TATA STEEL

Image Analytics of Construction Project BBS Diagram Rebar Shapes to extract data

Challenges faced by Tata Steel

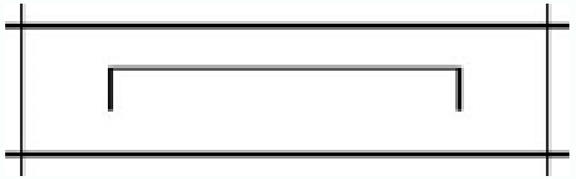
While handling numerous bbs documents, from clients :

- **Error Prone:** Counting the number of bends is error prone which can lead to inaccuracies in the data

- **Time Taking Process:** It is a time consuming process since BBS documents generally have more than 100 pages

- **Lack of count check:** Varied formulae for complex rebar shapes makes the counting of bends non-uniform across multiple users

Final Pipeline



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

Overview of the initial parts of the Pipeline

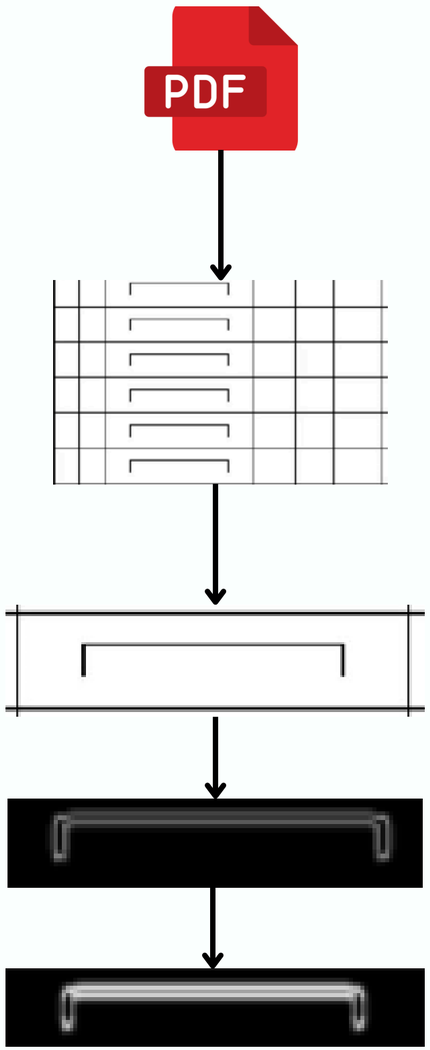


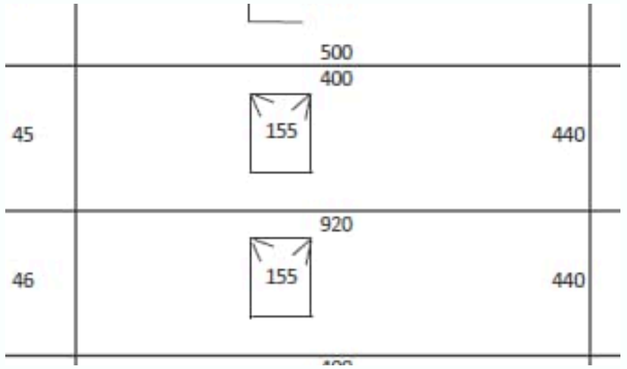
Image files are extracted from pdf, using standard python pdf module, then preprocessed (like blurring, inversion and dilation, for tackling issues with main img).

Diagrams are extracted after using threshold based column/row identification and serial wise iteration to get specified row/column.

Edge is detected using canny edge detection (Using OpenCV in python) and then resulting image is dilated.

Challenges faced

Row or column detection:



Many pdf files contains images, overheads, headers, footers which is irrelevant to us, but interferes with our code.

To counter this we initially approached manual reference based cropping methods, automatic cropping methods using ml models, etc.

Another issue was the presence of numbers between diagrams.

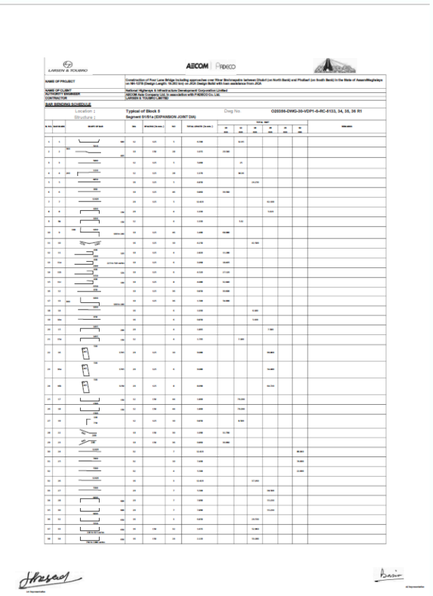
Techniques including tilt invariant, line invariant CNN models were trained.

PDF to Image Conversion

Turning the BBS file into Images for Processing:

- PyMuPDF for efficient conversion: We leverage the PyMuPDF library to effortlessly convert each page of the BBS PDF into a high-quality image.
- Sharpened Images: To ensure clarity, we increase the DPI (dots per inch) during conversion, resulting in sharper images.
- Preserving Details: A quality parameter is used to minimize JPEG compression artifacts, retaining important details from the original document.
- Flexible Manipulation: By converting the image to a NumPy array, we enable manual cropping if needed for further processing.
- BGR for Compatibility: The color space is converted from RGB to BGR to enhance compatibility with various image editing tools like OpenCV.
- Organized for Easy Use: Finally, the processed images are saved, and a dictionary is created for efficient access throughout the analysis.

