exVision: FeatureCraft

Youssef A. Shawki, Nada Omran, Nouran M. Khattab, Kareem Noureddine, Mohamed Samy

Supervised by **Dr. Ahmed Badawy** as part of the Classical Computer Vision course, Systems and Biomedical Engineering Faculty, Cairo University (2023/2024)

SIFT

The Scale-Invariant Feature Transform (SIFT) algorithm plays a crucial role in computer vision tasks by extracting image features that are invariant to translation, rotation, scaling and changes in image quality. Those descriptive features can be used to recognize certain objects or for matching pairs of images.

Steps:

1. Scale-space Extrema Detection
2. Keypoint Localization
3. Orientation Assignment
4. Keypoint Descriptor
5. Matching Descriptors

Space-scale Extrema Detection

Detecting invariant features to scale changes is accomplished by searching for stable features across multiple scales. This process is achieved by constructing a scale space representation using a Gaussian pyramid organized into octaves. Each octave consists of a series of increasingly blurred Gaussian images, the standard divisions of two nearby Gaussian images are separated by a constant multiplicative factor (k).

The base image of the first octave is generated by upsampling the original image by a factor of two, to double the number of keypoints, and the base image of each of the following octaves is generated by downsampling the Gaussian image from the previous octave that has twice the initial standard division (2σ).

The scale space of an image is produced from the convolution of a variable-scale Gaussian kernel, $G(x, y, \sigma)$, with an input image, $I(x, y)$:

$$L(x,y,\sigma) = G(x,y,\sigma) * I(x,y)$$

$$G(x,y,\sigma) = \frac{1}{2\Pi\sigma^2} . e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

The size of the kernel is determined based on the standard division (σ), so that it covers a significant portion of the Gaussian distribution. In the SIFT algorithm, a common approach is to choose a kernel radius that covers around 95% of the Gaussian distribution. This ensures that the kernel captures most of the important information for feature detection.

The Gaussian Pyramid structure

1. Four octaves
2. Five scales per octave: Five increasingly blurred images, every two nearby scales are separated by a constant multiplicative factor k, the five scales are defined as σ, kσ, k^2σ, k^3σ, and k^4σ, where σ is set to 1.6 and k is $\sqrt{2}$ following recommendations from the literature.

Detection of Extrema

Experiments showed that the extrema of the scale-normalized Laplacian of Gaussian, where the scale factor is σ^2, produces the most stable image features compared to a range of other possible image functions, but it is computationally intensive. Therefore, the Difference of Gaussian (DoG) function is utilized as an approximation. The Difference of the Gaussian function is simply generated by subtracting two nearby Gaussian images. This approach provides a computationally efficient approximation to the scale-normalized Laplacian of Gaussian.

Differences of Gaussian extrema are detected through looping over the pixels of the DoG image and generating 9x9x9 patches, centred at the current pixel, then detecting the index of the maximum and minimum values in the patch, if any of them is the middle index then the current pixel is an extremum and considered a potential keypoint.

Keypoint Localization

In the SIFT algorithm, keypoints of an image are defined as the 3D extrema of the DoG. However, due to the discrete nature of digital images, computing continuous 3D extrema can't be achieved directly. Instead, the algorithm detects discrete extrema in the digital DoG representation, and then precise localization is performed through using Taylor expansion (up to the quadratic terms) of the Difference of Gaussian function, shifted so that the origin is at the discrete extrema. This also helps in discarding unstable keypoints caused by low contrast or poor localization.

$$D(x) = D + \frac{\partial D^T}{\partial x} x + \frac{1}{2} x^T . \frac{\partial^2 D}{\partial X^2} x$$

Where: x is the pixel location of the Potential discrete keypoint.

The location of the extremum, Candidate keypoint offset ($\hat{x}$), is determined by taking the derivative of this function with respect to x and setting it to zero.

$$\hat{x} = -\frac{\partial^2 D^{-1}}{\partial x^2} . \frac{\partial D}{\partial x}$$

If the computed offset value is greater than 0.5, this means that the extremum lies closer to a different sample point, so the keypoint is discarded.

Experiments showed that this localization provides a substantial improvement in matching and stability.

