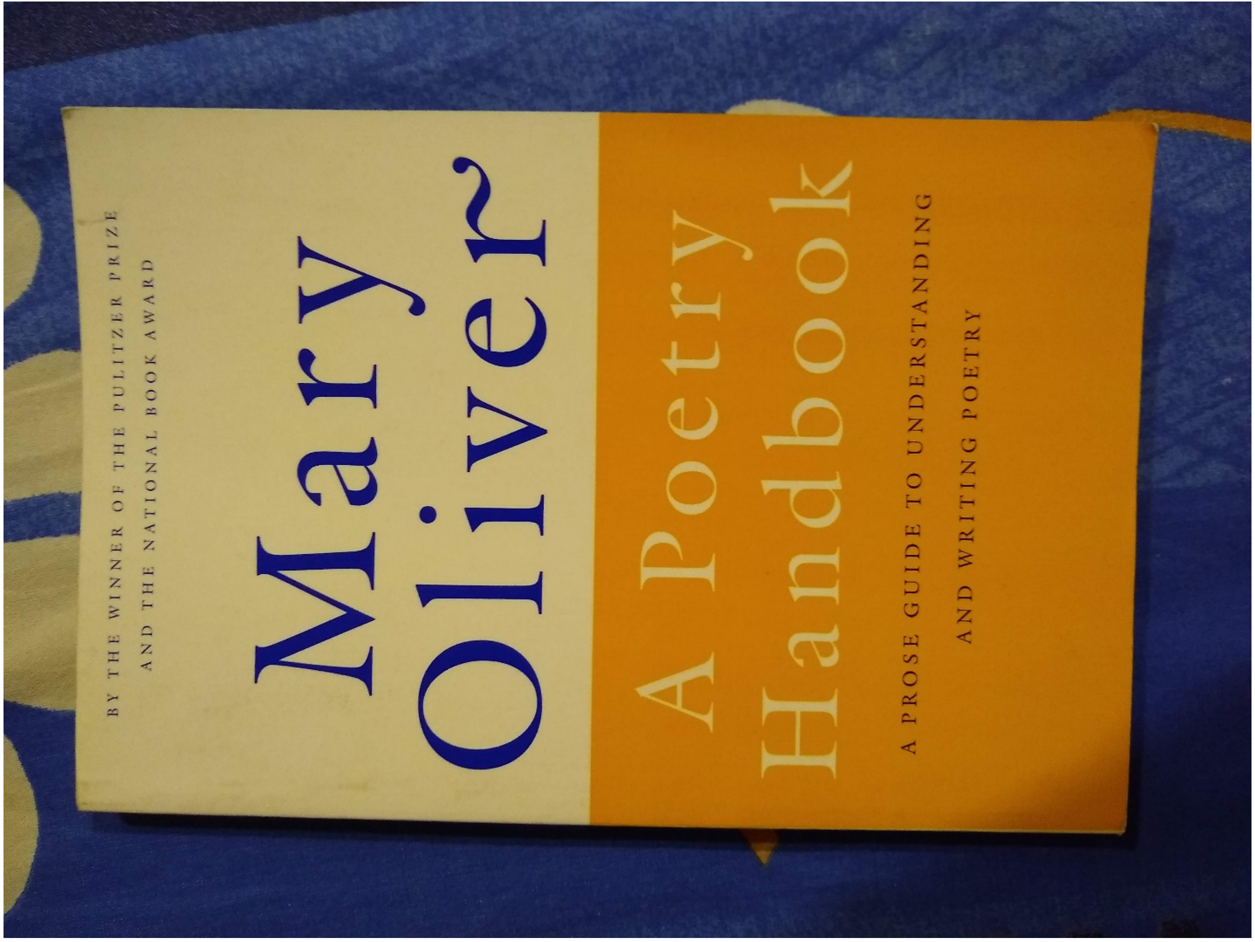
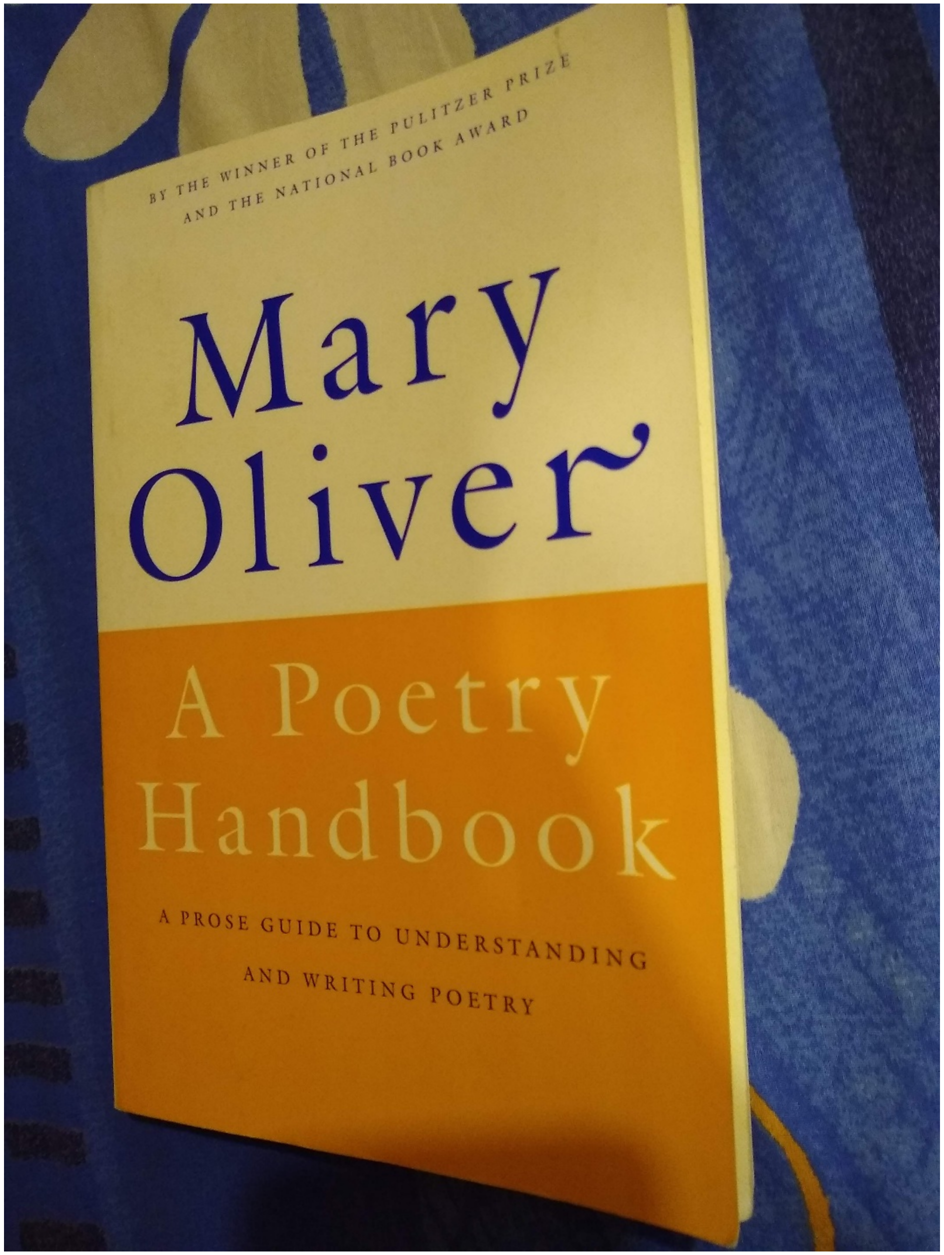
Image registration is a digital image processing technique that helps us align different images of the same scene. For instance, one may click the picture of a book from various angles. Below are a few instances that show the diversity of camera angles.

Now, we may want to “align” a particular image to the same angle as a reference image. In the images above, one may consider the first image to be an “ideal” cover photo, while the second and third images do not serve well for book cover photo purposes. The image registration algorithm helps us align the second and third pictures to the same plane as the first one.


