**Date:** 04/03/2021

**CS 6643 -** Computer Vision, Spring 2021 **Prof.James Fishbaugh**

# PROJECT II: CONVOLUTION, CROSS- CORRELATION AND IMAGE PANORAMAS

**Pruthviraj R Patil, Computer Science, MS.**
**New York University, Tandon School of Engineering**
**Email: prp7650@nyu.edu**

## 1. Introduction

Image processing models' pipelines contain various preprocessing technologies. In the case of Computer Vision, where we interpret the visual world, a better quality of dataset is quintessential no matter how good the architecture of the data model is. But, nowadays data is outsourced from zillions of devices worldwide, which makes it difficult to be cleaned after mining it. Hence, whenever the image data model is created, the preprocessing techniques are applied to the data to enhance the image quality. In this work, three main types of preprocessing technologies are discussed.

In the case of preprocessing, there might be a need for smoothening, edge detection, and image stitching (panorama) in order to affect the pipeline's efficiency and robustness. In this work, the technologies of convolution, cross-correlation, edge detection, and image stitching methodologies are illustrated.

## 2. Description and Illustration

### 2.1 Convolution and Derivative Filters

Convolution is an image preprocessing technique that uses filters that are applied over images to highlight various features. This helps the image processing model to classify images effectively. One of the main types of features that can be extracted using these filters is the edge detection technique. This can be extracted using the derivative filters. But, before that, the image has to be blurred in order to denoise the image and impose a generalization factor to induce robustness into the model.

### 2.12 Denoise the image with a Gaussian filter

As mentioned previously, before finding the edges, there is a need for denoising the image before feature extraction. The generalization is induced into the model by smoothing the image as well as reducing the details of the image. In this case, one of the famous filters used is the Gaussian filter.

The intuition behind the Gaussian filter is that in order to smoothen the image, there is a need for the image to convolve with the filter that inculcates the property of the 'n' number of images. This can be quantified using the central limit theorem, which is when we collect a large

number of images, the samples altogether start to look like the normal distribution with the mean of μ and the variance of $\frac{\sigma^2}{n}$ where n is the number of pixels in the image. Using this intuition, Gaussian distribution is formulized and is shown in equation 2.11 and its filter is created. In this illustration, the standard deviation **σ**is set to be 21 as shown in figure 2.11. The code used to create this filter is shown below as well.

$$g(x, y; \sigma) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

**equation 2.11**

```
def createGaussianFilter(n, stdev):
    #status: complete

    #task: creates a square gaussian filter
    #parameters: n= rows and column count, stdev= standard deviation
    #returns: gaussian kernel of size (n,n) and standard dev of stdev

    variance=stdev*stdev #variance
    gaussian_filter=np.zeros((n, n))
    x=np.linspace(-n/2.0, n/2.0, n)
    y=np.linspace(-n/2.0, n/2.0, n)

    for i in range(0, n):
        for j in range(0, n):
            gaussian_filter[i, j] =
(1/math.sqrt((2*math.pi*variance)))*math.exp(-(x[i]*x[i]+y[j]*y[j])/(2*variance))
    return gaussian_filter/np.sum(gaussian_filter.flatten())
```

The filter of the size (21, 21) is then convolved over the image shown in figure 2.11 beside it. This process is carried on with respect to the convolution method using the equation shown below in equation 2.12 where w(s, t) is the filter and f(x-s, y-t) is the image patch on every epoch.

$$g(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t) f(x - s, y - t)$$

**equation 2.12**


**Figure 2.11. The input image, Gaussian filter with a standard deviation of 21, and the output image after convolving image using this Gaussian filter**

The output of the convolution using the Gaussian filter is shown in figure 2.11. Hare, the image is denoised and its details are reduced. The code used to compute this is shown below.

```
def Convolve(image, Mask):
    #status: complete
    #task: perform Convolution or CrossCorrelation
    #parameters: image, Mask=kernel
    #returns: convolved image

    (init_x_size, init_y_size)=image.shape
    x_pad, y_pad, image, Mask, max_pad=adjustInputs(image, Mask, True)
       output_image= CommonUtil(image, init_x_size, init_y_size, max_pad, x_pad,
y_pad, Mask, False)
    return output_image

def adjustInputs(image, Mask, opType):
    #status: complete
    #task: adjust inputs by padding
    #parameters: image, Mask=kernel,
    #returns: x pad size, y pad size, Mask

    (init_x_size, init_y_size)=image.shape
    Mask=np.flipud(np.fliplr(Mask))

    #output_size= (input_size- filter_size + 2*Padding_size)+1
     #if input and output size are same: then Padding_size=(filter_size-1)/2 (ceil
it)

    x_pad_size=math.ceil((Mask.shape[0]-1)/2)
    y_pad_size=math.ceil((Mask.shape[1]-1)/2)
```

```
    pad_sizes=[x_pad_size, y_pad_size]
    image=np.pad(image, (max(pad_sizes)), mode='constant')

    return x_pad_size, y_pad_size, image, Mask, max(pad_sizes)


def  CommonUtil(image,  init_x_size,  init_y_size,  max_pad,  x_pad,  y_pad,  Mask,
isGrad):
    #status: complete
    #task: utility function to multiply kernel to image patch
     #parameters: image, x size of image, y size of image, maximum padding size, x
pad size, y pad size, Mask to convolve, isGrad= if the process is gradient finding
or just normal convolution
    #output: adjusted image

    if(isGrad):
        grad_output_image= np.zeros((init_x_size+2*y_pad, init_y_size+2*x_pad))
    else:
        grad_output_image= np.zeros((init_x_size, init_y_size))

    out_row=0
    out_col=0

    for x_pivot in range(x_pad, init_x_size+x_pad):
        out_col=0
        for y_pivot in range(y_pad, init_y_size+y_pad):
                  patch=image[x_pivot-max_pad:  x_pivot+max_pad+1,  y_pivot-max_pad:
y_pivot+max_pad+1]
            grad_output_image[out_row][out_col] = np.sum(np.multiply(Mask, patch))
            out_col+=1
        out_row+=1
    return grad_output_image
```

In the above code, there are three functions. The ‘Convolve’ function is used as the base function to convolve the image using the Gaussian filter created above. The parameters passed through this function are the input image and the gaussian filter (Mask). The image has to be further padded in order to get the output image to have the same as the input image. This is done using the function ‘adjustInputs’ function. Here, the x padding size, y padding size, and the padded image are returned.