The first and second derivatives of the Difference of Gaussian function at the candidate key-point are approximated using differences of neighbouring pixels:

$$\frac{\partial D}{\partial x} = \frac{D(x+1,y,s) - D(x-1,y,s)}{2}, \frac{\partial D}{\partial y} = \frac{D(x,y+1,s) - D(x,y-1,s)}{2}, \frac{\partial D}{\partial s} = \frac{D(x,y,s+1) - D(x-1,y,s-1)}{2}$$

$$\frac{\partial D^2}{\partial x^2} = D(x+1,y,s) - 2D(x,y,s) + D(x-1,y,s)$$

$$\frac{\partial D^2}{\partial y^2} = D(x,y+1,s) - 2D(x,y,s) + D(x,y-1,s)$$

$$\frac{\partial D^2}{\partial s^2} = D(x,y,s+1) - 2D(x,y,s) + D(x,y,s-1)$$

$$\frac{\partial D^2}{\partial x\, \partial y} = \frac{[D(x+1,y+1,s) - D(x-1,y+1,s)] - [D(x+1,y-1,s) - D(x-1,y-1,s)]}{4}$$

$$\frac{\partial D^2}{\partial x\, \partial s} = \frac{[D(x+1,y,s+1) - D(x-1,y,s+1)] - [D(x+1,y,s-1) - D(x-1,y,s-1)]}{4}$$

$$\frac{\partial D^2}{\partial y\, \partial s} = \frac{[D(x,y+1,s+1) - D(x,y-1,s+1)] - [D(x,y+1,s-1) - D(x,y-1,s-1)]}{4}$$

The function value at the extremum, $D(\hat{x})$, is used to reject unstable extrema with low contrast.

$$D(\hat{x}) = D + \frac{1}{2} . \frac{\partial D^T}{\partial x} \hat{x}$$

All candidate keypoints with absolute $D(\hat{X})$ smaller than a certain threshold value, controlled by the user through a slider in the UI, are discarded.

In the scale-normalized Laplacian of Gaussian, edges produce high contrast responses, however, they cannot represent a keypoint, so they need to be discarded. To do so, we use the computed derivatives to form a 2x2 Hessian matrix.

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$$

$$Tr(H) = D_{xx} + D_{yy}$$

$$Det(H) = D_{xx}D_{yy} + D_{xy}{}^{2}$$

The ratio of principal curvatures is computed as follows:

$$r = \frac{Tr(H)^2}{Det(H)}$$

If the ratio of principal curvatures is above a certain threshold, controlled by the user through a slider in the UI, then the keypoint lies along an edge and is discarded.

Unfortunately, we implemented the keypoints localization and filtering function after finishing the implementation of the rest of the algorithm, and we faced a challenge when we were trying to integrate it into the algorithm. Before applying localization of keypoints, the scale (σ) of any keypoint in any octave is one of the five scales that are stored in the Gaussian Pyramid, so we can directly access its scale image without needing any further computations. However, after localization, an offset value is added to the scale of the keypoint, so the scale image needs to be calculated based on the new scale. Since the number of keypoints is huge, this computation is impractical. As a result, we decided to comment on their implementation temporarily to explore suitable solutions for this problem.

Orientation Assignment

After extracting the keypoints info, namely the location (x, y), octave index, and scale index. Now, we are missing a very essential piece of information, which is the dominant orientation of the keypoint. This piece of information besides the scale of detection, would be used later in generating a descriptor that is invariant to rotation and scale.

