Introduction

We are required to come up with written-from-scratch algorithms to perform the following computer vision tasks specifically in python:

1. Feature detectors - Identify the interest points
2. Feature descriptors - Extract feature vector descriptor surrounding each interest point
3. Feature matching - Determine correspondence between descriptors in 2 views

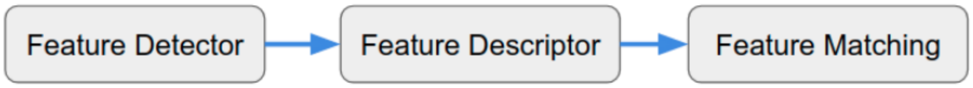
Why do we need to learn feature matching and what is the application of it?

To provide you with a clear answer to this question, let's come up with an example. take a look at the following images: This is the Episcopal Palace building in Spain. You can see the same building from different angles and distances.

You as a human can simply identify that there is one unique building in 2 images from 2 different angles and distances. But for a computer, this similarity identification is very difficult and sometimes impossible to find the answer. We are learning Feature Matching to enable a computer to find similar features in different images. Ultimately, the computer will able to provide the following applications:

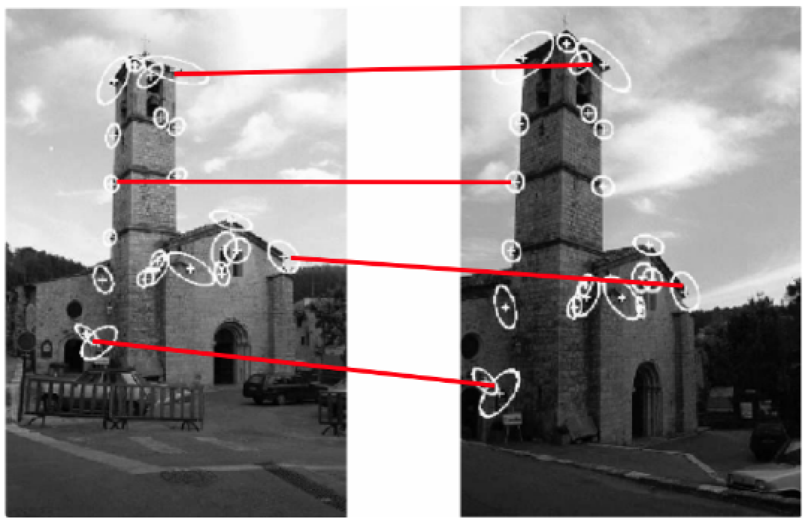
1. Automate object tracking
2. Point matching for computing disparity
3. Stereo calibration
4. Motion-based segmentation
5. Recognition
6. 3D object reconstruction
7. Robot navigation
8. image indexing

In order to write a specific algorithm for feature detection, the followings are required:



Harris Corner Detector

The feature detector or interest point detector algorithm is responsible for finding the most concrete and reliable features in an image that can be easily located in the second image. For instance, the following image pair is illustrated in the following figure:

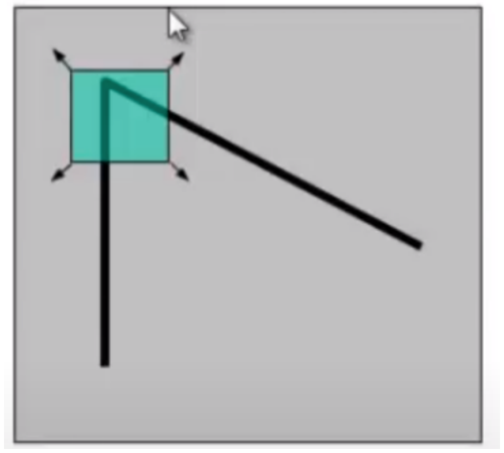


A good interest point detector should be able to:

1. Detect most of the true interest points
2. deal with noise
3. time-efficient

Mathematical Idea

In this section, we will be using the **Harris Corner detector** algorithm to find interest points in an image. In this algorithm, a small window will be shifted all over the image to find corner points in the image.



