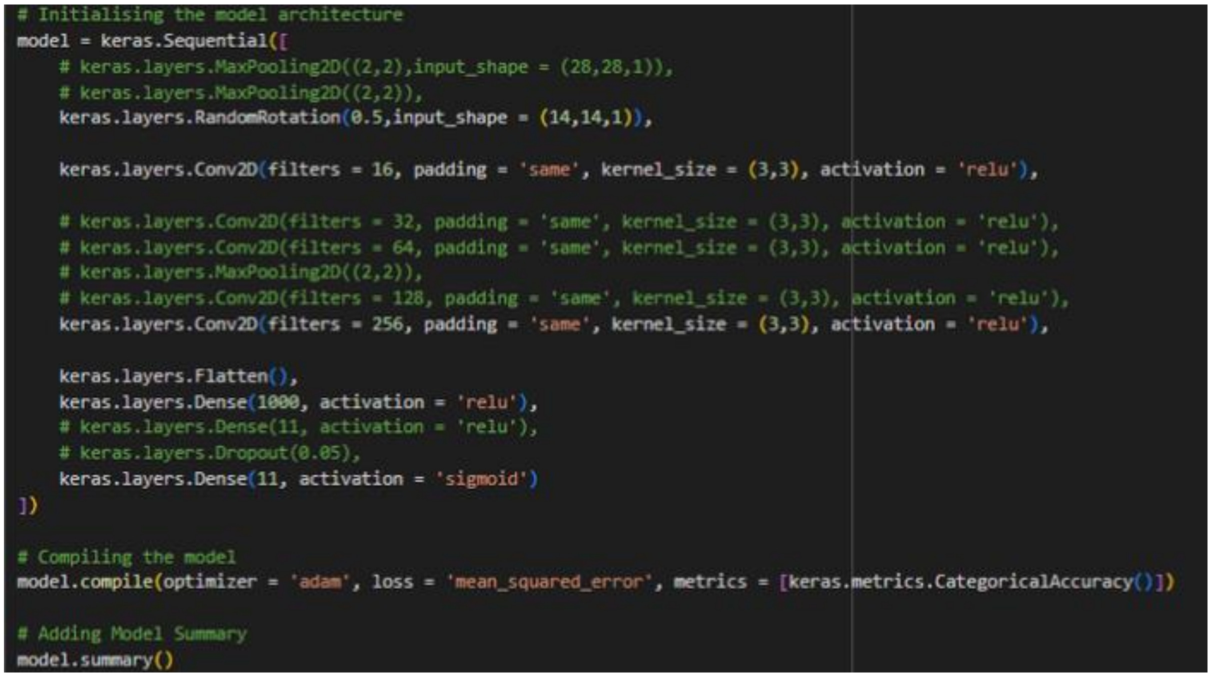


```python
def crop_and_save_pdf_pages(pdf_path, output_format='png', dpi=300, quality=100, vertical_crop=None, horizontal_crop=None):
    # Open the PDF file
    pdf_document = fitz.open(pdf_path)

    # Dictionary to store PIL Image objects for each page
    page_images = {}

    # Iterate through each page of the PDF
    for page_number in range(len(pdf_document)):
        page = pdf_document.load_page(page_number)

        # Render the page as a Pixmap with higher DPI
        zoom_x = dpi / 72.0
        zoom_y = dpi / 72.0
        mat = fitz.Matrix(zoom_x, zoom_y)

        # Render the page as an image
        pix = page.get_pixmap(matrix=mat)

        # Convert the Pixmap to a NumPy array
        img = np.frombuffer(pix.samples, dtype=np.uint8).reshape((pix.height, pix.width, 3))

        # Convert the color space if necessary (optional)
        img = cv2.cvtColor(img, cv2.COLOR_RGB2BGR)

        # Apply vertical crop if specified
        if vertical_crop:
            height = img.shape[0]
            top_margin = int(height * vertical_crop[0])
            bottom_margin = int(height * vertical_crop[1])
            img = img[top_margin:height - bottom_margin, :]

        # Apply horizontal crop if specified
        if horizontal_crop:
            width = img.shape[1]
            left_margin = int(width * horizontal_crop[0])
            right_margin = int(width * horizontal_crop[1])
            img = img[:, left_margin:width - right_margin]

        # Print dimensions of the cropped image for debugging
        print(f"Cropped image dimensions: {img.size}")

        # Save the page as an image file
        output_filename = f"page_{page_number + 1}.{output_format}"
        cv2.imwrite(output_filename, img, [int(cv2.IMWRITE_JPEG_QUALITY), quality])
        #img.save(output_filename)

        # Store PIL Image object in the dictionary
        page_images[page_number + 1] = img

    # Close the PDF document
    pdf_document.close()
```

- This is the part of the code where number removal was implemented, this shows the sliding window CNN function.
- The following code shows the CNN model initialization part, for number removal.