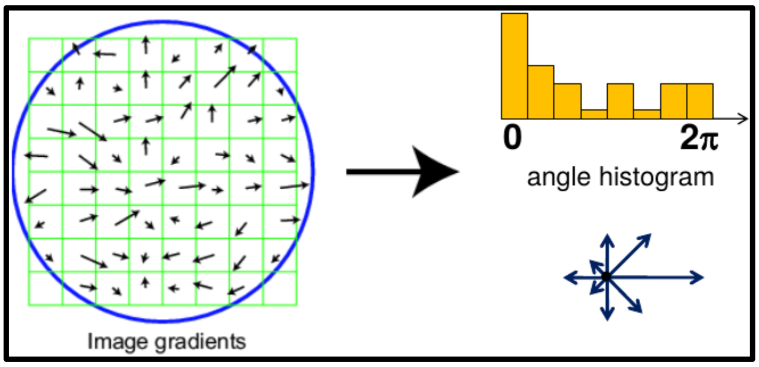


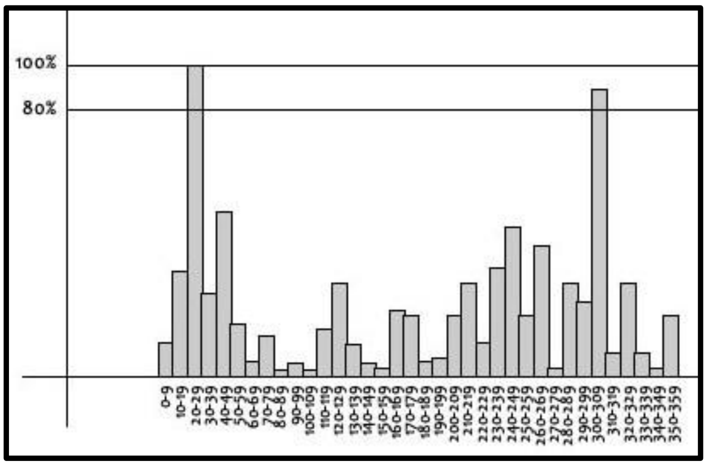
Figure 1, determines the dominant orientation of the keypoint using a histogram of gradient direction, each weighted by the gradient magnitude after being modulated using Gaussian kernel proportional to the scale of the keypoint, ***indicated by the blue circle****.*

To assign the orientation to each keypoint, we have computed the gradient of the blurred image at which we could catch up with the keypoint. Afterwards, we have quantized the gradient direction to **36** bins, paving the way to construct a histogram of the gradient direction. To construct the histogram, we will extract a window of size **4 sigma** at which the keypoint is detected from the magnitude and quantized direction of size proportional to the sigma value of the keypoint. The height of the bins will be weighted by the gradient magnitude. For that reason, we have to construct a Gaussian kernel of sigma equal to **(1.5 x Sigma of the keypoint)**. Then, we used such a kernel to modulate the gradient magnitude used to construct the histogram. This makes sure the far pixel orientation gets lower votes compared to the near ones.

Note that we have used the sigma value of the keypoint in the determination of the keypoint dominant orientation. **This contributes to making the dominant orientation invariant to scale change.**

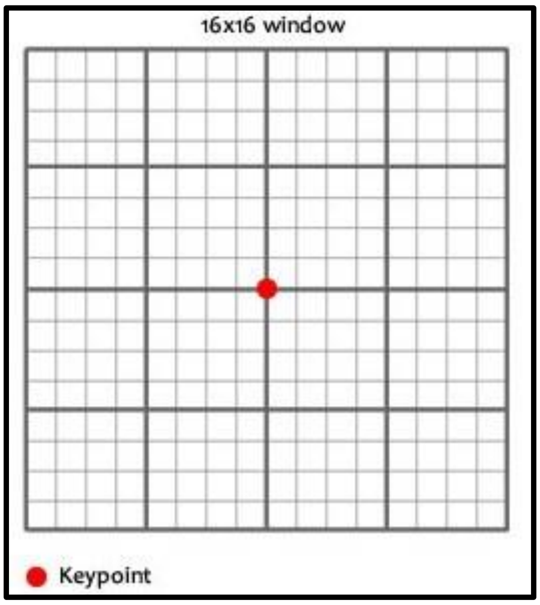
After constructing a histogram that is similar to that in figure no.2. We start to extract the directions that get the highest votes. Towards that end, we have extracted the direction of the max vote and any other direction of vote of at least **80%** of the max vote. **This indicates that we might have more than one orientation to a keypoint, thus, more than one descriptor.**

By the end of this step, we have added another info to each keypoint, which is the dominant orientation.

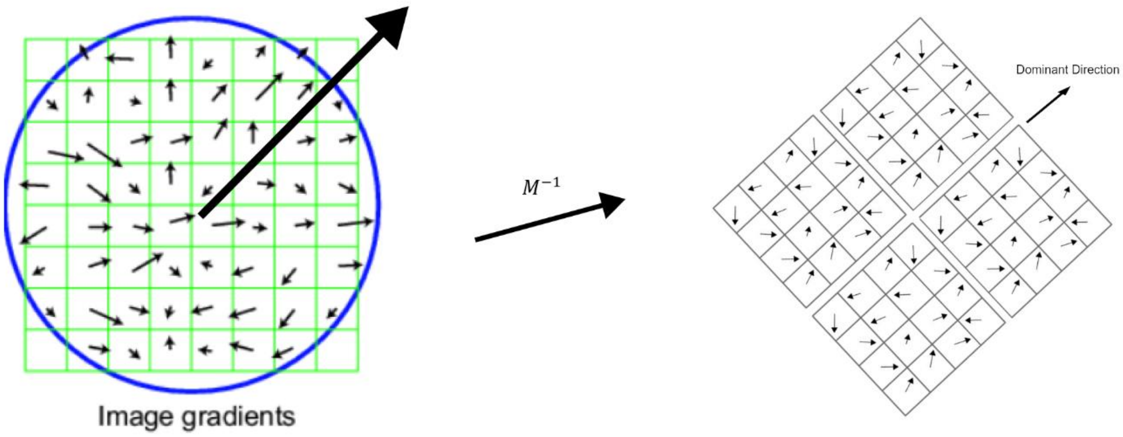


Keypoint Descriptor

After gathering all possible information about each keypoint. We are now ready to express these keypoints in a more robust way that is further variant to scale as well as orientation. **Remember**, we have already contributed to mitigating the scale dependency in orientation assignment by multiplying the magnitude window by a Gaussian kernel that is proportional to the scale of the keypoint itself. Now, we must ensure that the effect of orientation variance is eliminated as much as possible. Towards that end, we would have to extract a 16 x 16 window centred on the keypoint, then divide this 16 x 16 block into 16 regions of 4x4 as depicted by figure no.3.