Now the question is how to find corner points in an image. To answer this question, we need to dig into the mathematical idea behind Harris Corner Detector:

$$E(u,v) = \sum_{x,y} w(x,y)\,(I(x+u,\ y+v)\ -\ I(x,\ y))^2$$

w(x, y): Window Function

I(x, y): Pixel intensity in the image I

I(x+u, y+v): shifted intensity in the image I

We can simply use the Taylor series to find an approximation for I(x+u, y+v):

$$Shifted\ Intensity \rightarrow I(x+u, y+v) = I(x,y)\ +\ uI_x\ +\ vI_y$$

$$E(u,v) = \sum_{x,y} w(x,\ y)\left(uI_x\ +\ vI_y\right)^2$$

$$E(u,v) = \sum_{x,y} w(x,y)\left[\begin{bmatrix} u & v \end{bmatrix}\begin{bmatrix} I_x \\ I_y \end{bmatrix}\right]^2$$

$$E(u,v) = \sum_{x,y} w(x,y)\ \begin{bmatrix} u & v \end{bmatrix}\begin{bmatrix} I_x \\ I_y \end{bmatrix}\begin{bmatrix} I_x & I_y \end{bmatrix}\begin{bmatrix} u \\ v \end{bmatrix}$$

$$E(u,v) = \begin{bmatrix} u & v \end{bmatrix}\left(\sum_{x,y} w(x,y)\begin{bmatrix} I_x \\ I_y \end{bmatrix}\begin{bmatrix} I_x & I_y \end{bmatrix}\right)\begin{bmatrix} u \\ v \end{bmatrix}$$

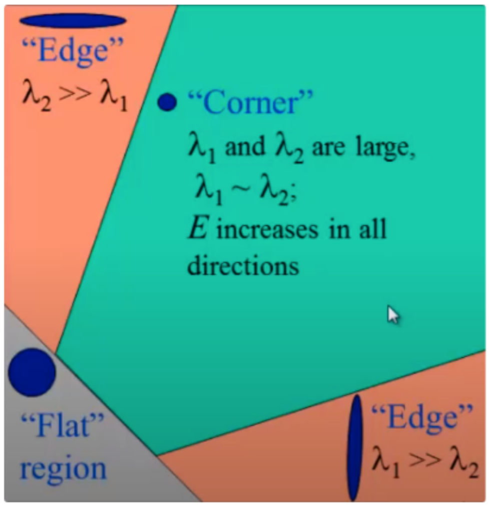
$$E(u,v) = \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix}$$

$$M = \sum_{x,y} w(x,y)\begin{bmatrix} I_xI_x & I_xI_y \\ I_xI_y & I_yI_y \end{bmatrix}$$

Now, we need to find the eigenvalues and eigenvectors of the derivative matrix M:

$$det(M - \lambda I)\,x = 0 \rightarrow \lambda_1\ \&\ \lambda_2$$

using the eigenvalues for each pixel, we can decide which pixel is the corner point. This concept is illustrated in the following figure:


"Edge"
$\lambda_2 >> \lambda_1$
"Corner"
λ_1 and λ_2 are large,
$\lambda_1 \sim \lambda_2$;
E increases in all directions
"Flat" region
"Edge"
$\lambda_1 >> \lambda_2$

Having the eigenvalues calculated, we can simply calculate the cornerness of every corner point using the Forstner-Harris equation. with α = 0.06. Unlike eigenvalue analysis, this quantity does not require the use of square roots and yet is still rotationally invariant and also downweights edge-like features.

$$det(A) - \alpha \times trace(A) = \lambda_0 \lambda_1 - \alpha \left(\lambda_0 + \lambda_1 \right)^2$$

Alpha value → 0.06.

