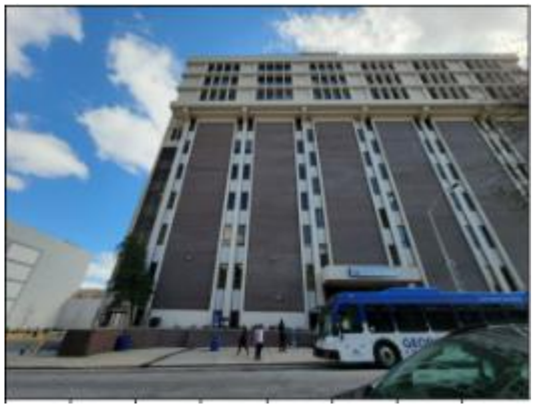
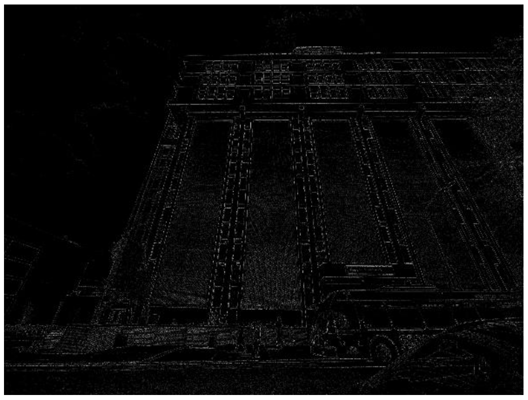
CV Assignment-2

Part-A

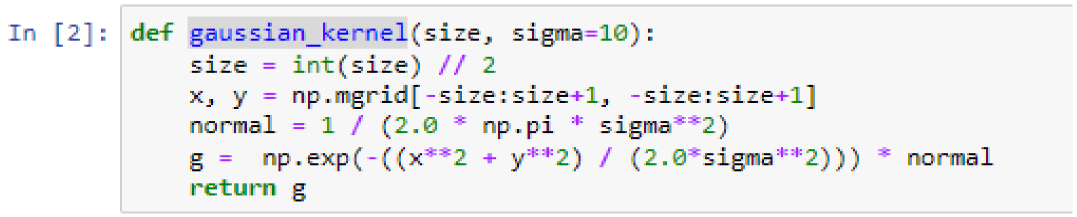
Canny Edge Detection





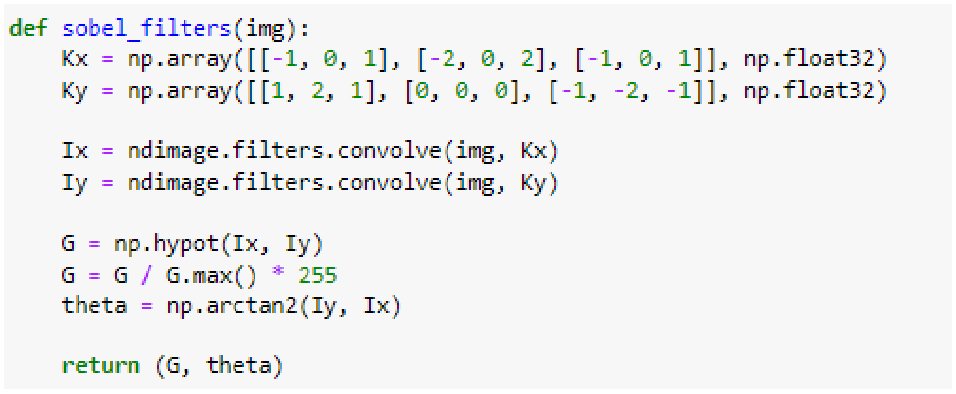
Original Image

Canny edge detected Image



```
In [2]: def gaussian_kernel(size, sigma=10):
            size = int(size) // 2
            x, y = np.mgrid[-size:size+1, -size:size+1]
            normal = 1 / (2.0 * np.pi * sigma**2)
            g =  np.exp(-((x**2 + y**2) / (2.0*sigma**2))) * normal
            return g
```