Input PDF of a BBS file

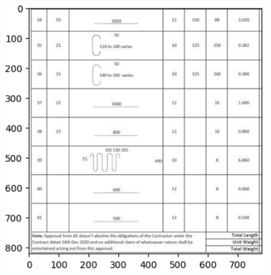


Editable image of each page of the PDF

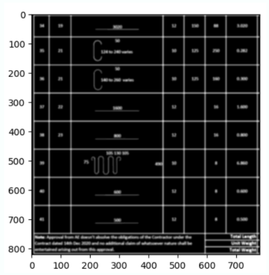
Diagram Extraction

Row or column detection:

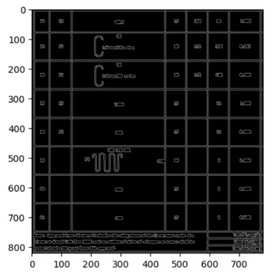
- Isolating Content: We begin by converting the PDF page image to grayscale, which simplifies processing.
- Noise Reduction: A Gaussian blur is applied to reduce background noise and enhance clarity for the next step.
- Edge Detection: The Canny edge detection algorithm is used to precisely identify the boundaries of important elements in the image, like table lines and diagrams.
- Smart Cropping: By analyzing the detected edges, we can automatically locate and extract the specific regions containing diagrams, eliminating unnecessary background.



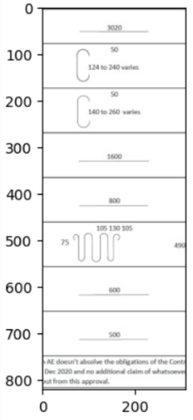
Grayscale image



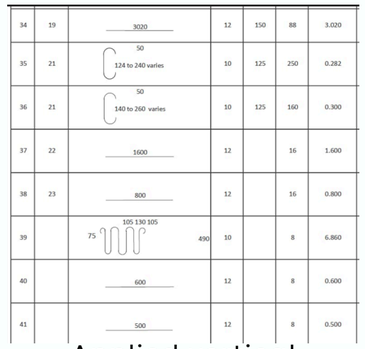
Gaussian blur applied on image and pixels inversed



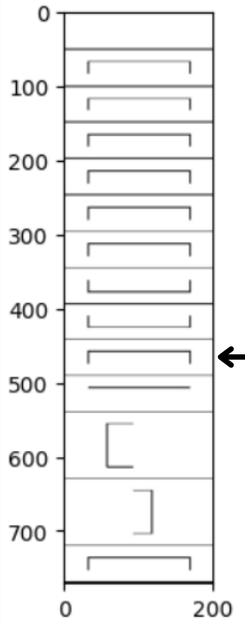
Canny operator applied and edges detected



Applied horizontal cropping between 2nd and 3rd column



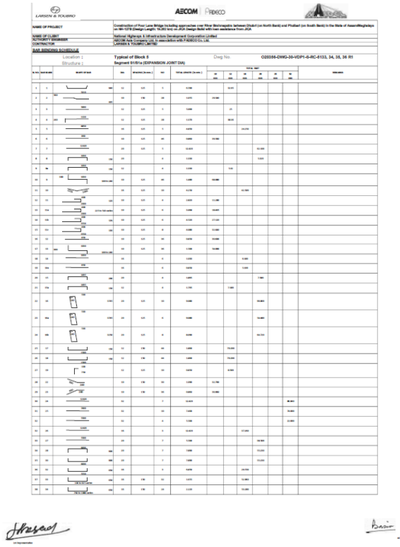
Applied vertical cropping to extract image between required rows



Final Image

Challenges in Automated Diagram Extraction

- Inconsistent Layout: BBS files lack a standardized format for diagram placement, making it difficult to write code that relies on specific locations. Even within a single PDF, diagrams can appear anywhere.
- Mimicking Random Lines: Unfortunately, diagrams often visually resemble other lines within the document, making them hard to automatically distinguish from tables.



Approaches Explored (These did not achieve consistent results):

- Serial Number Search: We attempted to identify diagrams by searching for a unique serial number, but this method proved unreliable due to potential variations in numbering schemes.
- Table Detection and Column Extraction: We explored locating tables and extracting the column with the most space between lines, assuming diagrams might reside there. However, this approach wasn't robust enough for all cases.
- Hough Transform with Distance Filtering: We experimented with using the Hough Transform to detect table lines and then filter them based on distance. However, this method required a precise threshold value and struggled with layout variations.