BY THE WINNER OF THE PULITZER PRIZE
AND THE NATIONAL BOOK AWARD
Mary
Oliver
A Poetry
Handbook
A PROSE GUIDE TO UNDERSTANDING
AND WRITING POETRY

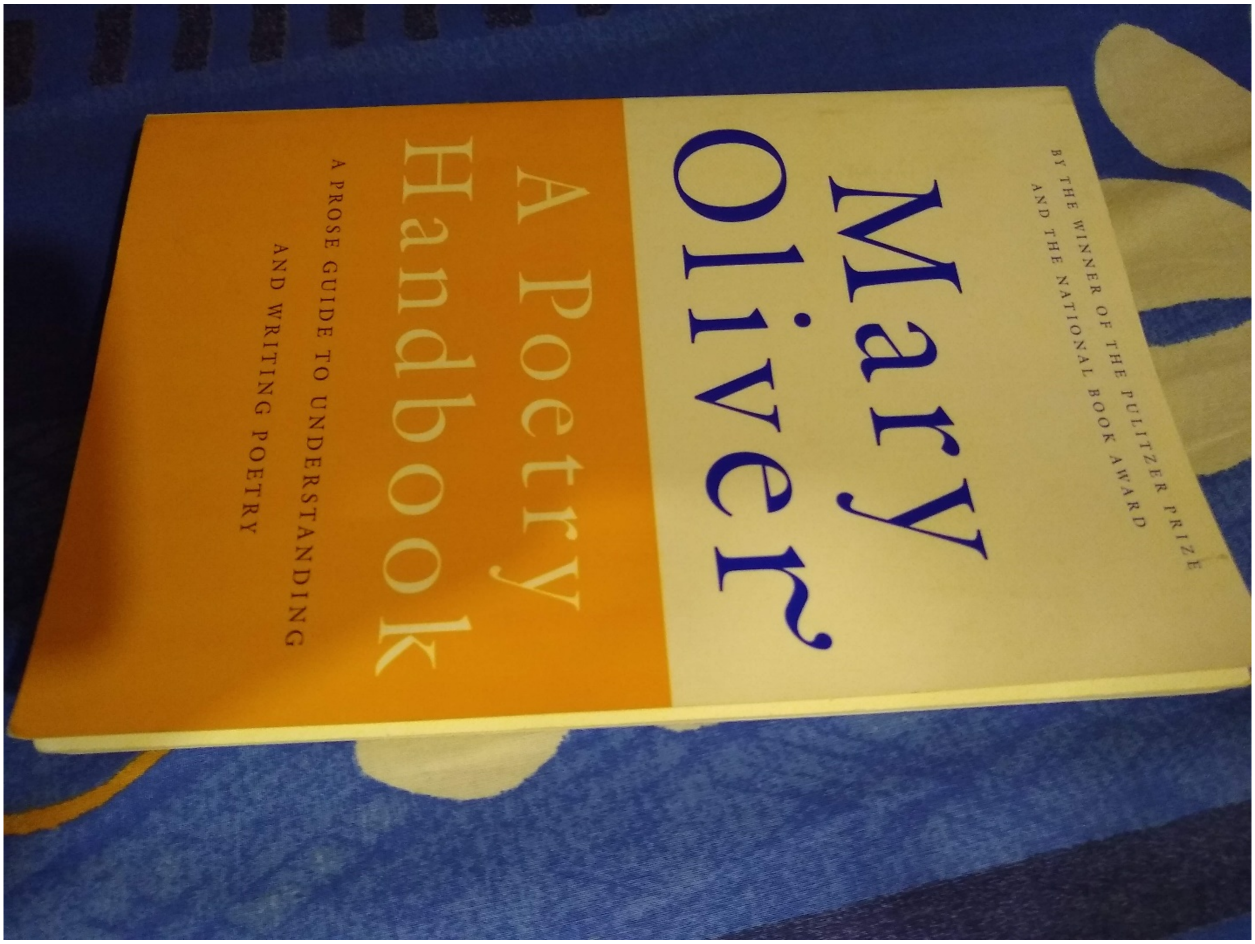


BY THE WINNER OF THE PULITZER PRIZE
AND THE NATIONAL BOOK AWARD

Mary Oliver

A Poetry Handbook

A PROSE GUIDE TO UNDERSTANDING
AND WRITING POETRY


BY THE WINNER OF THE PULITZER PRIZE
AND THE NATIONAL BOOK AWARD
Mary Oliver
A Poetry Handbook
A PROSE GUIDE TO UNDERSTANDING
AND WRITING POETRY

How does image registration work?

Alignment can be looked at as a simple coordinate transform. The algorithm works as follows:

- Convert both images to grayscale.
- Match features from the image to be aligned, to the reference image and store the coordinates of the corresponding key points. Keypoints are simply the selected few points that are used to compute the transform (generally points that stand out), and descriptors are histograms of the image gradients to characterize the appearance of a keypoint. In this post, we use ORB (Oriented FAST and Rotated BRIEF) implementation in the OpenCV library, which provides us with both key points as well as their associated descriptors.
- Match the key points between the two images. In this post, we use BFMatcher, which is a brute force matcher. BFMatcher.match() retrieves the best match, while BFMatcher.knnMatch() retrieves top K matches, where K is specified by the user.
- Pick the top matches, and remove the noisy matches.
- Find the homomorphy transform.

- Apply this transform to the original unaligned image to get the output image.

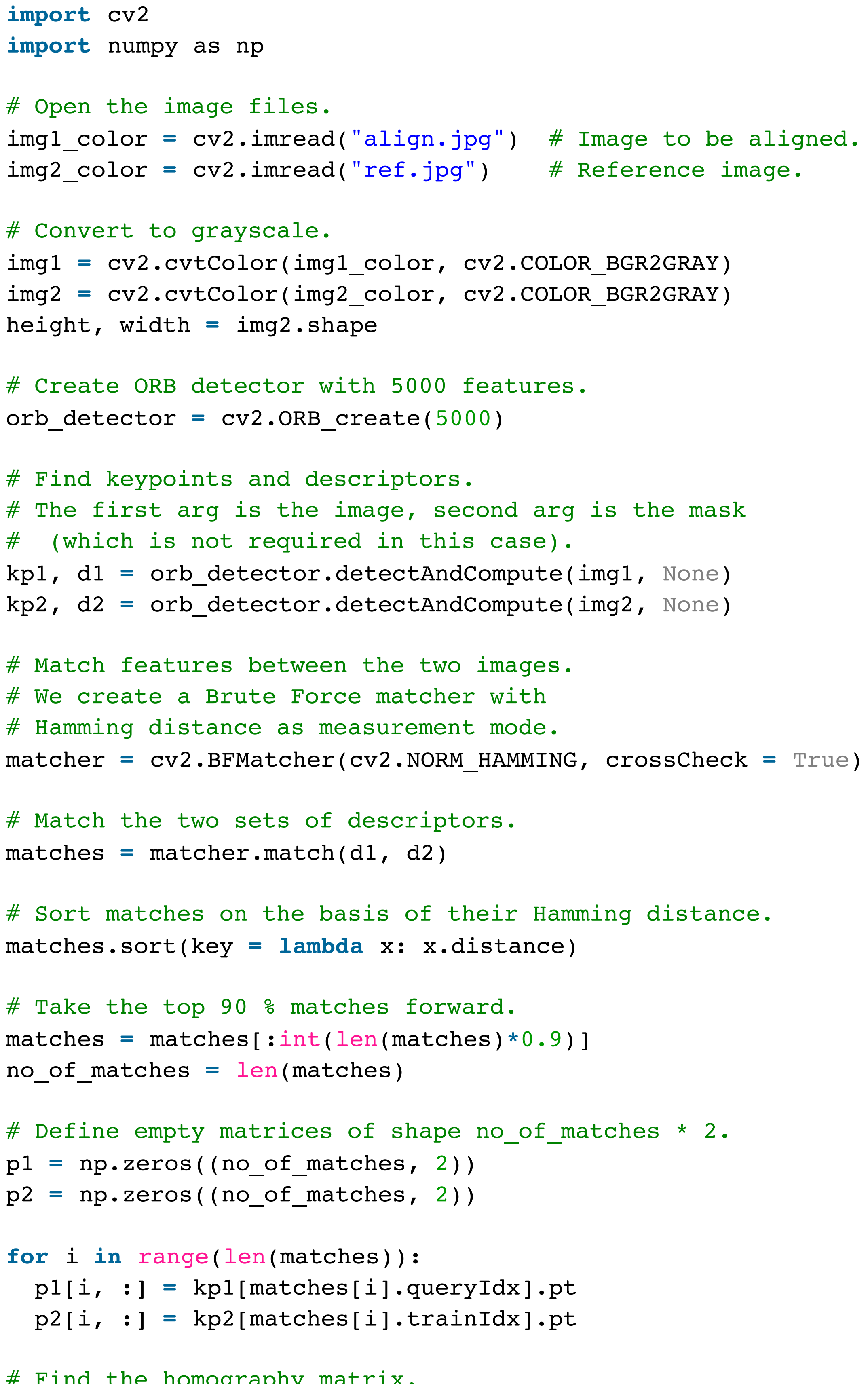
Applications of Image Registration –

Some of the useful applications of image registration include:

- Stitching various scenes (which may or may not have the same camera alignment) together to form a continuous panoramic shot.
- Aligning camera images of documents to a standard alignment to create realistic scanned documents.
- Aligning medical images for better observation and analysis.