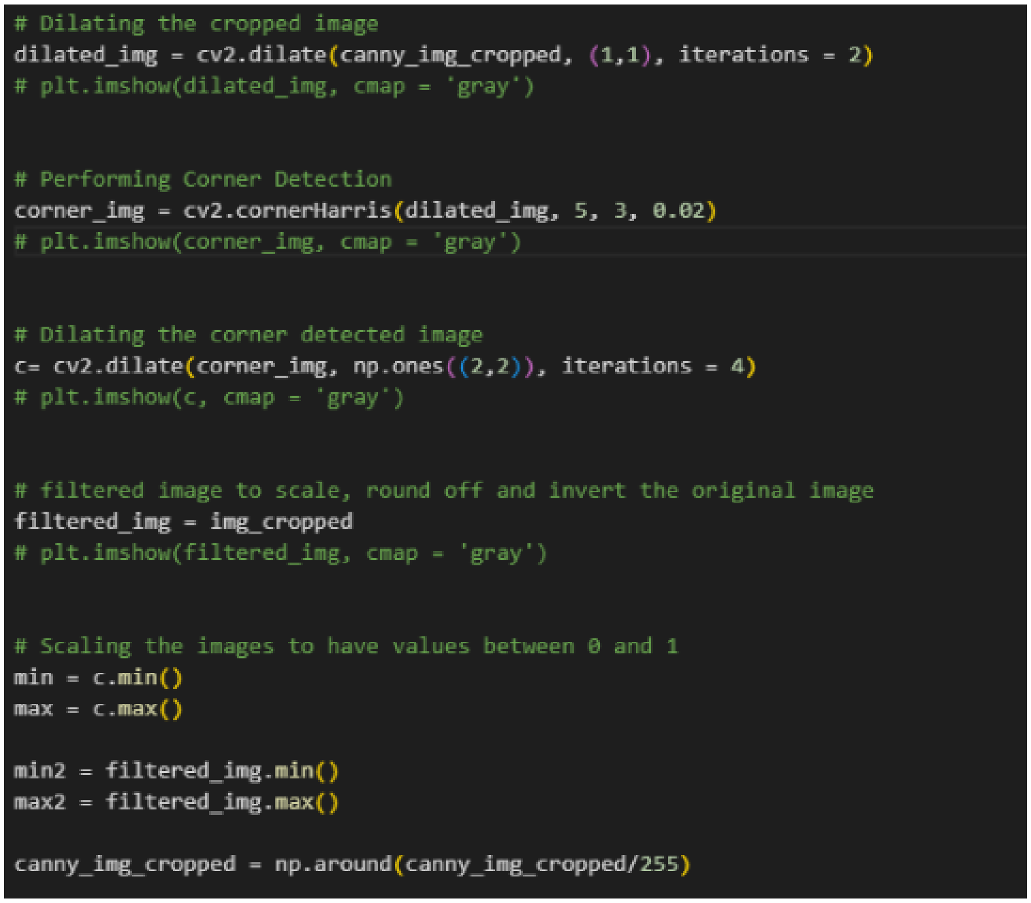


```python
# sliding window cnn function
def slide_cnn(model, arr, k_size, itr_size=None, stride_b=1, stride_h=1):

  # getting the breadth,height
  b = arr.shape[1]
  h = arr.shape[0]

  # setting window = kernel size if it is None
  if itr_size == None:
    itr_size = k_size

  # calculating number of iterations along breadth and height
  h_itr = (b - (itr_size[0] - 1) - 1)//stride_h + 1
  b_itr = (h - (itr_size[1]-1) - 1)//stride_b + 1

  # setting last 0th neuron prediction (indicating absence of a number) as 1, thus number is absent in start
  y_l = 1

  # pos array to store start of presence of numbers
  pos = []

  # iterating the window and sliding it
  for i in range(b_itr):
    for j in range(h_itr):

      # taking the window from the image array
      img = arr[i*stride_h:itr_size[0]+i*stride_h, j*stride_b:itr_size[1]+j*stride_b]

      img2 = cv2.cvtColor(arr, cv2.COLOR_GRAY2BGR)
      img2 = cv2.rectangle(img2, (i*stride_h,j*stride_b), (itr_size[0]+i*stride_h,itr_size[1]+j*stride_b), (0,0,255), 2)

      cv2_imshow(img2)

      # resizing it to the kernel size
      img_r = cv2.resize(img, (k_size[0], k_size[1]), interpolation=cv2.INTER_AREA)

      # reshaping into a 1 elemental 3d array
      img_f = np.zeros((1,img_r.shape[0],img_r.shape[1]))
      img_f[0,:,:] = img_r

      # making predictions from model
      prediction = model.predict(img_f)

      # converting prediction to integers
      y_pred = np.array([np.argmax(i) for i in prediction])

      # getting y_pred_pr or prediction for presence or absence of a number, 0 means present, 1 means absent
      y_pred_pr = np.zeros(y_pred.shape)
      y_pred_pr[y_pred != 0] = 1

      # checking if previous value was 0 (present) and current is 1 (absent), thus indicating end of a continous stretch of numbers (so we got the desired region to remove)
```