Further, ‘CommonUtil’ function is used to multiply image pixel-wise with the corresponding filter value. In every epoch, the patch of the image equal to the size of the filter is extracted and is passed into this function along with the Gaussian filter. The output of this function is the convoluted image patch.

**2.13. Compute derivative images (with respect to x and with respect to y) using the separable derivative filter.**

The preprocessing pipeline many times includes the edge detection techniques in which the information in between the edges becomes useless. In such cases, we must find the derivative of the image. In this illustration, we use the Prewitt filter. It is shown below.

| | | |
|---|---|---|
| -1 | 0 | 1 |
| -1 | 0 | 1 |
| -1 | 0 | 1 |

$h_x$

| | | |
|---|---|---|
| -1 | -1 | -1 |
| 0 | 0 | 0 |
| 1 | 1 | 1 |

$h_y$

**Prewitt filters hx and hy to find vertical and horizontal edges**

The x derivative (gx) and y derivative (gy) of the image are found by convolving these filters over the image. The output is shown in figure 2.12. Using these gradient images, the magnitude of the image is found. That means the intersection of the vertical and the horizontal paths are found. This, on the other hand, irrespective of direction, gives the image that says where the edges in the image occur i.e. where there is the sudden change in the pixel intensity occurs is visualized via this process. The magnitude is found using equation 2.13.

$$magnitude = \sqrt{gx^2 + gy^2}$$

equation 2.13

The code to run this process is also shown below. The 'FindDerivativeImages' function is the base of finding the derivatives of the image by convolving the x and y Prewitt filters over it.

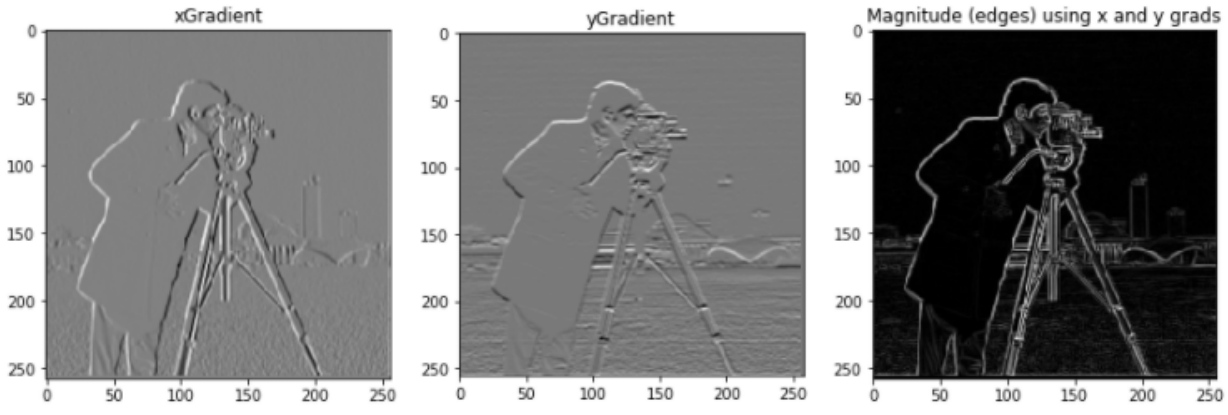


**Figure 2.12: xGradient, yGradient, Magnitude of the input image after convolving with the Prewitt filter**

### 2.131. Creating binary edge images from the gradient magnitude image using several thresholds

The image can also be binarized in order to get the proper edges in the output. In this illustration, we use several thresholds in order to binarize the magnitude output. The threshold array used is [1, 2, 0.5].

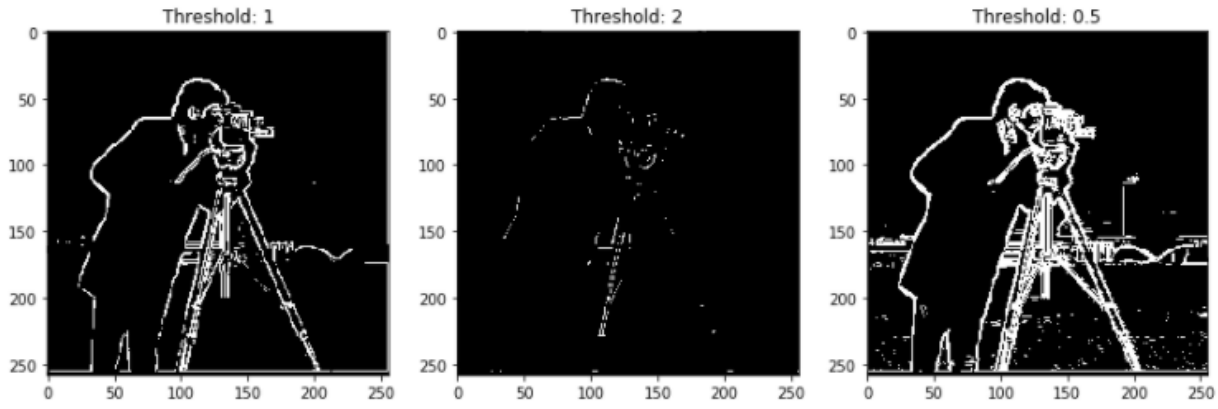


**Figure: 2.13: Binary image output with different thresholds**

```
def FindDerivativeImages(image, xDerivativeMask, yDerivativeMask, thresholds):
    #status: complete
    #task: finds x and y derivatives, magnitude of the derivatives of the image
    #parameters: image, xDerivative, yDerivative
    #returns: xgradient, ygradient, magnitude

    init_x_size, init_y_size=image.shape
    x_pad, y_pad, ximage, xDerivativeMask, max_pad=adjustInputs(image,
xDerivativeMask, True)
    xgrad_output_image = CommonUtil(ximage, init_x_size, init_y_size, max_pad,
x_pad, y_pad, xDerivativeMask, True)

    x_pad, y_pad, yimage, yDerivativeMask, max_pad=adjustInputs(image,
yDerivativeMask, True)
    ygrad_output_image = CommonUtil(yimage, init_x_size, init_y_size, max_pad,
x_pad, y_pad, yDerivativeMask, True)

    joined_filter=np.multiply(xDerivativeMask, yDerivativeMask)
    grad_output_image = CommonUtil(yimage, init_x_size, init_y_size, max_pad,
x_pad, y_pad, yDerivativeMask, True)

    magnitude=findMagnitude(xgrad_output_image, ygrad_output_image, [], False)
```

```
    magnitudes_by_thresholds=findMagnitude(xgrad_output_image, ygrad_output_image,
thresholds, True)

    return xgrad_output_image, ygrad_output_image, magnitude,
magnitudes_by_thresholds


def adjustInputs(image, Mask, opType):
    #status: complete
    #task: adjust inputs by padding
    #parameters: image, Mask=kernel, opType= True if Convolution False if
CrossCorrelation
    #returns: x pad size, y pad size, Mask

    (init_x_size, init_y_size)=image.shape
    if(opType==True):
        Mask=np.flipud(np.fliplr(Mask))

    #output_size= (input_size- filter_size + 2*Padding_size)+1
    #if input and output size are same: then Padding_size=(filter_size-1)/2 (ceil
it)

    x_pad_size=math.ceil((Mask.shape[0]-1)/2)
    y_pad_size=math.ceil((Mask.shape[1]-1)/2)
    pad_sizes=[x_pad_size, y_pad_size]
    image=np.pad(image, (max(pad_sizes)), mode='constant')

    return x_pad_size, y_pad_size, image, Mask, max(pad_sizes)
```