Extracting feature descriptors

Once the interest points are detected, the next step is finding a correspondence feature descriptor for each interest point. To find feature descriptors, Scale Invariant Feature Transform (**SIFT**) is used.

Now the question is what is SIFT and the idea behind it? Let’s answer this question by designing a very straightforward algorithm.

student_sift.py is a python file in which the SIFT algorithm is implemented. get_features(image, x, y, feature_width, scales=None) function receives an input image and its detected interest points.

1. Find the image X(Ix) & Y(Iy) gradient and save them.
2. Use a Gaussian filter to smooth the gradients.
3. Around each interest point(x, y), consider a 16x16 or 32x32 window
4. Calculate the orientation of each pixel. E.g., atan2(Iy/Ix)
5. Extract the histogram of that window. Consider 45 degrees for each histogram bin
6. This Histogram will be your feature descriptor for each interest point
7. Repeat the same process for each interest point and then return all the feature descriptors

Feature Matching

In this section, the K-Nearest Neighbor method is used to find the best match for each feature. Now the question is what is the mathematical idea behind this method. Well, it is very simple to understand and more simple to implement.

Let's say we have 100 features on each image. Here is the algorithm:

1. Pick feature#0 from image#1 and calculate its distance from all features on image#2.
2. Select the minimum distance and its correspondence indexes on both images. These two indexes match each other by the highest probability.
3. If that minimum is very near to another one, remove both features from the data.
4. Repeat the same process for all the features and find the best match for each.

Implementation of Harris corner detector

In this section of the report, the Harris corner detector is implemented in Python. ***student_harris.py*** is the file where the algorithm is implemented. get_interest_points(image, feature_width) receives an image and a number indicated feature width for finding local maxima.

At the very first step, for the input image, the first gradient in X and Y directions is calculated as follows:

```python
gaussian_filter_size = 3
Gausian_standard_diviation = 1
gaussian_low_pass_filter = cv2.getGaussianKernel(gaussian_filter_size,Gausian_standard_diviation)
img_guassian_filtered = cv2.filter2D(image,-1,gaussian_low_pass_filter)
img_grad_y, img_grad_x = np.gradient(img_guassian_filtered)
```

Then the second gradients are calculated as follows:

```python
img_grad_x_2 = img_grad_x**2
img_grad_x_y = img_grad_x*img_grad_y
img_grad_y_2 = img_grad_y**2
```

Then the resulted image gradients are convolved with a larger gaussian kernel:

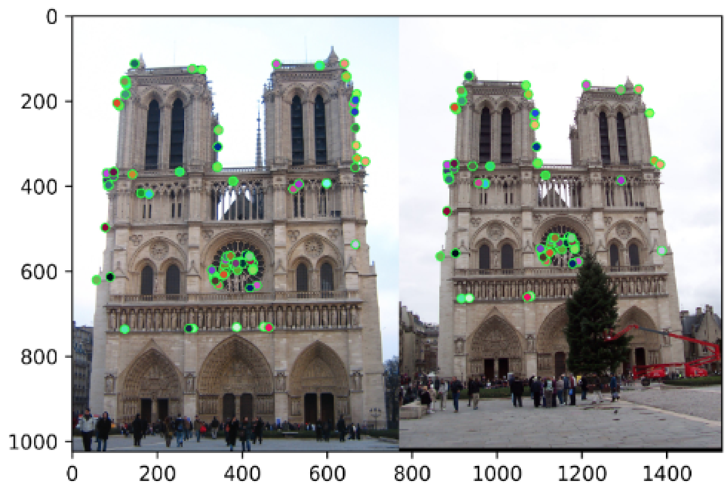
```python
larger_gaussian_filter_size = 7
Gausian_standard_diviation = 1
gaussian_larger_filter = cv2.getGaussianKernel(larger_gaussian_filter_size,Gausian_standard_diviation)
img_grad_x_2_filtered = cv2.filter2D(img_grad_x_2, -1, gaussian_larger_filter)
img_grad_x_y_filtered = cv2.filter2D(img_grad_x_y, -1, gaussian_larger_filter)
img_grad_y_2_filtered = cv2.filter2D(img_grad_y_2, -1, gaussian_larger_filter)