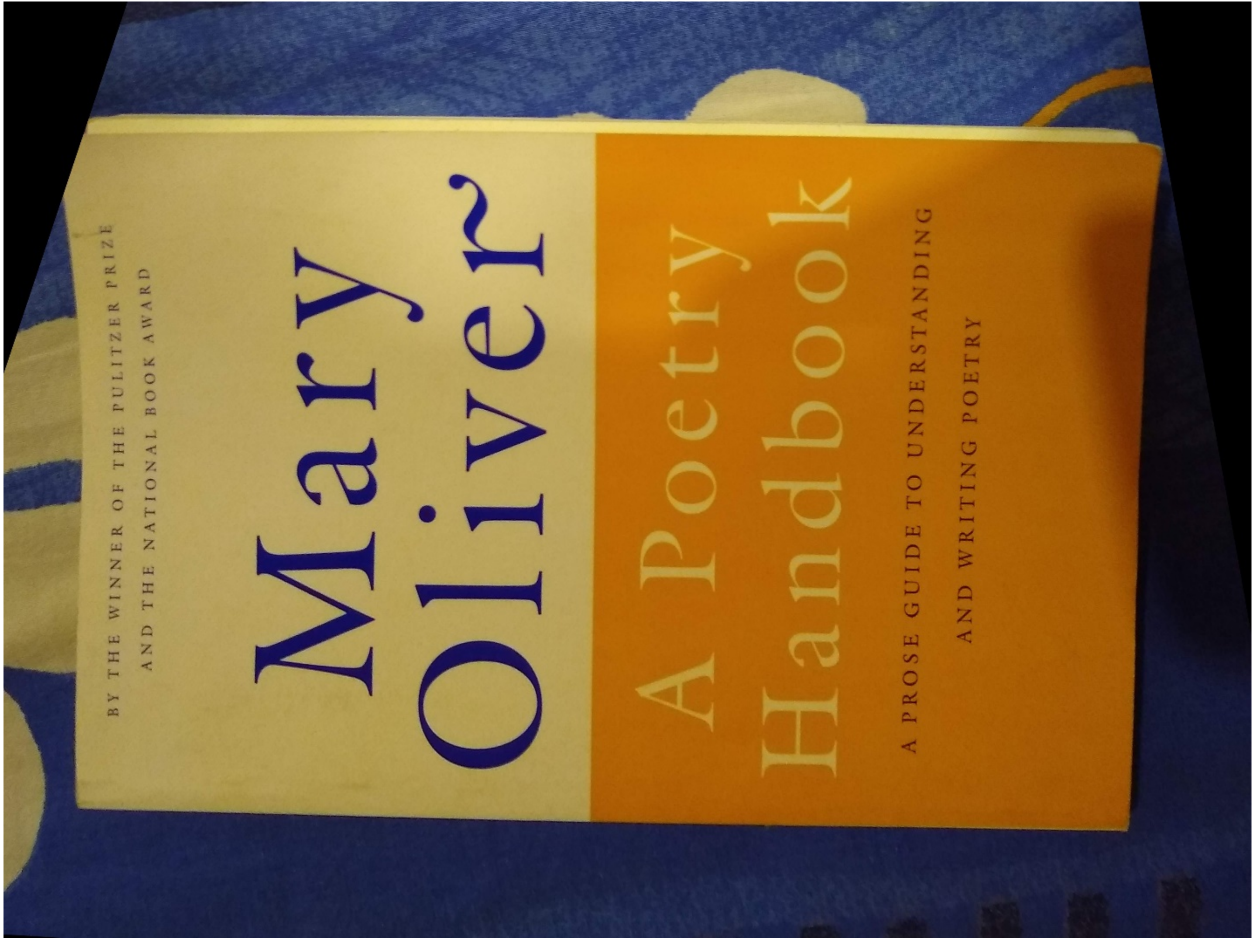
Below is the code for image registration. We have aligned the second image with reference to the third image.

- Python



```python
import cv2
import numpy as np

# Open the image files.
img1_color = cv2.imread("align.jpg")  # Image to be aligned.
img2_color = cv2.imread("ref.jpg")    # Reference image.

# Convert to grayscale.
img1 = cv2.cvtColor(img1_color, cv2.COLOR_BGR2GRAY)
img2 = cv2.cvtColor(img2_color, cv2.COLOR_BGR2GRAY)
height, width = img2.shape

# Create ORB detector with 5000 features.
orb_detector = cv2.ORB_create(5000)

# Find keypoints and descriptors.
# The first arg is the image, second arg is the mask
#  (which is not required in this case).
kp1, d1 = orb_detector.detectAndCompute(img1, None)
kp2, d2 = orb_detector.detectAndCompute(img2, None)

# Match features between the two images.
# We create a Brute Force matcher with
# Hamming distance as measurement mode.
matcher = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck = True)

# Match the two sets of descriptors.
matches = matcher.match(d1, d2)

# Sort matches on the basis of their Hamming distance.
matches.sort(key = lambda x: x.distance)

# Take the top 90 % matches forward.
matches = matches[:int(len(matches)*0.9)]
no_of_matches = len(matches)

# Define empty matrices of shape no_of_matches * 2.
p1 = np.zeros((no_of_matches, 2))
p2 = np.zeros((no_of_matches, 2))

for i in range(len(matches)):
  p1[i, :] = kp1[matches[i].queryIdx].pt
  p2[i, :] = kp2[matches[i].trainIdx].pt

# Find the homography matrix.
```

Output:


BY THE WINNER OF THE PULITZER PRIZE
AND THE NATIONAL BOOK AWARD
Mary Oliver
A Poetry Handbook
A PROSE GUIDE TO UNDERSTANDING
AND WRITING POETRY

Explanation

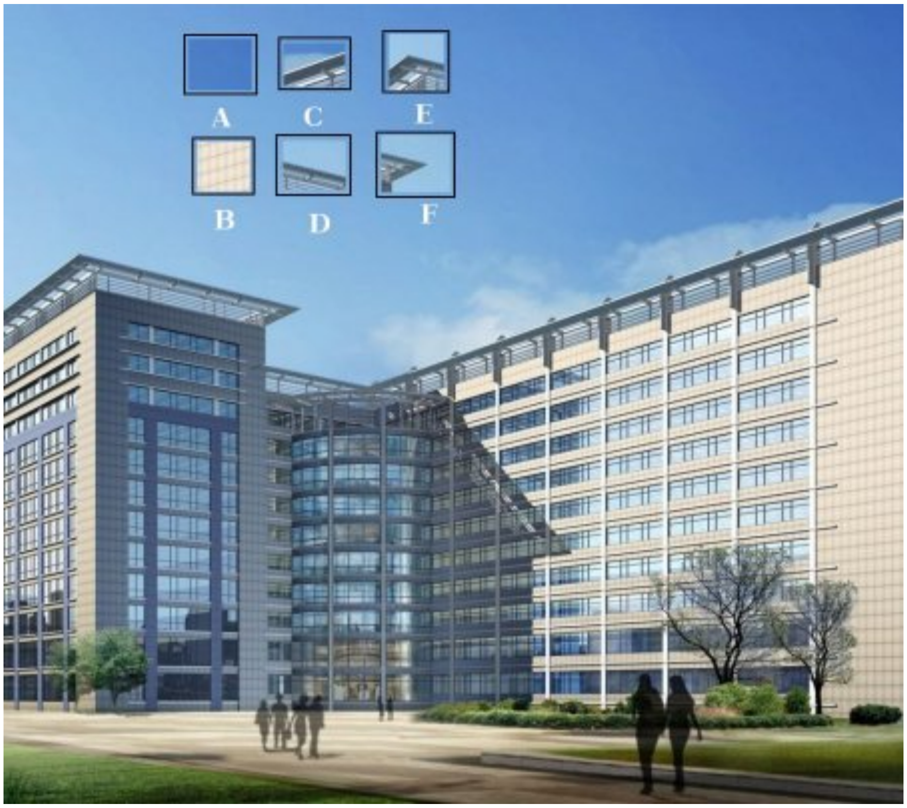
Most of you will have played the jigsaw puzzle games. You get a lot of small pieces of an image, where you need to assemble them correctly to form a big real image. **The question is, how you do it?** What about the projecting the same theory to a computer program so that computer can play jigsaw puzzles? If the computer can play jigsaw puzzles, why can't we give a lot of real-life images of a good natural scenery to computer and tell it to stitch all those images to a big single image? If the computer can stitch several natural images to one, what about giving a lot of pictures of a building or any structure and tell computer to create a 3D model out of it?

Well, the questions and imaginations continue. But it all depends on the most basic question: How do you play jigsaw puzzles? How do you arrange lots of scrambled image pieces into a big single image? How can you stitch a lot of natural images to a single image?