Code snippets

```
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
```

```
# Specify the path to the PDF file
pdf_file_path = "Image Processing in Python Algorithms, Tools, and Methods You Should Know.pdf"

# Set vertical crop margins (e.g., cropping 10% from the top and bottom)
vertical_crop_margins = (0.1, 0.5)

# Set horizontal crop margins (e.g., cropping 10% from the left and right)
horizontal_crop_margins = (0.1, 0.1)

# Specify the output image format (either 'jpg' or 'png')
output_image_format = 'png'
```

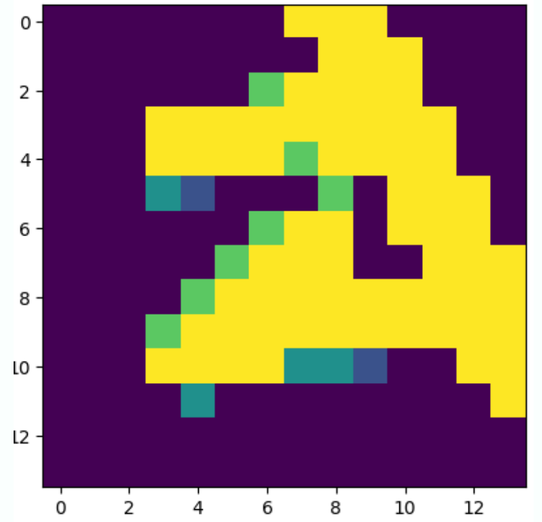
```
# Set the quality of the output image (0-100, higher is better)
output_image_quality = 100
```

```
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()
```

Number Removal

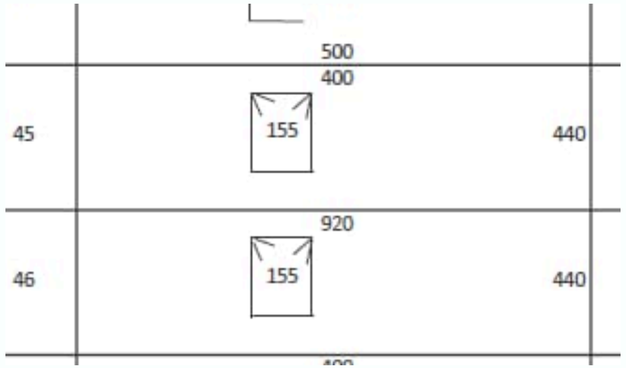
Number Removal:

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 recognise numbers and detect its presence.

Using the model we achieved ~96% accuracy on recognising numbers, ~100% accuracy on detecting numbers.



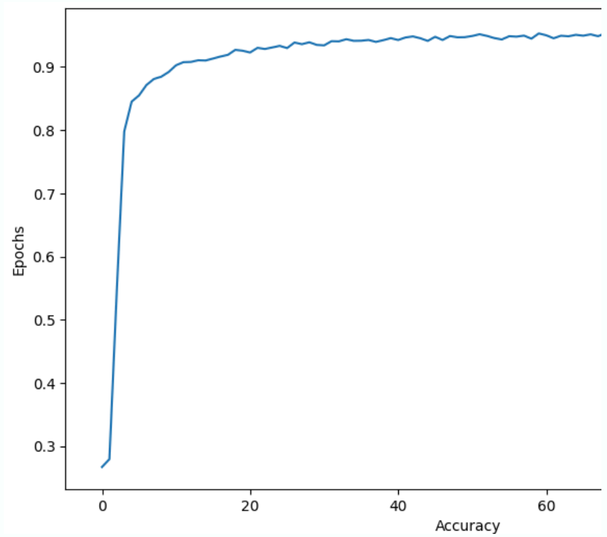
Number Removal

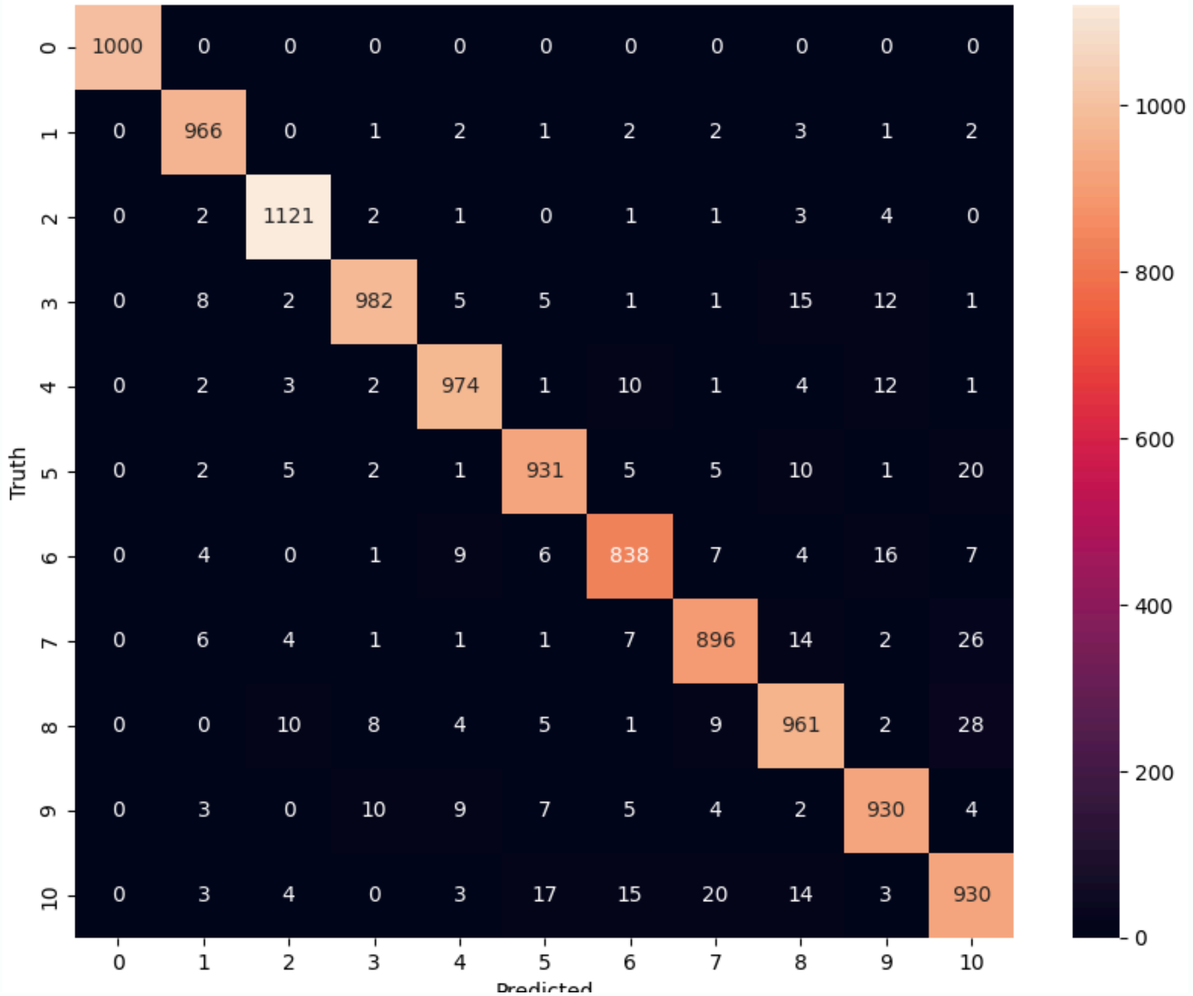
Number Removal:

The confusion matrix for the same.

100% acc for (0 detection - presence/absence of numbers)

Training loss graph



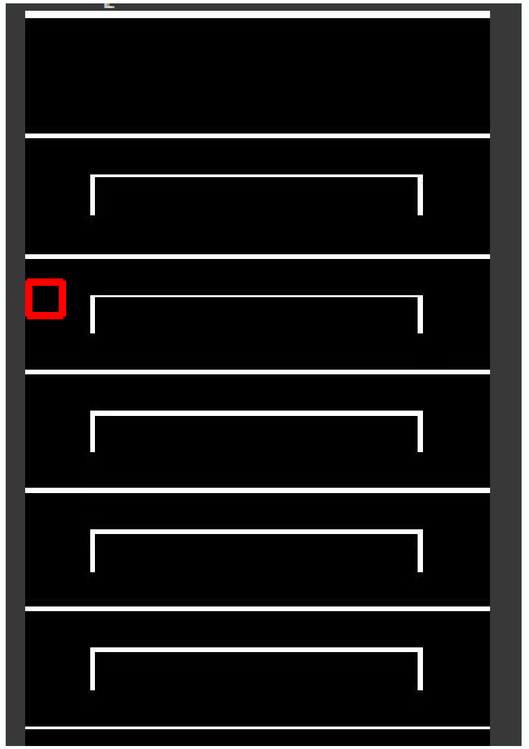


Number Removal

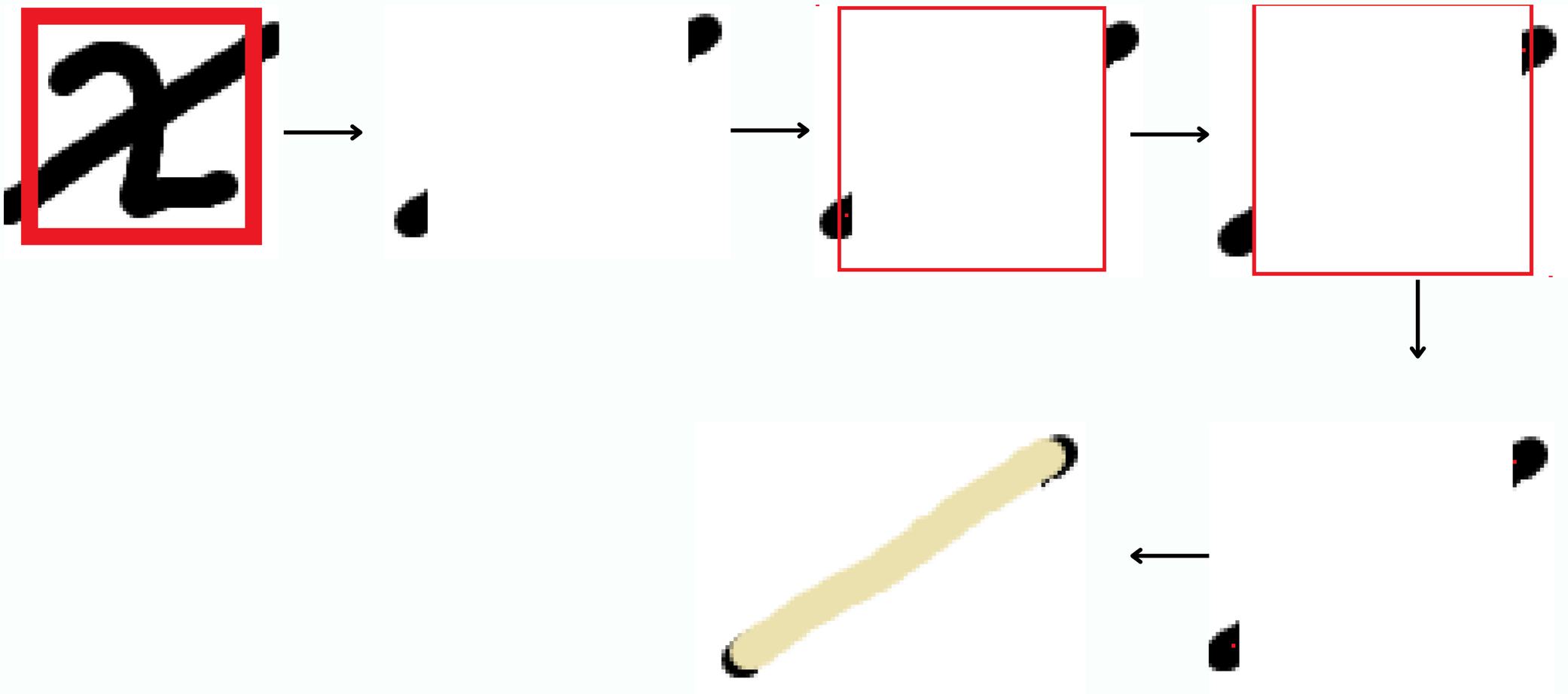
Finally the code was implemented using sliding window cnn to detect number patch, and cv2.line() was used to fill the line up. After the patch (number +line) was removed.