Custom Gaussian Kernel with 5*5 filter

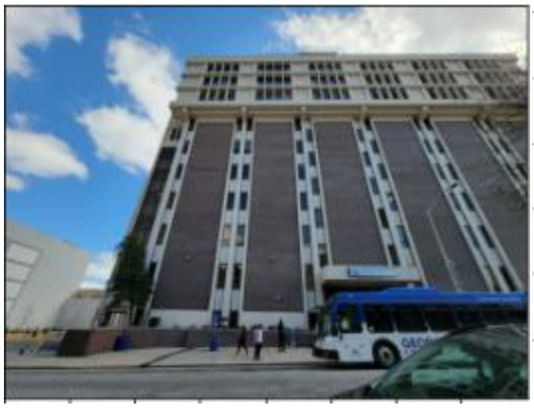
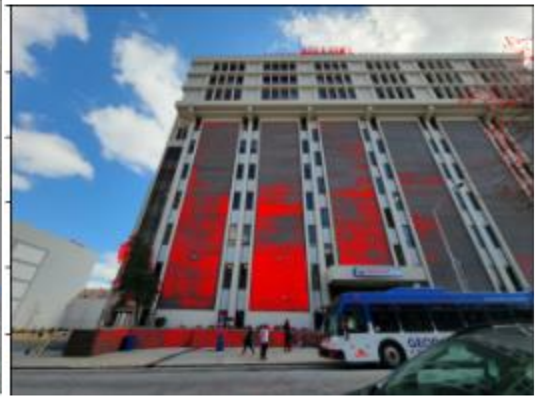


```
def sobel_filters(img):
    Kx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], np.float32)
    Ky = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], np.float32)

    Ix = ndimage.filters.convolve(img, Kx)
    Iy = ndimage.filters.convolve(img, Ky)

    G = np.hypot(Ix, Iy)
    G = G / G.max() * 255
    theta = np.arctan2(Iy, Ix)

    return (G, theta)
```

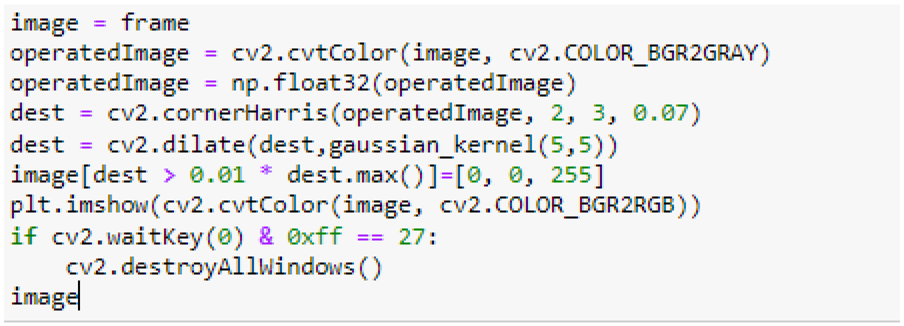
Custom build Sobel_filters

Using these custom functions, we can manipulate our image and do a non-max suppression then select only pixels that meets a weak and strong threshold value range. Also magnitudes and directions are taken and multiplied along the direction to get an edge. This is how we can get edges using canny edge detection algorithm.

Harris Corner Detection

Original Image Harris Corner Detection

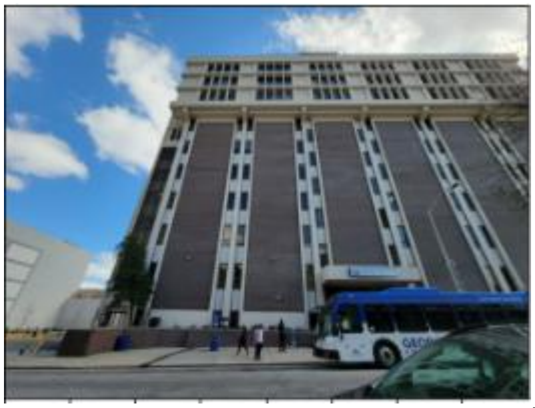
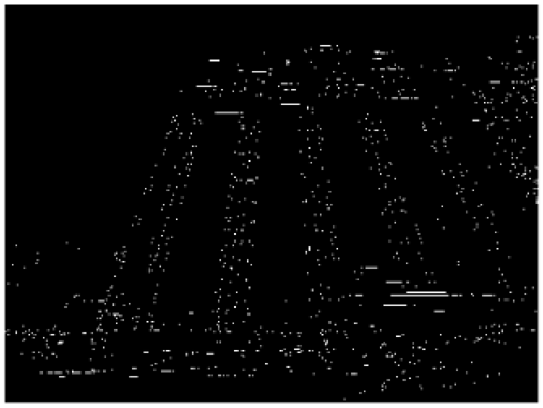


```
image = frame
operatedImage = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
operatedImage = np.float32(operatedImage)
dest = cv2.cornerHarris(operatedImage, 2, 3, 0.07)
dest = cv2.dilate(dest,gaussian_kernel(5,5))
image[dest > 0.01 * dest.max()]=[0, 0, 255]
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
if cv2.waitKey(0) & 0xff == 27:
    cv2.destroyAllWindows()
image
```