These blocks will help us represent the key points in a fully invariant manner. But, to get a more descriptive 16 x 16 window, we would extract such a window such that it is aligned with the dominant orientation. To achieve this we would apply affine transformation to the 16 x 16 window above, to make it aligned with the dominant orientation.

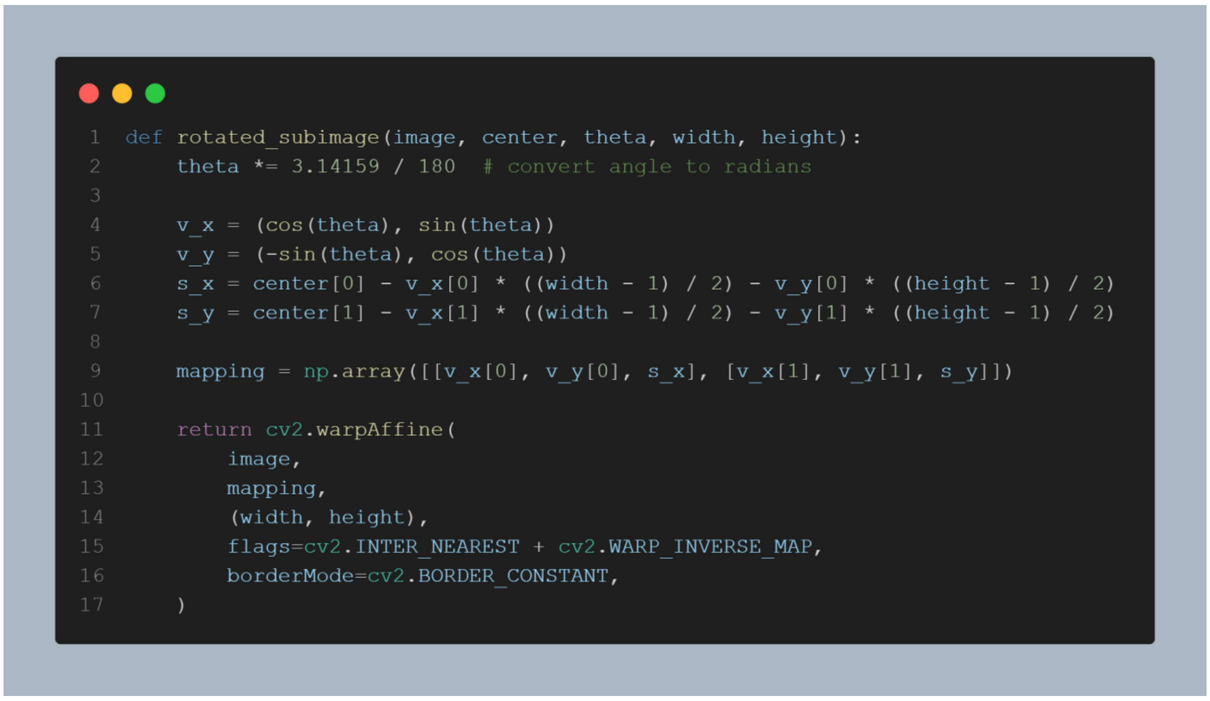


$$M = \begin{bmatrix} cos(\theta) & -sin(\theta) & t_x \\ sin(\theta) & cos(\theta) & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

Where:

- θ is the Dominant orientation angle.
- t_x & t_y the translation components that we want to counteract.

The translation component of the affine transformation would guarantee that the centre of the keypoint window would be at the location of keypoint detection (x, y).