The centroid points of line just outside the patch was detected and line drawn.

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



Number Removal



Code snippets

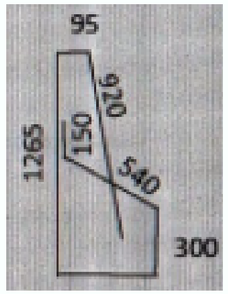
```python
def detect_rows(img):
  row_threshold = 10
  row_width_max = 10
  last_i = 0
  imgs = 0
  rows = []

  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
```

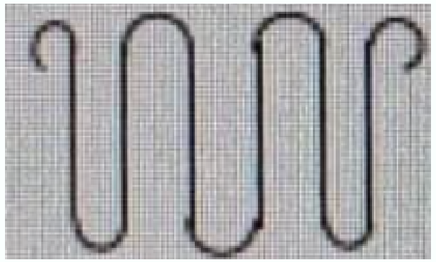
```python
# sliding window cnn function
def slide_cnn(model, arr, k_size, itr_size=None, stride_b=1, stride_h=1):

  # 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)
```

Code snippets

```
# 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()
```

Corner detection and bend counting

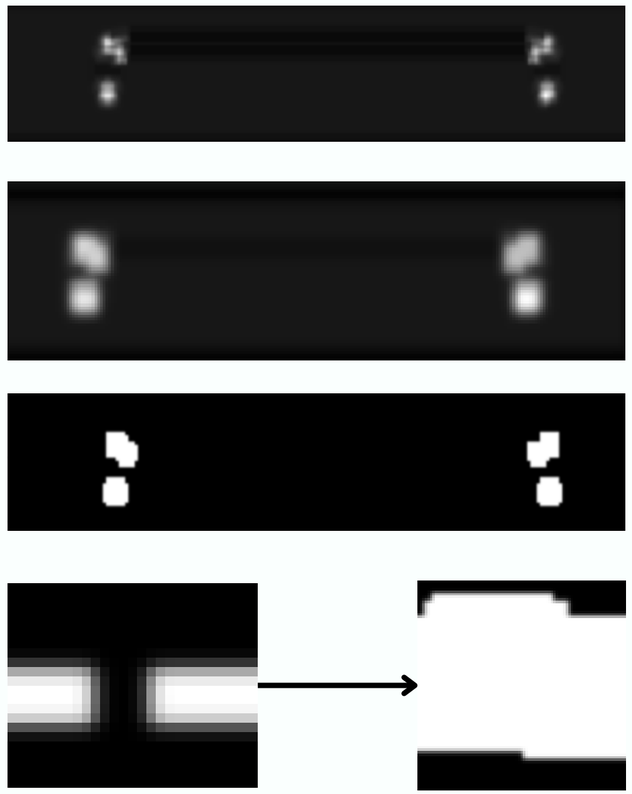




Corners are detected using Shi-Tomasi method

There are certain edge cases remaining, for overlapping, intersecting, circular and spiral bends which are countered using various different approaches.

Initial approach to corner detection



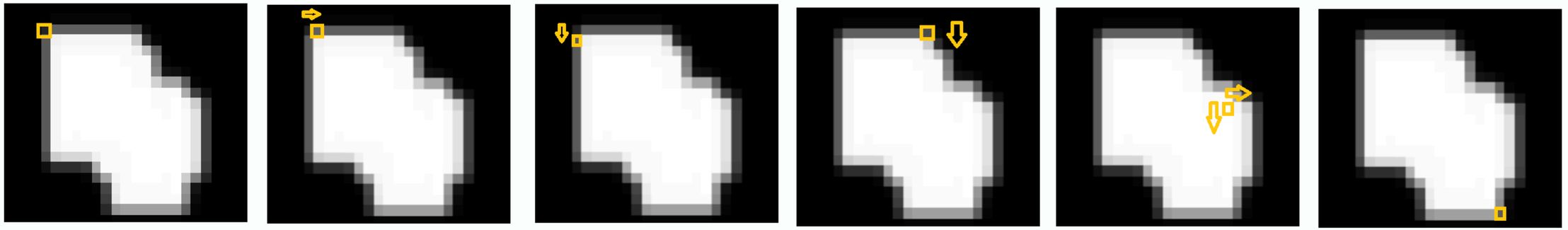
Corners were initially detected using Harrison corner detection method. The Corner Harrison detection method returns a patch based on the probability of a pixel being a corner.