In this case, the 'FindDerivativeImage' function takes in the parameters of Image, x Prewitt mask, y Prewitt mask, and threshold array. First, the image is adjusted to be padded using the function 'adjustInputs'. 'CommonUtil' is the function that is used to x_grad_output_image, y_grad_output_image contains the xgradient, ygradient images.

```
def CommonUtil(image, init_x_size, init_y_size, max_pad, x_pad, y_pad, Mask,
isGrad):
    #status: complete
    #task: utility function to multiply kernel to image patch
    #output: adjusted image

    if(isGrad):
        grad_output_image= np.zeros((init_x_size+2*y_pad, init_y_size+2*x_pad))
    else:
```

```
        grad_output_image= np.zeros((init_x_size, init_y_size))

    out_row=0
    out_col=0

    for x_pivot in range(x_pad, init_x_size+x_pad):
        out_col=0
        for y_pivot in range(y_pad, init_y_size+y_pad):
            patch=image[x_pivot-max_pad: x_pivot+max_pad+1, y_pivot-max_pad:
y_pivot+max_pad+1]
            grad_output_image[out_row][out_col] = np.sum(np.multiply(Mask, patch))
            out_col+=1
        out_row+=1
    return grad_output_image
```

Further, when the gradient images are returned from that function, then the magnitudes of the convolved image with different thresholds are found using the function 'findMagnitude'. Then the magnitude of the image is returned by using proper thresholds from that function that gives viable edges of the image using it's gradient images.

```
def findMagnitude(xGrad, yGrad, thresholds, useThresholds):
    #status: complete
    #params: xGrad, yGrad
    #returns: magnitude using the x and y gradients

    if(useThresholds):
        magnitudes=[]
        for i in range(len(thresholds)):
            magnitudes.append(np.zeros(xGrad.shape, np.float32))

        x_len=xGrad.shape[1]
        y_len=yGrad.shape[0]

        thresCount=0
        for mag in magnitudes:
            for i in range(x_len):
                for j in range(y_len):
                    temp=np.sqrt(xGrad[i][j]**2 + yGrad[i][j]**2)%255
                    if(temp>=thresholds[thresCount]):
                        mag[i][j]=255
                    else:
                        mag[i][j]=0
            thresCount+=1
        return magnitudes
```

```
    else:
        mags=np.zeros(xGrad.shape, np.float32)
        x_len=xGrad.shape[1]
        y_len=yGrad.shape[0]
        for i in range(x_len):
            for j in range(y_len):
                mags[i][j]=np.sqrt(xGrad[i][j]**2 + yGrad[i][j]**2)%255
        return mags
```

(Note: The main function calls these functions. This can be seen in the **References** section in this work where whole code has been put up)

### 2.2 Cross-correlation and Template Matching

In the contemporary world, computer vision plays a major role in providing security and maintenance by multiple ways like image classification, pattern matching, template matching etc. The process of template matching is one of the important methodologies in computer vision to find out if the particular patch of the image is present in it or not. This is done by using the image patch (of the same width and height that has to be present in the image and hovering it over the image to find out the cross correlation of it with the corresponding image patches as it hovers over the image.

In this illustration, we try to analyse if in the given set of keys, if the particular key is present or not by using this process. The set of keys and the key to be found is shown in the figure 2.21.

**Figure 2.21: Set of keys and the Image patch (template) key**

The image with the collection of keys is of the size (460, 612) whereas the image with the template key is of the size: (235, 108). Firstly, the input image and the template image are binarized using the function shown below.

```
def binarize(image, isTemplate):
    #status: complete
    #task: Convert image to binary image using the threshold of 200
    #params: image, isTemplate: if it is template or not (in case of cross
correlation)
    #output: binarized image
    image=image*255
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            if(image[i][j]<200):
                image[i][j]=1
            else:
                if(isTemplate):
                    image[i][j]=-1
                else:
                    image[i][j]=0
    return image
```

The threshold of 200 is used in-order to determine the value (0 or 1) to be assigned to the pixels in the image. The output when the image and the template are binarized are shown below in the figure 2.22. In the function two parameters are passed. One is the image and the second if it's the template image or not. If it is the template image, then unlike the pixel value assigned as 1 in the input image image if the pixel value crosses the threshold of 200, we assign -1. This is done to make the process easier.