```

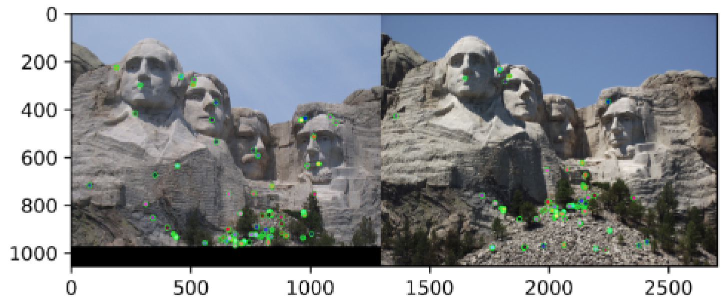
Cornerness of the interest points is calculated by Forstner-Harris euation:

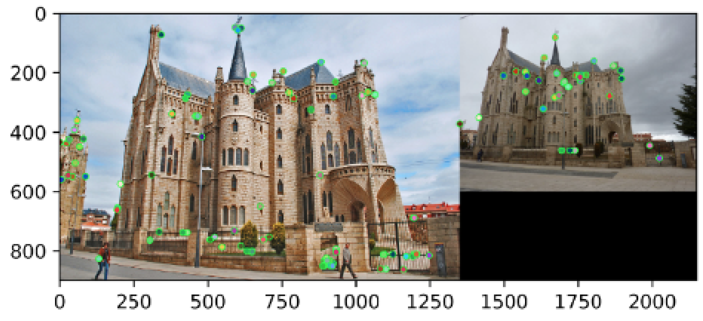
```python
factor_alpha = 0.06
det_A = img_grad_x_2_filtered * img_grad_y_2_filtered - img_grad_x_y_filtered **2
trace_A = img_grad_x_2_filtered + img_grad_y_2_filtered
Forstner_Harris = det_A -factor_alpha * trace_A ** 2

h, w = Forstner_Harris.shape
interest_points = []
fh_index = []

for f1 in range(0, h):
    fh_index = []
    for f2 in range(0, w):
            fh_index = [Forstner_Harris[f1,f2]]
            fh_index.append(f2)
            fh_index.append(f1)
            interest_points.append(fh_index)
```

Given the feature width, the local maxima can be calculated within every feature window. This method is called **Adaptive non-maxima suppression(ANMS)**:

```
interest_points_sort_list = sorted(interest_points, reverse = True)
number_of_desired_interest_points = 1000
interest_points_sort_list = interest_points_sort_list[0:number_of_desired_interest_points]
interest_point_neighborhood_list = []

for i in range(0, number_of_desired_interest_points):
    local_nibrhd = float('inf')
    first_location = interest_points_sort_list[i]
    for j in range(0, i):
        second_location = interest_points_sort_list[j]
        euclidean_distance = (first_location[1]-second_location[1])**2 +(first_location[2]-second_locat
        local_nibrhd = min(euclidean_distance, local_nibrhd)
    interest_point_neighborhood_list.append([np.sqrt(local_nibrhd), first_location[0], first_location[1
x_list = []
y_list = []
for interest_point_neighborhood in sorted(interest_point_neighborhood_list, reverse = True):
    x_list.append(interest_point_neighborhood[2])
    y_list.append(interest_point_neighborhood[3])
x = np.array(x_list)
y = np.array(y_list)

return x,y, confidences, scales, orientations
```







Implementation of the feature descriptor

According to the aforementioned algorithm for extracting feature descriptors, the Scale Invariant Feature Transform method is used to find the feature descriptor around each interest point.

