CHILD'S PLAY

ABSTRACT

Visual inspection is the image-based inspection of parts where a camera scans the part under test for both failures and quality defects. Automated inspection and defect detection are critical for high-throughput quality control in production systems. Visual inspection systems with high-resolution cameras efficiently detect microscale or even nanoscale defects that are difficult for human eyes to pick up. Hence, they are widely adopted in many industries for detection of flaws on manufactured surfaces such as metallic rails, semiconductor wafers, and contact lenses.
In this case I built visual inspection for teddy bears and poker cards.

Ofir Herrera
HIT- Computer Science

Table of Contents

Introduction

Visual inspection is the image-based inspection of parts where a camera scans the part under test for both failures and quality defects. Automated inspection and defect detection are critical for high-throughput quality control in production systems. Visual inspection systems with high-resolution cameras efficiently detect microscale or even nanoscale defects that are difficult for human eyes to pick up. Hence, they are widely adopted in many industries for detection of flaws on manufactured surfaces such as metallic rails, semiconductor wafers, and contact lenses.

I took card and teddy bear objects because I like to play poker and because I like teddy bears.
In addition, the geometric shape of a card is quite simple, and the geometric shape of a teddy bear is more complex. Therefore, I was interested to see the complexity of the test for each one of the objects.

I built up a story for my project:

The story behind the project built because of the need of a certain factory, which produces poker cards and teddy bears, use the Visual inspection system called "Child's play" that I built.
"Child's play" in English refers to something that is simple to do . I chose this name for my inspection system because I wanted to show that detect defects is easy to do and because teddy bears are meant to be child's game in contrast for Poker cards.

Key words:

Image processing, computer vision, Visual inspection system, Registration, Homography.

Project Description

This project describes a visual inspection system of a factory produces poker cards and teddy bears.

"Child's game" system will help to identify defective cards and teddy bears.

Motivation

Identify defects in original cards or teddy bear images created by the factory machine according to a template image in order to create fair and clear items.

Input

The Photos were taken from my iPhone camera (the sensor of the factory).
The images were taken from the same angle to simulate a sensor camera.

Target

Analyze which of all images are damaged.

Pipeline

1) Load photos

2) Create images with variance as part of the dataset.

3) Data pre- processing.

4) Registration.

5) Check if the image aligned to the template.

Step 1 - Load photos

Tools

Programming Language: Python.

IDE: Jupyter Notebook.

Libraries: OpenCV, NumPy, matplotlib.

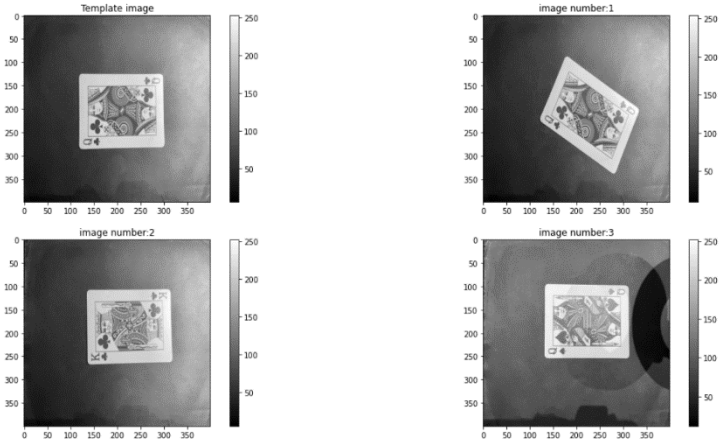
Description

The datasets contains pictures of Poker cards and teddy bears taken through a sensor- the camera of my iPhone.

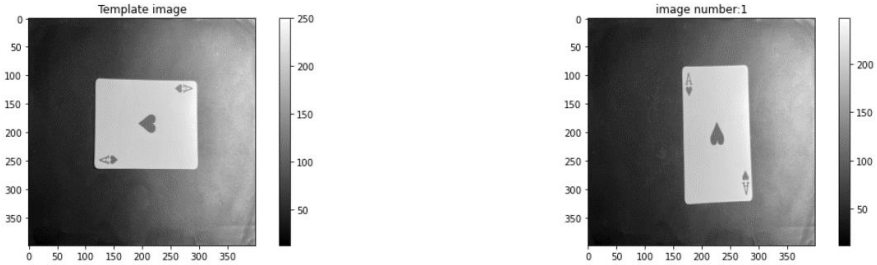
The photos were taken against a black background except one image, this is a way of simulating photographing action on production line of factory.

There are 3 separate datasets:

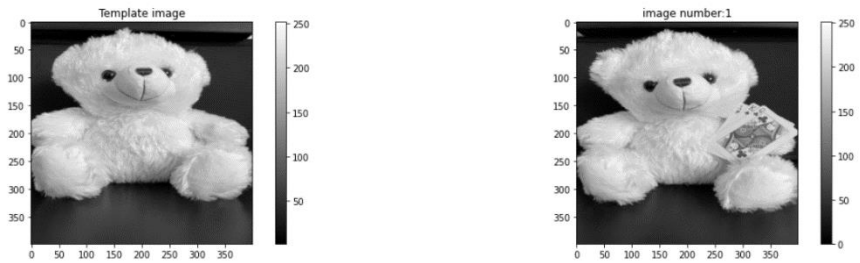
Dataset number 1:



Dataset number 2:



Dataset number 3:



I used two functions that I build: loadRescaleImages() and plotImages().

loadRescaleImages():

List is passed as a parameter, the function returns list of images. Each image is loaded and transferred to gray scale.

```python
# Load the images

# Images files of 3 datasets

file_names1=["template_queen.JPG","noise_queen.JPG","king.JPG","queen2.JPG"]
file_names2=["As_template.JPG","as3.JPG","as2.JPG"]
file_names3=["mishka_template.JPG","mishka4.JPG"]


def loadRescaleImages(file_names):

    """
    List is passed as a parameter, the function returns list of images.
    Each image is loaded and transferrd to gray scale.
    Note: The first image is the template image.
    """
    # List contains the images
    instances = []

    # Load all images and keep it in 'instances', convert them to grayscale, and blur them slightly
    for file_name in file_names:
        img = cv2.imread(file_name)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        instances.append(img)


    # Scaling the images to the same size
    for i in range(len(instances)):
        instances[i] = cv2.resize(instances[i],(400,400), interpolation = cv2.INTER_AREA)
    return(instances)
```

Figure number 1

Note: The first image is the template image.

plotImages():

Plot the images.

```
32 def plotImages(instances, row, column):
33
34     """
35     List of images, row and column passed as parameters, the function plots images.
36     row and column represents the location of ploting each image.
37     """
38     # Plot the images
39     w = 10
40     h = 10
41     fig = plt.figure(figsize=(20, 10))
42     columns = column
43     rows = row
44     # ax enables access to manipulate each of subplots
45     ax = []
46     for i in range(columns*rows):
47         # Create subplot and append to ax
48             ax.append(fig.add_subplot(rows, columns, i+1))
49             plt.imshow(instances[i],cmap = 'gray')
50             plt.colorbar()
51             if (i==0):
52                 ax[-1].set_title("Template image")
53             else:
54                 ax[-1].set_title("image number:"+str(i))  # Set title
55     plt.show()  # Finally, render the plot
```

Figure number 2

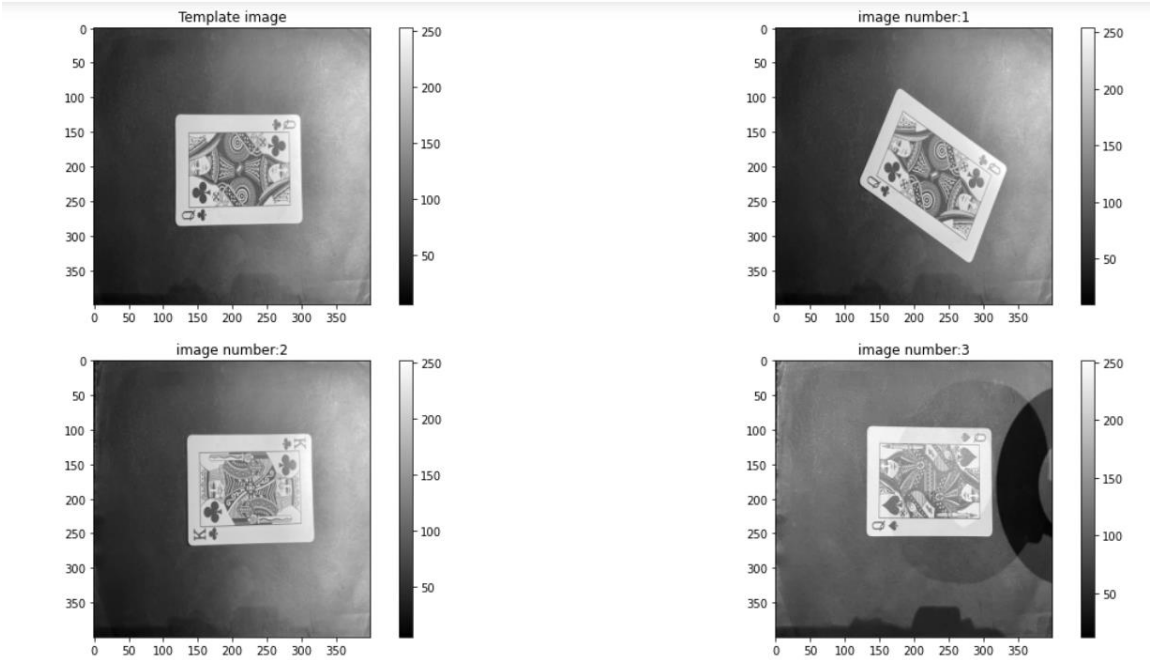


Figure number 3:Dataset number 1

Step 2 - Create images with variance as part of the dataset

Tools

Programming Language: Python.

IDE: Jupyter Notebook.

Libraries: OpenCV, NumPy, matplotlib, Imutils, Random.

Description

Create a list called " variance_instances" containing all variance images.

Create several functions to make high variance images.

These functions can help us simulate a real situation where there is a lot of noise generated from the environment or the sensor.

Next, I would like to check how the algorithms deal with noise.

Functions:

1) Create rotated images.

2) Create defects over the template image.

3) Create noisy(Salt and pepper noise) images.

rotateImage():

The steps required to rotate image are the direction (clockwise or counterclockwise around the center or any other point) and angle(between -360 to 360). I used imutlis.rotate_bound() function. This Auxiliary function instead of rotation function to ensure that all parts of the image are in the rotated image.

```
def rotateImage(image):

    """
    Image is passed as a parameter and retured as rotated image.
    """
    # Make a copy image called temp
    temp = image.copy()
    # Pick a random angle
    angle = random.randint(-360,360)
    # Rotate temp according to the angle
    rotated = imutils.rotate_bound(temp, angle)
    return(rotated)
```

Figure number 4

createBlackSquare():

The purpose of this function is intentionally create a defect in the image.

Note: On dataset number 1 I created white square. It is the same function but, I only changed the value to '255' instead of '0' in line 23

```
14 def createBlackSquare(image):
15     """
16     Image is passed as a parameter and returned with a black square on  it at a specific locatio
17     """
18     # Get the shape of the image
19     rows,cols = image.shape
20     # Locate few pixels in the photo and change them to 0
21     # The range of the pixels that changed to 0
22     # In Dataset number 3 I changed the value to 255 instead of 0 to make the square white
23     image[int(rows/2)-30:int(rows/2)+30,int(cols/2)-30:int(cols/2)+30]=0
24     return(image)
```

Figure number 5

createSaltAndPepperNoise():

Salt-and-pepper noise definition explained in part 3.

```
def createSaltAndPepperNoise(image):
    """
    Image is passed as a parameter and returned with Salt&Pepper noise.
    """
    # Getting the dimensions of the image
    rows , cols = image.shape

    # Randomly pick some pixels in the
    # image for coloring them white
    # Pick a random number between 300 and 10000
    number_of_pixels = random.randint(300, 10000)

    for i in range(number_of_pixels):

        # Pick a random y coordinate
        y_coord=random.randint(0, rows - 1)

        # Pick a random x coordinate
        x_coord=random.randint(0, cols - 1)

        # Color that pixel to white
        image[y_coord][x_coord] = 255

    # Randomly pick some pixels in
    # the image for coloring them black
    # Pick a random number between 300 and 10000
    number_of_pixels = random.randint(300 , 10000)

    for i in range(number_of_pixels):

        # Pick a random y coordinate
        y_coord=random.randint(0, rows - 1)

        # Pick a random x coordinate
        x_coord=random.randint(0, cols - 1)

        # Color that pixel to black
        image[y_coord][x_coord] = 0
    return image
```

Figure number 6

At this point I created new data and append each image to the ' variance_instances' list.

```
# Create a list contains the original images
variance_instances = [template_img.copy(),instances[1].copy() ]

# Create variance images
img1 = rotateImage(template_img.copy())
img2 = createSaltAndPepperNoise(template_img.copy())
img3 = createWhiteSquare(instances[2].copy())
img4 = instances[3].copy()

# Append the images to the list
variance_instances.append(img1)
variance_instances.append(img2)
variance_instances.append(img3)
variance_instances.append(img4)
```

Figure number 7: List of variance images made from dataset number 1

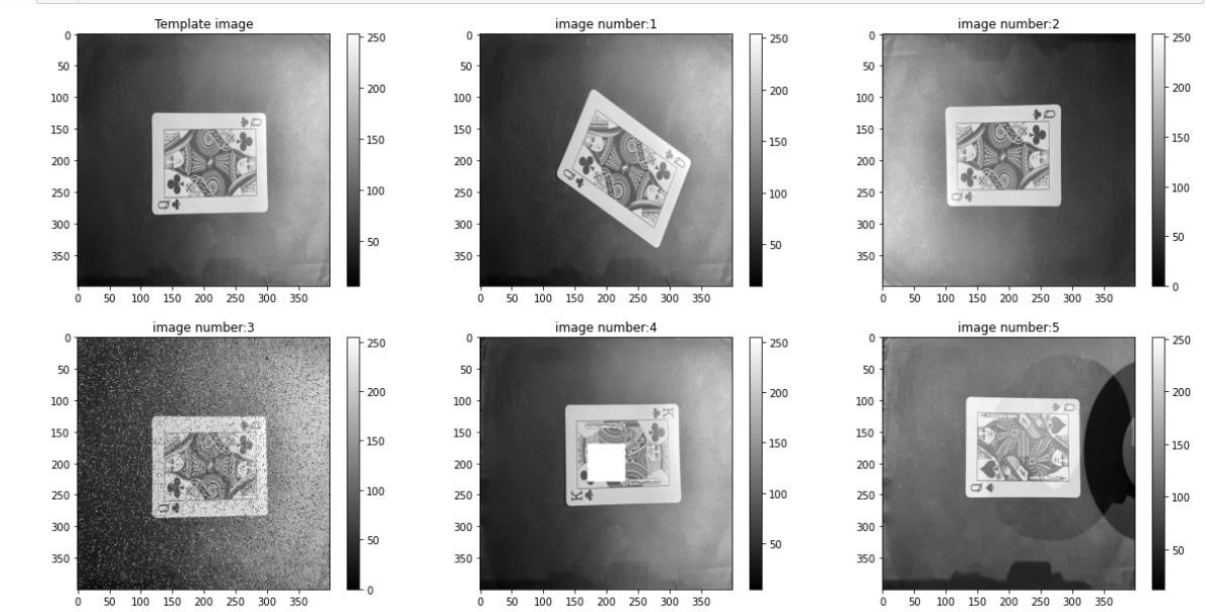


Figure number 8: plot variance images of cards

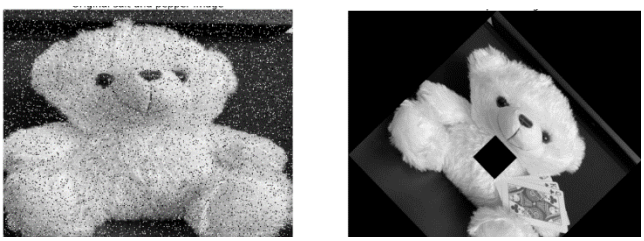


Figure number 9: plot variance images of teddy bears

Step 3 – Data pre- processing

Tools

Programming Language: Python.

IDE: Jupyter Notebook.

Libraries: OpenCV, matplotlib.

Description

Pre- processing images includes:

- Converting images to gray scale.
- Rescaling images (resize to 400X400).
- Analyze an image using Histogram.
- Use median filter to reduce salt and pepper noise.

Note: Part of the process has already been done (Rescaling images, Converting images to grey scale).