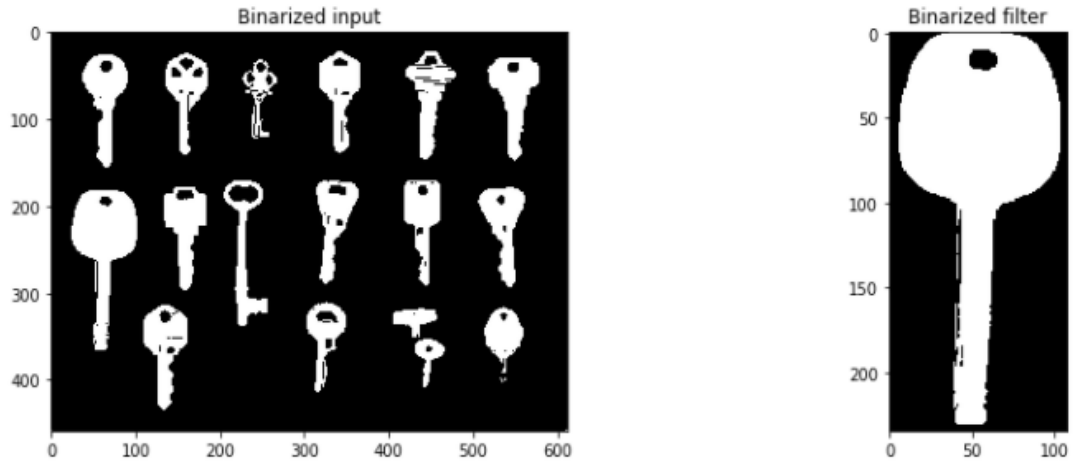


**Figure 2.22: Binarized input image and the template image.**

Further, the padding of the input image is done using the function 'adjustInputs'. This function takes three parameters - Image, Mask, opType. opType is the parameter that is assigned False here because it is cross correlation. This is because based on this variable itself, the filter(template) is flipped/not flipped before padding.

```
def adjustInputs(image, Mask, opType):
    #status: complete
    #task: adjust inputs by padding
    #parameters: image, Mask=kernel, opType= True if Convolution False if
CrossCorrelation
    #returns: x pad size, y pad size, Mask
    #output_size= (input_size- filter_size + 2*Padding_size)+1
    #if input and output size are same: then Padding_size=(filter_size-1)/2 (ceil
it)

    (init_x_size, init_y_size)=image.shape
    if(opType==True):
        Mask=np.flipud(np.fliplr(Mask))

    x_pad_size=math.ceil((Mask.shape[0]-1)/2)
    y_pad_size=math.ceil((Mask.shape[1]-1)/2)
    pad_sizes=[x_pad_size, y_pad_size]
    image=np.pad(image, (max(pad_sizes)), mode='constant')

    return x_pad_size, y_pad_size, image, Mask, max(pad_sizes)
```

Further, the cross correlation is applied over the padded input image using the template. The cross correlation operation is performed using the equation 2.21. The padded input and the marked output is shown in the figure 2.23. The output is marked where the intensity is peaked. There are various intensity peaks in the output image but they are local peaks. Therefore, the peak suggests that the template matches the patch of the image.

$$I'(X,Y) = \sum_{j=-k}^{k} \sum_{i=-k}^{k} F(i,j)\, I\,(X+i, Y+j)$$

**equation 2.21**

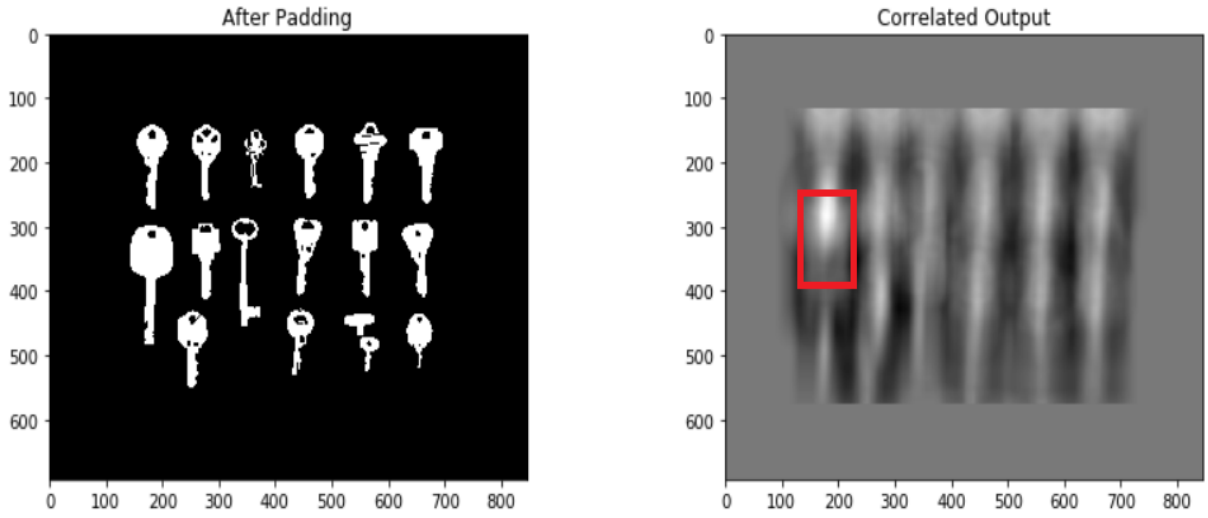


**Figure 2.23: Padded input and the correlated output.**

```
def correlate(image, template):
    #status: complete
    #params: image, template
    #returns: correlated image

    (init_x_size, init_y_size)=image.shape
    x_pad, y_pad, image, Mask, max_pad=adjustInputs(image, template)
    grad_output_image=np.zeros((image.shape))
    x, y=(0, 0)
    maxim=-1000
    for x_pivot in range(0, image.shape[0]-Mask.shape[0]):
        for y_pivot in range(0, image.shape[1]-Mask.shape[1]):
            patch=image[x_pivot: x_pivot+Mask.shape[0], y_pivot:
y_pivot+Mask.shape[1]]
            temp=np.sum(np.multiply(patch, Mask))
            if(maxim<=temp):
                maxim=temp
                x, y=(x_pivot+(Mask.shape[0])//2, y_pivot+Mask.shape[1]//2)
            grad_output_image[x_pivot+(Mask.shape[0])//2][y_pivot+Mask.shape[1]//2]
= temp
    return grad_output_image, x, y, image
```

## 2.3 Image Panorama

The process of Panorama formation is nothing but stitching the two images by using the Direct Linear Transformation method. The given input images differ by rotation only. The images used in the process is shown in figure 2.31.