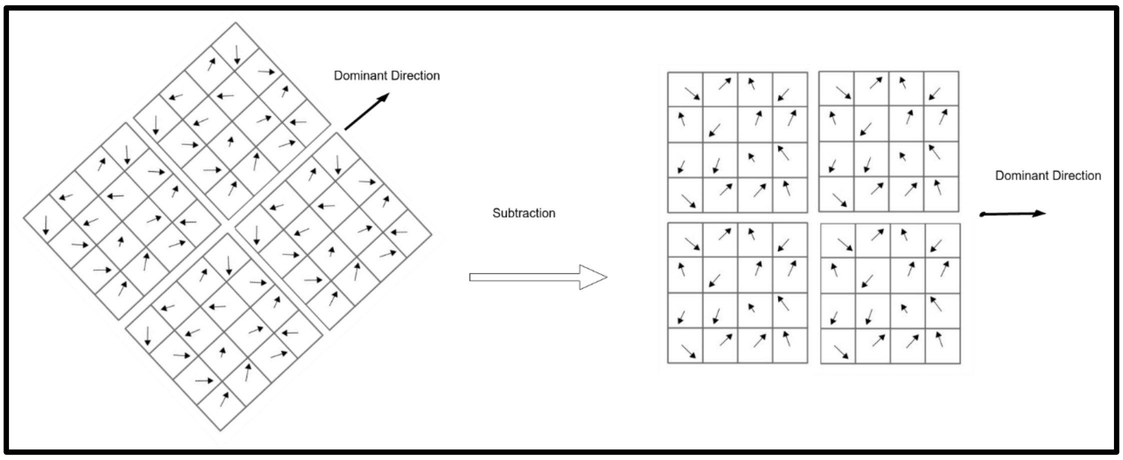


```python
def rotated_subimage(image, center, theta, width, height):
    theta *= 3.14159 / 180  # convert angle to radians

    v_x = (cos(theta), sin(theta))
    v_y = (-sin(theta), cos(theta))
    s_x = center[0] - v_x[0] * ((width - 1) / 2) - v_y[0] * ((height - 1) / 2)
    s_y = center[1] - v_x[1] * ((width - 1) / 2) - v_y[1] * ((height - 1) / 2)

    mapping = np.array([[v_x[0], v_y[0], s_x], [v_x[1], v_y[1], s_y]])

    return cv2.warpAffine(
        image,
        mapping,
        (width, height),
        flags=cv2.INTER_NEAREST + cv2.WARP_INVERSE_MAP,
        borderMode=cv2.BORDER_CONSTANT,
    )
```

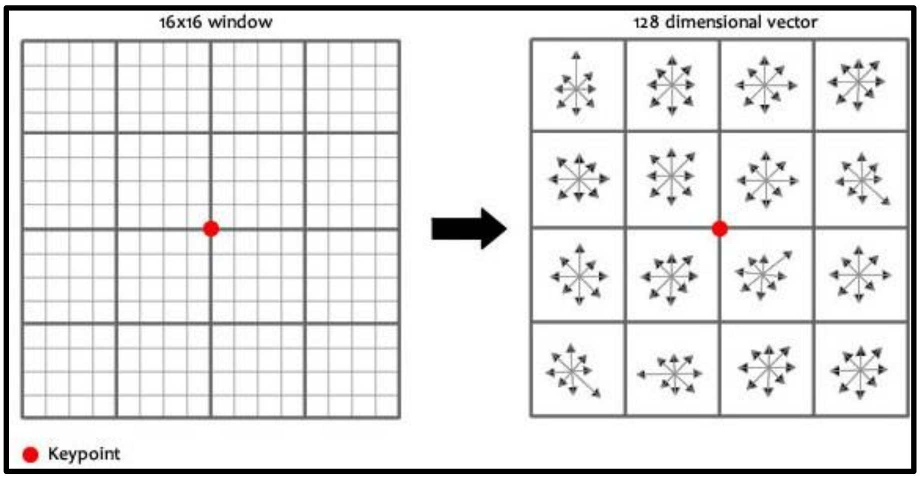
Figure 2, Sample Code of the Above Affine Transformation

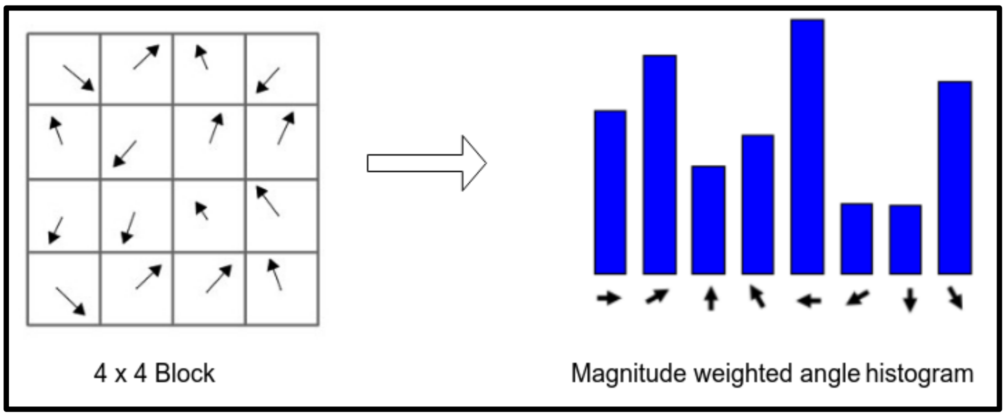
The above affine transformation uses the K-nearest neighbour as an interpolation method.