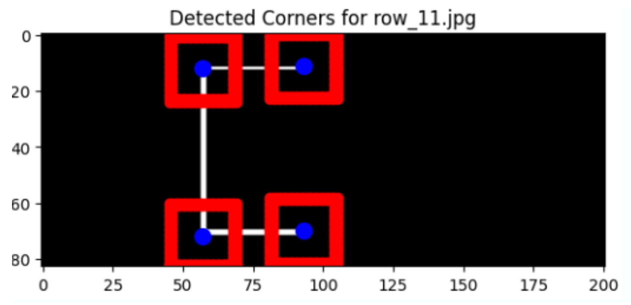


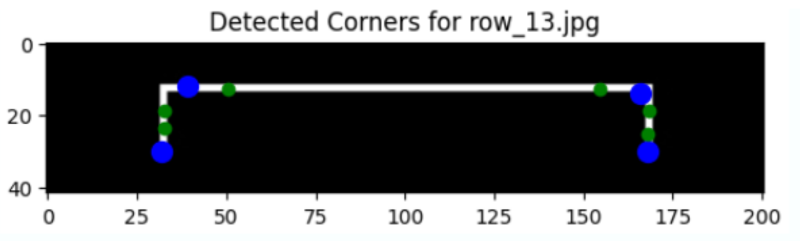
```
# Initialising the model architecture
model = keras.Sequential([
    # keras.layers.MaxPooling2D((2,2),input_shape = (28,28,1)),
    # keras.layers.MaxPooling2D((2,2)),
    keras.layers.RandomRotation(0.5,input_shape = (14,14,1)),

    keras.layers.Conv2D(filters = 16, padding = 'same', kernel_size = (3,3), activation = 'relu'),

    # keras.layers.Conv2D(filters = 32, padding = 'same', kernel_size = (3,3), activation = 'relu'),
    # keras.layers.Conv2D(filters = 64, padding = 'same', kernel_size = (3,3), activation = 'relu'),
    # keras.layers.MaxPooling2D((2,2)),
    # keras.layers.Conv2D(filters = 128, padding = 'same', kernel_size = (3,3), activation = 'relu'),
    keras.layers.Conv2D(filters = 256, padding = 'same', kernel_size = (3,3), activation = 'relu'),

    keras.layers.Flatten(),
    keras.layers.Dense(1000, activation = 'relu'),
    # keras.layers.Dense(11, activation = 'relu'),
    # keras.layers.Dropout(0.05),
    keras.layers.Dense(11, activation = 'sigmoid')
])

# Compiling the model
model.compile(optimizer = 'adam', loss = 'mean_squared_error', metrics = [keras.metrics.CategoricalAccuracy()])

# Adding Model Summary
model.summary()
```

- This is the part of the code for image preprocessing, dilation, scaling and applying Harrison corner detection.
- The following images are part of the model outputs for corner detecting and perimeter pixel cluster identification (PPCI) respectively.

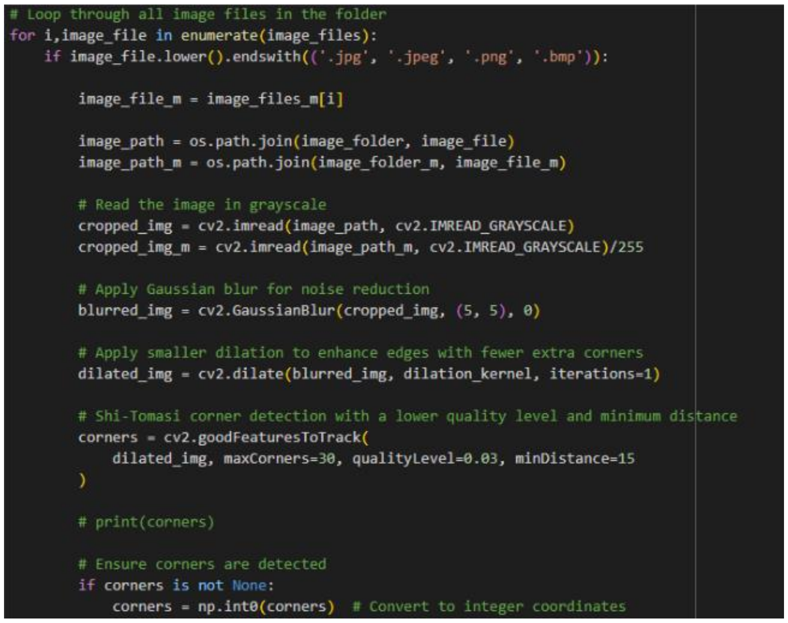


```
# Dilating the cropped image
dilated_img = cv2.dilate(canny_img_cropped, (1,1), iterations = 2)
# plt.imshow(dilated_img, cmap = 'gray')


# Performing Corner Detection
corner_img = cv2.cornerHarris(dilated_img, 5, 3, 0.02)
# plt.imshow(corner_img, cmap = 'gray')


# Dilating the corner detected image
c= cv2.dilate(corner_img, np.ones((2,2)), iterations = 4)
# plt.imshow(c, cmap = 'gray')


# filtered image to scale, round off and invert the original image
filtered_img = img_cropped
# plt.imshow(filtered_img, cmap = 'gray')


# Scaling the images to have values between 0 and 1
min = c.min()
max = c.max()

min2 = filtered_img.min()
max2 = filtered_img.max()

canny_img_cropped = np.around(canny_img_cropped/255)
```