Analyze an image using Histogram

Histogram is considered as a graph or plot which is related to frequency of pixels in a gray scale Image with pixel values (ranging from 0 to 255).
Grayscale image is an image in which the value of each pixel is a single sample, that is, it carries only intensity information where pixel value varies from 0 to 255. Images of this sort, also known as black-and-white, are composed exclusively of shades of gray, varying from black at the weakest intensity to white at the strongest where Pixel can be considered as a every point in an image.
we get intuition about contrast, brightness, intensity distribution etc. of that image. Histogram, which is drawn for grayscale image, not color image. Left region of histogram shows the amount of darker pixels in image and right region shows the amount of brighter pixels.

```
# Create figure
plt.figure(figsize=(10,5))

# Create histogram for template image
hist_template = template_img.copy()
plt.hist(hist_template.flatten(),255,[1,256])
plt.grid()
plt.title("Template_image_histogram")

```

```
Text(0.5, 1.0, 'Template_image_histogram')
```

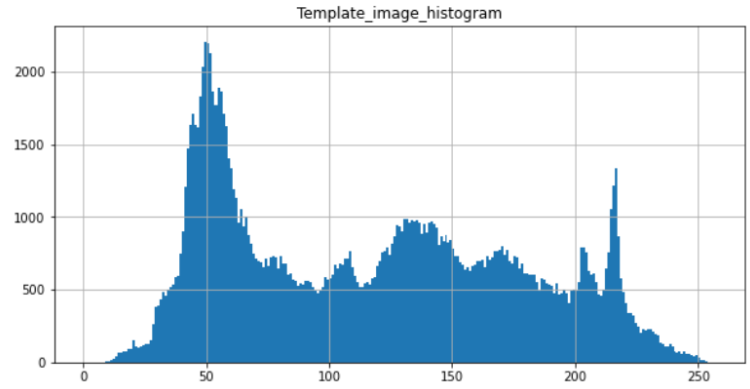


Figure number 10: histogram of "template" of dataset number 1

In this histogram ("Template_image_histogram"), it can be seen that there is high amount of pixels more prone to dark But still there is a variety of the pixels range. Also, most pixels do not tend to the edges (0 or 255).

```
# Salt and pepper image analysis
fig = plt.figure(figsize=(40,30))

# Plot image of 'Salt and Pepper' noise
fig.add_subplot(2, 2, 1)
plt.imshow(variance_instances[3].copy())
plt.title("Original salt and pepper image" ,fontsize=25)
plt.axis('off')

# Plot hitogram of 'Salt and Pepper' noise
fig.add_subplot(2, 2, 2)
plt.hist((variance_instances[3].copy()).flatten(),255,[1,256])
plt.title("Original salt and pepper image histogram" ,fontsize=25)
plt.grid()


# Plot hitogram of correction to'Salt and Pepper' noise using median filtering
fig.add_subplot(2, 2, 3)
median_filter_img = cv2.medianBlur(variance_instances[3].copy(), 3)
plt.hist(median_filter_img.flatten(),255,[1,256])
plt.title("Median blur filtering histogram",  fontsize=25)
plt.grid()

# Plot the 2 images before and after filtering
fig.add_subplot(2, 2, 4)
compare = np.concatenate((median,variance_instances[3].copy()), axis=1) #side by side comparison
plt.imshow(compare)
plt.title("Left: Median filtering. Right: Original image.",  fontsize=25)
plt.axis('off')
```

Figure number 11: histogram of the "salt and pepper" template image dataset 1

Now, observe "salt and pepper" image but first, I will explain what is "salt and pepper noise" and what is "median filtering":

Salt and Pepper noise is A type of noise commonly seen in photographs is salt and pepper noise. It manifests as white and black pixels that appear at random intervals. Errors in data transfer cause this form of noise to appear. The values a and b in salt pepper noise are different. Each has a probability of less than 0.1 on average. The corrupted pixels are alternately set to the minimum and highest value, giving the image a “salt and pepper” appearance.
The use of a median filter, is an effective noise eradication strategy for this type of noise.

The median filter is a non-linear digital filtering technique, often used to remove noise from an image. Such noise reduction is a typical pre-processing step to improve the results of later processing (for example, edge detection on an image). Median filtering is very widely used in digital image processing because, under certain conditions, it preserves edges while removing noise.

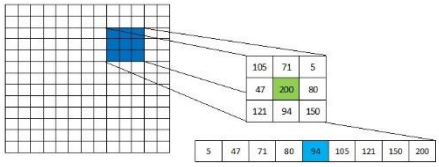


Figure number 12 Example to how median filtering works

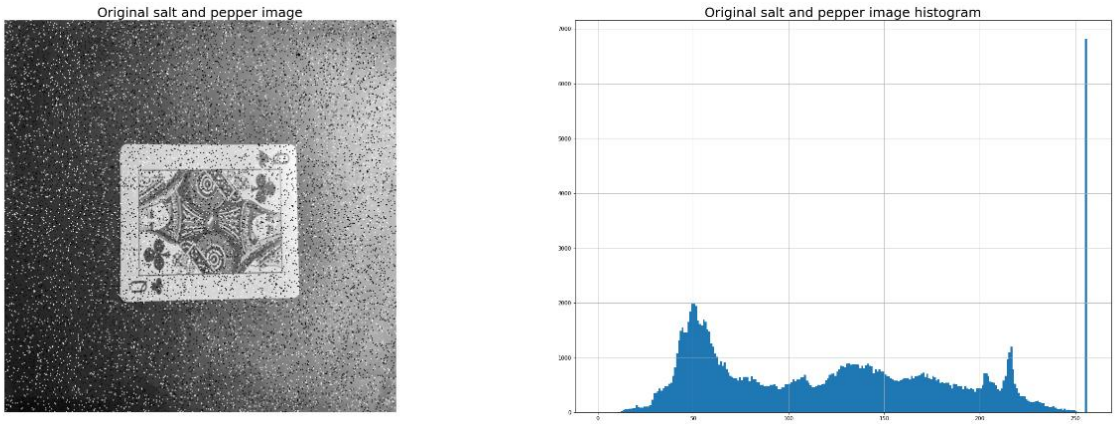


Figure number 13: histogram of the "salt and pepper" template image dataset 1

In this histogram ("Template_image_histogram"), it can be seen that there is high amount of pixels more prone to bright, the right region of the histogram shows that there is high amount of brighter pixels specially 255.

As can be seen from figure 11, line 17 and further, I used the median filtering function(line 19) with kernel (filter) size of 3X3.

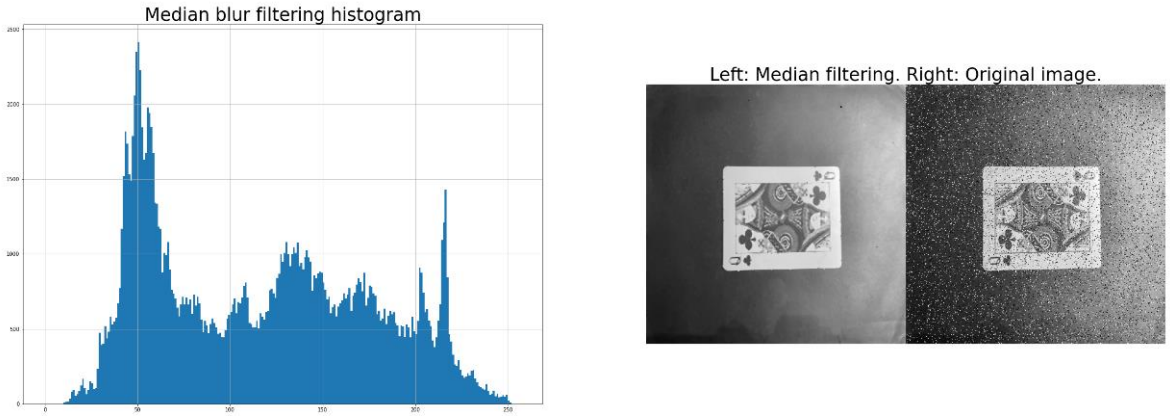


Figure number 14: histogram of performing median filtering over "salt and pepper" template image dataset 1

It can be seen that the filtered image is "smoother" than the original one.

```
1  # White Square image analysis
2  fig = plt.figure(figsize=(40,30))
3
4  # Plot image of White Square
5  fig.add_subplot(2, 2, 1)
6  plt.imshow(variance_instances[5].copy())
7  plt.title("White Square image",  fontsize=25)
8  plt.axis('off')
9
10 # Plot hitogram of 'Salt and Pepper' noise
11 fig.add_subplot(2, 2, 2)
12 plt.hist((variance_instances[5].copy()).flatten(),255,[1,256])
13 plt.title("White Square image histogram",  fontsize=25)
14 plt.grid()
```

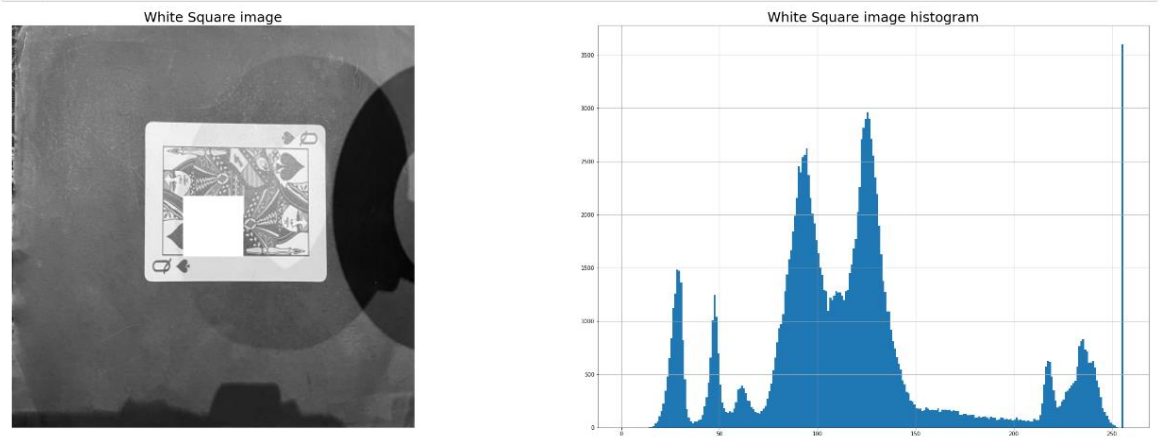


Figure number 15: histogram of image with white square as flaw image dataset 1

It can be seen that the right region of the histogram shows high amount of brighter pixels. The amount of the brightest pixels is very high In contrast to the "template_img" which make sense because the function createWhiteSquare() converted the pixels to 255.

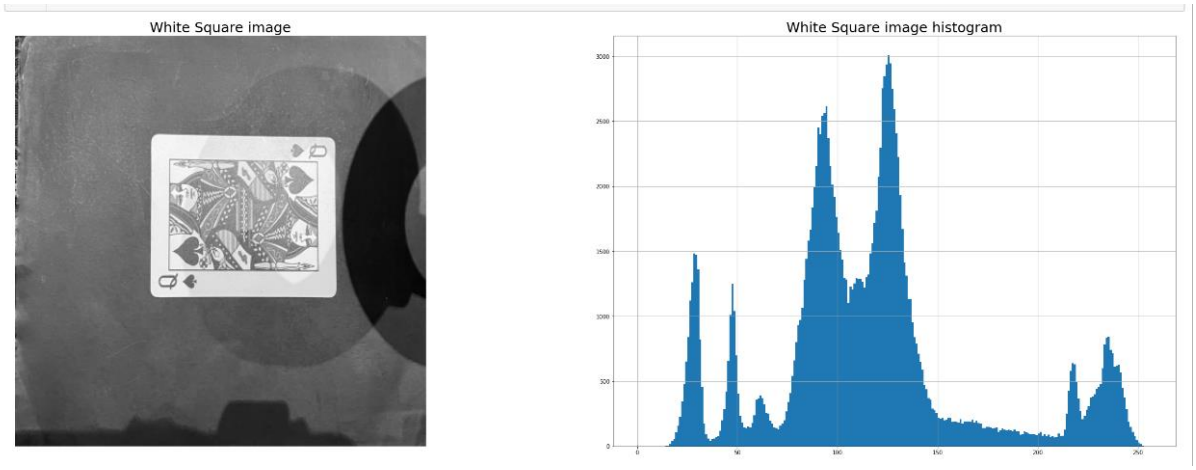


Figure number 16

The represented in figure 16, taken with different lighting. There are many pixels between dark and bright.

Demonstrate the histograms for the dataset3:

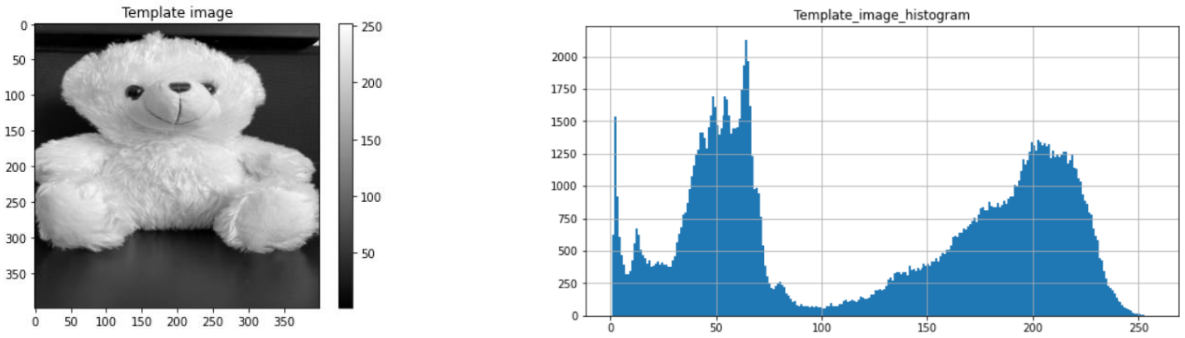


Figure number 17: histogram of "template" of dataset number 3

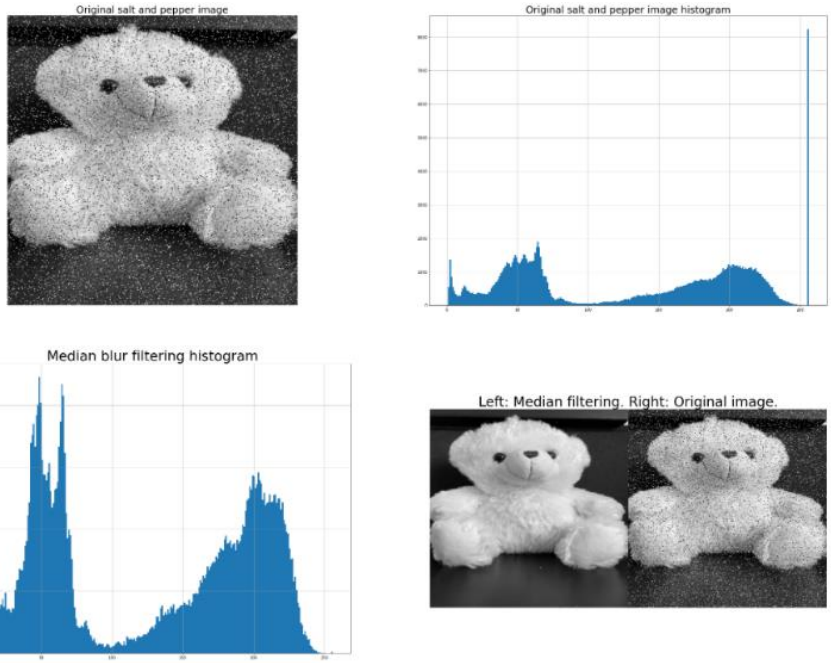


Figure number 18: histograms of "salt and pepper" template image dataset 3 and median filtering over "salt and pepper" template image dataset 3

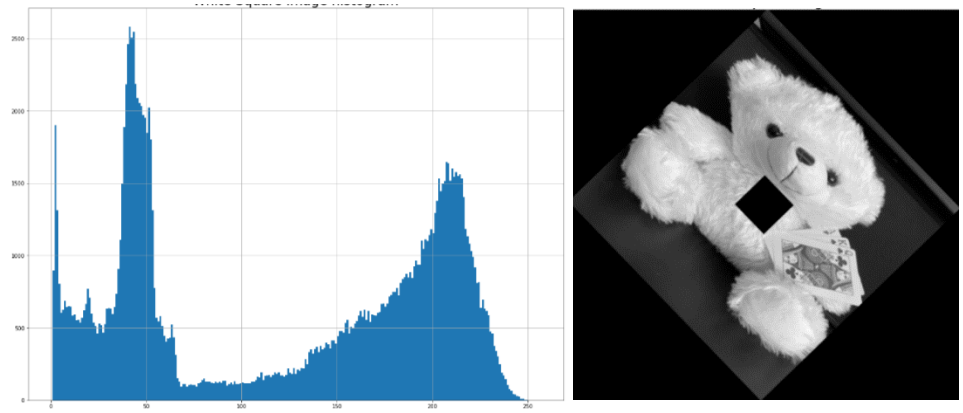


Figure number 19: histogram of template image both rotated and with black square as flaw dataset 3

Step 4 - Registration

Tools

Programming Language: Python.