The answer is, we are looking for specific patterns or specific features which are unique, can be easily tracked and can be easily compared. If we go for a definition of such a feature, we may find it difficult to express it in words, but we know what they are. If someone asks you to point out one good feature which can be compared across several images, you can point out one. That is why even small children can simply play these games. We search for these features in an image, find them, look for the same features in other images and align them. That's it. (In jigsaw puzzle, we look more into continuity of different images). All these abilities are present in us inherently.

So our one basic question expands to more in number, but becomes more specific. **What are these features?**. (The answer should be understandable also to a computer.)

It is difficult to say how humans find these features. This is already programmed in our brain. But if we look deep into some pictures and search for different patterns, we will find something interesting. For example, take below image:



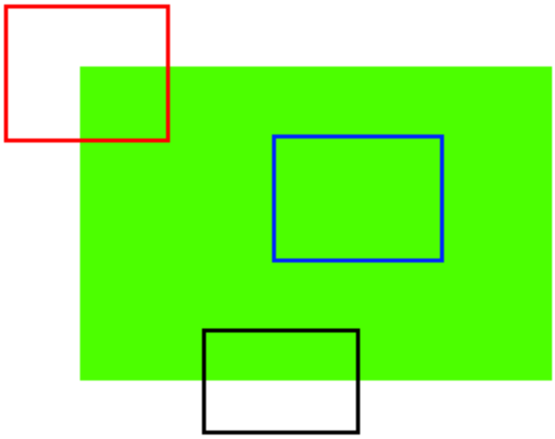
image

The image is very simple. At the top of image, six small image patches are given. Question for you is to find the exact location of these patches in the original image. How many correct results can you find?

A and B are flat surfaces and they are spread over a lot of area. It is difficult to find the exact location of these patches.

C and D are much more simple. They are edges of the building. You can find an approximate location, but exact location is still difficult. This is because the pattern is same everywhere along the edge. At the edge, however, it is different. An edge is therefore better feature compared to flat area, but not good enough (It is good in jigsaw puzzle for comparing continuity of edges).

Finally, E and F are some corners of the building. And they can be easily found. Because at the corners, wherever you move this patch, it will look different. So they can be considered as good features. So now we move into simpler (and widely used image) for better understanding.



image

Just like above, the blue patch is flat area and difficult to find and track. Wherever you move the blue patch it looks the same. The black patch has an edge. If you move it in vertical direction (i.e. along the gradient) it changes. Moved along the edge (parallel to edge), it looks the same. And for red patch, it is a corner. Wherever you move the patch, it looks different, means it is unique. So basically, corners are considered to be good features in an image. (Not just corners, in some cases blobs are considered good features).

So now we answered our question, "what are these features?". But next question arises. How do we find them? Or how do we find the corners?. We answered that in an intuitive way, i.e., look for the regions in images which have maximum variation when moved (by a small amount) in all regions around it. This would be projected into computer language in coming chapters. So finding these image features is called **Feature Detection**.

We found the features in the images. Once you have found it, you should be able to find the same in the other images. How is this done? We take a region around the feature, we explain it in our own words, like "upper part is blue sky, lower part is region from a building, on that building there is glass etc" and you search for the same area in the other images. Basically, you are describing the feature. Similarly, a computer also should describe the region around the feature so that it can find it in other images. So called description is called **Feature Description**. Once you have the features and its description, you can find same features in all images and align them, stitch them together or do whatever you want.

Goal

In this chapter,

- We will understand the concepts behind Harris Corner Detection.
- We will see the following functions: **cv.cornerHarris()**, **cv.cornerSubPix()**

Theory

In the last chapter, we saw that corners are regions in the image with large variation in intensity in all the directions. One early attempt to find these corners was done by **Chris Harris & Mike Stephens** in their paper **A Combined Corner and Edge Detector** in 1988, so now it is called the Harris Corner Detector. He took this simple idea to a mathematical form. It basically finds the difference in intensity for a displacement of (u,v) in all directions. This is expressed as below:

$$E(u,v)=\sum_{x,y} w(x,y)\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\text{window function}[I(x+u,y+v)\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\text{shifted intensity}-I(x,y)\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\boxed{?}\text{intensity}]2$$

The window function is either a rectangular window or a Gaussian window which gives weights to pixels underneath.

We have to maximize this function $E(u,v)$ for corner detection. That means we have to maximize the second term. Applying Taylor Expansion to the above equation and using some mathematical steps (please refer to any standard text books you like for full derivation), we get the final equation as:

$$E(u,v)\approx[uv]M[uv]$$

where

$$M=\sum_{x,y} w(x,y)[I_xI_xI_xI_yI_xI_yI_yI_y]$$

Here, I_x and I_y are image derivatives in x and y directions respectively. (These can be easily found using **cv.Sobel()**).

Then comes the main part. After this, they created a score, basically an equation, which determines if a window can contain a corner or not.

$$R=det(M)-k(trace(M))_2$$

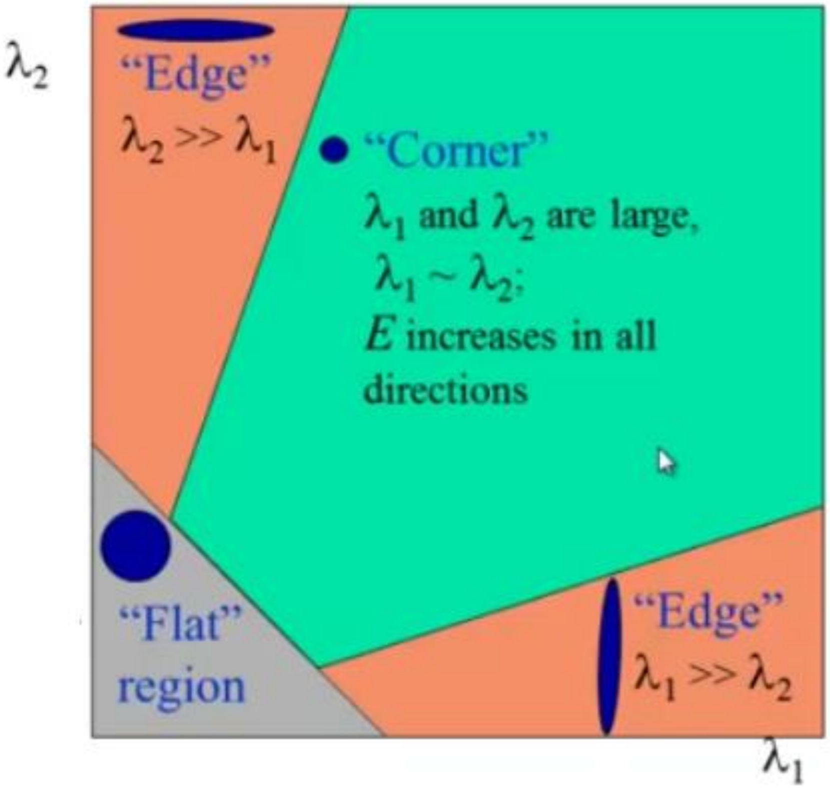
where

- $det(M)=\lambda_1\lambda_2$
- $trace(M)=\lambda_1+\lambda_2$
- λ_1 and λ_2 are the eigenvalues of M

So the magnitudes of these eigenvalues decide whether a region is a corner, an edge, or flat.

- When $|R|$ is small, which happens when λ_1 and λ_2 are small, the region is flat.
- When $R<0$, which happens when $\lambda_1>>\lambda_2$ or vice versa, the region is edge.
- When R is large, which happens when λ_1 and λ_2 are large and $\lambda_1\sim\lambda_2$, the region is a corner.

It can be represented in a nice picture as follows:



image

So the result of Harris Corner Detection is a grayscale image with these scores. Thresholding for a suitable score gives you the corners in the image. We will do it with a simple image.