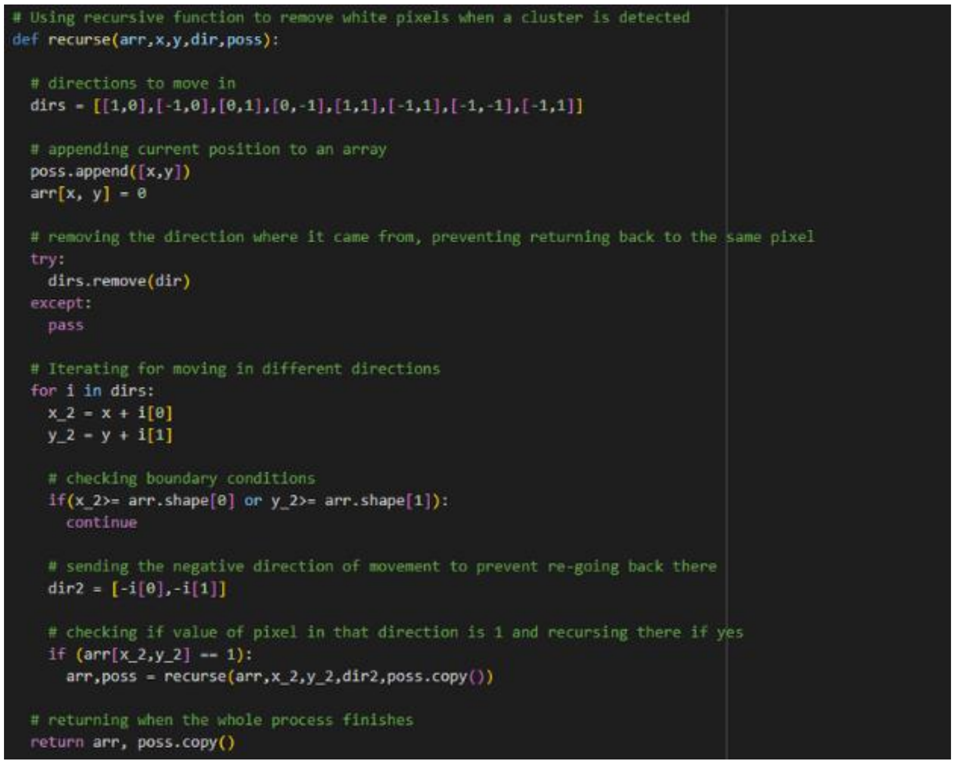




- This is the part of the code for image loading, blurring, dilation and applying the final method utilized, Shi-Tomasi corner detection.
- The following code is for the recurse function, used to iterate over the image to identify and remove pixel clusters, for patch counting.