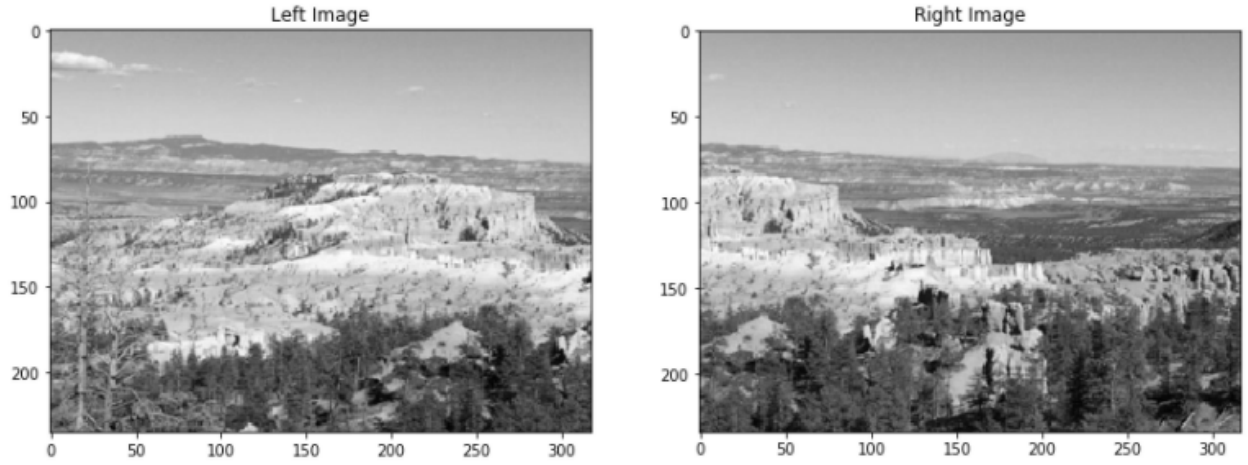


**Figure 2.31 Input images in panorama process**

The image panorama process is carried out by feature mapping. The linear transformation matrix P is formed using the 6 parameters p11, p12, p13, p21, p22, p23, p31, p32, p33. These parameters are found by using the feature points taken from the left and the right image. These are saved as the coordinates in the L_arr, R_arr arrays. Features parameters contain a number of features to be used. Here, in this illustration, outputs for 4, 5, 6, 7 number of features are used. This is done by using "CreateAffine" function.

Linear transformation with matrix P

$$\bar{x}^* = P\bar{x} \qquad P = \begin{pmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & 1 \end{pmatrix} \qquad \begin{aligned} x^* &= p_{11}x + p_{12}y + p_{13} \\ y^* &= p_{21}x + p_{22}y + p_{23} \\ z^* &= p_{31}x + p_{32}y + 1 \end{aligned}$$

Perspective equivalence

$$\begin{aligned} x' &= \frac{p_{11}x + p_{12}y + p_{13}}{p_{31}x + p_{32}y + 1} \\ y' &= \frac{p_{21}x + p_{22}y + p_{23}}{p_{31}x + p_{32}y + 1} \end{aligned}$$

Multiply by denominator and reorganize terms

$$\begin{aligned} p_{31}xx' + p_{32}yx' - p_{11}x - p_{12}y - p_{13} &= -x' \\ p_{31}xy' + p_{32}yy' - p_{21}x - p_{22}y - p_{23} &= -y' \end{aligned}$$

Linear system, solve for P

$$\begin{pmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1x'_1 & y_1x'_1 \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2x'_2 & y_2x'_2 \\ & & & \vdots & & & & \\ -x_N & -y_N & -1 & 0 & 0 & 0 & x_Nx'_N & y_Nx'_2 \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1y'_1 & y_1y'_1 \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2y'_2 & y_2y'_2 \\ & & & \vdots & & & & \\ 0 & 0 & 0 & -x_N & -y_N & -1 & x_Ny'_N & y_Ny'_N \end{pmatrix} \begin{pmatrix} p_{11} \\ p_{12} \\ p_{13} \\ p_{21} \\ p_{23} \\ p_{23} \\ p_{31} \\ p_{32} \end{pmatrix} = \begin{pmatrix} -x'_1 \\ -x'_2 \\ \vdots \\ -x'_N \\ -y'_1 \\ -y'_2 \\ \vdots \\ -y'_N \end{pmatrix}$$

```
def CreateAffine(L_arr, R_arr, features):
    #status: complete
    #Task: create affine transformation matrix using the fetaures arrays
(co-ordinates of left and right image)
    #parameters: Left image co-ordinates, Right image co-ordinates, features
    #return Transformation matrix, inverse transformation matrix
    row=0
    col=0
    X=[]
    for i in range(features):
        temp=[-(L_arr[i][0]), -(L_arr[i][1]), -1, 0, 0, 0,
(L_arr[i][0])*(R_arr[i][0]), (L_arr[i][1])*(R_arr[i][0])]
        X.append(temp)
    for i in range(features):
        temp=[0, 0, 0, -(L_arr[i][0]), -(L_arr[i][1]), -1,
(L_arr[i][0])*(R_arr[i][1]), (L_arr[i][1])*(R_arr[i][1])]
        X.append(temp)
    X=np.array(X)

    XDash=[]
    for i in range(features):
        XDash.append(R_arr[i][0])
    for i in range(features):
        XDash.append(R_arr[i][1])
    XDash=np.array(XDash)

    params=np.linalg.lstsq(X, XDash)[0]
    params=np.append(params, 1)
    T=params.reshape(3, 3)
    Tinv = np.linalg.inv(T)

    return T, Tinv
```

To form the panorama image, the output canvas is first created and then, the transformation is applied over the left image. The transformed output of the left image is first pasted on the canvas. This is done using the function - “adjustments”. Then, the right image is added over the output image in the overlapped position by replacing the gray values.

```
def adjustments(im1, T):
    # Status: complete
    # Task: Create output canvas using the left image and the Transformation matrix
    # Return: min rows, max rows, min col, max col, and size (rows, columns) of
output canvas

    rows, cols=(im1.shape)
```

```
    # Top left corner
    top_left = T.dot(np.array([0.0, 0.0, 1.0], float))
    top_left = top_left // top_left[2]

    # Bottom left corner
    bottom_left = T.dot(np.array([rows-1.0, 0.0, 1.0], float))
    bottom_left = bottom_left // bottom_left[2]

    # Top right corner
    top_right = T.dot(np.array([0.0, cols-1.0, 1.0], float))
    top_right = top_right // top_right[2]

    # Bottom right corner
    bottom_right = T.dot(np.array([rows-1.0, cols-1.0, 1.0], float))
    bottom_right = bottom_right // bottom_right[2]

    # Calculate tight bounding box around the transformed corners
    min_rows = np.min([top_left[0], bottom_left[0], top_right[0], bottom_right[0]])
    max_rows = np.max([top_left[0], bottom_left[0], top_right[0], bottom_right[0]])

    min_cols = np.min([top_left[1], bottom_left[1], top_right[1], bottom_right[1]])
    max_cols = np.max([top_left[1], bottom_left[1], top_right[1], bottom_right[1]])

    out_sampling_rows = np.linspace(math.floor(min_rows), math.ceil(max_rows),
math.ceil(max_rows) - math.floor(min_rows)+1)
    out_sampling_cols = np.linspace(math.floor(min_cols), math.ceil(max_cols),
math.ceil(max_cols) - math.floor(min_cols)+1)

    return min_rows, max_rows, min_cols, max_cols, out_sampling_cols,
out_sampling_rows
```