Harris Corner Detector in OpenCV

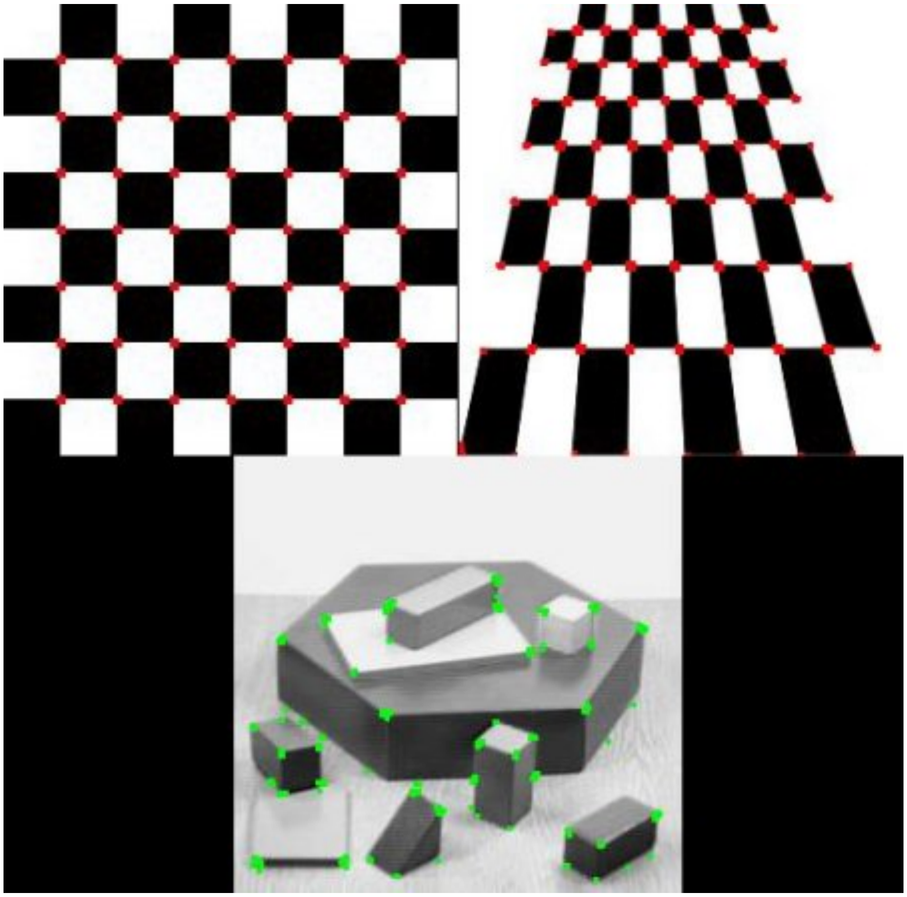
OpenCV has the function cv.cornerHarris() for this purpose. Its arguments are:

- **img -** Input image. It should be grayscale and float32 type.
- **blockSize -** It is the size of neighbourhood considered for corner detection
- **ksize -** Aperture parameter of the Sobel derivative used.
- **k -** Harris detector free parameter in the equation.

See the example below:

```
import numpy as np

import cv2 as cv

filename = 'chessboard.png'

img = cv.imread(filename)

gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
gray = np.float32(gray)

dst = cv.cornerHarris(gray,2,3,0.04)

#result is dilated for marking the corners, not important

dst = cv.dilate(dst,None)

# Threshold for an optimal value, it may vary depending on the image.

img[dst>0.01*dst.max()]=[0,0,255]

cv.imshow('dst',img)

if cv.waitKey(0) & 0xff == 27:

cv.destroyAllWindows()
```

Below are the three results:



image

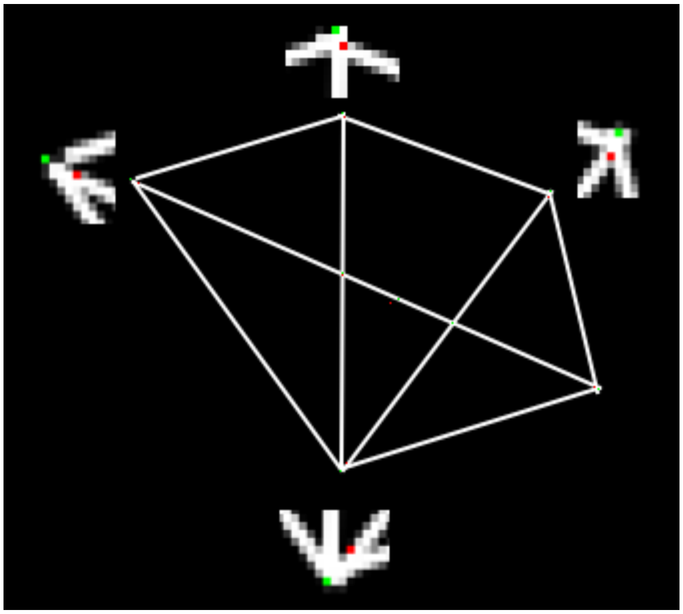
Corner with SubPixel Accuracy

Sometimes, you may need to find the corners with maximum accuracy. OpenCV comes with a function cv.cornerSubPix() which further refines the corners detected with sub-pixel accuracy.

Below is an example. As usual, we need to find the Harris corners first. Then we pass the centroids of these corners (There may be a bunch of pixels at a corner, we take their centroid) to refine them. Harris corners are marked in red pixels and refined corners are marked in green pixels. For this function, we have to define the criteria when to stop the iteration. We stop it after a specified number of iterations or a certain accuracy is achieved, whichever occurs first. We also need to define the size of the neighbourhood it searches for corners.

```
import numpy as np
import cv2 as cv
filename = 'chessboard2.jpg'
img = cv.imread(filename)
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
# find Harris corners
gray = np.float32(gray)
dst = cv.cornerHarris(gray,2,3,0.04)
dst = cv.dilate(dst,None)
ret, dst = cv.threshold(dst,0.01*dst.max(),255,0)
dst = np.uint8(dst)
# find centroids
ret, labels, stats, centroids = cv.connectedComponentsWithStats(dst)
# define the criteria to stop and refine the corners
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 100, 0.001)
corners = cv.cornerSubPix(gray,np.float32(centroids),(5,5),(-1,-1),criteria)
# Now draw them
res = np.hstack((centroids,corners))
res = np.int0(res)
img[res[:,1],res[:,0]]=[0,0,255]
img[res[:,3],res[:,2]] = [0,255,0]
cv.imwrite('subpixel5.png',img)
```

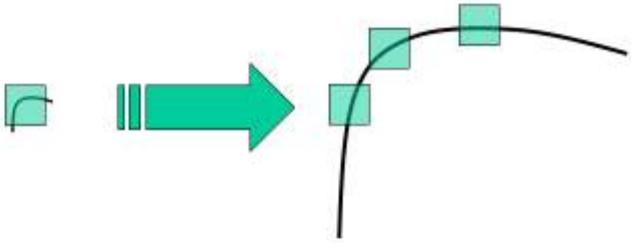
Below is the result, where some important locations are shown in the zoomed window to visualize:



image

Theory

In last couple of chapters, we saw some corner detectors like Harris etc. They are rotation-invariant, which means, even if the image is rotated, we can find the same corners. It is obvious because corners remain corners in rotated image also. But what about scaling? A corner may not be a corner if the image is scaled. For example, check a simple image below. A corner in a small image within a small window is flat when it is zoomed in the same window. So Harris corner is not scale invariant.



image

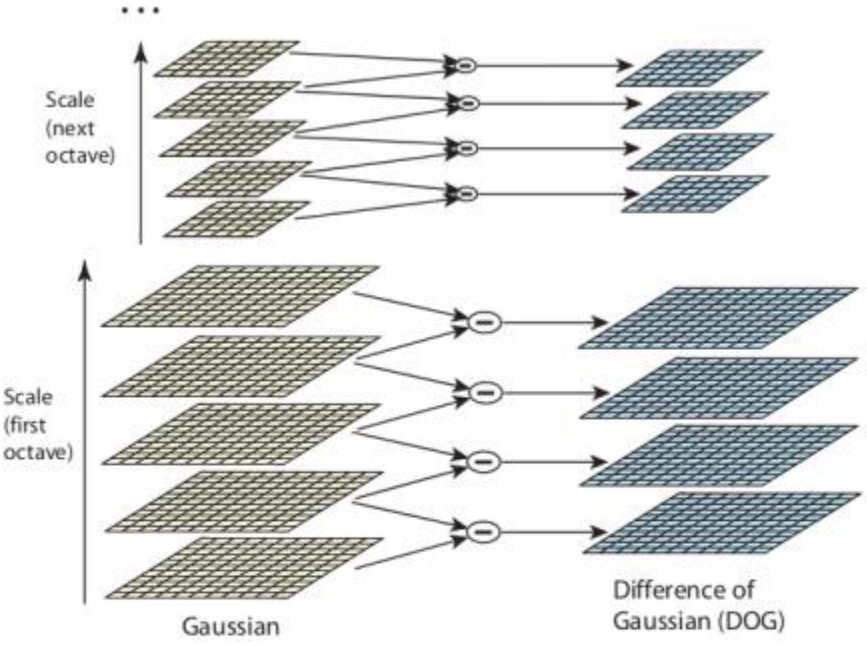
In 2004, **D.Lowe**, University of British Columbia, came up with a new algorithm, Scale Invariant Feature Transform (SIFT) in his paper, **Distinctive Image Features from Scale-Invariant Keypoints**, which extract keypoints and compute its descriptors. *(This paper is easy to understand and considered to be best material available on SIFT. This explanation is just a short summary of this paper)*.

There are mainly four steps involved in SIFT algorithm. We will see them one-by-one.