After applying the transformation, we would subtract the dominant orientation from the gradient directions of the window that is centred on the keypoint. Thus, we have achieved rotation invariance.



After applying all of the previous steps on gradient and orientation. Now, we are ready to construct the descriptor of each keypoint. Towards that end, we would create a histogram of the gradient direction of each 4x4 block. But before this, we need to smooth the transformed magnitude using a Gaussian kernel of sigma value equal to **1.5 x scale of keypoint** and of size 16. Now, we can create the 16-magnitude weighted histogram of each block





After constructing the 16 histograms of 16 blocks each of 8-directions bins. We would concatenate them and come up with a (16 x 8 = 128) long descriptor for each keypoint.

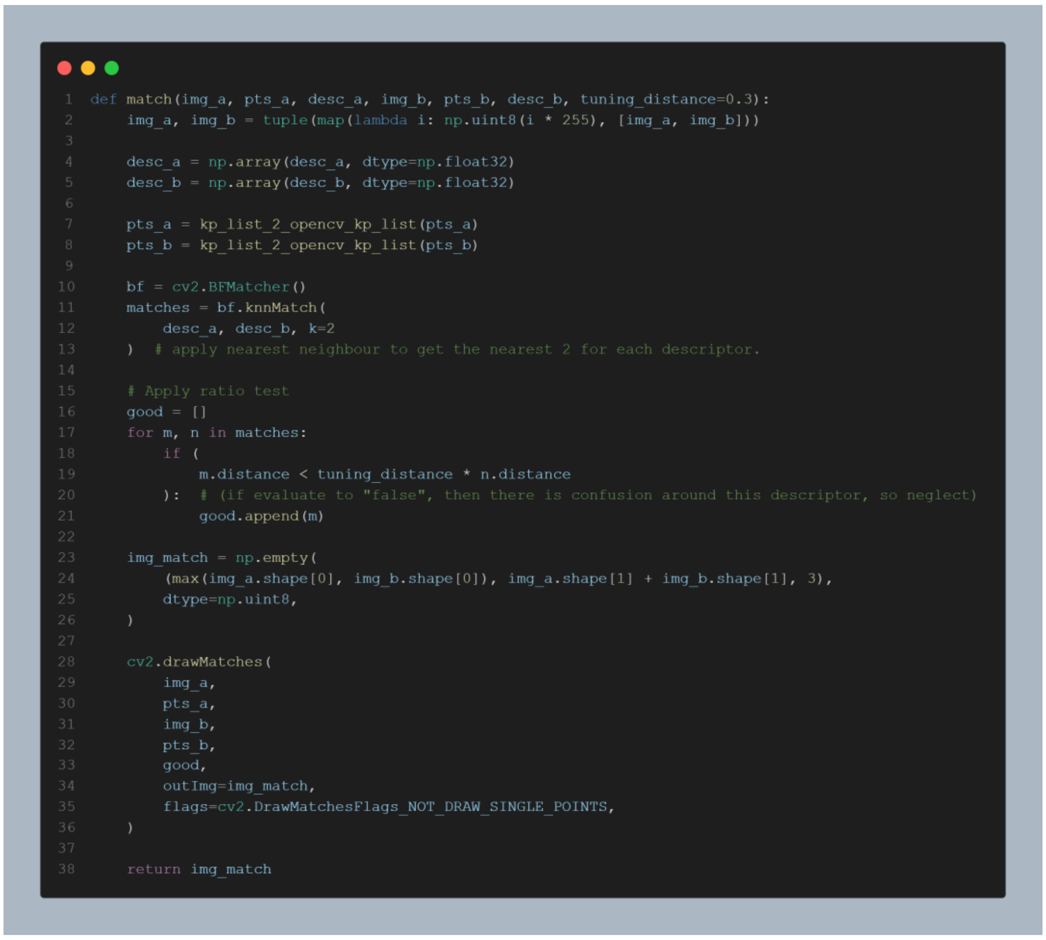
Remember, we might have more than one descriptor for the same keypoint.