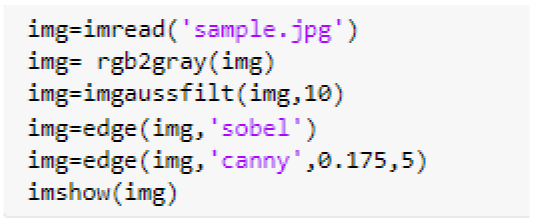
```
array([[[211, 208, 204],
        [211, 208, 204],
        [211, 208, 204],
        ...,
        [217, 224, 221],
        [217, 224, 221],
        [217, 224, 221]],

       [[211, 208, 204],
        [211, 208, 204],
        [211, 208, 204],
        ...,
        [217, 224, 221],
        [217, 224, 221],
        [217, 224, 221]],

       [[211, 208, 204],
        [211, 208, 204],
        [211, 208, 204],
        ...,
        [217, 224, 221],
        [217, 224, 221],
        [217, 224, 221]],

       ...,
```

Part-B

MATLAB Internal Canny edge detector

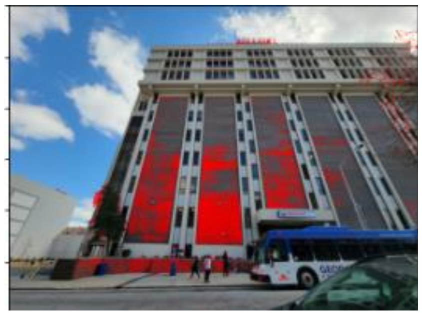
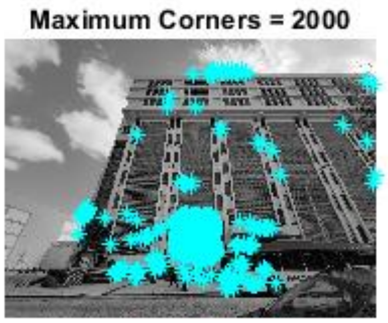



Original Image Canny edge detected Image



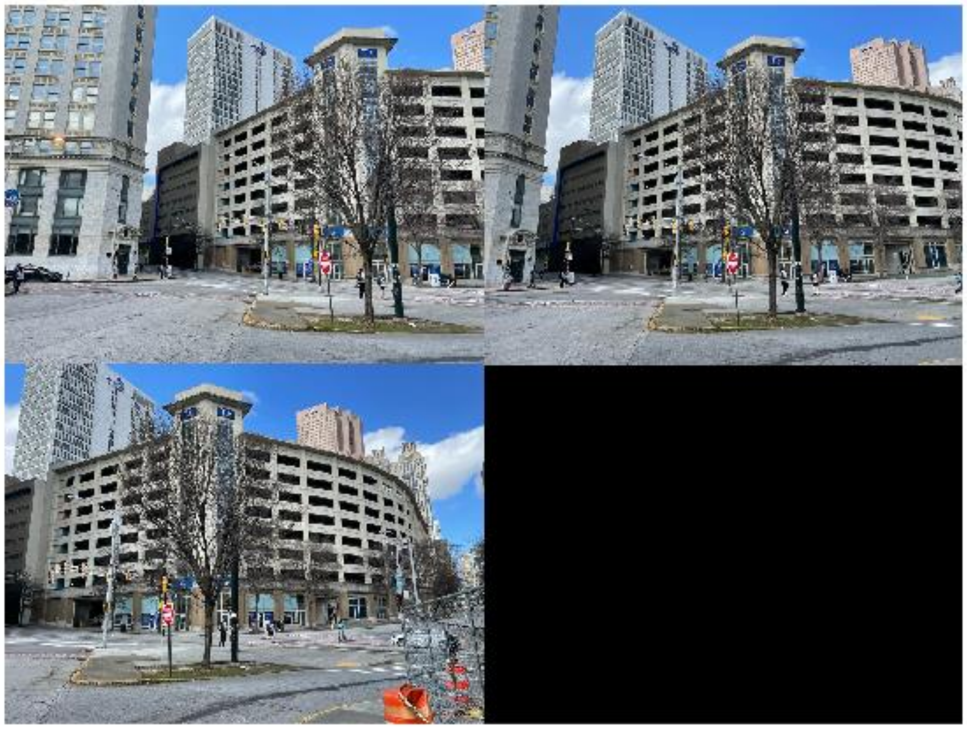
```
img=imread('sample.jpg')
img= rgb2gray(img)
img=imgaussfilt(img,10)
img=edge(img,'sobel')
img=edge(img,'canny',0.175,5)
imshow(img)
```