1. Scale-space Extrema Detection

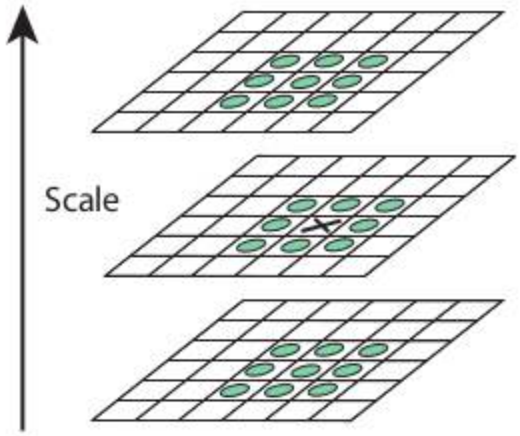
From the image above, it is obvious that we can't use the same window to detect keypoints with different scale. It is OK with small corner. But to detect larger corners we need larger windows. For this, scale-space filtering is used. In it, Laplacian of Gaussian is found for the image with various σ values. LoG acts as a blob detector which detects blobs in various sizes due to change in σ. In short, σ acts as a scaling parameter. For eg, in the above image, gaussian kernel with low σ gives high value for small corner while gaussian kernel with high σ fits well for larger corner. So, we can find the local maxima across the scale and space which gives us a list of (x,y,σ) values which means there is a potential keypoint at (x,y) at σ scale.

But this LoG is a little costly, so SIFT algorithm uses Difference of Gaussians which is an approximation of LoG. Difference of Gaussian is obtained as the difference of Gaussian blurring of an image with two different σ, let it be σ and $k\sigma$. This process is done for different octaves of the image in Gaussian Pyramid. It is represented in below image:



image

Once this DoG are found, images are searched for local extrema over scale and space. For eg, one pixel in an image is compared with its 8 neighbours as well as 9 pixels in next scale and 9 pixels in previous scales. If it is a local extrema, it is a potential keypoint. It basically means that keypoint is best represented in that scale. It is shown in below image:



image

Regarding different parameters, the paper gives some empirical data which can be summarized as, number of octaves = 4, number of scale levels = 5, initial $\sigma=1.6$, $k=2-\sqrt{}$ etc as optimal values.

2. Keypoint Localization

Once potential keypoints locations are found, they have to be refined to get more accurate results. They used Taylor series expansion of scale space to get more accurate location of extrema, and if the intensity at this extrema is less than a threshold value (0.03 as per the paper), it is rejected. This threshold is called **contrastThreshold** in OpenCV

DoG has higher response for edges, so edges also need to be removed. For this, a concept similar to Harris corner detector is used. They used a 2x2 Hessian matrix (H) to compute the principal curvature. We know from Harris corner detector that for edges, one eigen value is larger than the other. So here they used a simple function,

If this ratio is greater than a threshold, called **edgeThreshold** in OpenCV, that keypoint is discarded. It is given as 10 in paper.

So it eliminates any low-contrast keypoints and edge keypoints and what remains is strong interest points.

3. Orientation Assignment

Now an orientation is assigned to each keypoint to achieve invariance to image rotation. A neighbourhood is taken around the keypoint location depending on the scale, and the gradient magnitude and direction is calculated in that region. An orientation histogram with 36 bins covering 360 degrees is created (It is weighted by gradient magnitude and gaussian-weighted circular window with σ equal to 1.5 times the scale of keypoint). The highest peak in the histogram is taken and any peak above 80% of it is also considered to calculate the orientation. It creates keypoints with same location and scale, but different directions. It contribute to stability of matching.

4. Keypoint Descriptor

Now keypoint descriptor is created. A 16x16 neighbourhood around the keypoint is taken. It is divided into 16 sub-blocks of 4x4 size. For each sub-block, 8 bin orientation histogram is created. So a total of 128 bin values are available. It is represented as a vector to form keypoint descriptor. In addition to this, several measures are taken to achieve robustness against illumination changes, rotation etc.

5. Keypoint Matching

Keypoints between two images are matched by identifying their nearest neighbours. But in some cases, the second closest-match may be very near to the first. It may happen due to noise or some other reasons. In that case, ratio of closest-distance to second-closest distance is taken. If it is greater than 0.8, they are rejected. It eliminates around 90% of false matches while discards only 5% correct matches, as per the paper.

This is a summary of SIFT algorithm. For more details and understanding, reading the original paper is highly recommended.

SIFT in OpenCV

Now let's see SIFT functionalities available in OpenCV. Note that these were previously only available in the opencv contrib repo, but the patent expired in the year 2020. So they are now included in the main repo. Let's start with keypoint detection and draw them. First we have to construct a SIFT object. We can pass different parameters to it which are optional and they are well explained in docs.

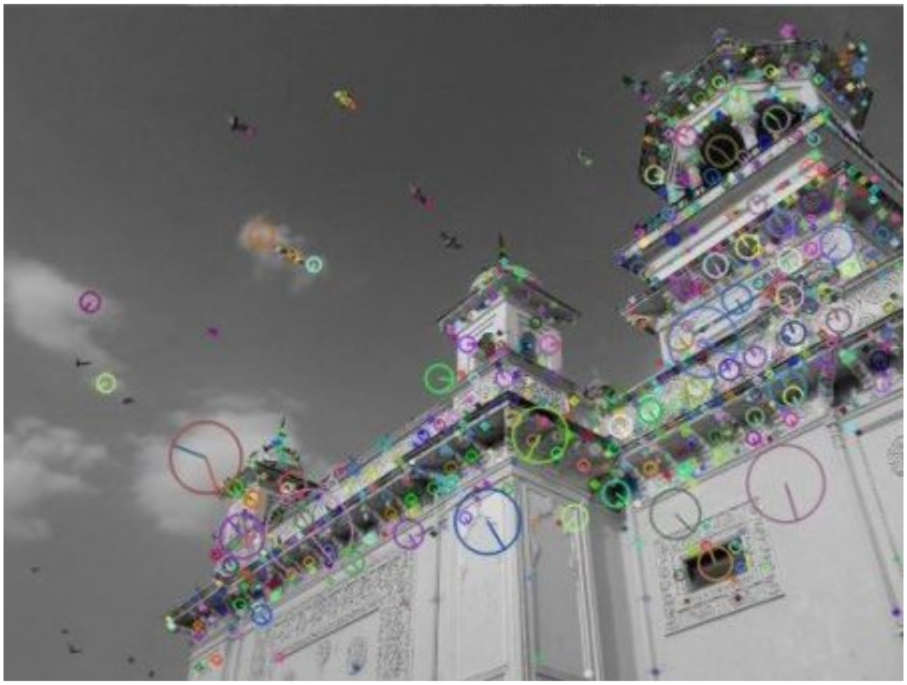
```
import numpy as np

import cv2 as cv

img = cv.imread('home.jpg')

gray= cv.cvtColor(img,cv.COLOR_BGR2GRAY)

sift = cv.SIFT_create()

kp = sift.detect(gray,None)

img=cv.drawKeypoints(gray,kp,img)

cv.imwrite('sift_keypoints.jpg',img)
```

sift.detect() function finds the keypoint in the images. You can pass a mask if you want to search only a part of image. Each keypoint is a special structure which has many attributes like its (x,y) coordinates, size of the meaningful neighbourhood, angle which specifies its orientation, response that specifies strength of keypoints etc.

OpenCV also provides **cv.drawKeyPoints()** function which draws the small circles on the locations of keypoints. If you pass a flag, **cv.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS** to it, it will draw a circle with size of keypoint and it will even show its orientation. See below example.

```
img=cv.drawKeypoints(gray,kp,img,flags=cv.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOI
NTS)

cv.imwrite('sift_keypoints.jpg',img)
```

See the two results below:



image

Now to calculate the descriptor, OpenCV provides two methods.

1. Since you already found keypoints, you can call **sift.compute()** which computes the descriptors from the keypoints we have found. Eg: kp,des = sift.compute(gray,kp)
2. If you didn't find keypoints, directly find keypoints and descriptors in a single step with the function, **sift.detectAndCompute()**.

We will see the second method:

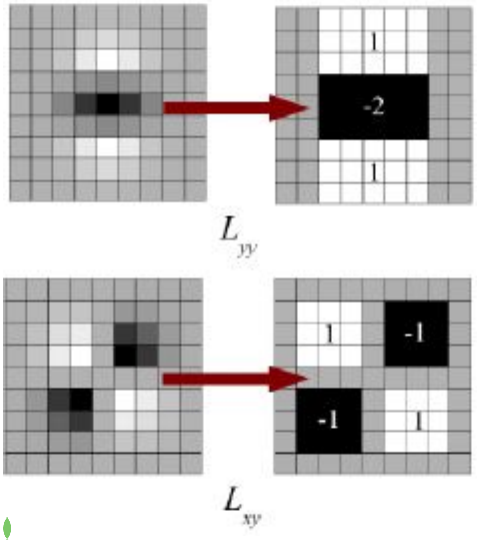
```
sift = cv.SIFT_create()

kp, des = sift.detectAndCompute(gray,None)
```

Here kp will be a list of keypoints and des is a numpy array of shape $(\text{Number of Keypoints}) \times 128$.