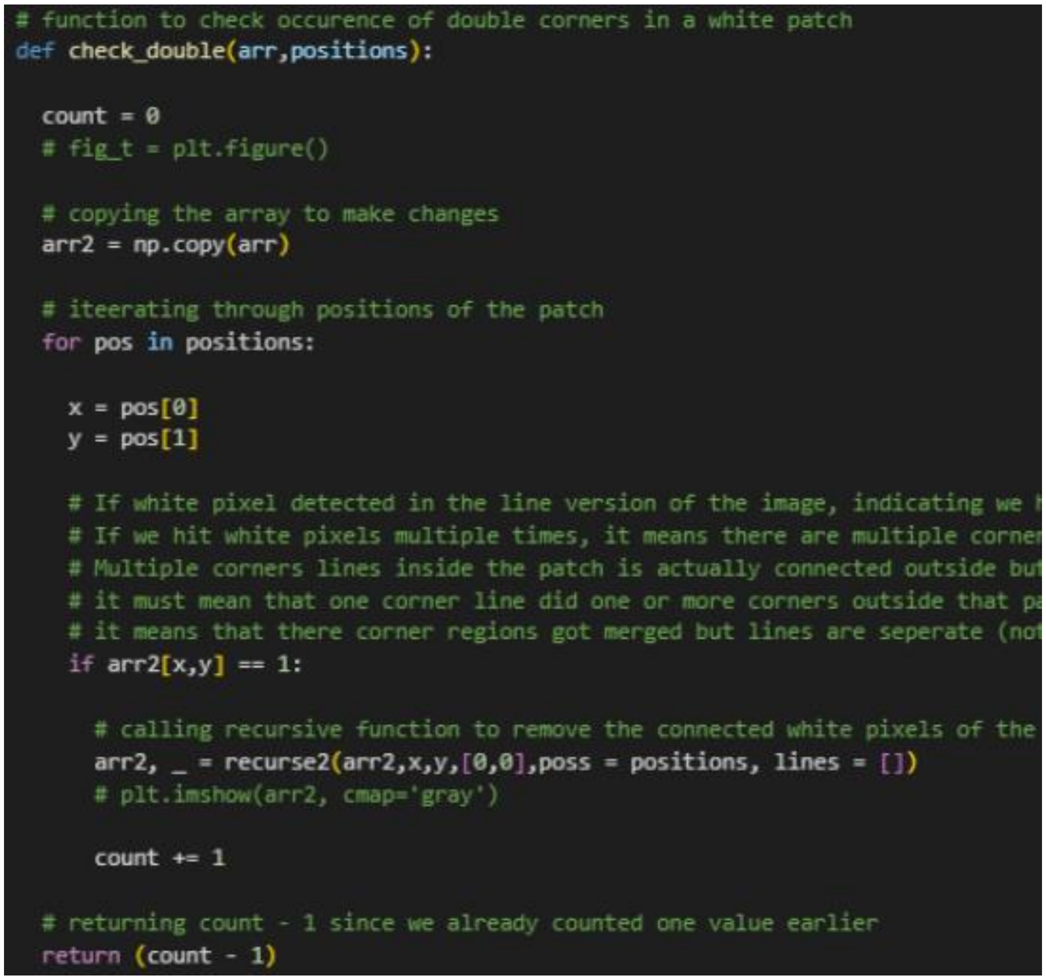


```python
# Loop through all image files in the folder
for i,image_file in enumerate(image_files):
    if image_file.lower().endswith(('.jpg', '.jpeg', '.png', '.bmp')):

        image_file_m = image_files_m[i]

        image_path = os.path.join(image_folder, image_file)
        image_path_m = os.path.join(image_folder_m, image_file_m)

        # Read the image in grayscale
        cropped_img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
        cropped_img_m = cv2.imread(image_path_m, cv2.IMREAD_GRAYSCALE)/255

        # Apply Gaussian blur for noise reduction
        blurred_img = cv2.GaussianBlur(cropped_img, (5, 5), 0)

        # Apply smaller dilation to enhance edges with fewer extra corners
        dilated_img = cv2.dilate(blurred_img, dilation_kernel, iterations=1)

        # Shi-Tomasi corner detection with a lower quality level and minimum distance
        corners = cv2.goodFeaturesToTrack(
            dilated_img, maxCorners=30, qualityLevel=0.03, minDistance=15
        )

        # print(corners)

        # Ensure corners are detected
        if corners is not None:
            corners = np.int0(corners)  # Convert to integer coordinates
```



```python
# Using recursive function to remove white pixels when a cluster is detected
def recurse(arr,x,y,dir,poss):

  # directions to move in
  dirs = [[1,0],[-1,0],[0,1],[0,-1],[1,1],[-1,1],[-1,-1],[-1,1]]

  # appending current position to an array
  poss.append([x,y])
  arr[x, y] = 0

  # removing the direction where it came from, preventing returning back to the same pixel
  try:
    dirs.remove(dir)
  except:
    pass

  # Iterating for moving in different directions
  for i in dirs:
    x_2 = x + i[0]
    y_2 = y + i[1]

    # checking boundary conditions
    if(x_2>= arr.shape[0] or y_2>= arr.shape[1]):
      continue

    # sending the negative direction of movement to prevent re-going back there
    dir2 = [-i[0],-i[1]]

    # checking if value of pixel in that direction is 1 and recursing there if yes
    if (arr[x_2,y_2] == 1):
      arr,poss = recurse(arr,x_2,y_2,dir2,poss.copy())

  # returning when the whole process finishes
  return arr, poss.copy()
```