IDE: Jupyter Notebook.

Libraries : OpenCV, NumPy, matplotlib, Imutils.

Algorithms: SIFT, BFMatcher, RANSAC.

Description

Image registration is the process of transforming different sets of data into one coordinate system. Data may be multiple photographs, data from different sensors, times, depths, or viewpoints.

To perform a registration process there are several steps to follow:

1) Detect feature – using SIFT.
2) Match corresponding feature- using BFMatcher.
3) Infer geometric transformation.
4) Use the geometric transformation to align one image to another – using Homography.

This part is a crucial in checking whether an image is defective or not because the two images must be aligned well. To do that I had to choose appropriate algorithms that will detect the objects well and will match between the source object and the template object.

I will explain each step separately both the theory and the implementation of the code.
At the end I will represent the complete ImageRegistration() function.
Results represented at the results section.

Step 1: detect features:

In order to detect the objects of the images I used 'SIFT' descriptor.

SIFT- Transformation scale-invariant feature (SIFT) is an algorithm used to locate and describe local features in digital images. It locates certain key points and then provides them with quantitative information (so-called outlines) that can be used for example to identify objects. The outlines are supposed to be immutable against different transformations that may make the images look different even though they represent the same objects.

Keypoints which describes each image will be found.

One pixel in an image is compared with its 8 neighbors as well as 9 pixels in the next scale and 9 pixels in previous scales. This way, a total of 26 checks are made. If it is a local extrema, it is a potential keypoint.

Now, for both images, I am going to generate the SIFT features. First, I must construct a SIFT object and then use detectAndCompute() function to get the keypoints. It will return two values – the keypoints and the descriptors.

```
def ImageRegistration(source, template):

    """
    Two images are passed as a parameters.
    The function returns an aligned image to the template
    """
    # create figure
    fig = plt.figure(figsize=(30,20))

    src_img = source
    reference_img = template

    max_fetures = 500
    good_match_percent = 0.5

    # Initiate SIFT detector
    # max_fetures = The number of best features to retain.
    # The features are ranked by their scores (measured in SIFT algorithm as the local contrast)
    sift = cv2.SIFT_create(max_fetures)

    # find the keypoints and descriptors with SIFT
    kp1, desc1 = sift.detectAndCompute(src_img, mask=None)
    kp2, desc2 = sift.detectAndCompute(reference_img, mask=None)

    plt.subplot(121)
    plt.imshow(desc1[:100],cmap='hot') # plot 100 rows
    plt.title("Left: descriptors of source image.",  fontsize=25)
    plt.axis('off')

    plt.subplot(122)
    plt.imshow(desc2[:100], cmap='Blues')
    plt.title("Right: descriptors of template image.",  fontsize=25)
    plt.axis('off')
```

Figure number 19

In figure 19, I created the function ImageRegistration(). At first, the function creates SIFT detector with 'max_fetures' represents the number of best features to retain.
The features are ranked by their scores (measured in SIFT algorithm as the local contrast).
Then, I used the function detectAndCompute() with parameters: the image selected and

mask -specifying that there is no need to look for keypoints which initialized as 'None'. The function *detects keypoints and computes their descriptors*.

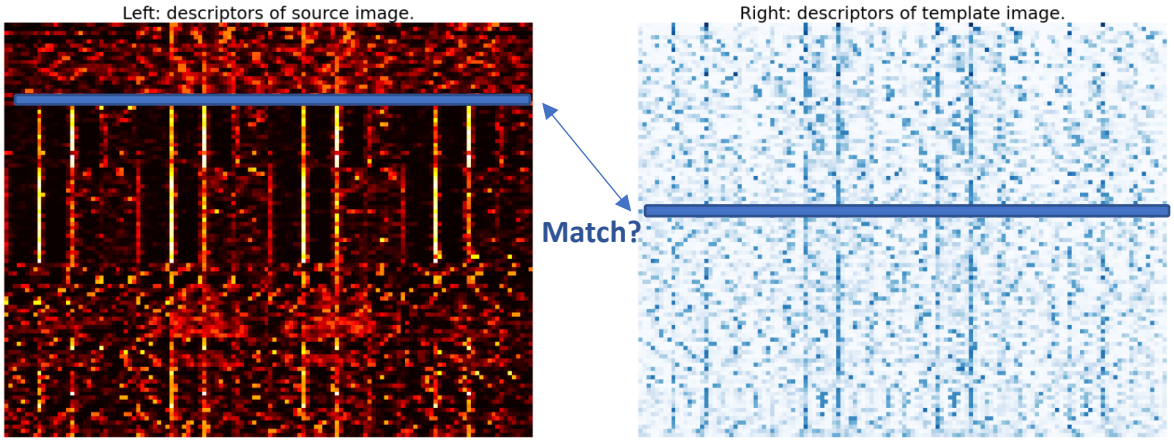


Figure number 20

In figure number 20, I plotted the descriptors of each image(2 matrices NxD). Descriptors are returned as rows in a N x D matrix (N – number of keypoints, D – dimension of the descriptor vector). In that case there are 427 keypoints and the dimension of the descriptor vector is 128. I would like to see which of the descriptors are matched.

Step 2 : Match corresponding feature:

Next step is to find the best matches of the descriptors.

Brute-Force matcher - Takes the descriptor of one feature in first set and is matched with all other features in second set using some distance calculation. And the closest one is returned.

For BF matcher, first created the BFMatcher object using BFMatcher(). It takes two optional params, first one is normType. It specifies the distance measurement to be used. By default, it is "NORM_L2". It is good for SIFT.

Second param is Boolean variable, crossCheck if it is true, Matcher returns only those matches with value (i,j) such that i-th descriptor in set A has j-th descriptor in set B as the best match and vice-versa. That is, the two features in both sets should match each other.

Once it is created, there is important method BFMatcher.match(), this method returns the best match. I sorted the matches according to the minimum distance and removed 0.5 of them not so good matches.

cv.drawMatches() helps to draw the matches. It stacks two images horizontally and draw lines from first image to second image showing best matches.

Then, I extracted the location of "good matches" in order to us them for geometric transformation to align one image to another.

```
    # create BFMatcher object (brute-force)
    bf = cv2.BFMatcher(crossCheck=True)

    # Match descriptors.
    matches = bf.match(desc1,desc2)
    matches = sorted(matches, key = lambda x:x.distance)

    # Remove not so good matches
    num_good_matches = int(len(matches) * good_match_percent)
    matches = matches[:num_good_matches]

    # Draw top matches
    im_matches = cv2.drawMatches(src_img, kp1, reference_img, kp2, matches, None)

    # Adds a subplot and show the matches
    plt.imshow(im_matches)
    plt.title("imMatches")
    plt.axis('off')

    plt.show()

    # Extract location of good matches
    points1 = np.zeros((len(matches), 2), dtype=np.float32)
    points2 = np.zeros((len(matches), 2), dtype=np.float32)

```

Figure number 21

The implementation of the steps finding best matches of the descriptors is described above, figure 21.

Step 3 : Infer geometric transformation:

To perform geometric transformation, I chose Homography.

Homography-

mapping between any two projection planes with the same center of projection.

Two images of a scene are related by a homography under two conditions.

1) The two images are that of a plane (e.g. sheet of paper, card etc.).

2)The two images were acquired by rotating the camera about its optical axis. We take such images while generating panoramas.

Homography is a 3×3 matrix:

$$\begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}$$

Let (x'_1, x'_2, x'_3) be a point in the first image, $x'_3 = 1$ and (x_1, x_2, x_3), $x_3 = 1$, be the coordinates of the same physical point in the second image. Then, the Homography H relates them in the following way:

$$\begin{pmatrix} x'_1 \\ x'_2 \\ x'_3 \end{pmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

When the homography is known, it could be applied to all the pixels of one image to obtain a warped image that is aligned with the second image.

To find the homography matrix it is required to find 4 or more corresponding points in the two images(points1 and points2 variables), I used OpenCV function findHomography() (line 65 figure 22) which internally solves a linear system of equations in order to find numerical approximation to the homography matrix because points1 and points2 variables which represent the good matching features (lines 56-65 figure 22) , does not always fit 100%.

In order to overcome this inaccuracy findHomography() method utilizes a robust estimation technique called Random Sample Consensus (RANSAC) (line 65 figure 22) which produces the right result even in the presence of large number of bad matches.

Random sample consensus (RANSAC)- Iterative method to estimate parameters of a

mathematical model from a set of observed data that contains outliers, when outliers are to be accorded no influence on the values of the estimates. Therefore, it also can be interpreted as an outlier detection method.

RANSAC: Inliers and Outliers.

• RANSAC loop:

1. Get four point correspondences (randomly).2. Compute H (DLT)
3. Count inliers
4. Keep if largest number of inliers

* Recompute H using all inliers

Estimating homography using RANSAC:

Once the homography has been calculated, the transformation can be applied to all pixels in one image to map it to the other image. This is done using the warpPerspective() function (line 70 figure 22) in OpenCV. The function gets the source_image (image that has to be aligned the template), the homography matrix, and the shape of the template image.

```
55
56      # Extract location of good matches
57      points1 = np.zeros((len(matches), 2), dtype=np.float32)
58      points2 = np.zeros((len(matches), 2), dtype=np.float32)
59
60      for i, match in enumerate(matches):
61          points1[i, :] = kp1[match.queryIdx].pt
62          points2[i, :] = kp2[match.trainIdx].pt
63
64      # Find homography
65      h, mask = cv2.findHomography(points1, points2, cv2.RANSAC)
66
67      # Use homography to wrap the source image to destination.
68      # hight and width are the size of destination image (reference_img).
69      height, width = reference_img.shape
70      im1Reg = cv2.warpPerspective(src_img, h, (width, height))
71
72      return(im1Reg)
```

Figure number 22

Full code of the function I built:

```python
def ImageRegistration(source, template):

    """
    Two images are passed as a parameters.
    The function returns an aligned image to the template
    """
    # create figure
    fig = plt.figure(figsize=(30,20))

    src_img = source
    reference_img = template

    max_fetures = 500
    good_match_percent = 0.5

    # Initiate SIFT detector
    # max_fetures = The number of best features to retain.
    # The features are ranked by their scores (measured in SIFT algorithm as the local contrast)
    sift = cv2.SIFT_create(max_fetures)

    # find the keypoints and descriptors with SIFT
    kp1, desc1 = sift.detectAndCompute(src_img, mask=None)
    kp2, desc2 = sift.detectAndCompute(reference_img, mask=None)
#     plt.subplot(121)
#     plt.imshow(desc1[:100],cmap='hot') # plot 100 rows
#     plt.title("Left: descriptors of source image.",  fontsize=25)
#     plt.axis('off')

#     plt.subplot(122)
#     plt.imshow(desc2[:100], cmap='Blues')
#     plt.title("Right: descriptors of template image.",  fontsize=25)
#     plt.axis('off')

    # create BFMatcher object (brute-force)
    bf = cv2.BFMatcher(crossCheck=True)
```

```python
    # Match descriptors.
    matches = bf.match(desc1,desc2)
    matches = sorted(matches, key = lambda x:x.distance)

    # Remove not so good matches
    num_good_matches = int(len(matches) * good_match_percent)
    matches = matches[:num_good_matches]

    # Draw top matches
    im_matches = cv2.drawMatches(src_img, kp1, reference_img, kp2, matches, None)

    # Adds a subplot and show the matches
    plt.imshow(im_matches)
    plt.title("imMatches")
    plt.axis('off')

    plt.show()

    # Extract location of good matches
    points1 = np.zeros((len(matches), 2), dtype=np.float32)
    points2 = np.zeros((len(matches), 2), dtype=np.float32)

    for i, match in enumerate(matches):
        points1[i, :] = kp1[match.queryIdx].pt
        points2[i, :] = kp2[match.trainIdx].pt

    # Find homography
    h, mask = cv2.findHomography(points1, points2, cv2.RANSAC)

    # Use homography to wrap the source image to destination.
    # hight and width are the size of destination image (reference_img).
    height, width = reference_img.shape
    im1Reg = cv2.warpPerspective(src_img, h, (width, height))

    return(im1Reg)
```

Figure number 23

Step 5 – Check if the image aligned to the template

Tools

Programming Language: python.

IDE: Jupyter Notebook.

Libraries: OpenCV, NumPy, matplotlib.

Algorithms: Canny, thresholding, Morphological operators.

Description

In order to check if the source image aligned to the template and contains the exact same object, I created function CardVisualInspectionSystem() which uses functions edgeDetection() and ImageRegistration() I built.

I will represent and explain edgeDetection() function, then explain the use of the function CardVisualInspectionSystem(), which represents the main function of the inspection system.

edgeDetection():

Image and a string are passed as a parameter.
The function returns filtered image according to the method chosen (thresholding/Canny).
I choose to use 2 different methods to edge detection.

The edge detection of image helps to separate the object in the image and later it can be used to check if the object of image is the same to another.

Canny edge detector(line 17 – 16 figure 24):
I chose to use **Canny edge detector** because it has high results.
The **Canny edge detector** is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images. The algorithm is based on grayscale pictures. composed of 5 steps:

1) Smooth image with a Gaussian filter - to get rid of the noise on the image, I used Gaussian blur to smooth it with kernel 5X5.
2) Compute gradients (for each pixel – magnitude and orientation) - The Gradient calculation step detects the edge intensity and direction by calculating the gradient of the image using edge detection operators.

3) Non-maxima suppression (NMS)- the final image should have thin edges. Thus, we must perform non-maximum suppression to thin out the edges.
The algorithm goes through all the points on the gradient intensity matrix and finds the pixels with the maximum value in the edge directions.

4) Thresholding and linking (with two thresholds – low and high) – Used to identifying strong, weak, and non-relevant pixels. High threshold is used to identify the strong pixels (intensity higher than the high threshold) and low threshold is used to identify the non-relevant pixels (intensity lower than the low threshold).

Thresholding and morphological operators(line 28 – 48 figure 24):

Thresholding is a type of image segmentation, where the pixels of an image are being changed to make the image easier to analyze. Most frequently, thresholding used to select areas of interest of an image, while ignoring the parts we are not concerned with.
I used adaptive thresholding first in order select the areas containing the object. This method where the threshold value is calculated for smaller regions and therefore, there will be different threshold values for different regions.

Parameters of the function adaptiveThreshold():

src − source image.
maxVal – max value that can be assigned to a pixel.
adaptiveMethod − A variable of integer the type representing the adaptive method to be used. This will be either of the following two values.
ADAPTIVE_THRESH_MEAN_C − threshold value is the mean of neighborhood area.
thresholdType − A variable of integer type representing the type of threshold to be used.
blockSize − A variable of the integer type representing size of the pixelneighborhood used to calculate the threshold value.
C − A variable of double type representing the constant used in the both methods (subtracted from the mean or weighted mean).

Morphological erosion removes floating pixels and thin lines so that only substantive objects remain. Remaining lines appear thinner, and shapes appear smaller.
I used erosion to reduce unnecessary areas which do not contains the object(line 38).

Morphological dilation makes objects more visible and fills in small holes in objects. Lines appear thicker, and filled shapes appear larger.
I used dilation to highlight the edges of the object(line 44).

```python
def edgeDetection(image, algorithm = 'thresh'):

    """
    image, string are passed as a parameters.
    The function returns filterd image according to the method choosen (thresh/Canny).
    """
    # Plot image process:
    plt.figure(figsize=(18,8))
    plt.subplot(1,4,1)
    plt.imshow(image)
    plt.title('Original image')

    # load the image, convert it to grayscale, and blur it slightly
    img = image.copy()

    # USE Canny algorithm to get bold lines
    if (algorithem == 'Canny'):

        # Smooth image with a Gaussian filter
        img = cv2.GaussianBlur(image, (5, 5), 0)
        edge =  cv2.Canny(img,400,0)

        #Plot Canny
        plt.subplot(1,4,2)
        plt.imshow(edge)
        plt.title('Canny algorithm')

    else:

        # threshold the image, then perform a series of erosions +
        # dilations to remove any small regions of noise
        edge = cv2.adaptiveThreshold(img,250, cv2.ADAPTIVE_THRESH_MEAN_C,cv2.THRESH_BINARY,21,-30)
        # Plot thresholding
        plt.subplot(1,4,2)
        plt.imshow(edge)
        plt.title('thresh')

        edge = cv2.erode(edge, None, iterations=2)
        # Plot erode
        plt.subplot(1,4,3)
        plt.imshow(edge)
        plt.title('erode')

        edge = cv2.dilate(edge, None, iterations=2)
        # Plot dilate
        plt.subplot(1,4,4)
        plt.imshow(edge)
        plt.title('dilate')


    return(edge)
```

Figure number 24

CardVisualInspectionSystem():

The function gets Two images after pre-processing, string which passed as parameters.