After extracting the descriptor, we would normalize it to be robust for differences in linear illumination. Afterwards, we would clip all the values that are above (0.2), this will help us to eliminate the non-linear difference in illumination. Finally, we would renormalize the descriptor vector. Now, it becomes ready for matching.

Matching

To extract matches between the keypoints descriptors of the two input images (target & template). We have used the SSD as our computational method in the context of the k-nearest neighbour algorithm with (k = 2), meaning that we get the nearest 2 descriptors between the template and the target image. Then, using a confusion factor we can exclude or include the match into our set of matches. Phrased differently, we compute the distance between the descriptor of the template and the 2 possible matches in the target image. If the distances differ significantly, we include the match of the least distance. Otherwise, exclude the match to avoid confusion. **If the confusion factor is set to high value this means more matches are included that might be incorrect and vice versa.**

To enhance the computation time, we have used a built-in kNN algorithm that is more efficient in terms of computational time. Because applying such brute force matching algorithms using SSD would have been extremely expensive. Therefore, we have used **"cv2.BFMatcher.knnMatch()"**



```python
def match(img_a, pts_a, desc_a, img_b, pts_b, desc_b, tuning_distance=0.3):
    img_a, img_b = tuple(map(lambda i: np.uint8(i * 255), [img_a, img_b]))

    desc_a = np.array(desc_a, dtype=np.float32)
    desc_b = np.array(desc_b, dtype=np.float32)

    pts_a = kp_list_2_opencv_kp_list(pts_a)
    pts_b = kp_list_2_opencv_kp_list(pts_b)

    bf = cv2.BFMatcher()
    matches = bf.knnMatch(
        desc_a, desc_b, k=2
    )  # apply nearest neighbour to get the nearest 2 for each descriptor.

    # Apply ratio test
    good = []
    for m, n in matches:
        if (
            m.distance < tuning_distance * n.distance
        ):  # (if evaluate to "false", then there is confusion around this descriptor, so neglect)
            good.append(m)

    img_match = np.empty(
        (max(img_a.shape[0], img_b.shape[0]), img_a.shape[1] + img_b.shape[1], 3),
        dtype=np.uint8,
    )

    cv2.drawMatches(
        img_a,
        pts_a,
        img_b,
        pts_b,
        good,
        outImg=img_match,
        flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS,
    )

    return img_match
```

Figure 3, Sample Code of Matching Function

Output Samples

Moscow Cathedral Mosque

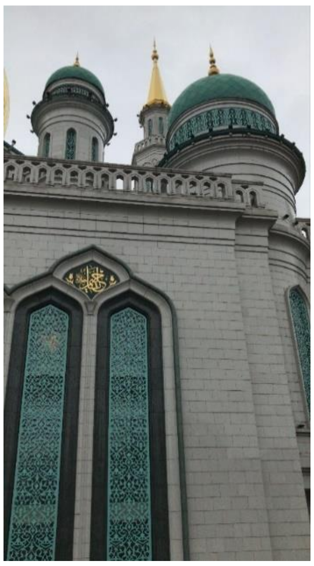


Figure 4, Target Image

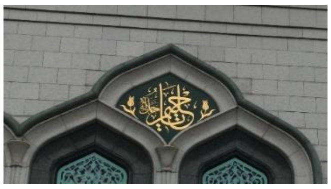


Figure 5, Template Image

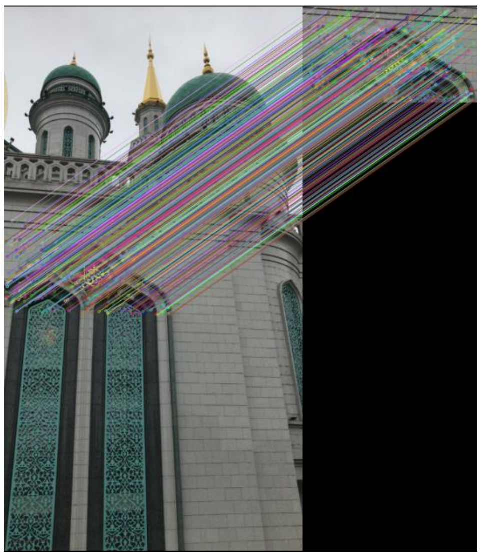


Figure 6, the matching output shows invariance to scale. The matching process has been conducted at a confusion factor of 0.3. This results in including more matches that are incorrect

You can think of the confusion factor as our confidence factor. A lower value indicates higher confidence in the accuracy of the matches.