At the very first step, the image padding function will be used to pad the input image in order to prepare it for Gaussian filtering:

```python
def img_padding(img_data, padding_number):

  # if condition: grayscale images
  if True:
    i, j= img_data.shape
    zeros_column = np.zeros((1, i))
    zeros_row =  np.zeros((1, j + 2*padding_number))
    # for loop: to add zero padding to columns
    counter = 1
    for l in range(0, padding_number):

      img_data = np.insert(img_data, 0, zeros_column, axis=1)
      img_data = np.insert(img_data, j + counter , zeros_column, axis=1)
      counter = counter + 1

    # for loop: to add zero padding to rows
    counter = 1
    for p in range(0, padding_number):

      img_data = np.insert(img_data, 0, zeros_row, axis=0)
      img_data = np.insert(img_data, i + counter , zeros_row, axis=0)
      counter = counter + 1
    return img_data
```

The next step is to write two functions for taking image gradients:

```python
def get_Ix(inp_img):
    height, width = inp_img.shape
    h, w = inp_img.shape
    Ix = np.zeros((h-1, w-1))
    for row in range(0, h-1):
        for col in range(0, w-1):
            Ix[row, col] = inp_img[row, col + 1] - inp_img[row, col]
    return Ix

def get_Iy(inp_img):
    h, w = inp_img.shape
    Iy = np.zeros((h-1, w-1))
    for col in range(0, w-1):
        for row in range(0, h-1):
            Iy[row, col] = inp_img[row + 1, col] - inp_img[row, col]
    return Iy
```

Now, we are ready to dig into the main function of the feature descriptor:

Don't Panic! 🙂 I wrote this code very simple so that everyone can make sense of it. After taking the X and Y derivatives of the image, we will define the bins of the histogram. Then we will start looping over the image, stopping at each interest point, and then extract the feature descriptor, histogram of orientation and magnitude. This way the descriptor vector for each interest point is extracted and we return them in normalized shapes.

```python
def get_features(image, x, y, feature_width, scales=None):
    padding_number = int(abs(feature_width/2))
    image = img_padding(image, padding_number)
    dx = get_Ix(image)
    dy = get_Iy(image)
    features_descriptors=[]
    number_of_interest_points = x.size
    bin_val_1 = -1
    bin_val_2 = 1
    bin_val_3 = 2
    for interest_point in range(0, number_of_interest_points):
        o = int(abs(feature_width/4))
        pixel_magintude_all = np.zeros((o,o,8))
        for j in range(0, feature_width):
            for i in range(0, feature_width):
                ix_value = dx[(int)(y[interest_point]) + j, (int)(x[interest_point]) +i]
                iy_value = dy[(int)(y[interest_point]) +j, (int)(x[interest_point]) +i]
                pixel_angle = np.arctan2(iy_value, ix_value)
                if pixel_angle > bin_val_2:
                    pixel_angle = bin_val_3
                if pixel_angle < bin_val_1:
                    pixel_angle = bin_val_1
                pixel_magintude = np.sqrt(ix_value**2 + iy_value**2)
                if ix_value >0:
                    pixel_magintude_all[(int)(j/4), (int)(i/4), math.ceil(pixel_angle+1)] += pixel_magi
                else:
                    pixel_magintude_all[(int)(j/4), (int)(i/4), math.ceil(pixel_angle+5)] += pixel_magi
        single_feat = np.reshape(pixel_magintude_all,(1,o*o*8))
        single_feat = single_feat/(single_feat.sum())
        features_descriptors.append(single_feat)
    fv = np.array(features_descriptors)
    return fv
```

Implementation of feature matching

The feature matching is our last algorithm to implement in this assignment. According to the aforementioned algorithm for feature matching:

1. start looping over all the returned feature descriptors.
2. Pick the feature descriptor from feature1 array and compare them with all feature descriptor from feature2.
3. Detect the first and second minima. If they are very close to each other, ignore this interest points. Otherwise, save the indexes.
4. calculate the confidence by dividing the first and second minima in each feature descriptor.
5. Repeat the same process for all feature descriptors.
6. return matches and confidence.