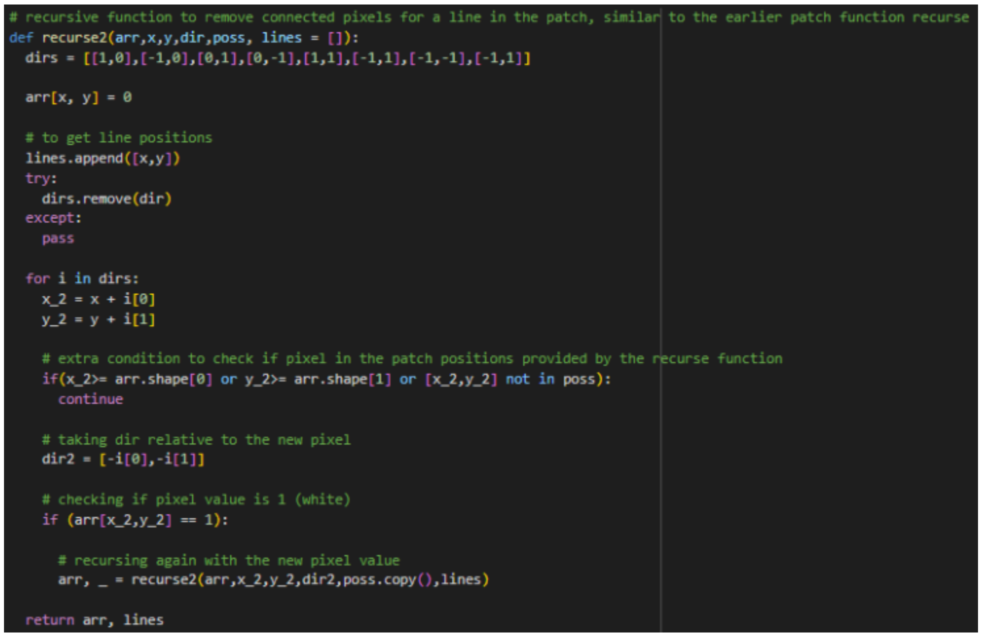
➢ Function to check occurrence of double corners (non-overlapping) in a white patch.



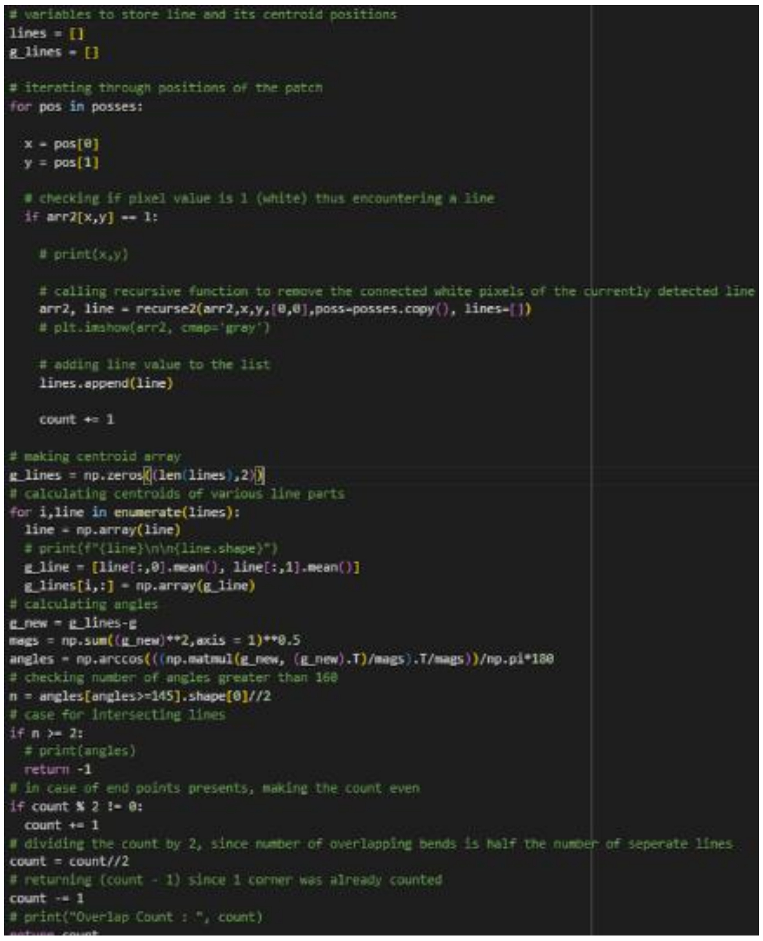
```python
# function to check occurence of double corners in a white patch
def check_double(arr,positions):

  count = 0
  # fig_t = plt.figure()

  # copying the array to make changes
  arr2 = np.copy(arr)

  # iteerating through positions of the patch
  for pos in positions:

    x = pos[0]
    y = pos[1]

    # If white pixel detected in the line version of the image, indicating we
    # If we hit white pixels multiple times, it means there are multiple corne
    # Multiple corners lines inside the patch is actually connected outside bu
    # it must mean that one corner line did one or more corners outside that p
    # it means that there corner regions got merged but lines are seperate (no
    if arr2[x,y] == 1:

      # calling recursive function to remove the connected white pixels of the
      arr2, _ = recurse2(arr2,x,y,[0,0],poss = positions, lines = [])
      # plt.imshow(arr2, cmap='gray')

      count += 1

  # returning count - 1 since we already counted one value earlier
  return (count - 1)
```

➢ Part of the code to get line centroids calling recurse2 function (to get line pixel clusters), calculating cluster centroids, finding angles, returning if intersecting (at least one pair ~180 deg), and to get overlapping corners, after dividing the number of clusters by 2. (one is subtracted since it was already counted for)

➢ Following is the recurse2 function, which creates two boxes around corners, gets pixels between them, and recurses in the region like recurse one, but only in that region and not the whole image. Further it iterates on the original image, from the centroid boxes, in between the box area, centroid is calculated from the positions obtained from the patch positions, after iterating recurse function on it (recurse function will return the positions after iterating).