The function check if image1 and image2 has the exact same object.

The function returns aligned image to image2 and substruction of aligned image and template.

Steps:

1) Use ImageRegistration() function previously explained to perform registration on the two images, source and template and receive the align_image, an image which the card in the image is aligned to the template card(line 16 figure 25).
2) Detect the edges of the align_image and image2(template) in order to extract the object from the image without background.
   Use edgeDetection() function to perform the edge detection.
   edgeDetection() gets an image and string value which indicates the method we prefer to use between two – thresholding and morphological operators- erosion and dilation or Canny edge detection algorithm (line 28-29 figure 25).
3) Create test, new image, which is a subtraction of align_image_edge, img2_edge. If they are perfectly aligned, test will be a black image – a zero matrix (line 32 figure 25).
4) Determine a threshold, if the number of zeros of the test image is bigger than the threshold chosen, so the image is determined to a defect image and the card should not be offered for sell (line 32-45 figure 25).

```
1  def CardVisualInspectionSystem(image1, image2, algorithm = 'thresh'):
2
3      """
4      Two images, string are passed as a parameters.
5      The function check if image1 and image2 has the exact same object.
6      The function returns aligned image to image2 and substruction of aligned image and template.
7      """
8      # Plot image process:
9      plt.figure(figsize=(20,30))
10
11     # Use copy of the images
12     img1 = image1.copy()
13     img2 = image2.copy()
14
15     # Perform registration
16     align_image = ImageRegistration(img1, img2)
17
18     plt.imshow(align_image)
19     plt.subplot(121)
20     plt.imshow(img2)
21     plt.title('Original image')
22
23     plt.subplot(122)
24     plt.imshow(img1)
25     plt.title('Image to be align')
26
27     # Detect the edges of the images
28     align_image_edge = edgeDetection(align_image, algorithm)
29     img2_edge = edgeDetection(img2, algorithm)
30
31     # Subtraction of images
32     test = cv2.subtract(align_image_edge, img2_edge)
33
34     # Check whether the zero matrix was obtained
35     if (not np.any(test)):
36         print("The images are perfectly the same")
37     else:
38         print("The images are NOT perfectly the same")
39
40         if(np.count_nonzero(test)>509): # Define threshold
41             print("There are {} non-zero pixels".format(np.count_nonzero(test)))
42             print("DEFECT IMAGE")
43         else:
44             print("There are {} non-zero pixels".format(np.count_nonzero(test)))
45             print("SELL THE CARD")
46
47
48     return(test, align_image)
```

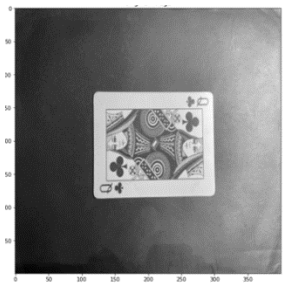
Figure number 25

Results

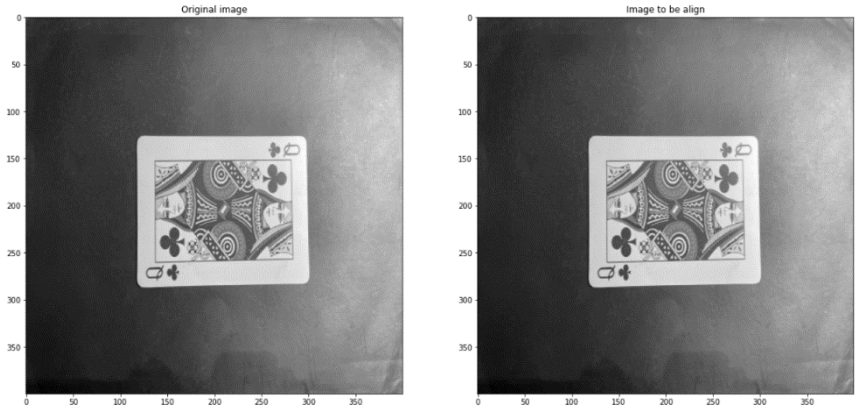
Note: Template is variance_instances[0] for each run.

First run of CardVisualInspectionSystem() on dataset number 1:

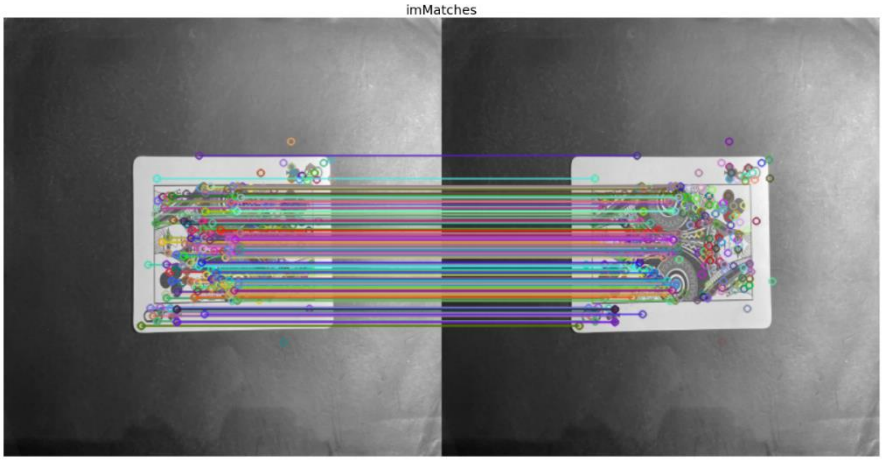
Template image:

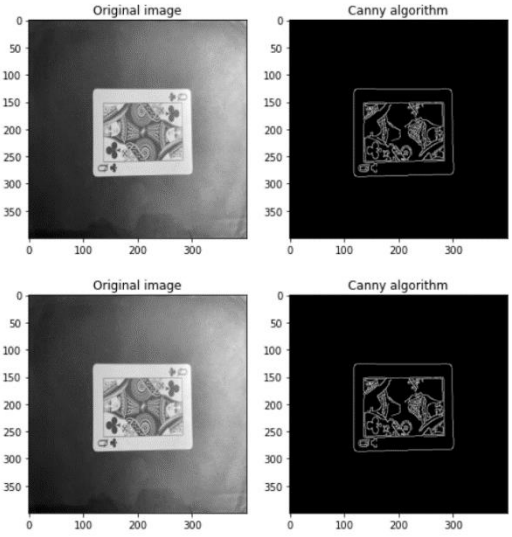


template image Vs. template image:

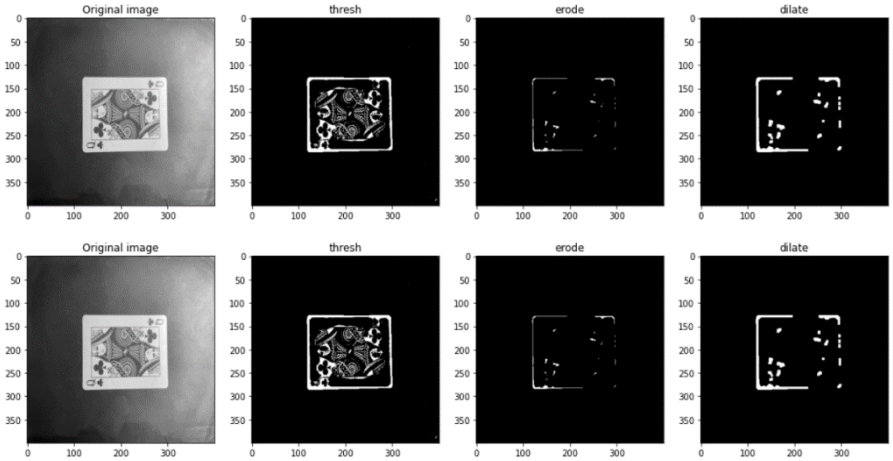


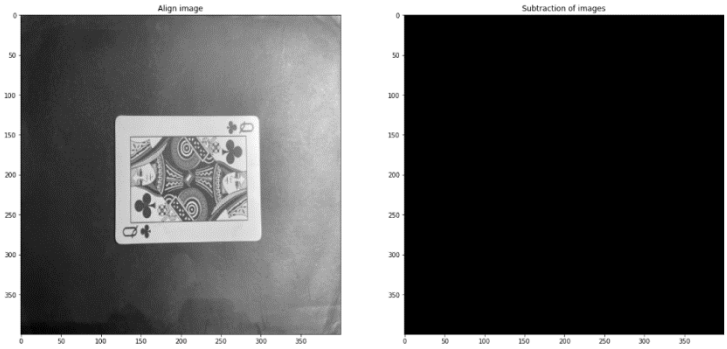

```
test_image, align_image = CardVisualInspectionSystem(variance_instances[0], variance_instances[0],'Canny')
```






```
1 test_image, align_image = CardVisualInspectionSystem(variance_instances[0], variance_instances[0])
```

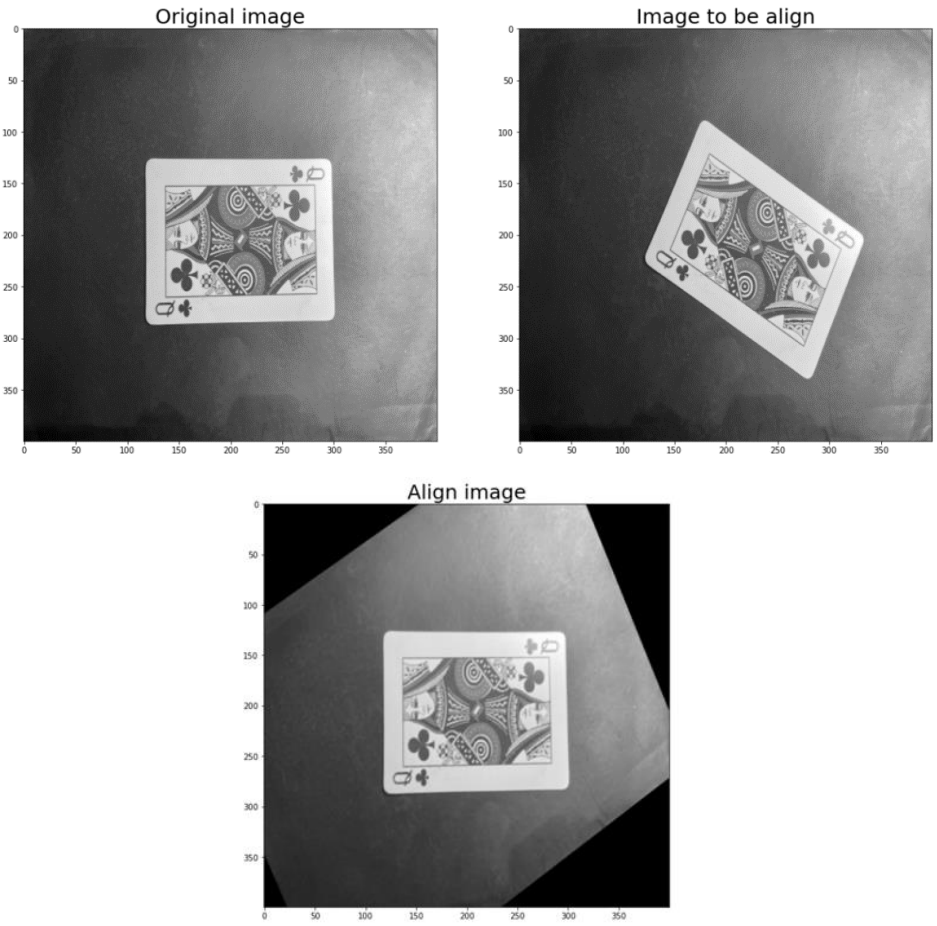




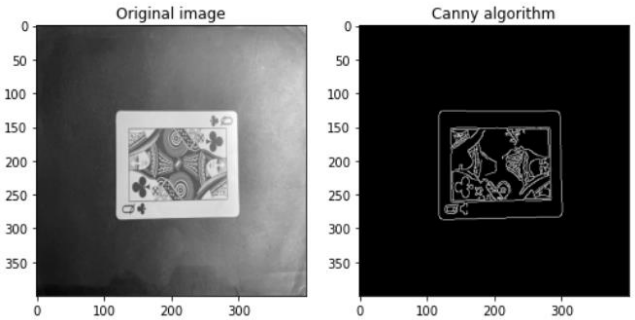
The images are perfectly the same

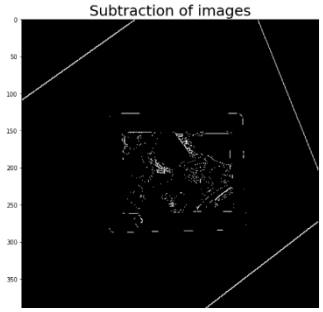
Subtraction matrix is zero matrix. So, the source image is good which make sense because it is compared to itself. In this case True Positive.

variance_instances[1] Vs. template image:



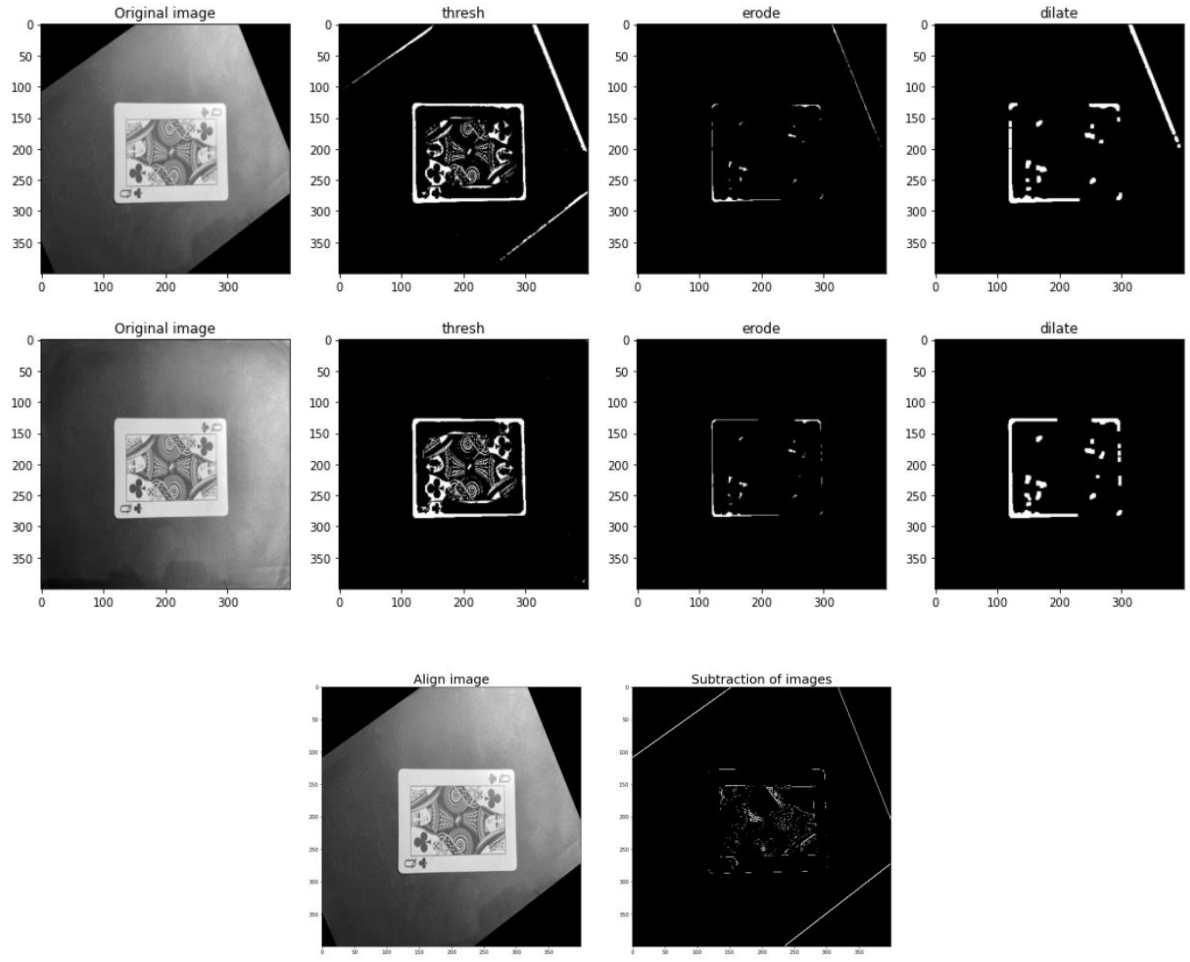

```
test_image, align_image = CardVisualInspectionSystem(variance_instances[1], variance_instances[0],'Canny')
```






```
The images are NOT perfectly the same
There are 1731 non-zero pixels
SELL THE CARD
```