MATLAB Internal Canny edge detector

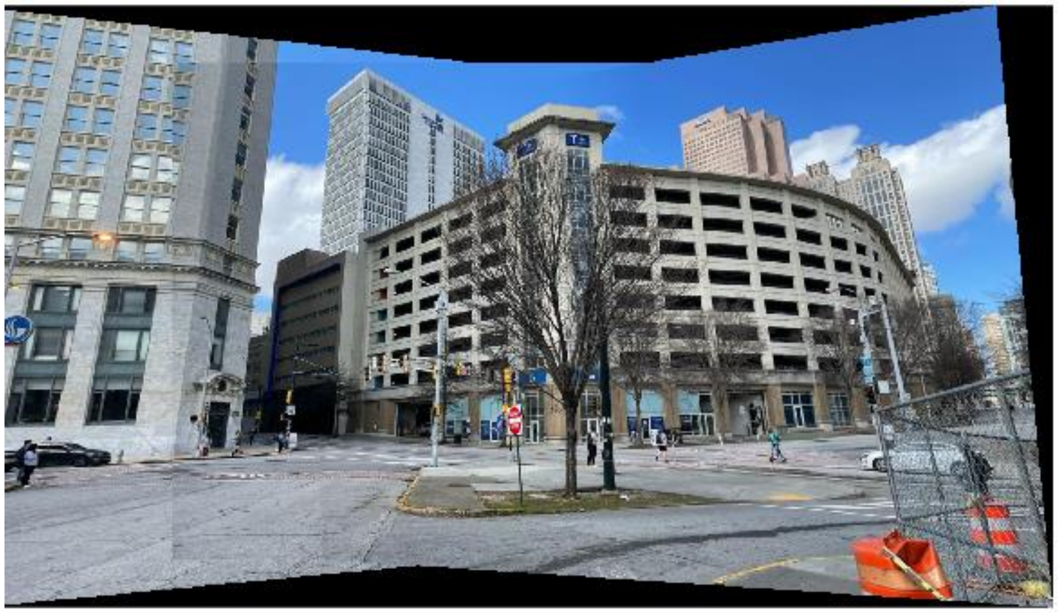



As we can see the corners are shown as dots and stars MATLAB Harris corner detection works on how many corners you want. So, if I give 200 corners, it only see first 200 corners. In the above image I took around 2000 corners as my threshold value.