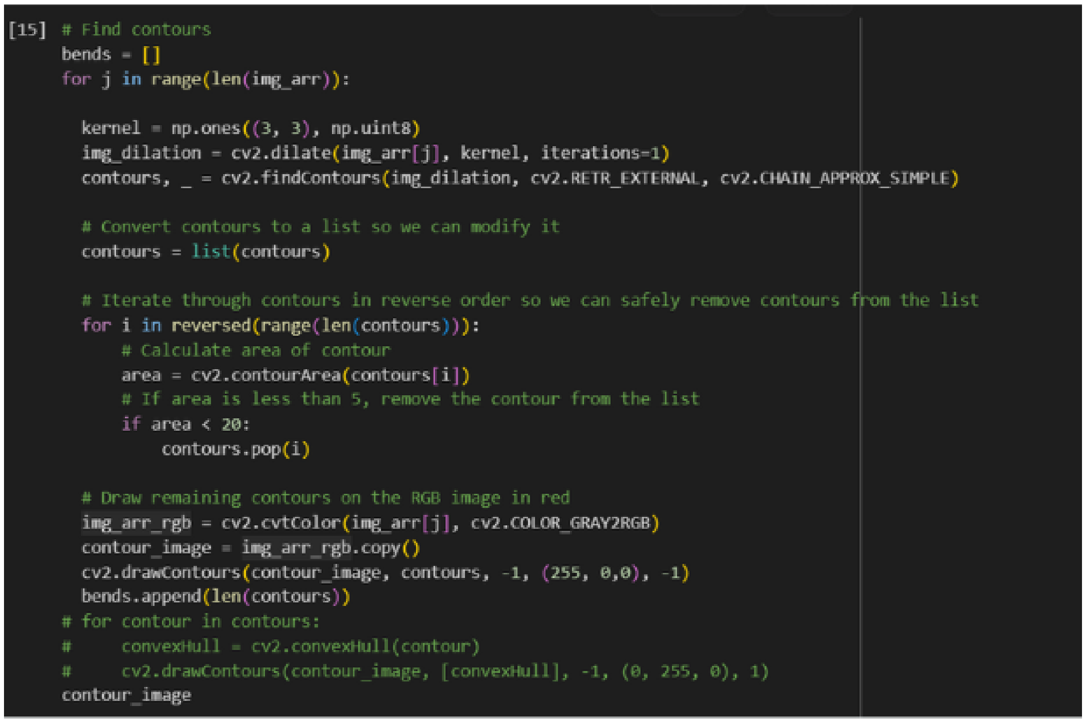


```python
# recursive function to remove connected pixels for a line in the patch, similar to the earlier patch function recurse
def recurse2(arr,x,y,dir,poss, lines = []):
  dirs = [[1,0],[-1,0],[0,1],[0,-1],[1,1],[-1,1],[-1,-1],[-1,1]]

  arr[x, y] = 0

  # to get line positions
  lines.append([x,y])
  try:
    dirs.remove(dir)
  except:
    pass

  for i in dirs:
    x_2 = x + i[0]
    y_2 = y + i[1]

    # extra condition to check if pixel in the patch positions provided by the recurse function
    if(x_2>= arr.shape[0] or y_2>= arr.shape[1] or [x_2,y_2] not in poss):
      continue

    # taking dir relative to the new pixel
    dir2 = [-i[0],-i[1]]

    # checking if pixel value is 1 (white)
    if (arr[x_2,y_2] == 1):

      # recursing again with the new pixel value
      arr, _ = recurse2(arr,x_2,y_2,dir2,poss.copy(),lines)

  return arr, lines
```