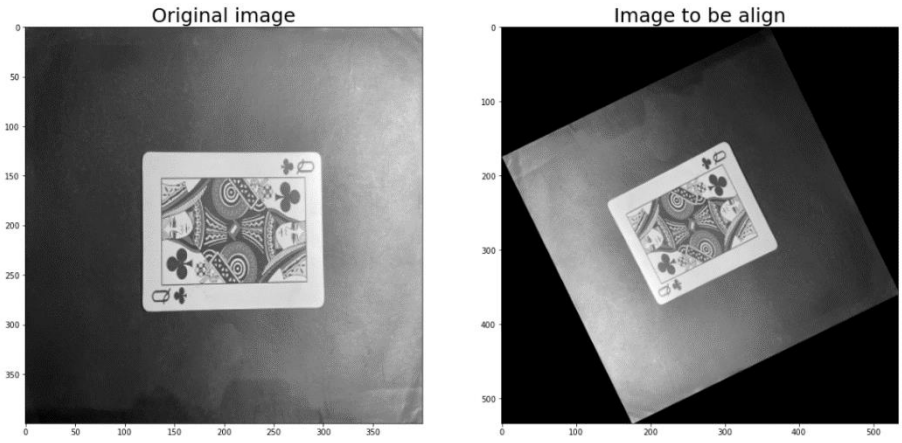
```
test_image, align_image = CardVisualInspectionSystem(variance_instances[1], variance_instances[0])
```




```
The images are NOT perfectly the same
There are 1415 non-zero pixels
SELL THE CARD
```

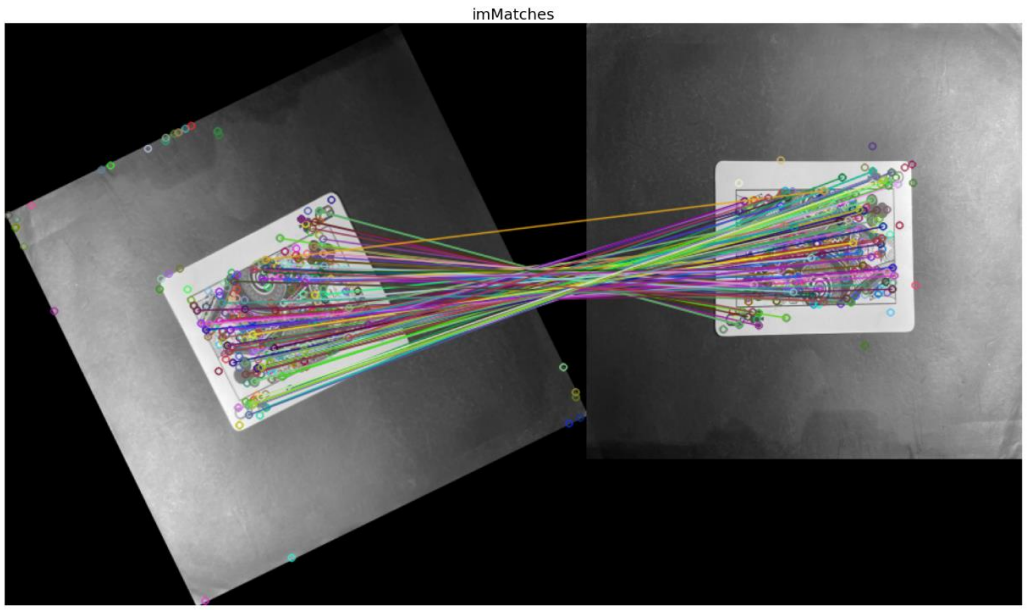
The source image is identical to the template except the image taken as rotated.
We can see the homography performed well.
subtraction matrix is not perfectly non-zeros matrix when we used Canny edge detector but, it has fewer non-zeros values than the threshold. True Positive.
The thresholding method performed better than Canny.
The subtraction image tends to be more like zero- matrix.

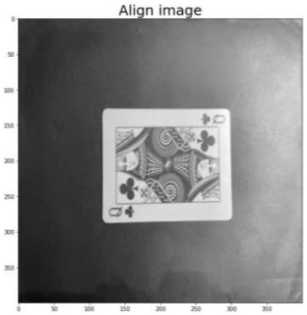
variance_instances[2] Vs. template image:

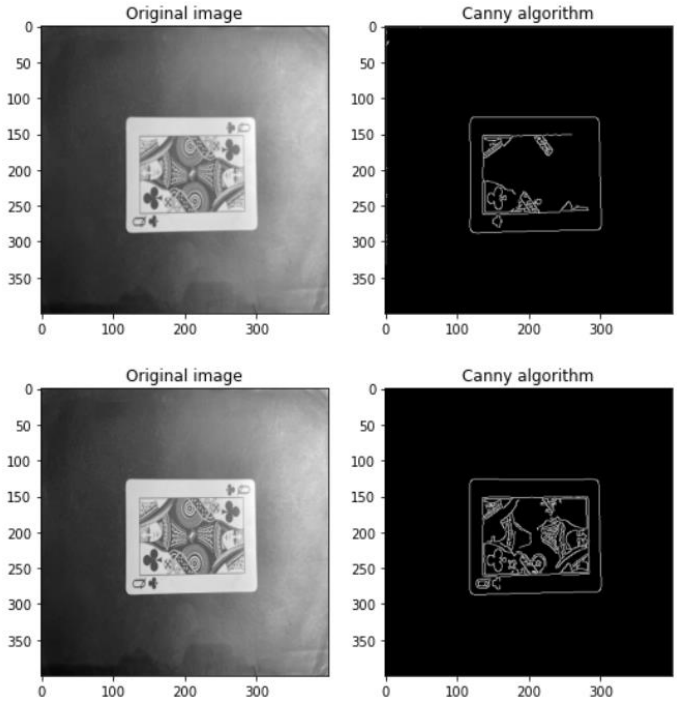


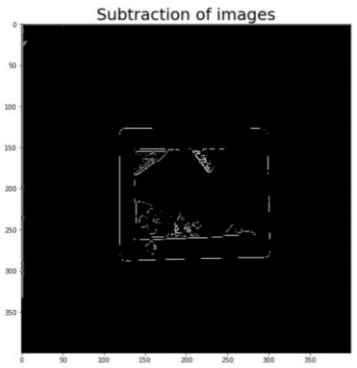
Reminder: the source image(variance_instances[2]) was rotated by the function rotateImage().

```
1 test_image, align_image = CardVisualInspectionSystem(variance_instances[2], variance_instances[0],'Canny')
```





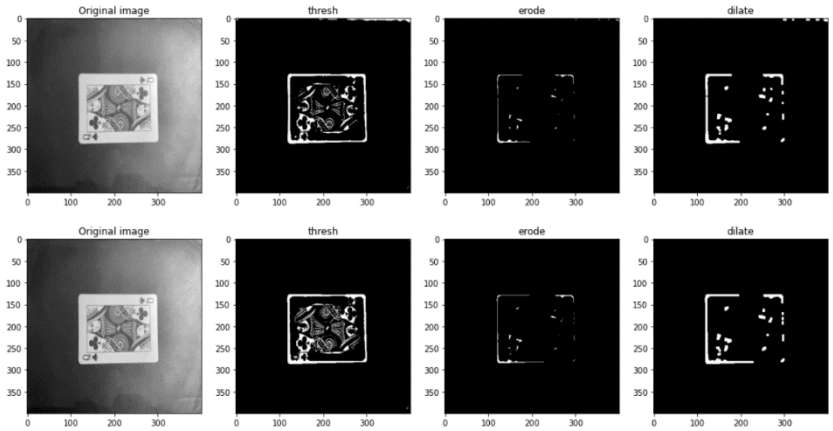


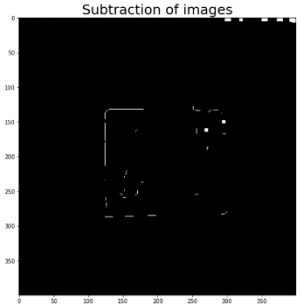



```
The images are NOT perfectly the same
There are 1767 non-zero pixels
DEFECT IMAGE
```

According to the running with the use of Canny's algorithm, we got that the source image has defect but, it was just rotated. So, this is a False Negative result.
As we can see the source image aligned well to the template so, the problem with the result is because the Canny edge detection.

```
1 test_image, align_image = CardVisualInspectionSystem(variance_instances[2], variance_instances[0])
```

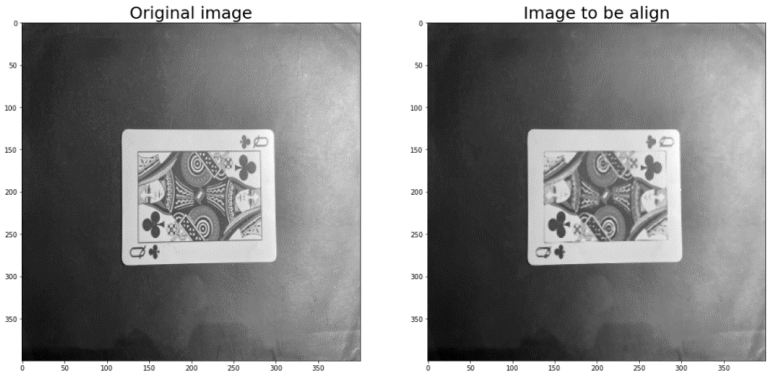




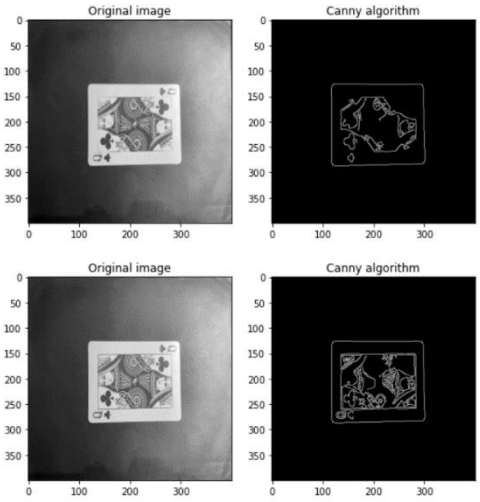

```
The images are NOT perfectly the same
There are 538 non-zero pixels
SELL THE CARD
```

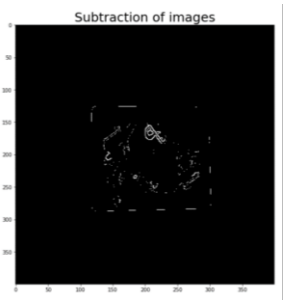
In contrast to Canny's algorithm thresholding method worked well. True Positive.

median_filter_img Vs. template image:



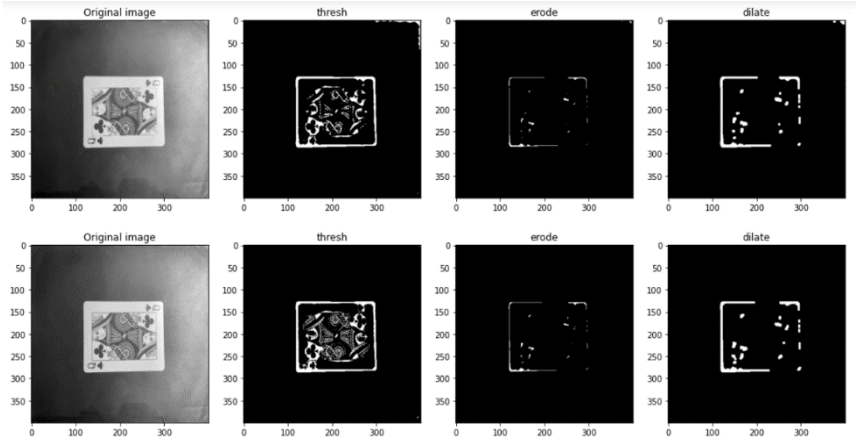

```
1 test_image, align_image = CardVisualInspectionSystem(median_filter_img, variance_instances[0],'Canny')
```

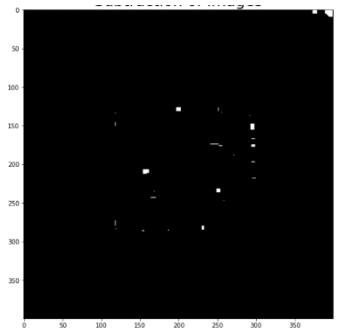





```
The images are NOT perfectly the same
There are 667 non-zero pixels
SELL THE CARD
```

```
1 test_image, align_image = CardVisualInspectionSystem(median_filter_img, variance_instances[0])
```

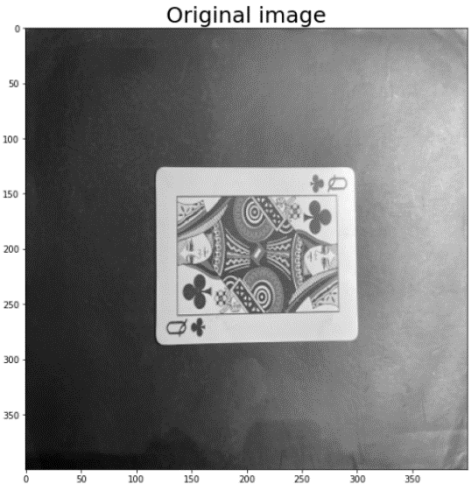


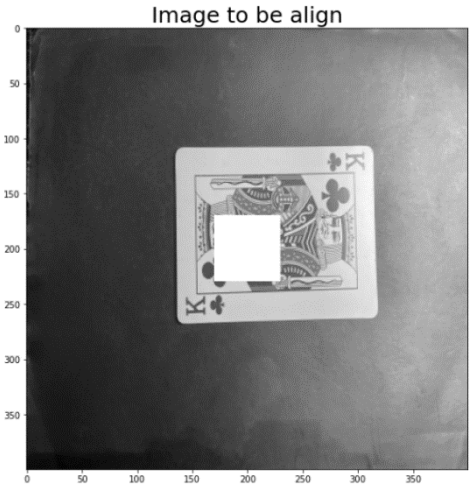


```
The images are NOT perfectly the same
There are 343 non-zero pixels
SELL THE CARD
```

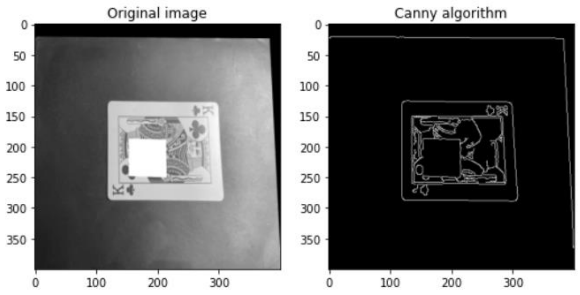
Both methods shows that the card is good for sell. True Positive.