Theory

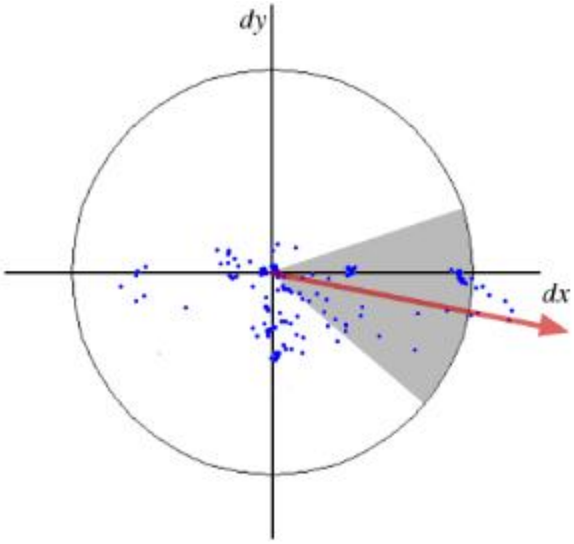
In last chapter, we saw SIFT for keypoint detection and description. But it was comparatively slow and people needed more speeded-up version. In 2006, three people, Bay, H., Tuytelaars, T. and Van Gool, L, published another paper, "SURF: Speeded Up Robust Features" which introduced a new algorithm called SURF. As name suggests, it is a speeded-up version of SIFT.

In SIFT, Lowe approximated Laplacian of Gaussian with Difference of Gaussian for finding scale-space. SURF goes a little further and approximates LoG with Box Filter. Below image shows a demonstration of such an approximation. One big advantage of this approximation is that, convolution with box filter can be easily calculated with the help of integral images. And it can be done in parallel for different scales. Also the SURF rely on determinant of Hessian matrix for both scale and location.



image

For orientation assignment, SURF uses wavelet responses in horizontal and vertical direction for a neighbourhood of size 6s. Adequate gaussian weights are also applied to it. Then they are plotted in a space as given in below image. The dominant orientation is estimated by calculating the sum of all responses within a sliding orientation window of angle 60 degrees. Interesting thing is that, wavelet response can be found out using integral images very easily at any scale. For many applications, rotation invariance is not required, so no need of finding this orientation, which speeds up the process. SURF provides such a functionality called Upright-SURF or U-SURF. It improves speed and is robust upto $\pm 15^{\circ}$. OpenCV supports both, depending upon the flag, **upright**. If it is 0, orientation is calculated. If it is 1, orientation is not calculated and it is faster.

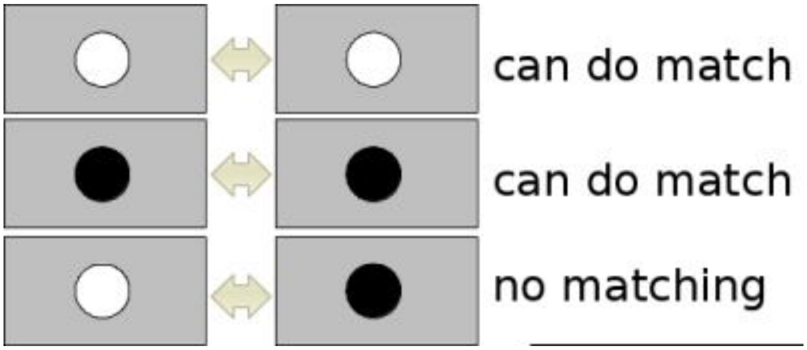


image

For feature description, SURF uses Wavelet responses in horizontal and vertical direction (again, use of integral images makes things easier). A neighbourhood of size 20sX20s is taken around the keypoint where s is the size. It is divided into 4x4 subregions. For each subregion, horizontal and vertical wavelet responses are taken and a vector is formed like this, $v=(\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|)$. This when represented as a vector gives SURF feature descriptor with total 64 dimensions. Lower the dimension, higher the speed of computation and matching, but provide better distinctiveness of features.

For more distinctiveness, SURF feature descriptor has an extended 128 dimension version. The sums of d_x and $|d_x|$ are computed separately for $d_y<0$ and $d_y \geq 0$. Similarly, the sums of d_y and $|d_y|$ are split up according to the sign of d_x , thereby doubling the number of features. It doesn't add much computation complexity. OpenCV supports both by setting the value of flag **extended** with 0 and 1 for 64-dim and 128-dim respectively (default is 128-dim)

Another important improvement is the use of sign of Laplacian (trace of Hessian Matrix) for underlying interest point. It adds no computation cost since it is already computed during detection. The sign of the Laplacian distinguishes bright blobs on dark backgrounds from the reverse situation. In the matching stage, we only compare features if they have the same type of contrast (as shown in image below). This minimal information allows for faster matching, without reducing the descriptor's performance.



image

In short, SURF adds a lot of features to improve the speed in every step. Analysis shows it is 3 times faster than SIFT while performance is comparable to SIFT. SURF is good at handling images with blurring and rotation, but not good at handling viewpoint change and illumination change.

SURF in OpenCV

OpenCV provides SURF functionalities just like SIFT. You initiate a SURF object with some optional conditions like 64/128-dim descriptors, Upright/Normal SURF etc. All the details are well explained in docs. Then as we did in SIFT, we can use SURF.detect(), SURF.compute() etc for finding keypoints and descriptors.

First we will see a simple demo on how to find SURF keypoints and descriptors and draw it. All examples are shown in Python terminal since it is just same as SIFT only.

```
>>> img = cv.imread('fly.png',0)

# Create SURF object. You can specify params here or later.

# Here I set Hessian Threshold to 400

>>> surf = cv.xfeatures2d.SURF_create(400)

# Find keypoints and descriptors directly

>>> kp, des = surf.detectAndCompute(img,None)

>>> len(kp)

699
```

1199 keypoints is too much to show in a picture. We reduce it to some 50 to draw it on an image. While matching, we may need all those features, but not now. So we increase the Hessian Threshold.

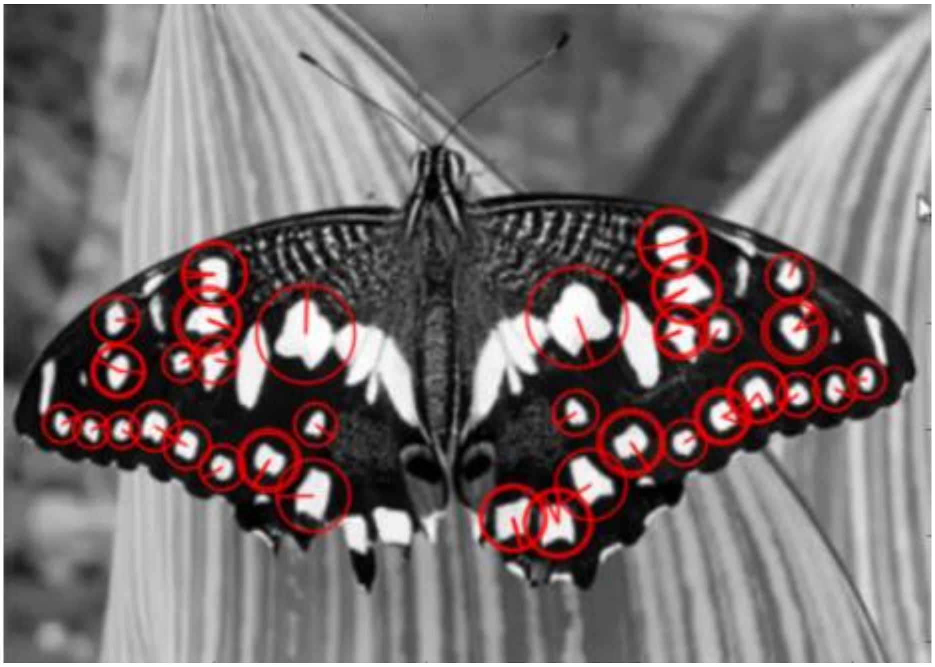
```
# Check present Hessian threshold

>>> print( surf.getHessianThreshold() )

400.0
```

```
# We set it to some 50000. Remember, it is just for representing in picture.
# In actual cases, it is better to have a value 300-500
>>> surf.setHessianThreshold(50000)
# Again compute keypoints and check its number.
>>> kp, des = surf.detectAndCompute(img,None)
>>> print( len(kp) )
47
```

It is less than 50. Let's draw it on the image.

```
>>> img2 = cv.drawKeypoints(img,kp,None,(255,0,0),4)
>>> plt.imshow(img2),plt.show()
```

See the result below. You can see that SURF is more like a blob detector. It detects the white blobs on wings of butterfly. You can test it with other images.