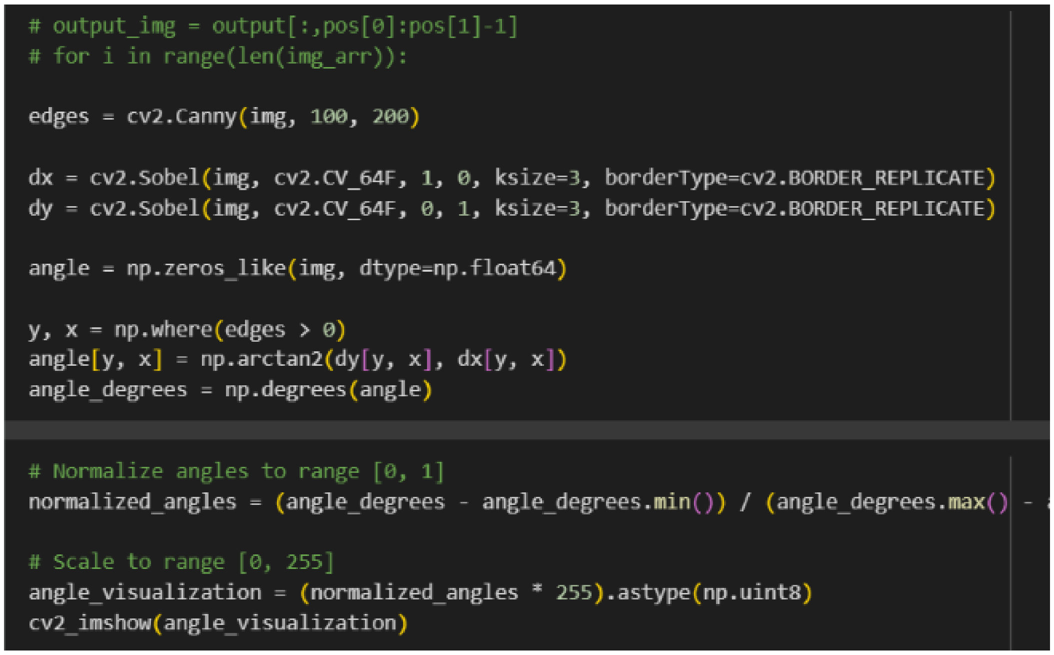


```python
# variables to store line and its centroid positions
lines = []
g_lines = []

# iterating through positions of the patch
for pos in posses:

  x = pos[0]
  y = pos[1]

  # checking if pixel value is 1 (white) thus encountering a line
  if arr2[x,y] == 1:

    # print(x,y)

    # calling recursive function to remove the connected white pixels of the currently detected line
    arr2, line = recurse2(arr2,x,y,[0,0],poss=posses.copy(), lines=[])
    # plt.imshow(arr2, cmap='gray')

    # adding line value to the list
    lines.append(line)

    count += 1

# making centroid array
g_lines = np.zeros((len(lines),2))
# calculating centroids of various line parts
for i,line in enumerate(lines):
  line = np.array(line)
  # print(f"{line}\n\n{line.shape}")
  g_line = [line[:,0].mean(), line[:,1].mean()]
  g_lines[i,:] = np.array(g_line)
# calculating angles
g_new = g_lines-g
mags = np.sum((g_new)**2,axis = 1)**0.5
angles = np.arccos(((np.matmul(g_new, (g_new).T)/mags).T/mags))/np.pi*180
# checking number of angles greater than 160
n = angles[angles>=145].shape[0]//2
# case for intersecting lines
if n >= 2:
  # print(angles)
  return -1
# in case of end points presents, making the count even
if count % 2 != 0:
  count += 1
# dividing the count by 2, since number of overlapping bends is half the number of seperate lines
count = count//2
# returning (count - 1) since 1 corner was already counted
count -= 1
# print("Overlap Count : ", count)
```

- Next is the Sobel Operator usage with normalisation of angle detecting non-sharp gradual terms as bends.
- Following that is the Image Processing used to calculate the number of bends detected and draw them on the image



```
[15] # Find contours
     bends = []
     for j in range(len(img_arr)):

       kernel = np.ones((3, 3), np.uint8)
       img_dilation = cv2.dilate(img_arr[j], kernel, iterations=1)
       contours, _ = cv2.findContours(img_dilation, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

       # Convert contours to a list so we can modify it
       contours = list(contours)

       # Iterate through contours in reverse order so we can safely remove contours from the list
       for i in reversed(range(len(contours))):
           # Calculate area of contour
           area = cv2.contourArea(contours[i])
           # If area is less than 5, remove the contour from the list
           if area < 20:
               contours.pop(i)

       # Draw remaining contours on the RGB image in red
       img_arr_rgb = cv2.cvtColor(img_arr[j], cv2.COLOR_GRAY2RGB)
       contour_image = img_arr_rgb.copy()
       cv2.drawContours(contour_image, contours, -1, (255, 0,0), -1)
       bends.append(len(contours))
     # for contour in contours:
     #     convexHull = cv2.convexHull(contour)
     #     cv2.drawContours(contour_image, [convexHull], -1, (0, 255, 0), 1)
     contour_image
```



```
# output_img = output[:,pos[0]:pos[1]-1]
# for i in range(len(img_arr)):

edges = cv2.Canny(img, 100, 200)

dx = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3, borderType=cv2.BORDER_REPLICATE)
dy = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=3, borderType=cv2.BORDER_REPLICATE)

angle = np.zeros_like(img, dtype=np.float64)

y, x = np.where(edges > 0)
angle[y, x] = np.arctan2(dy[y, x], dx[y, x])
angle_degrees = np.degrees(angle)
```

```
# Normalize angles to range [0, 1]
normalized_angles = (angle_degrees - angle_degrees.min()) / (angle_degrees.max() -

# Scale to range [0, 255]
angle_visualization = (normalized_angles * 255).astype(np.uint8)
cv2_imshow(angle_visualization)
```