Challenges faced: Multiple different corners due to edges both sides. (Dilated image to tackle it). Further thresholded it to make it binary (for easy calculations further on).

Further challenge of nearby endpoints merging is faced.

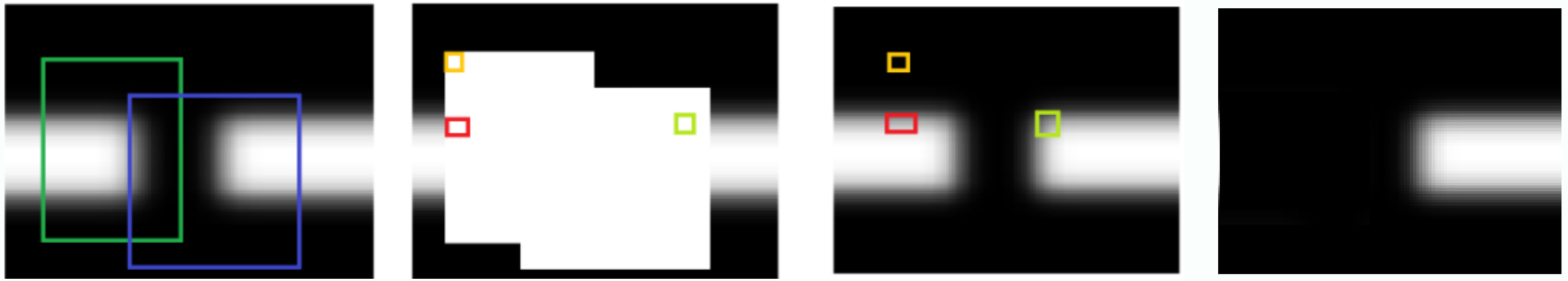
Initial Challenges

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 instance of a line in the patch then that confirms that corners were merged.



Tackling Challenges

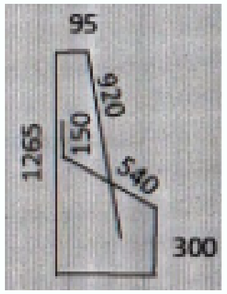
Multiple instances are detected by removing the connected line pixels (white pixels) as soon as one is detected, and recursing further. If more line pixels are detected, that means multiple line instances in a patch.



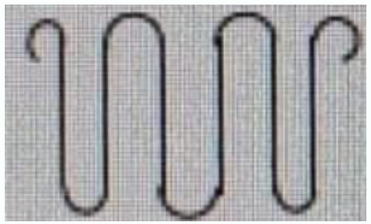
The green and blue boxes merge to give the white patch (2nd), Yellow is when the first algorithm recurses on it. Red box is when the first line is detected (shown in 3), thus removing its connected pixels (4). Green box detects the second instance of the line.

Final Method

Corners are detected using 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.



Code Snippets

```
# 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)
```

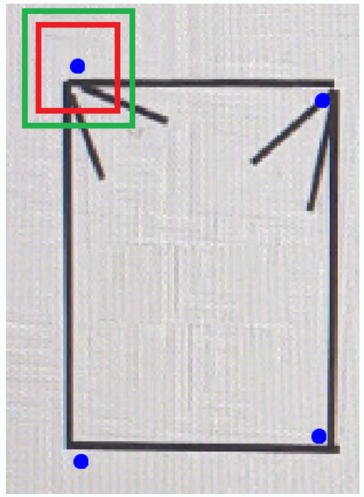
```
# Using recursive function to remove white pixels when a cluster is detected
def recurse(arr,x,y,dir,poss):

  # 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()
```

Code Snippets

```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
```

Overlapping Bends

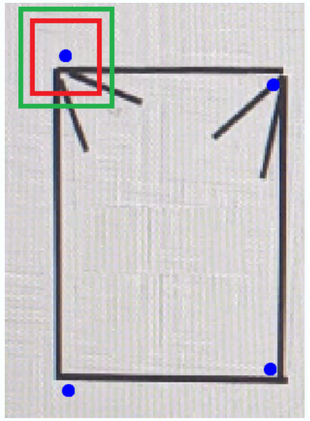
Overlapping bends:
Overlapping bends are those, which are actually two or more bends behind each other, but appear to be the same in a 2d space.



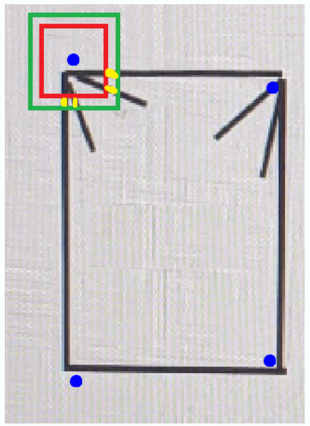
To tackle this we devised a technique we name as “perimeter pixel-cluster identification” (PPCI).