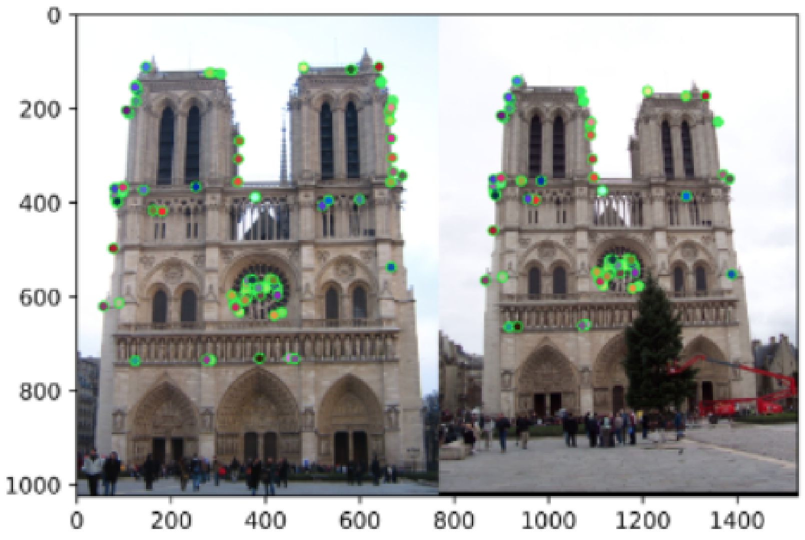
```python
def match_features(features1, features2, x1, y1, x2, y2):
    d_all = []
    feat_1_match_index = []
    feat_2_match_index = []
    feat_match = []
    mathing_confidence = []

    for i in range(features1.shape[0]):
        for j in range(features2.shape[0]):
            local_distance = np.linalg.norm(features1[i]-features2[j])
            d_all.append(local_distance)
        first_min = min(d_all)
        min_1_index = d_all.index(first_min)
        del d_all[min_1_index]
        second_min = min(d_all)
        min_2_index = d_all.index(second_min)
        nearest_neighbor_distance_ratio = first_min /second_min
        match_index = min_1_index
        feature_index_1 = i
        feat_1_match_index.append(i)
        feat_2_match_index.append(match_index)
        mathing_confidence.append(nearest_neighbor_distance_ratio)
        feat_match.append([nearest_neighbor_distance_ratio, feature_index_1, match_index])
        d_all = []
    match_list = []
    confidence_list = []
    for mt in sorted(feat_match)[:100]:
            match_list.append([mt[1], mt[2]])
            confidence_list.append(mt[0])

    matches = np.array(match_list)
    confidences = np.array(confidence_list)

    matches = matches.astype(int)
    return matches, confidences
```

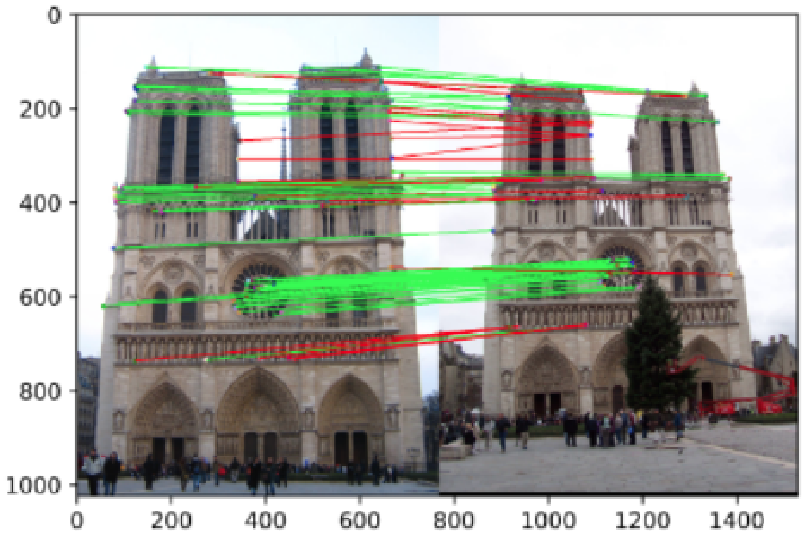
Harris corner detection - Results

The output of the aforementioned algorithm is the XY location of all interest points within the processed image. The following image pairs are processed:

Notre Dame Building: accuracy

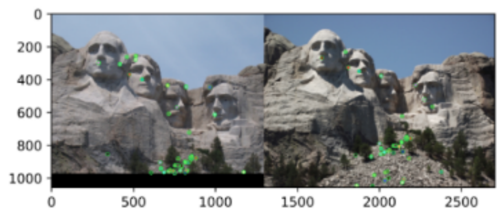
Settings to get the best results:
Feature Width: 32
Alpha = 0.06
Gaussian Distribution 3x3 → Standard Deviation: 1.0 → applied on the input image
Gaussian Distribution 7*7 → Standard Deviation: 7.0 → applied on image gradients
Accuracy: 73%

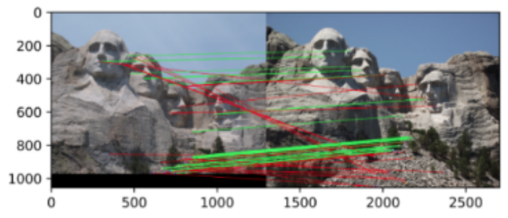




Mount Rushmore:

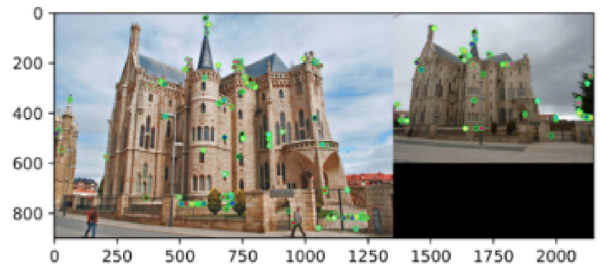
Settings to get the best results:
Feature Width: 32
Alpha = 0.005
Gaussian Distribution 3x3 → Standard Deviation: 1.0 → applied on the input image
Gaussian Distribution 7*7 → Standard Deviation: 10 → applied on image gradients
Accuracy: 61%

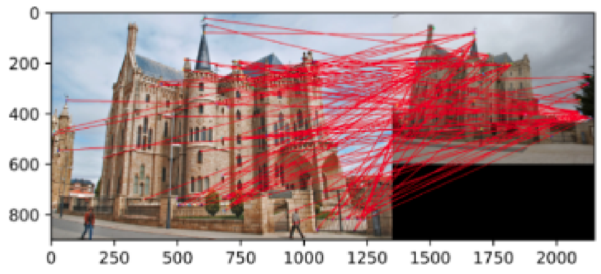




Episcopal Gaudi:

Settings to get the best results:
Feature Width: 32
Alpha = 0.005
Gaussian Distribution 3x3 → Standard Deviation: 1.0 → applied on the input image
Gaussian Distribution 7*7 → Standard Deviation: 10 → applied on image gradients
Accuracy: 0.0 %





References

[1]: Richard Szeliski. 2010. Computer Vision: Algorithms and Applications (1st. ed.). Springer-Verlag, Berlin, Heidelberg.

[2]. Abdel-Hakim, A. E. and Farag, A. A. (2006). CSIFT: A SIFT descriptor with color invariant characterstics. In IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'2006), pp. 1978– 1983, New York City, NY.

[3]. Adelson, E. H., Simoncelli, E., and Hingorani, R. (1987). Orthogonal pyramid transforms for image coding. In SPIE Vol. 845, Visual Communications and Image Processing II, pp. 50–58, Cambridge, Massachusetts.