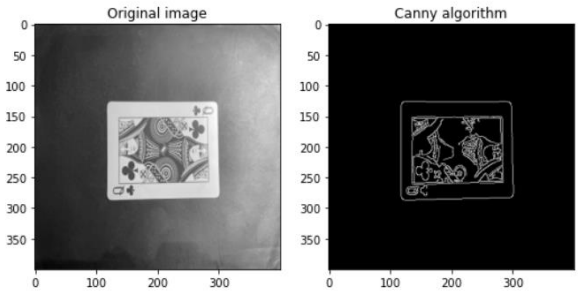
variance_instances[4] Vs. template image

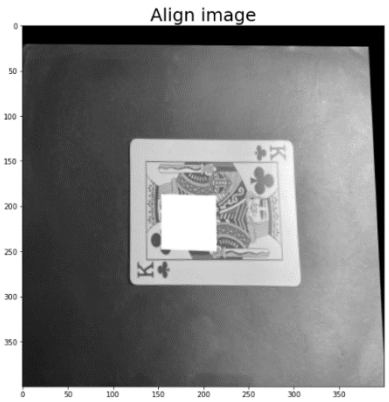


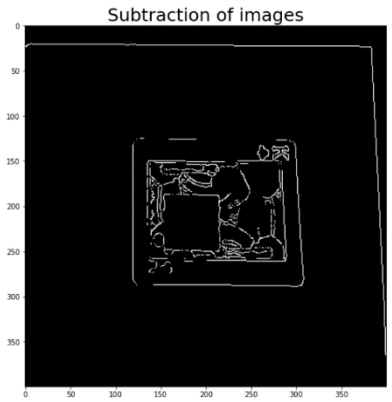



```
1 test_image, align_image = CardVisualInspectionSystem(variance_instances[4], variance_instances[0],'Canny')
```

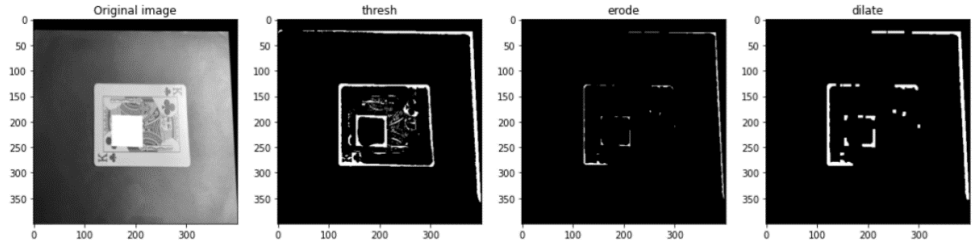


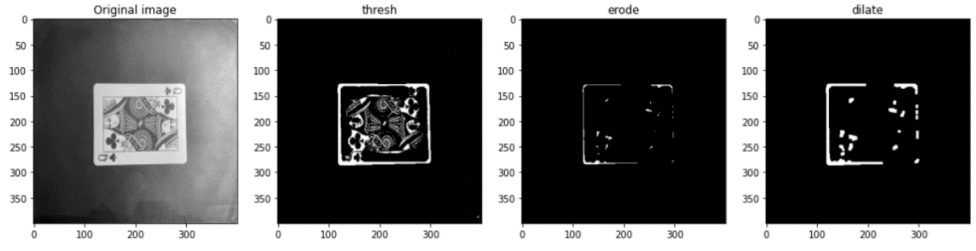


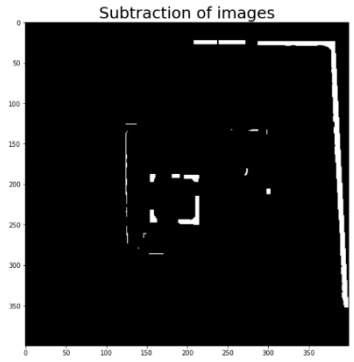





```
The images are NOT perfectly the same
There are 3299 non-zero pixels
DEFECT IMAGE
```







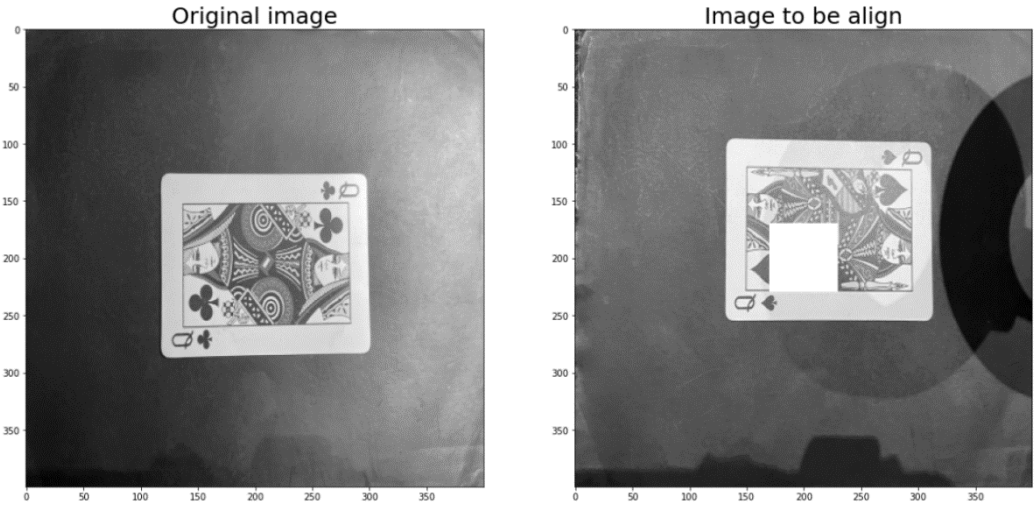

```
The images are NOT perfectly the same
There are 3896 non-zero pixels
DEFECT IMAGE
```

In this case we can see that the object represented at the source image is different than the template's (king and queen). Also, there is a white rectangle(defect) on the source image either way the Inspection system shouldn't be offered the card for sale.

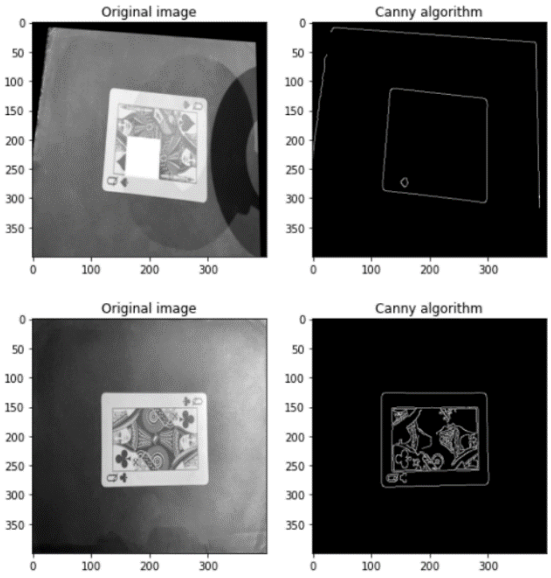
And indeed, using both methods it can be seen that the card is defect.

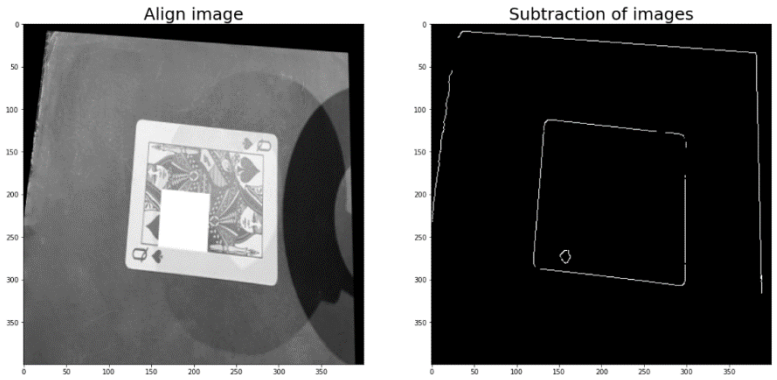
True Negative both cases.

variance_instances[5] Vs. template image:



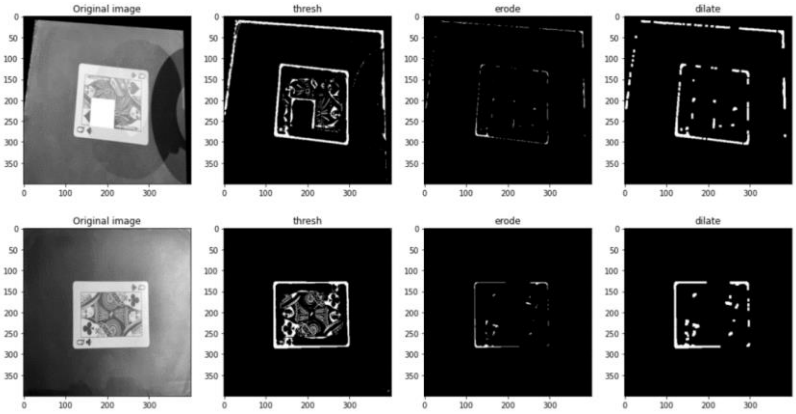

```
1 test_image, align_image = CardVisualInspectionSystem(variance_instances[5], variance_instances[0],'Canny')
```






```
The images are NOT perfectly the same
There are 1498 non-zero pixels
SELL THE CARD
```

```
1 test_image, align_image = CardVisualInspectionSystem(variance_instances[5], variance_instances[0])
```

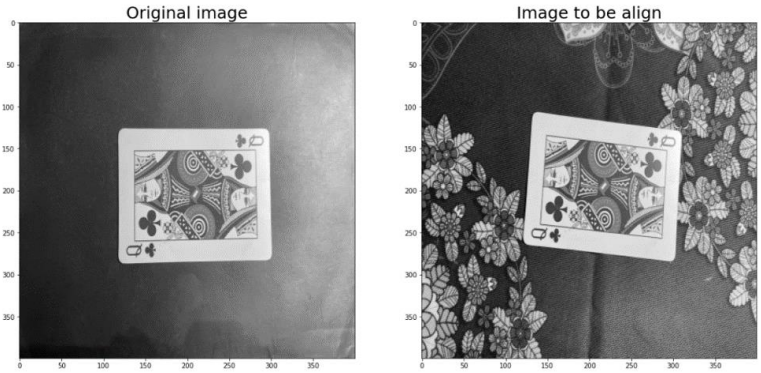



```
The images are NOT perfectly the same
There are 5134 non-zero pixels
DEFECT IMAGE
```

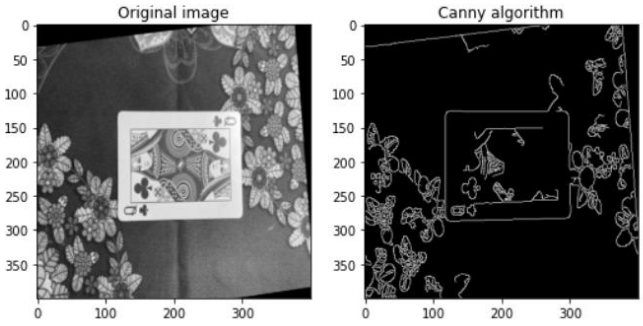
At this example we can see that the object represented at the source image has different lighting than the template. Also, there is a white rectangle(defect) on the source image the Inspection system shouldn't be offer the card for sale.

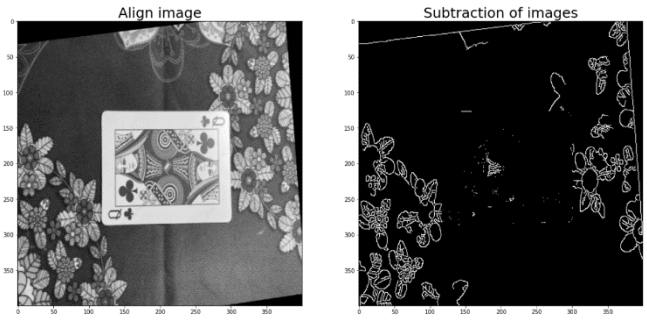
When running the algorithm with Canny's method, we got False Positive in contrast to thresholding method returned correctly True Negative.

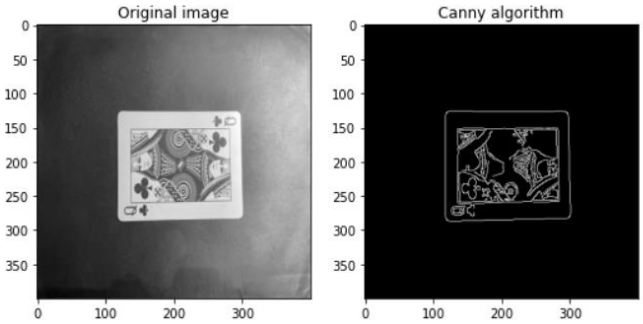
different_background Vs. template image:




```
1 test_image, align_image = CardVisualInspectionSystem(different_background, variance_instances[0],'Canny')
```

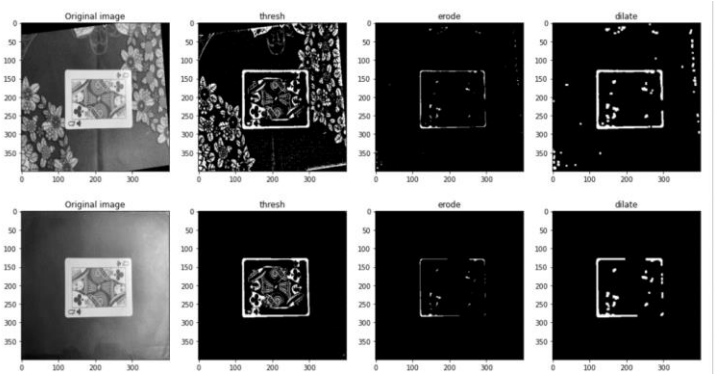


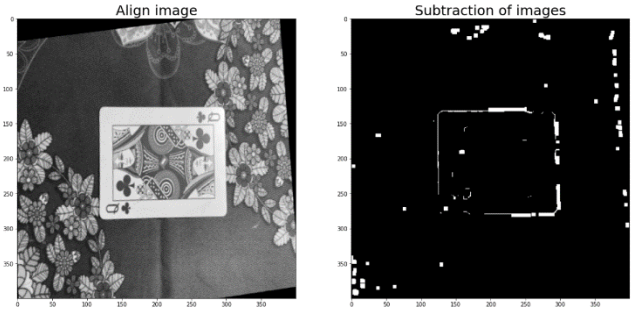





```
The images are NOT perfectly the same
There are 7847 non-zero pixels
DEFECT IMAGE
```

```
1 test_image, align_image = CardVisualInspectionSystem(different_background, variance_instances[0])
```



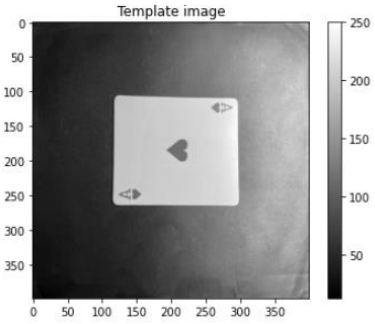



```
The images are NOT perfectly the same
There are 3168 non-zero pixels
DEFECT IMAGE
```

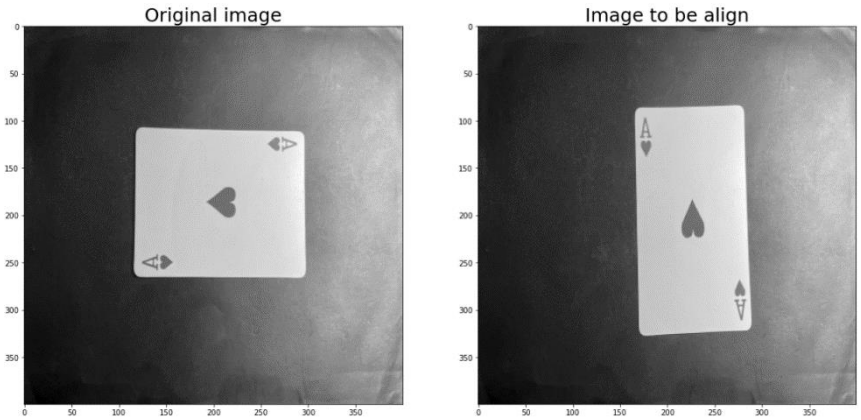
This case is very interesting, the source image was taken on noisy background and slightly rotated but, the object is identical to the template's object. Shortly, this image is True Positive.
Both, Canny's algorithm and thresholding method didn't work well. False Negative.