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 to it inside the box, thus getting each to cluster seperately once.

Tackling challenges

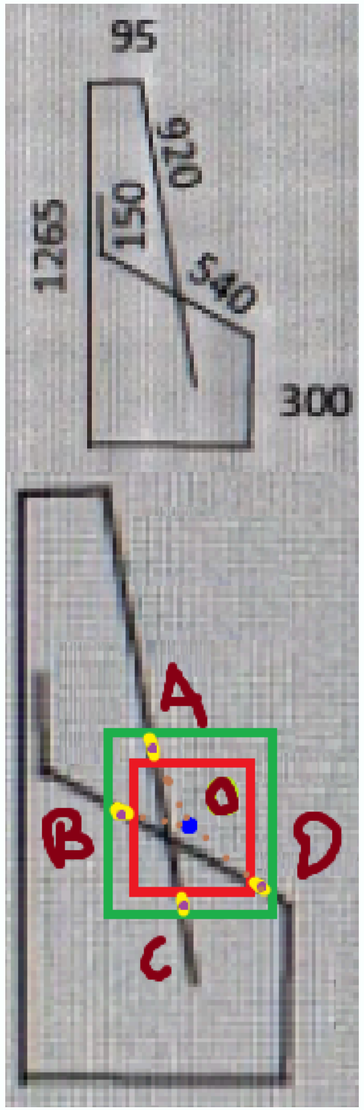


Now we have a number of pixel clusters. To identify a number of overlapping bends, we divide the number of pixel clusters by two. One bend is formed by two lines and thus two clusters, so clusters/two gives us number of corners.



Here we got 4 lines or 4 clusters (yellow) for top left corner, thus implying two bends on top of each other.

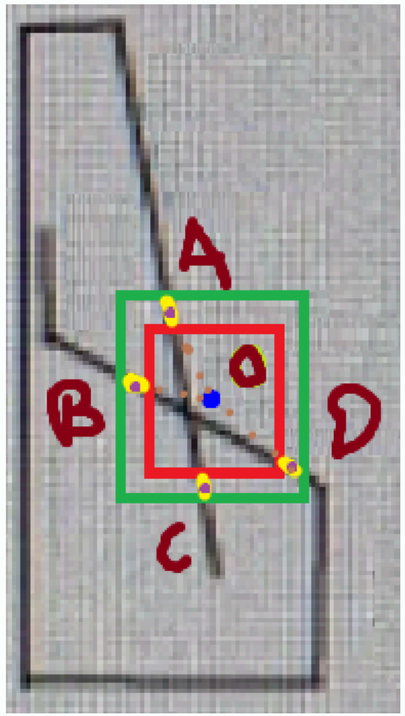
Intersecting bends



Intersecting bends:
Intersecting bends are solved by a similar technique, called as “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 cluster.

Tackling challenges

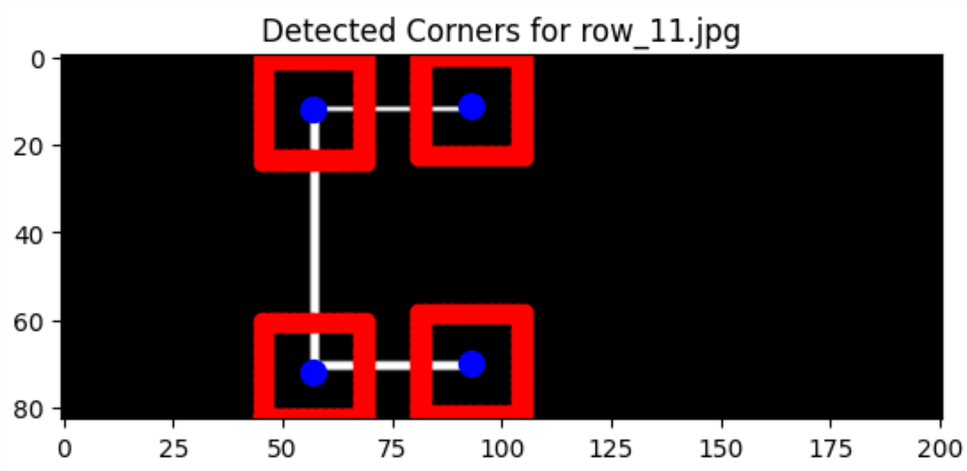


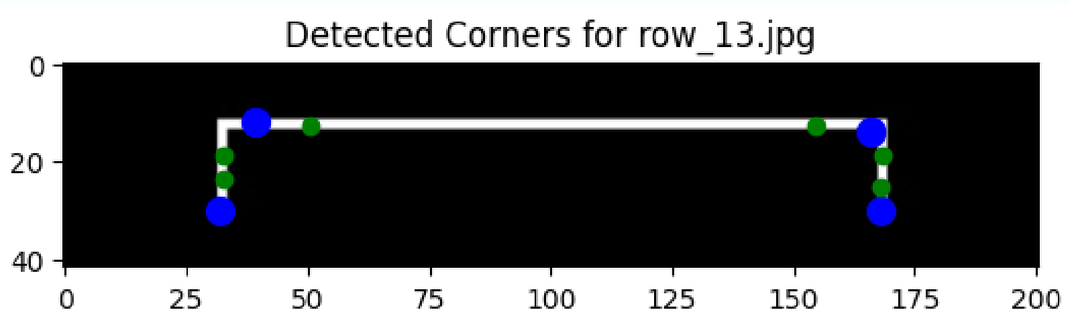
After finding the centroid we find the angles, between the corner point and centroid of cluster.