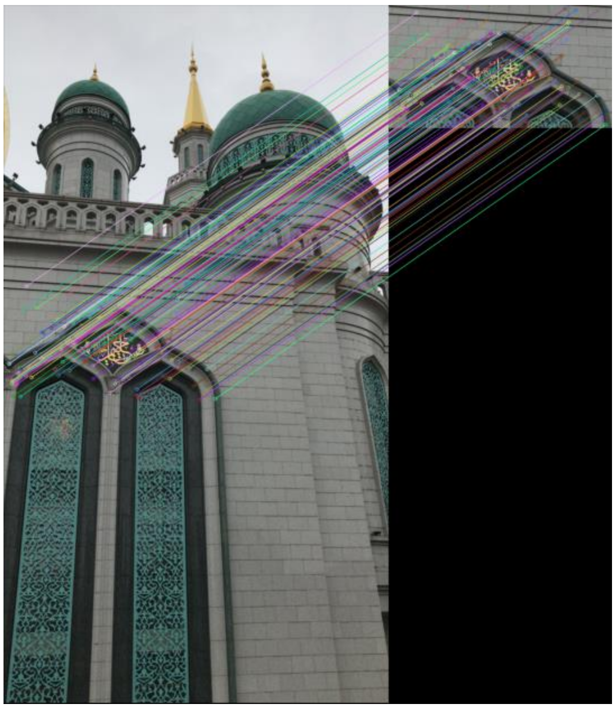


Figure 7, the matching process has been conducted at a confusion factor of 0.2. The matching process is more conservative leading to fewer matches, but higher accuracy

Brandenburg Gate

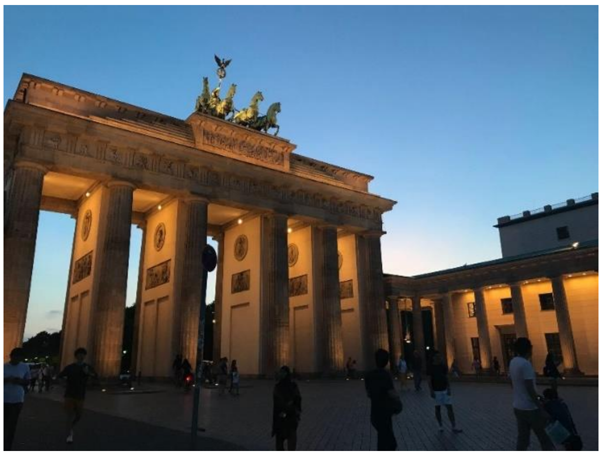


Figure 8, Target Image

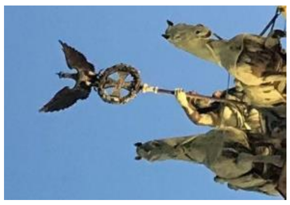


Figure 9, Template Image

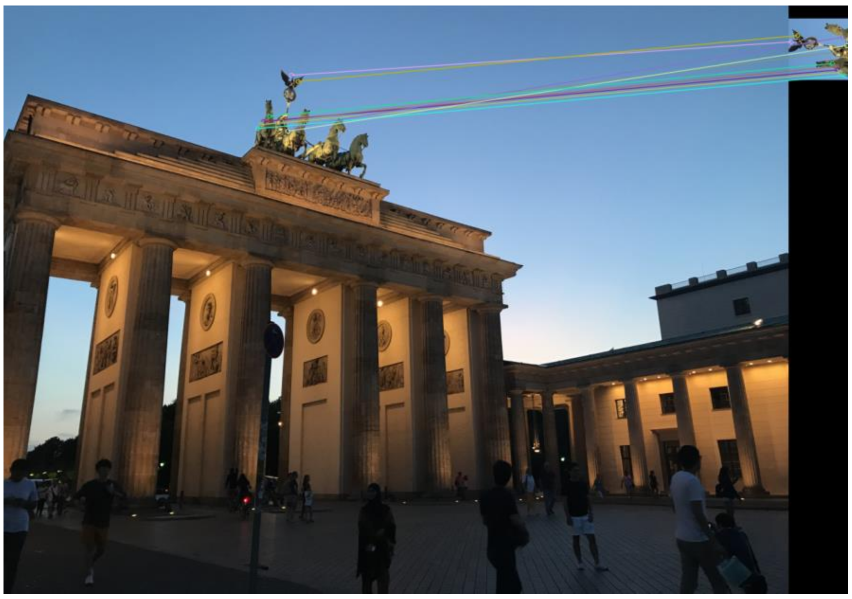


Figure 10, the matching output shows invariant in rotation, with high accuracy.

The computational time for the SIFT algorithm is in the range of **(195 – 205) seconds.**