Image Stitching in MATLAB



3 images that we took to stitch together

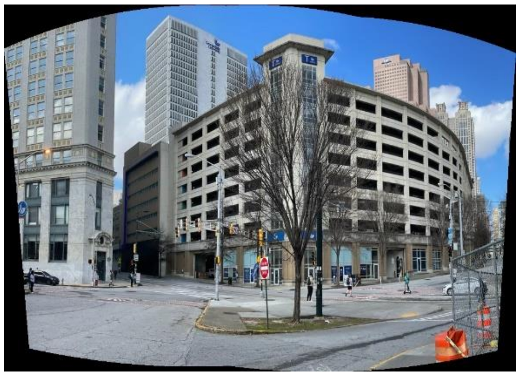


After the images are stitched and by the way it is T-deck 😉

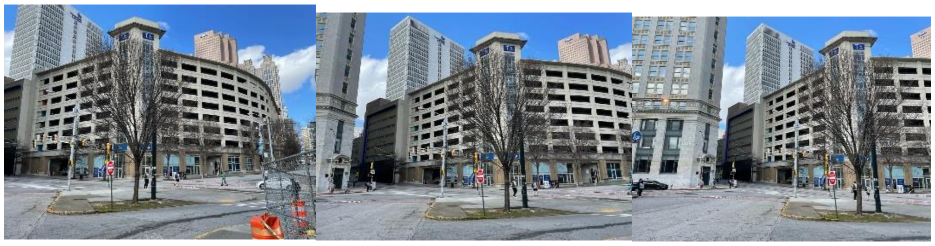
Function that can calculate integral image in real time

```python
def integral_image(image, *, dtype=None):
    if dtype is None and image.real.dtype.kind == 'f':
        dtype = np.promote_types(image.dtype, np.float64)

    S = image
    for i in range(image.ndim):
        S = S.cumsum(axis=i, dtype=dtype)
    return S


def integrate(ii, start, end):
    start = np.atleast_2d(np.array(start))
    end = np.atleast_2d(np.array(end))
    rows = start.shape[0]

    total_shape = ii.shape
    total_shape = np.tile(total_shape, [rows, 1])

    start_negatives = start < 0
    end_negatives = end < 0
    start = (start + total_shape) * start_negatives + \
             start * ~(start_negatives)
    end = (end + total_shape) * end_negatives + \
           end * ~(end_negatives)

    if np.any((end - start) < 0):
        raise IndexError('end coordinates must be greater or equal to start')

    S = np.zeros(rows)
    bit_perm = 2 ** ii.ndim
    width = len(bin(bit_perm - 1)[2:])
    for i in range(bit_perm):
        binary = bin(i)[2:].zfill(width)
        bool_mask = [bit == '1' for bit in binary]

        sign = (-1)**sum(bool_mask)

        bad = [np.any(((start[r] - 1) * bool_mask) < 0)
               for r in range(rows)]

        corner_points = (end * (np.invert(bool_mask))) + \
                        ((start - 1) * bool_mask)

        S += [sign * ii[tuple(corner_points[r])] if(not bad[r]) else 0
              for r in range(rows)]
    return S
```

Integral Image of live feed of the rgb camera….



Stitched Image



Parts of images to be stitched

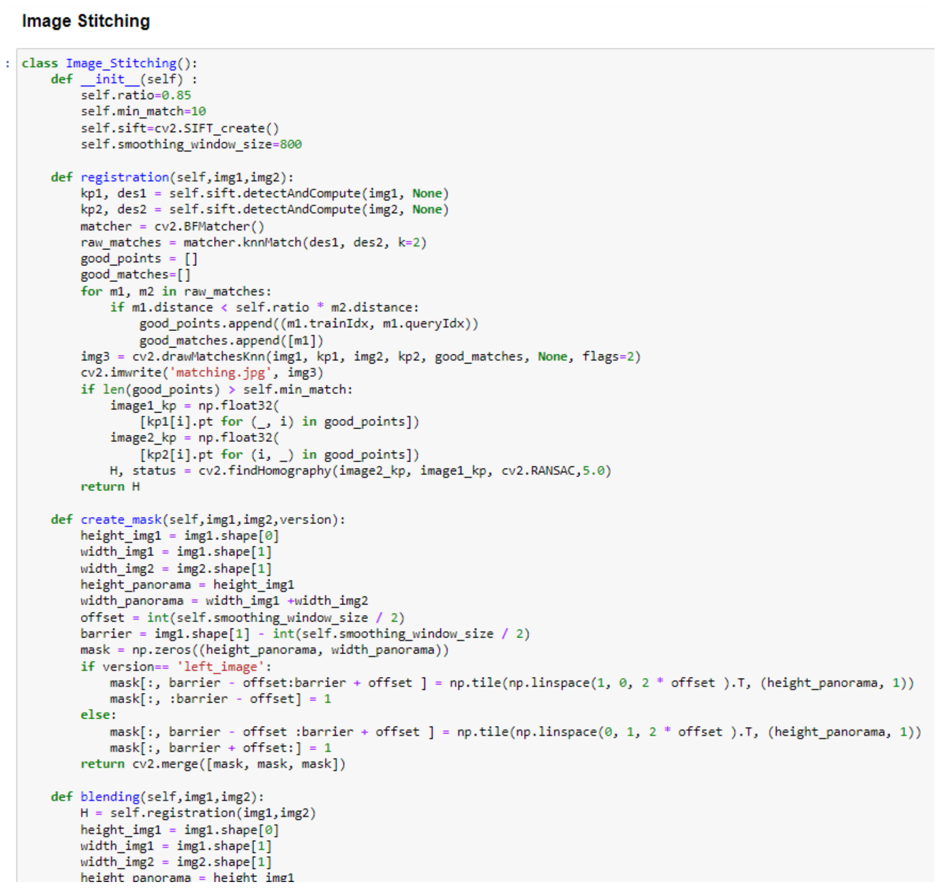
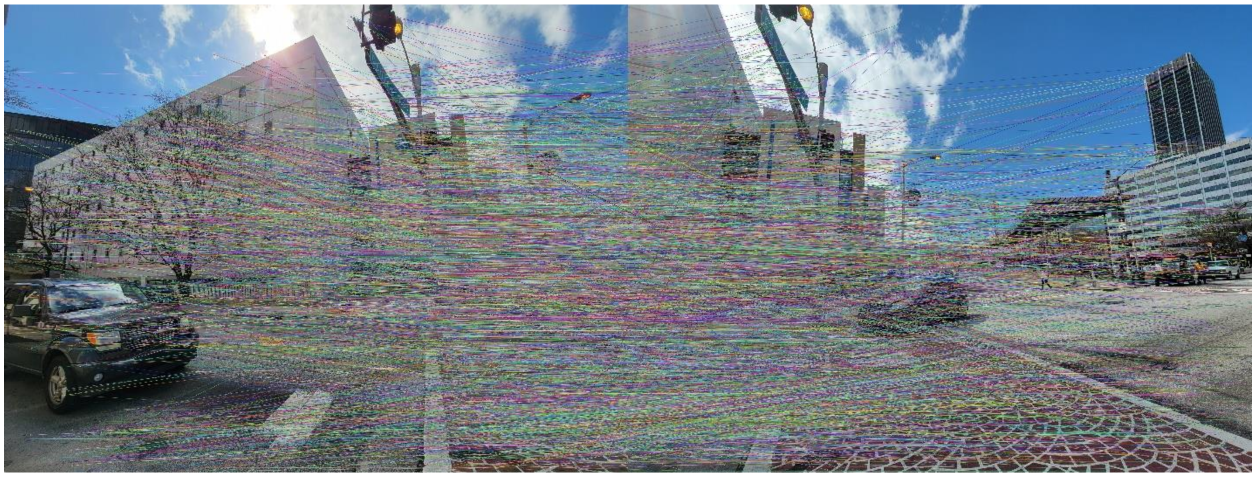