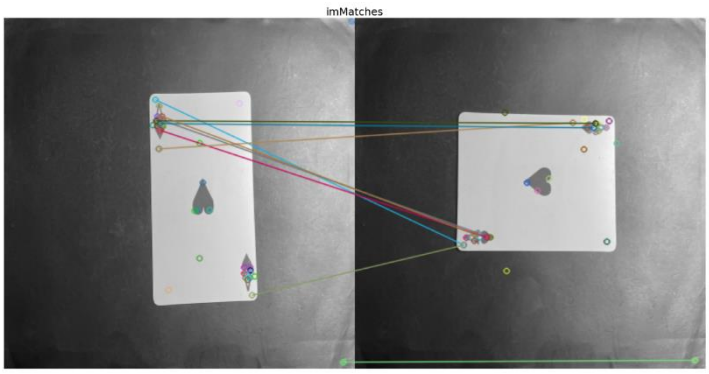
Second run of CardVisualInspectionSystem() on dataset number 2:

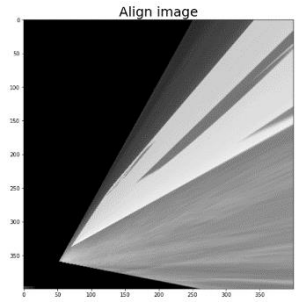
Template image:



variance_instances_[1] Vs. template image:

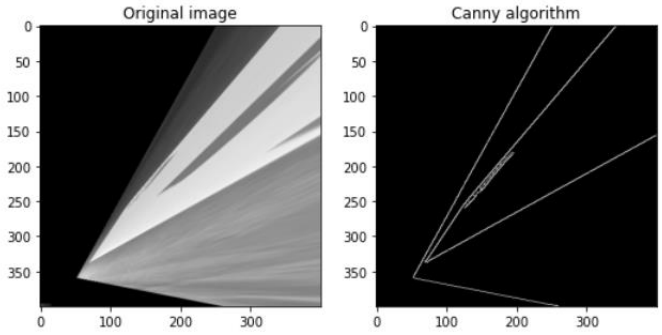


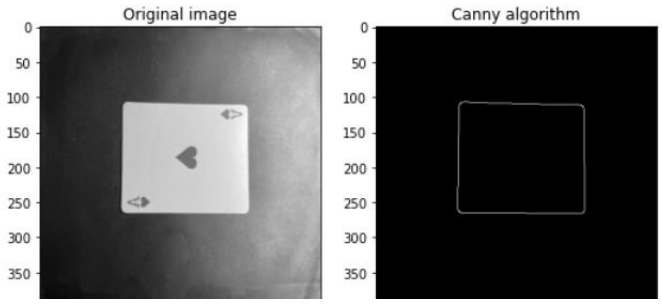


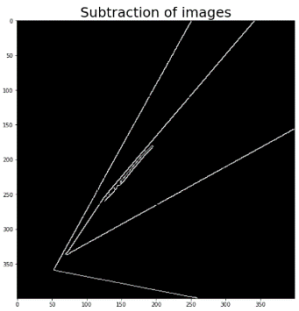


We can see that the alignment did not perform well. Apparently, this happened because the lack of matches were found between the descriptors because there wasn't enough features on each card.

```
1 test_image, align_image = CardVisualInspectionSystem(variance_instances[1], variance_instances[0],'Canny')
```





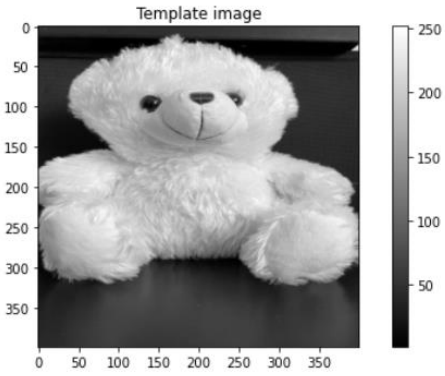



```
The images are NOT perfectly the same
There are 2076 non-zero pixels
DEFECT IMAGE
```

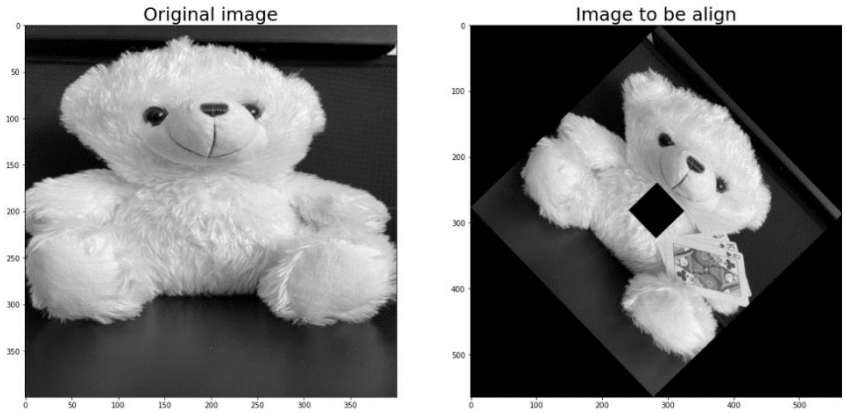
False negative output with Canny's method. The source card has no defects but, the inspection system said that it has.

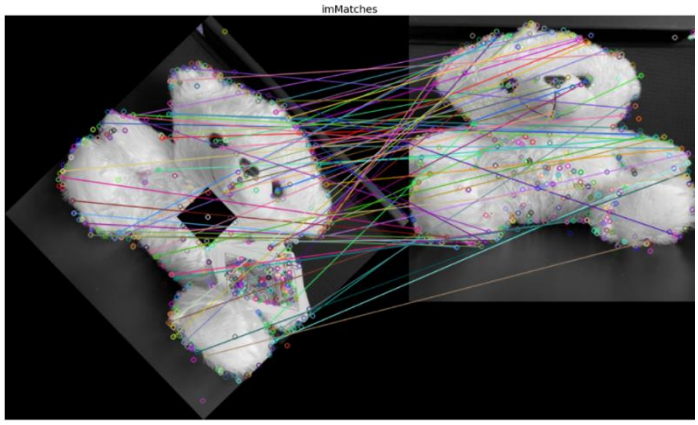
Third run of CardVisualInspectionSystem() on dataset number 3 :

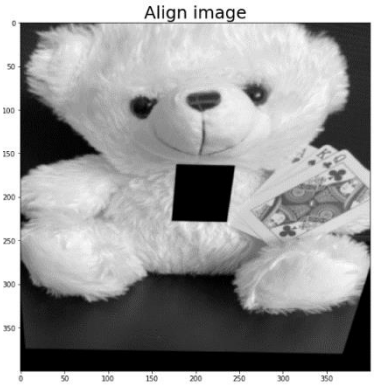
Template:



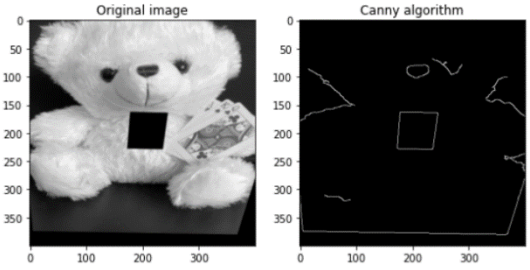
variance_instances_[5] Vs. template image:

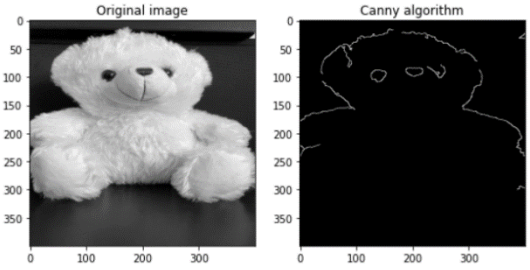


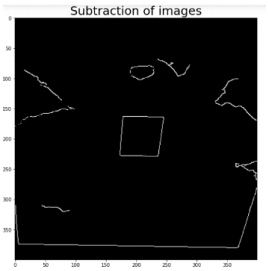





```
1 test_image, align_image = CardVisualInspectionSystem(variance_instances[5], variance_instances[0],'Canny')
```



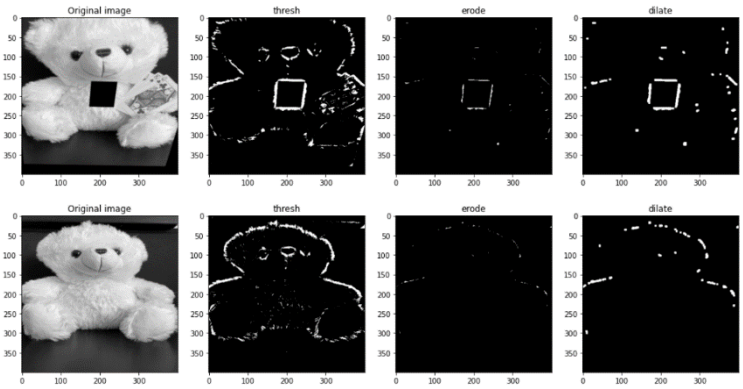


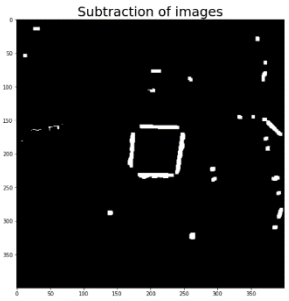



```
The images are NOT perfectly the same
There are 1573 non-zero pixels
SELL ITEM
```

In this example it can be seen a damaged teddy bear but, according to the algorithm with Canny's method the teddy bear should be offered for sell. False Positive example.

```
1 test_image, align_image = CardVisualInspectionSystem(variance_instances[5], variance_instances[0])
```

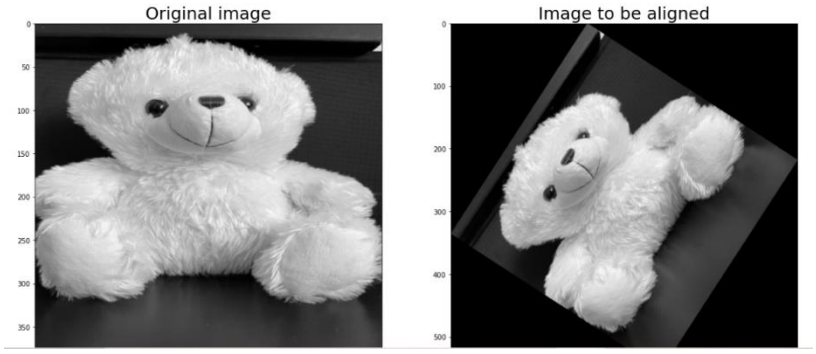




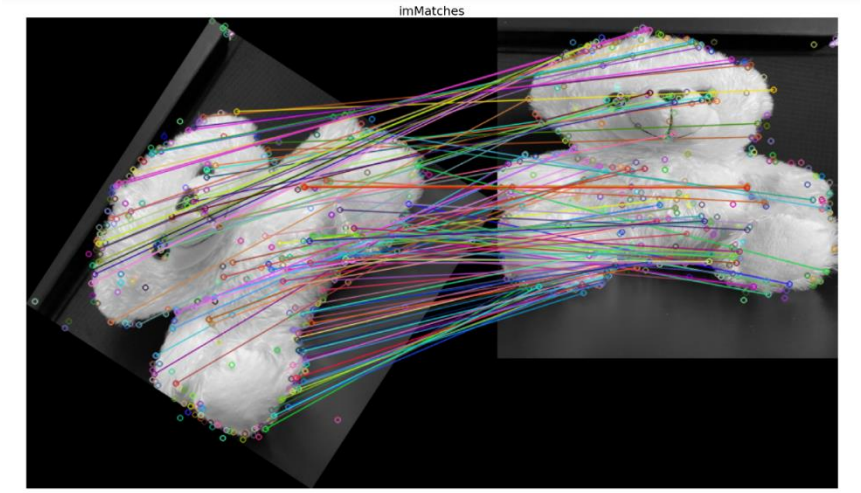

```
The images are NOT perfectly the same
There are 2831 non-zero pixels
DEFECT IMAGE
```

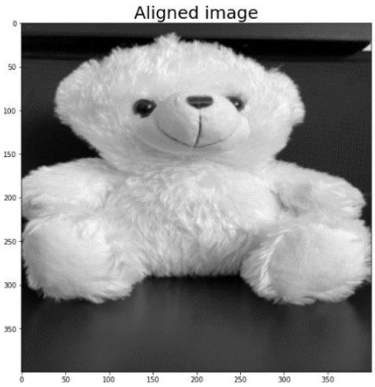
According to the thresholding method indeed the algorithm returned True Negative answer.

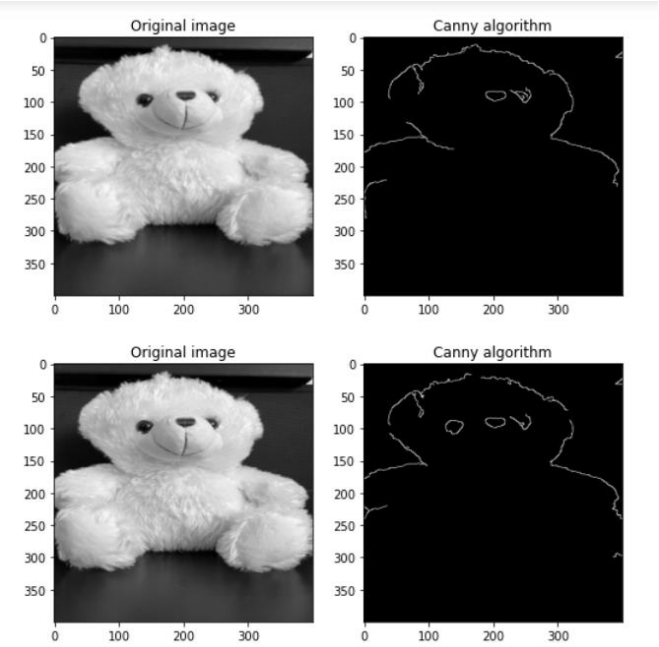
variance_instances_[2] Vs. template image:

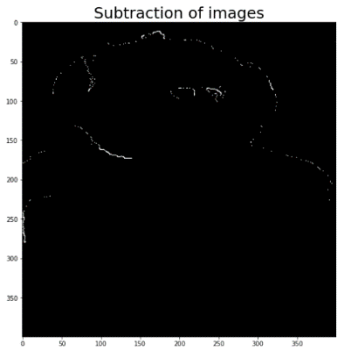



```
test_image, align_image = CardVisualInspectionSystem(variance_instances[2], variance_instances[0],'Canny')
```

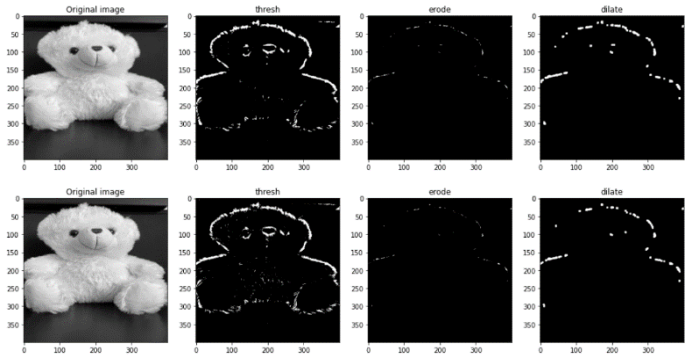


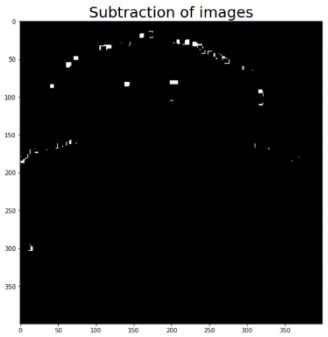







```
The images are NOT perfectly the same
There are 379 non-zero pixels
SELL ITEM
```



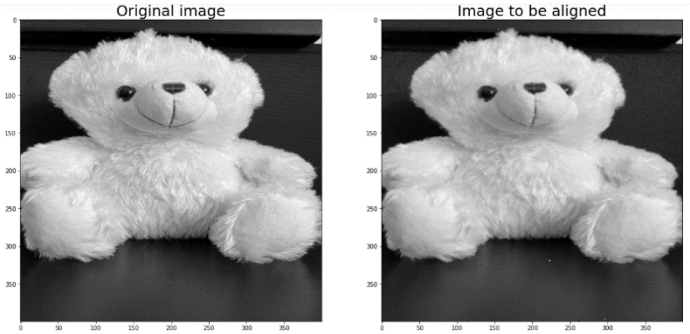


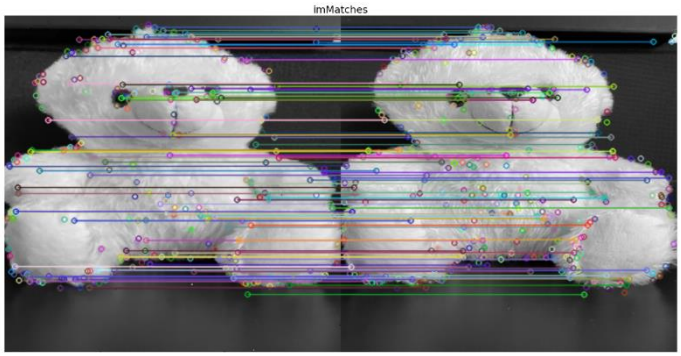

```
The images are NOT perfectly the same
There are 684 non-zero pixels
SELL ITEM
```

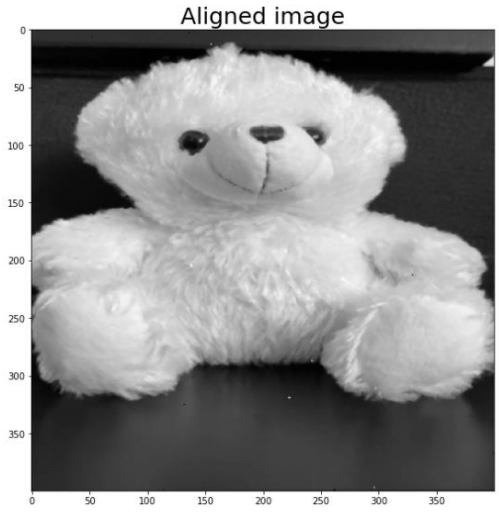
In this case, the source image is the template image rotated by rotateImage() function.
The output should be True Positive.

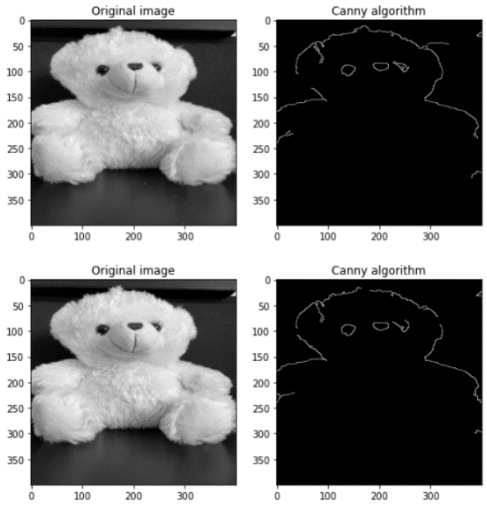
True Positive result received for running the algorithm with Canny's method and thresholding method.