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.

Outputs and code snippets

```
# 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 ha
    # If we hit white pixels multiple times, it means there are multiple corners
    # Multiple corners lines inside the patch is actually connected outside but
    # it must mean that one corner line did one or more corners outside that pat
    # it means that there corner regions got merged but lines are seperate (not
    if arr2[x,y] == 1:

      # calling recursive function to remove the connected white pixels of the c
      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)
```





Outputs and code snippets

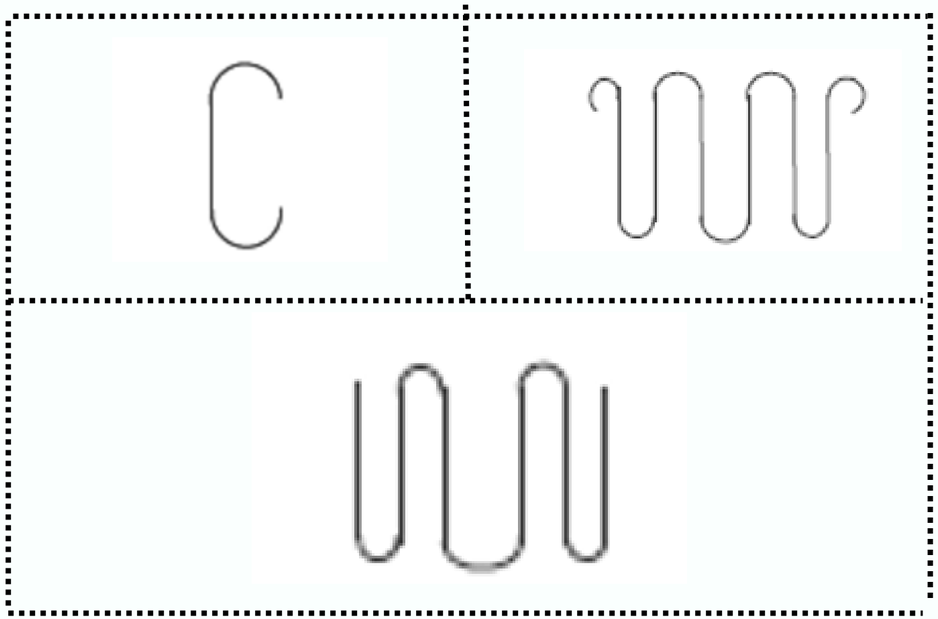
```python
# Function to check overlapping
def check_overlap(arr, g):

  g_r = [round(g[0]), round(g[1])]

  posses = []
  count = 0

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

  # setting default positions as 0 if centroid - checking region lower than 0
  pos1 = 0
  pos2 = 0

  # checking if top left corner beyond (0,0), else assigning pos1,pos2 its values
  if g_r[0] - 12 > 0 :
    pos1 = g_r[0] - 12

  if g_r[1] - 12 > 0 :
    pos2 = g_r[1] - 12

  # extracting region for checking multiple line instances
  # g_r[0] - 12 and g_r[1] - 12 checks for pos1, pos2 lying beyond 0 and adjusts accordingly
  for i in range(25 - (pos1 - (g_r[0] - 12))):
    for j in range(25- (pos2 - (g_r[1] - 12))):

      # for i values in range of initial 2 and final 2 rows, thus for all columns
      if (i < 2 - (pos1 - (g_r[0] - 12)) or i > 22 - (pos1 - (g_r[0] - 12))):

        # checking for both boundary exceed condition
        if pos1 + i > arr.shape[0] - 1  and pos2 + j > arr.shape[1] - 1 :
          break

        # checking for single boundary exceed conditions
        if pos1 + i > arr.shape[0] - 1 :
          posses.append([arr.shape[0] - 1, pos2 + j])
          continue

        if pos2 + j > arr.shape[1] - 1 :
          posses.append([pos1 + i, arr.shape[1] - 1])
          continue

        posses.append([pos1 + i, pos2 + j])

      # for i values in range between 2 and 11 thus in the limited column variation region
      elif (i >= 2 - (pos1 - (g_r[0] - 12)) and i <= 22 - (pos1 - (g_r[0] - 12))) and (j < 2 - (pos2 - (g_r[1] - 12))  or j > 22 - (pos2 - (g_r[1] - 12))):

        # checking for both boundary exceed condition
        if pos1 + i > arr.shape[0] - 1  and pos2 + j > arr.shape[1] - 1 :
          break
```

```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 ins
    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)
return count
```

Outputs and code snippets

```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
```

Circular Bend Counting



Examples of Circular Bends

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 (Code Snippet provided next slide)

We have applied angle normalisation using the Sobel Operator which gives us the following output -

Code Snippet

```
# 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() - a

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

Sobel Operator used with normalisation of angle detecting non-sharp gradual terms as bends

```
[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
```

Image Processing to calculate the number of bends detected and draw them on the image

Circular Bend Count Pipeline

No. of Bends - 6

No. of Bends - 2