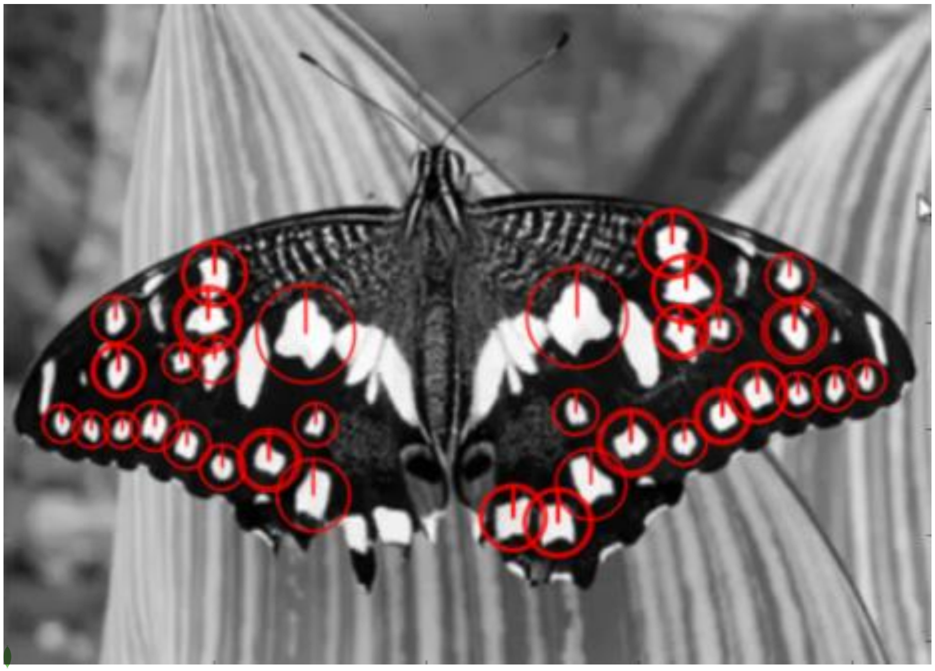
image

Now I want to apply U-SURF, so that it won't find the orientation.

```
# Check upright flag, if it False, set it to True
>>> print( surf.getUpright() )
False
>>> surf.setUpright(True)
# Recompute the feature points and draw it
```

```
>>> kp = surf.detect(img,None)

>>> img2 = cv.drawKeypoints(img,kp,None,(255,0,0),4)

>>> plt.imshow(img2),plt.show()
```

See the results below. All the orientations are shown in same direction. It is faster than previous. If you are working on cases where orientation is not a problem (like panorama stitching) etc, this is better.



image

Finally we check the descriptor size and change it to 128 if it is only 64-dim.

```
# Find size of descriptor

>>> print( surf.descriptorSize() )

64

# That means flag, "extended" is False.

>>> surf.getExtended()

False

# So we make it to True to get 128-dim descriptors.

>>> surf.setExtended(True)

>>> kp, des = surf.detectAndCompute(img,None)

>>> print( surf.descriptorSize() )

128
```

```
>>> print( des.shape )

(47, 128)
```

Theory

As an OpenCV enthusiast, the most important thing about the ORB is that it came from "OpenCV Labs". This algorithm was brought up by Ethan Rublee, Vincent Rabaud, Kurt Konolige and Gary R. Bradski in their paper **ORB: An efficient alternative to SIFT or SURF** in 2011. As the title says, it is a good alternative to SIFT and SURF in computation cost, matching performance and mainly the patents. Yes, SIFT and SURF are patented and you are supposed to pay them for its use. But ORB is not !!!

ORB is basically a fusion of FAST keypoint detector and BRIEF descriptor with many modifications to enhance the performance. First it use FAST to find keypoints, then apply Harris corner measure to find top N points among them. It also use pyramid to produce multiscale-features. But one problem is that, FAST doesn't compute the orientation. So what about rotation invariance? Authors came up with following modification.

It computes the intensity weighted centroid of the patch with located corner at center. The direction of the vector from this corner point to centroid gives the orientation. To improve the rotation invariance, moments are computed with x and y which should be in a circular region of radius r, where r is the size of the patch.
Now for descriptors, ORB use BRIEF descriptors. But we have already seen that BRIEF performs poorly with rotation. So what ORB does is to "steer" BRIEF according to the orientation of keypoints. For any feature set of n binary tests at location (x_i, y_i), define a $2 \times n$ matrix, S which contains the coordinates of these pixels. Then using the orientation of patch, θ, its rotation matrix is found and rotates the S to get steered(rotated) version S_θ.
ORB discretize the angle to increments of $2\pi/30$ (12 degrees), and construct a lookup table of precomputed BRIEF patterns. As long as the keypoint orientation θ is consistent across views, the correct set of points S_θ will be used to compute its descriptor.

BRIEF has an important property that each bit feature has a large variance and a mean near 0.5. But once it is oriented along keypoint direction, it loses this property and become more distributed. High variance makes a feature more discriminative, since it responds differentially to inputs. Another desirable property is to have the tests uncorrelated, since then each test will contribute to the result. To resolve all these, ORB runs a greedy search among all possible binary tests to find the ones that have both high variance and means close to 0.5, as well as being uncorrelated. The result is called **rBRIEF**.

For descriptor matching, multi-probe LSH which improves on the traditional LSH, is used. The paper says ORB is much faster than SURF and SIFT and ORB descriptor works better than SURF. ORB is a good choice in low-power devices for panorama stitching etc.

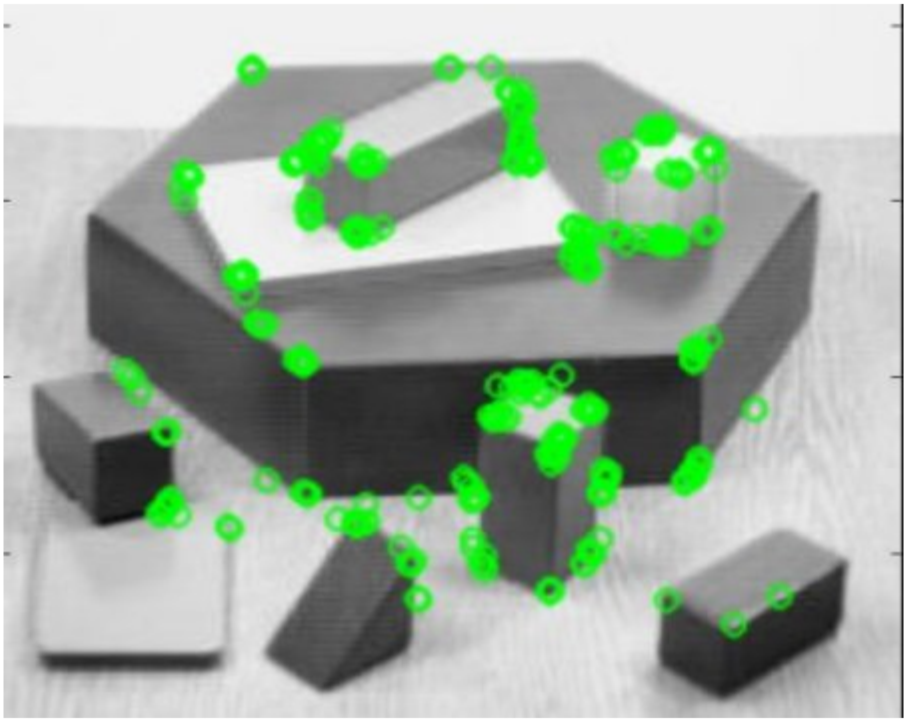
ORB in OpenCV

As usual, we have to create an ORB object with the function, cv.ORB() or using feature2d common interface. It has a number of optional parameters. Most useful ones are nFeatures which denotes maximum number of features to be retained (by default 500), scoreType which denotes whether Harris score or FAST score to rank the features (by default, Harris score) etc. Another parameter, WTA_K decides number of points that produce each element of the oriented BRIEF descriptor. By default it is two, ie selects two points at a time. In that case, for matching, NORM_HAMMING distance is used. If WTA_K is 3 or 4, which takes 3 or 4 points to produce BRIEF descriptor, then matching distance is defined by NORM_HAMMING2.

Below is a simple code which shows the use of ORB.

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('simple.jpg',0)
# Initiate ORB detector
orb = cv.ORB_create()
# find the keypoints with ORB
kp = orb.detect(img,None)
# compute the descriptors with ORB
kp, des = orb.compute(img, kp)
# draw only keypoints location,not size and orientation
img2 = cv.drawKeypoints(img, kp, None, color=(0,255,0), flags=0)
plt.imshow(img2), plt.show()
```

See the result below:



image