Image Stitching

```
class Image_Stitching():
    def __init__(self) :
        self.ratio=0.85
        self.min_match=10
        self.sift=cv2.SIFT_create()
        self.smoothing_window_size=800

    def registration(self,img1,img2):
        kp1, des1 = self.sift.detectAndCompute(img1, None)
        kp2, des2 = self.sift.detectAndCompute(img2, None)
        matcher = cv2.BFMatcher()
        raw_matches = matcher.knnMatch(des1, des2, k=2)
        good_points = []
        good_matches=[]
        for m1, m2 in raw_matches:
            if m1.distance < self.ratio * m2.distance:
                good_points.append((m1.trainIdx, m1.queryIdx))
                good_matches.append([m1])
        img3 = cv2.drawMatchesKnn(img1, kp1, img2, kp2, good_matches, None, flags=2)
        cv2.imwrite('matching.jpg', img3)
        if len(good_points) > self.min_match:
            image1_kp = np.float32(
                [kp1[i].pt for (_, i) in good_points])
            image2_kp = np.float32(
                [kp2[i].pt for (i, _) in good_points])
            H, status = cv2.findHomography(image2_kp, image1_kp, cv2.RANSAC,5.0)
        return H

    def create_mask(self,img1,img2,version):
        height_img1 = img1.shape[0]
        width_img1 = img1.shape[1]
        width_img2 = img2.shape[1]
        height_panorama = height_img1
        width_panorama = width_img1 +width_img2
        offset = int(self.smoothing_window_size / 2)
        barrier = img1.shape[1] - int(self.smoothing_window_size / 2)
        mask = np.zeros((height_panorama, width_panorama))
        if version== 'left_image':
            mask[:, barrier - offset:barrier + offset ] = np.tile(np.linspace(1, 0, 2 * offset ).T, (height_panorama, 1))
            mask[:, :barrier - offset] = 1
        else:
            mask[:, barrier - offset :barrier + offset ] = np.tile(np.linspace(0, 1, 2 * offset ).T, (height_panorama, 1))
            mask[:, barrier + offset:] = 1
        return cv2.merge([mask, mask, mask])

    def blending(self,img1,img2):
        H = self.registration(img1,img2)
        height_img1 = img1.shape[0]
        width_img1 = img1.shape[1]
        width_img2 = img2.shape[1]
        height_panorama = height_img1
```

Class to calculate panorama in realtime.



Sift Feature matching before creating a panorama. The feature selecting is done by SSD which is sum of squared distances.