The panorama image formed by using the function “Panorama”. This is done by stitching the transformed left image over the canvas and the right image beside the transformed left image.

```
def Panorama(im1, im2, Tinv, min_rows, max_rows, min_cols, max_cols,
out_sampling_cols, out_sampling_rows):
    #status: Complete
    #task: stitch the transformed left image over the canvas and the right image
beside the transformed left image.
    #return: input images, panorama image.

    rows, cols=(im1.shape[0], im1.shape[1])
    out_im2 = np.zeros((len(out_sampling_rows), im2.shape[1]+im1.shape[1]))

    im1=np.flipud(np.fliplr(im1))
```

```
    c_1=out_sampling_cols.shape[0]
    r_1=out_sampling_rows.shape[0]
    for cur_row in range(0, len(out_sampling_rows)):
        for cur_col in range(0, len(out_sampling_cols)):
            cur_pt = np.array([out_sampling_rows[cur_row],
out_sampling_cols[cur_col], 1.0])
            transformed_pt = Tinv.dot(cur_pt)
            transformed_pt = transformed_pt / transformed_pt[2]
            if (np.floor(transformed_pt[0]) < 0 or np.ceil(transformed_pt[0]) >=
rows or \
                np.floor(transformed_pt[1]) < 0 or np.ceil(transformed_pt[1]) >=
cols):
                continue
            new_row = int(round(transformed_pt[0]))
            new_col = int(round(transformed_pt[1]))
            out_im2[cur_row, cur_col] = im1[new_row, new_col]
            c_1-=1
        r_1-=1

    start_row=im2.shape[0]
    start_col=im2.shape[1]
    c_1, r_1=(0, 0)
    rows, cols=(out_im2.shape[0], out_im2.shape[1])

    dif_cols=out_sampling_cols.shape[0]-im1.shape[1]
    dif_rows=out_sampling_rows.shape[0]-im1.shape[0]

    for i in range(int(round(out_im2.shape[0]-im1.shape[0])/2),
dif_rows+im2.shape[0]):
        c_1=0
        for j in range(cols-start_col-dif_cols, cols-1):
            if(c_1>=im2.shape[1] or r_1>=im2.shape[0]):
                continue
            out_im2[i][j]=im2[r_1][c_1]
            c_1+=1
        r_1+=1

    return im1, im2, out_im2
```

The outputs formed by using 4, 5, 6, and 7 features are shown in the figure 2.32. Here, we can see that higher the number of features used, more the accurate outputs we get. This can be seen in the figure 2.33.

Number of feature points = 7

Number of feature points =6

Number of feature points = 5

Number of feature points =4

**Figure 2.32: Output images with 4, 5, 6, 7 feature points respectively.**

## 3. Reference - Complete Python Code.

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import math
from skimage import io
from skimage.color import rgb2gray
import matplotlib.patches as patches

def importImage(path):
    #status:complete
    #task: imports image, converts it into grayscale and returns image, #rows,
#cols
    #parameter: path
    #returns: grayscaled image

    im = io.imread(path)
    im = rgb2gray(im)
    im = ((im - np.min(im)) * (1/(np.max(im) - np.min(im)) *1.0)).astype('float')
    return im

def createGaussianFilter(n, stdev):
    #status: complete
    #task: creates a square gaussian filter
    #parameters: n= rows and column count, stdev= standard deviation
    #returns: gaussian kernel of size (n,n) and standard dev of stdev

    variance=stdev*stdev #variance
    gaussian_filter=np.zeros((n, n))
    x=np.linspace(-n/2.0, n/2.0, n)
    y=np.linspace(-n/2.0, n/2.0, n)

    for i in range(0, n):
        for j in range(0, n):
            gaussian_filter[i,
j]=(1/math.sqrt((2*math.pi*variance)))*math.exp(-(x[i]*x[i]+y[j]*y[j])/(2*variance)
)
    return gaussian_filter/np.sum(gaussian_filter.flatten())


def adjustInputs(image, Mask, opType):
    #status: complete
    #task: adjust inputs by padding
    #parameters: image, Mask=kernel, opType= True if Convolution False if
CrossCorrelation
```

```
    #returns: x pad size, y pad size, Mask
    #output_size= (input_size- filter_size + 2*Padding_size)+1
    #if input and output size are same: then Padding_size=(filter_size-1)/2 (ceil
it)

    (init_x_size, init_y_size)=image.shape
    if(opType==True):
        Mask=np.flipud(np.fliplr(Mask))

    x_pad_size=math.ceil((Mask.shape[0]-1)/2)
    y_pad_size=math.ceil((Mask.shape[1]-1)/2)
    pad_sizes=[x_pad_size, y_pad_size]
    image=np.pad(image, (max(pad_sizes)), mode='constant')

    return x_pad_size, y_pad_size, image, Mask, max(pad_sizes)


def FindDerivativeImages(image, xDerivativeMask, yDerivativeMask, thresholds):
    #status: complete
    #task: finds x and y derivatives, magnitude of the derivatives of the image
    #parameters: image, xDerivative, yDerivative
    #returns: xgradient, ygradient, magnitude

    init_x_size, init_y_size=image.shape

    x_pad, y_pad, ximage, xDerivativeMask, max_pad=adjustInputs(image,
xDerivativeMask, True)
    xgrad_output_image = CommonUtil(ximage, init_x_size, init_y_size, max_pad,
x_pad, y_pad, xDerivativeMask, True)

    x_pad, y_pad, yimage, yDerivativeMask, max_pad=adjustInputs(image,
yDerivativeMask, True)
    ygrad_output_image = CommonUtil(yimage, init_x_size, init_y_size, max_pad,
x_pad, y_pad, yDerivativeMask, True)

    joined_filter=np.multiply(xDerivativeMask, yDerivativeMask)
    grad_output_image = CommonUtil(yimage, init_x_size, init_y_size, max_pad,
x_pad, y_pad, yDerivativeMask, True)

    magnitude=findMagnitude(xgrad_output_image, ygrad_output_image, [], False)
    magnitudes_by_thresholds=findMagnitude(xgrad_output_image, ygrad_output_image,
thresholds, True)

    return xgrad_output_image, ygrad_output_image, magnitude,
magnitudes_by_thresholds
```