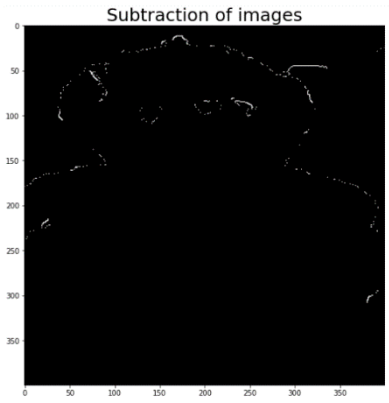
median_filter_img Vs. template image:



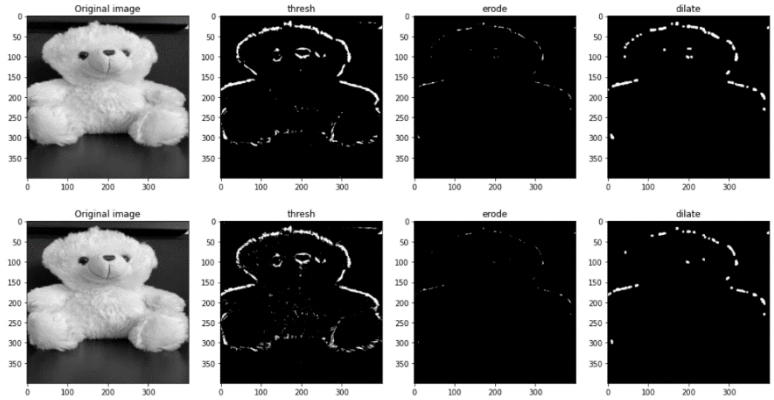


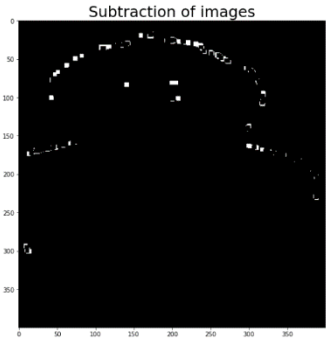







```
The images are NOT perfectly the same
There are 458 non-zero pixels
SELL ITEM
```






```
The images are NOT perfectly the same
There are 1026 non-zero pixels
SELL ITEM
```

In this case, the source image is the median_filter_img.
The output should be True Positive.

True Positive result received for running the algorithm with Canny's method and thresholding method.

Discussions and Conclusions

The course project helped me understand the complexity of classic image processing in diagnosing objects in images versus computer vision and deep learning which I learned in other courses. There is no doubt that image processing is base of computer vision and to solve optimization problems with high accuracy, related to images we need to use more advanced tools like CNN.

For example, I tried to perform the registration and detect the location points of the object for each image using contours but had difficulties to align the images.
Also, I had a difficulty to choose the threshold of the minimum non-zero values in the test image.

At figure first thresh image is the source image and the thresh image in the second row is the template. The images contains the exact same object but, the source image is rotated.

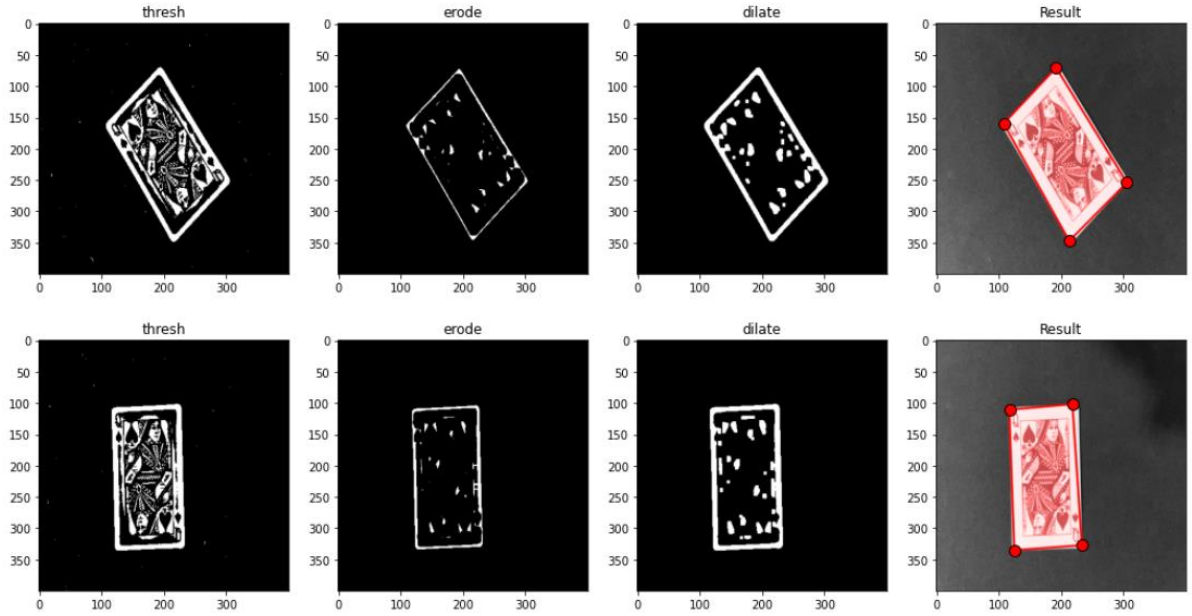


Figure number 1

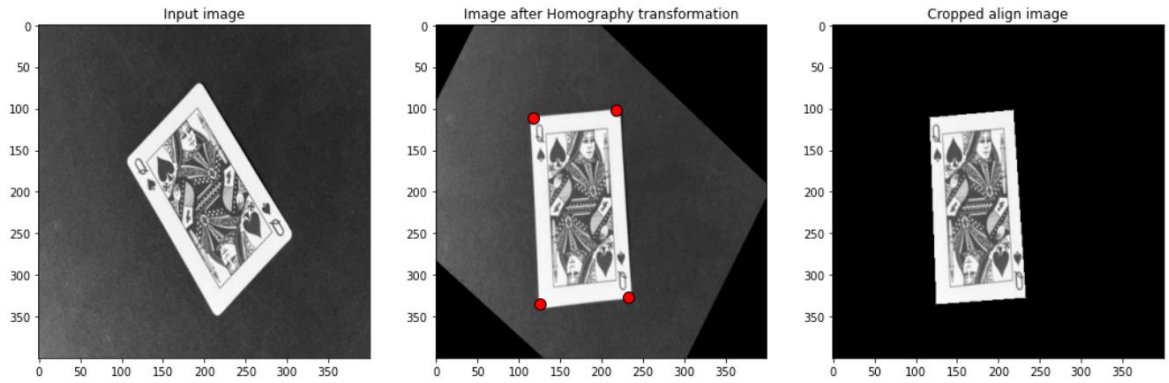


Figure number 2

In figure 1,2 it can be seen that the source image and the cropped align image after applied homography.

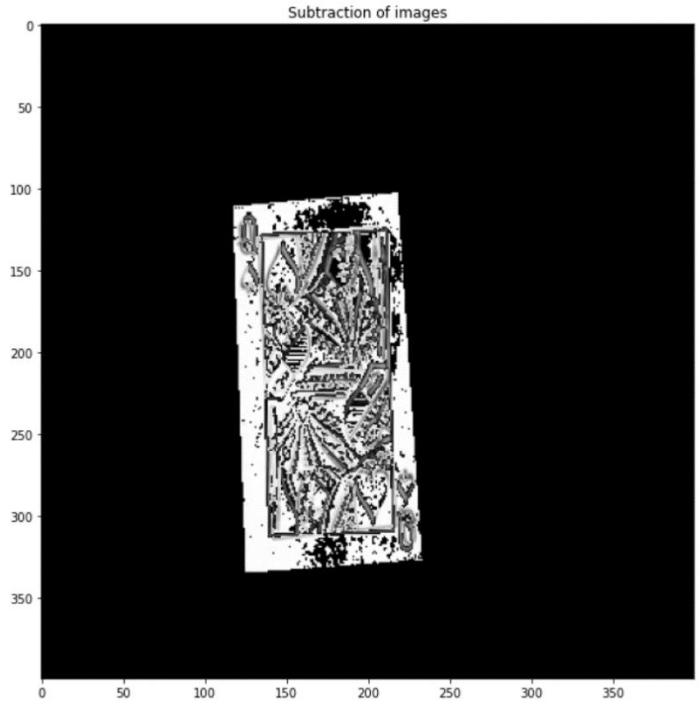


Figure number 3

This figure shows the test(substruction of align_image and template) image.
The test image should be zero matrix but, the optimization does not perform well.
I will represent the code on the appendixes.

Furthermore, I am happy that the project included high Linear Algebra analysis and it is amazing to see how the mathematics can be applied in real world. There is no doubt that I will work in the field of computer vision in the future.

Appendices

First prototype:

```
def detectFeature(image, algorithem = 'thresh'):

    # Plot image process:
    plt.figure(figsize=(18,8))

    # load the image, convert it to grayscale, and blur it slightly
    img = image.copy()

#     gray = cv2.GaussianBlur(image, (5, 5), 0)

    # USE Canny algorithm to get bold lines
    # threshold the image, then perform a series of erosions +
    # dilations to remove any small regions of noise
    if (algorithem == 'Canny'):

        edge =  cv2.Canny(img,800,0)
    #     plt.imshow(edge)
    #     thresh = cv2.threshold(gray,150,250, cv2.THRESH_BINARY)[1]

#         Plot Canny
        plt.subplot(1,4,1)
        plt.imshow(edge)
        plt.title('Canny algorithm')

        # find contours in Canny edge image, then grab the largest one
        cnts = cv2.findContours(edge.copy(), cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE)
        cnts = imutils.grab_contours(cnts)
        c = max(cnts, key=cv2.contourArea)

    else:

        thresh = cv2.adaptiveThreshold(img,250, cv2.ADAPTIVE_THRESH_MEAN_C,cv2.THRESH_BINARY,21,-21)
        # Plot erode
        plt.subplot(1,4,1)
        plt.imshow(thresh)
        plt.title('thresh')

        thresh = cv2.erode(thresh, None, iterations=2)
        # Plot erode
        plt.subplot(1,4,2)
        plt.imshow(thresh)
        plt.title('erode')

        thresh = cv2.dilate(thresh, None, iterations=2)
        # Plot dilate
        plt.subplot(1,4,3)
        plt.imshow(thresh)
        plt.title('dilate')
```

```
        # find contours in thresholded image, then grab the largest
        # one
        cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE)
        cnts = imutils.grab_contours(cnts)
        c = max(cnts, key=cv2.contourArea)

    # determine the most extreme points along the contour
    # Make list pts for contains all the points
    extLeft = tuple(c[c[:, :, 0].argmin()][0])
    extRight = tuple(c[c[:, :, 0].argmax()][0])
    extTop = tuple(c[c[:, :, 1].argmin()][0])
    extBot = tuple(c[c[:, :, 1].argmax()][0])
    pts = np.int32([[extLeft[0],extLeft[1]], [extTop[0],extTop[1]], [extRight[0],extRight[1]], [extBot[0],extBot[1]]])
#     pts = np.float32([[extLeft[0],extLeft[1]], [extTop[0],extTop[1]], [extRight[0],extRight[1]] ]).reshape(-1,1,2)

    # draw red frame
    img1_marks = cv2.polylines(cv2.cvtColor(img,cv2.COLOR_GRAY2RGB),[pts],True,(255,0,0),2, cv2.LINE_AA)

    # fill with 50% red
    img1_fill = cv2.fillPoly(np.zeros_like(cv2.cvtColor(img,cv2.COLOR_GRAY2RGB)),[pts],(255,0,0))
    img1_marks = cv2.addWeighted(img1_marks,1,img1_fill,0.5,0)


    # Plot after registretion
    plt.subplot(1,4,4)
    plt.imshow(img1_marks)
    plt.title('Result')

    # show the output image with extreme points
    plt.plot(extLeft[0],extLeft[1],'or',markeredgecolor='k',markersize=10)
    plt.plot(extRight[0],extRight[1],'or',markeredgecolor='k',markersize=10)
    plt.plot(extTop[0],extTop[1],'or',markeredgecolor='k',markersize=10)
    plt.plot(extBot[0],extBot[1],'or',markeredgecolor='k',markersize=10)



    return(img,pts)
```

```
def cropImage(image, pts):

    # Read a image
    crop_img = image.copy()

    # Define the polygon coordinates to use or the crop
    polygon = np.int32(pts.reshape(1,4,2))

    # First find the minX minY maxX and maxY of the polygon
    minX = crop_img.shape[1]
    maxX = -1
    minY = crop_img.shape[0]
    maxY = -1
    for point in polygon[0]:

        x = point[0]
        y = point[1]

        if x < minX:
            minX = x
        if x > maxX:
            maxX = x
        if y < minY:
            minY = y
        if y > maxY:
            maxY = y

    # Go over the points in the image if thay are out side of the emclosing rectangle put zero
    # if not check if thay are inside the polygon or not
    cropedImage = np.zeros_like(crop_img)
    for y in range(0,crop_img.shape[0]):
        for x in range(0, crop_img.shape[1]):

            if (x < minX or x > maxX or y < minY or y > maxY):
                continue

            if (cv2.pointPolygonTest(np.asarray(polygon),(x,y),False) >= 0):
                cropedImage[y, x] = crop_img[y, x]

    return(cropedImage)

```

```python
def geometricTransformation(source, template, pts_source, pts_template):

    # Plot image process:
    plt.figure(figsize=(18,8))

    rows, cols = template.shape

    M, mask = cv2.findHomography(pts_source, pts_template, method=cv2.RANSAC)
    dst = cv2.warpPerspective(source, M, (cols,rows))

    # Plot images
    plt.subplot(1,3,1)
    plt.imshow(source)
    plt.title('Input image')

    plt.subplot(1,3,2)
    plt.imshow(dst)
    plt.title('Image after Homography transformation')

    # Show the output image with extreme points
    points = pts_template.reshape(4,2)
    for point in points:
        plt.plot(point[0],point[1],'or',markeredgecolor='k',markersize=10)

    # Crop dst image
    crop_dst = cropImage(dst.copy(), pts_template)

    plt.subplot(1,3,3)
    plt.imshow(crop_dst)
    plt.title('Cropped align image')
    plt.show()

    return(crop_dst)
```

```python
def CardVisualInspectionSystem(image1, image2, algorithem = 'thresh'):

    img1 = image1.copy()
    img2 = image2.copy()

    # Part 1: Registration
    source, pts_source= detectFeature(img1, algorithem)
    template, pts_template= detectFeature(img2, algorithem)

    align_image = geometricTransformation(img1, img2, pts_source, pts_template)

    crop_template = cropImage(img2, pts_template)

    # Subtraction of images
    test = align_image-crop_template

    plt.imshow(test)
    plt.title('Subtraction of images')

    # Check whether the zero matrix was obtained
    if (not np.any(test)):
        print("The images are perfectly the same")
    else:
        print("The images are NOT perfectly the same")

        if(np.count_nonzero(test)>21875): # Define threshold
            print("There are {} non-zero pixels".format(np.count_nonzero(test)))
            print("DEFECT IMAGE")
        else:
            print("There are {} non-zero pixels".format(np.count_nonzero(test)))
            print("SELL THE CARD")
    return(test)
```

References

Visual Inspection system:

https://www.mathworks.com/discovery/visual-inspection

Median filtering:

https://en.wikipedia.org/wiki/Median_filter

https://medium.com/@florestony5454/median-filtering-with-python-and-opencv-2bce390be0d1

Salt and Pepper noise:

https://medium.com/@florestony5454/median-filtering-with-python-and-opencv-2bce390be0d1

RANSAC:

https://en.wikipedia.org/wiki/Random_sample_consensus#:~:text=Random%20sample%20consensus%20(RANSAC)%20is,as%20an%20outlier%20detection%20method.

SIFT:

https://www.analyticsvidhya.com/blog/2019/10/detailed-guide-powerful-sift-technique-image-matching-python/

Homography:

https://mattmaulion.medium.com/homography-transform-image-processing-eddbcb8e4ff7

https://en.wikipedia.org/wiki/Homography

- Multiple View Geometry in computer vision – Richard Hartley: page 31-44

Morphological Operations:

https://www.mathworks.com/help/images/morphological-dilation-and-erosion.html#:~:text=Dilation%20adds%20pixels%20to%20the,used%20to%20process%20the%20image.

Adaptive thresholding:

https://homepages.inf.ed.ac.uk/rbf/HIPR2/adpthrsh.htm#:~:text=Adaptive%20thresholding%20typically%20takes%20a,threshold%20has%20to%20be%20calculated.

Canny edge detector:

https://en.wikipedia.org/wiki/Canny_edge_detector