```
def CommonUtil(image, init_x_size, init_y_size, max_pad, x_pad, y_pad, Mask,
isGrad):
    #status: complete
    #task: utility function to multiply kernel to image patch
    #output: adjusted image

    if(isGrad):
        grad_output_image= np.zeros((init_x_size+2*y_pad, init_y_size+2*x_pad))
    else:
        grad_output_image= np.zeros((init_x_size, init_y_size))

    out_row=0
    out_col=0

    for x_pivot in range(x_pad, init_x_size+x_pad):
        out_col=0
        for y_pivot in range(y_pad, init_y_size+y_pad):
            patch=image[x_pivot-max_pad: x_pivot+max_pad+1, y_pivot-max_pad:
y_pivot+max_pad+1]
            grad_output_image[out_row][out_col] = np.sum(np.multiply(Mask, patch))
            out_col+=1
        out_row+=1
    return grad_output_image


def Convolve(image, Mask, opType):
    #status: complete
    #task: perform Convolution or CrossCorrelation
    #parameters: image, Mask=kernel, opType= True if Convolution False if
CrossCorrelation
    #returns: convolved image

    (init_x_size, init_y_size)=image.shape
    x_pad, y_pad, image, Mask, max_pad=adjustInputs(image, Mask, True)
    output_image= CommonUtil(image, init_x_size, init_y_size, max_pad, x_pad,
y_pad, Mask, False)
    return output_image


def findMagnitude(xGrad, yGrad, thresholds, useThresholds):
    #status: complete
    #params: xGrad, yGrad
    #returns: magnitude using the x and y gradients

    if(useThresholds):
        magnitudes=[]
        for i in range(len(thresholds)):
```

```
            magnitudes.append(np.zeros(xGrad.shape, np.float32))

        x_len=xGrad.shape[1]
        y_len=yGrad.shape[0]

        thresCount=0
        for mag in magnitudes:
            for i in range(x_len):
                for j in range(y_len):
                    temp=np.sqrt(xGrad[i][j]**2 + yGrad[i][j]**2)%255
                    if(temp>=thresholds[thresCount]):
                        mag[i][j]=255
                    else:
                        mag[i][j]=0
            thresCount+=1
        return magnitudes

    else:
        mags=np.zeros(xGrad.shape, np.float32)
        x_len=xGrad.shape[1]
        y_len=yGrad.shape[0]
        for i in range(x_len):
            for j in range(y_len):
                mags[i][j]=np.sqrt(xGrad[i][j]**2 + yGrad[i][j]**2)%255
        return mags


def correlate(image, template):
    #status: complete
    #params: image, template
    #returns: correlated image

    (init_x_size, init_y_size)=image.shape
    x_pad, y_pad, image, Mask, max_pad=adjustInputs(image, template, True)

    grad_output_image=np.zeros((image.shape))
    x, y=(0, 0)
    maxim=-1000
    for x_pivot in range(0, image.shape[0]-Mask.shape[0]):
        for y_pivot in range(0, image.shape[1]-Mask.shape[1]):
            patch=image[x_pivot: x_pivot+Mask.shape[0], y_pivot:
y_pivot+Mask.shape[1]]
            temp=np.sum(np.multiply(patch, Mask))
            if(maxim<=temp):
                maxim=temp
                x, y=(x_pivot+(Mask.shape[0])//2, y_pivot+Mask.shape[1]//2)
```

```
            grad_output_image[x_pivot+(Mask.shape[0])//2][y_pivot+Mask.shape[1]//2]
= temp
    return grad_output_image, x, y, image


def displayCorner(nRows, nCols, imgArray, nameArray):
    #status: complete
    #params: nrows, ncols, images array, names of the images,
    #returns: subplots of given format

    c = 1  # initialize plot counter
    fig = plt.figure(figsize=(14,10))
    for i in imgArray:
        plt.subplot(nRows, nCols, c)
        plt.title(nameArray[c-1])
        plt.imshow(imgArray[c-1], cmap='gray')
        c = c + 1
    plt.show()


def binarize(image, isTemplate):
    #status: complete
    #task: Convert image to binary image using the threshold of 200
    #params: image, isTemplate: if it is template or not (in case of cross
correlation)
    #output: binarized image

    image=image*255
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            if(image[i][j]<200):
                image[i][j]=1
            else:
                if(isTemplate):
                    image[i][j]=-1
                else:
                    image[i][j]=0
    return image


def CreateAffine(L_arr, R_arr, features):
    #status: complete
    #Task: create affine transformation matrix using the fetaures arrays
(co-ordinates of left and right image)
    #parameters: Left image co-ordinates, Right image co-ordinates, features
    #return Transformation matrix, inverse transformation matrix
    row=0
```

```
    col=0
    X=[]
    for i in range(features):
        temp=[-(L_arr[i][0]), -(L_arr[i][1]), -1, 0, 0, 0,
(L_arr[i][0])*(R_arr[i][0]), (L_arr[i][1])*(R_arr[i][0])]
        X.append(temp)
    for i in range(features):
        temp=[0, 0, 0, -(L_arr[i][0]), -(L_arr[i][1]), -1,
(L_arr[i][0])*(R_arr[i][1]), (L_arr[i][1])*(R_arr[i][1])]
        X.append(temp)
    X=np.array(X)

    XDash=[]
    for i in range(features):
        XDash.append(R_arr[i][0])
    for i in range(features):
        XDash.append(R_arr[i][1])
    XDash=np.array(XDash)

    params=np.linalg.lstsq(X, XDash)[0]
    params=np.append(params, 1)
    T=params.reshape(3, 3)
    Tinv = np.linalg.inv(T)

    return T, Tinv


def adjustments(im1, T):
    rows, cols=(im1.shape)
    # Top left corner
    top_left = T.dot(np.array([0.0, 0.0, 1.0], float))
    top_left = top_left // top_left[2]

    # Bottom left corner
    bottom_left = T.dot(np.array([rows-1.0, 0.0, 1.0], float))
    bottom_left = bottom_left // bottom_left[2]

    # Top right corner
    top_right = T.dot(np.array([0.0, cols-1.0, 1.0], float))
    top_right = top_right // top_right[2]

    # Bottom right corner
    bottom_right = T.dot(np.array([rows-1.0, cols-1.0, 1.0], float))
    bottom_right = bottom_right // bottom_right[2]

    # Calculate tight bounding box around the transformed corners
    min_rows = np.min([top_left[0], bottom_left[0], top_right[0], bottom_right[0]])
```

```
    max_rows = np.max([top_left[0], bottom_left[0], top_right[0], bottom_right[0]])

    min_cols = np.min([top_left[1], bottom_left[1], top_right[1], bottom_right[1]])
    max_cols = np.max([top_left[1], bottom_left[1], top_right[1], bottom_right[1]])

    out_sampling_rows = np.linspace(math.floor(min_rows), math.ceil(max_rows),
math.ceil(max_rows) - math.floor(min_rows)+1)
    out_sampling_cols = np.linspace(math.floor(min_cols), math.ceil(max_cols),
math.ceil(max_cols) - math.floor(min_cols)+1)

    return min_rows, max_rows, min_cols, max_cols, out_sampling_cols,
out_sampling_rows


def Panorama(im1, im2, Tinv, min_rows, max_rows, min_cols, max_cols,
out_sampling_cols, out_sampling_rows):
    #status: Complete
    #task: stitch the transformed left image over the canvas and the right image
beside the transformed left image.
    #return: input images, panorama image

    rows, cols=(im1.shape[0], im1.shape[1])
    out_im2 = np.zeros((len(out_sampling_rows), im2.shape[1]+im1.shape[1]))

    im1=np.flipud(np.fliplr(im1))

    c_1=out_sampling_cols.shape[0]
    r_1=out_sampling_rows.shape[0]
    for cur_row in range(0, len(out_sampling_rows)):
        for cur_col in range(0, len(out_sampling_cols)):
            cur_pt = np.array([out_sampling_rows[cur_row],
out_sampling_cols[cur_col], 1.0])
            transformed_pt = Tinv.dot(cur_pt)
            transformed_pt = transformed_pt / transformed_pt[2]
            if (np.floor(transformed_pt[0]) < 0 or np.ceil(transformed_pt[0]) >=
rows or \
                np.floor(transformed_pt[1]) < 0 or np.ceil(transformed_pt[1]) >=
cols):
                continue
            new_row = int(round(transformed_pt[0]))
            new_col = int(round(transformed_pt[1]))
            out_im2[cur_row, cur_col] = im1[new_row, new_col]
            c_1-=1
        r_1-=1

    start_row=im2.shape[0]
    start_col=im2.shape[1]
```

```
    c_1, r_1=(0, 0)
    rows, cols=(out_im2.shape[0], out_im2.shape[1])

    dif_cols=out_sampling_cols.shape[0]-im1.shape[1]
    dif_rows=out_sampling_rows.shape[0]-im1.shape[0]

    for i in range(int(round(out_im2.shape[0]-im1.shape[0])/2),
dif_rows+im2.shape[0]):
        c_1=0
        for j in range(cols-start_col-dif_cols, cols-1):
            if(c_1>=im2.shape[1] or r_1>=im2.shape[0]):
                continue
            out_im2[i][j]=im2[r_1][c_1]
            c_1+=1
        r_1+=1

    return im1, im2, out_im2


def main(cross_corr_paths, conv_image_path, filter_size, im_path, features,
n_features):
    #status: incomplete
    #params: conv_image_path= image path for convolution, filter_size= filter size

    #1. Convolution
    image=importImage(conv_image_path)
    gaussian_filter=createGaussianFilter(filter_size, 2) #stdev assumed 20
    Gaussian_conv=Convolve(image, gaussian_filter, True)
    y_prewitt_filter=(np.array([[-1, 0, 1]]).T)
    x_prewitt_filter=np.array([[-1, 0, 1]])
    thresholds= [1, 2, 0.5]
    xGradImage, yGradImage, magnitude, binary_mags=FindDerivativeImages(image,
x_prewitt_filter, y_prewitt_filter, thresholds)
    image_list=[image, gaussian_filter, Gaussian_conv, xGradImage, yGradImage,
magnitude]
    names_list=["Image Imported", "Gaussian Filter of size: ("+str(filter_size)+" ,
"+str(filter_size)+")", "Output Image", "xGradient", "yGradient", "Magnitude
(edges) using x and y grads"]
    displayCorner(2, 3, image_list, names_list)
    names_list=["Threshold: 1","Threshold: 2","Threshold: 0.5"]
    displayCorner(2, 3, binary_mags, names_list)


    #2. Cross Correlation
    image_corr=importImage(cross_corr_paths[0])
    image_corr=binarize(image_corr, False)
    filter_image=importImage(cross_corr_paths[1])
```

```
    filter_image=binarize(filter_image, True)
    out_image, x, y, in_image=correlate(image_corr, filter_image)
    displayCorner(2, 2, [image_corr, filter_image, in_image, out_image],
["Binarized input", "Binarized filter", "After Padding", "Correlated Output"])
    image = cv2.rectangle(out_image, (x, y), (x-2*filter_image.shape[1],
y+2*filter_image.shape[0]), (255, 0, 0), 2)
    plt.imshow(image, cmap="gray")


    #3. Image Panorama
    im_path1 = im_path[0]
    im_path2 = im_path[1]
    im1, im2=(importImage(im_path1), importImage(im_path2))
    Left_img_features=features[0]
    Right_img_features=features[1]
    T, Tinv=CreateAffine(Left_img_features, Right_img_features, n_features)
    min_rows, max_rows, min_cols, max_cols, out_sampling_cols, out_sampling_rows=
adjustments(im1, T)
    im1, im2, out_im= Panorama(im1, im2, Tinv, min_rows, max_rows, min_cols,
max_cols, out_sampling_cols, out_sampling_rows)
    displayCorner(1, 2, [im1, im2], ["Right image", "Left Image"])
    plt.imshow(out_im, cmap="gray")
    plt.title("Panorama Image")


if __name__=="__main__":
    im_path1 =
'C:\\Users\\pruth\\Desktop\\NYU\\Year-1\\Spring\\CV\\Assignments_notebooks\\m1.png'
    im_path2 =
'C:\\Users\\pruth\\Desktop\\NYU\\Year-1\\Spring\\CV\\Assignments_notebooks\\m2.png'
    Left_img_features=[(214, 35), (297, 182), (239, 170), (284, 98), (292,
109),(286, 131), (269, 141), (318, 79), (255, 111),]
    Right_img_features=[(8, 26), (95, 175), (38, 165), (81, 90), (89, 102), (82,
124), (66, 134), (1, 68), (53, 106)]
    n_features=4
    main(["multiplekeys.png", "key_2.png"],"cameraman.png", 21, [im_path1,
im_path2], [Left_img_features, Right_img_features], n_features)
```