Amirkabir University of Technology
(Tehran Polytechnic)
Department of Computer Engineering

Computer Vision

Assignment IV

Soroush Mahdi
99131050

supervised by
Dr. Reza Safabakhsh

December 17, 2021

Contents

1 How SIFT works?

1.1 What is SIFT?

SIFT, or Scale Invariant Feature Transform, is a feature detection algorithm in Computer Vision.

SIFT helps locate the local features in an image, commonly known as the ‘keypoints‘ of the image. These keypoints are scale and rotation invariant that can be used for various computer vision applications, like image matching, object detection, scene detection, etc.

We can also use the keypoints generated using SIFT as features for the image during model training. The major advantage of SIFT features, over edge features or hog features, is that they are not affected by the size or orientation of the image.

The keypoints found by SIFT are those that are very prominent and will not change due to lighting, affine transformation, noise and other factors, such as corner points, edge points, bright spots in dark areas and dark spots in bright areas.

1.2 Solvable Problems

The performance of image registration / target recognition and tracking is affected by the state of the target itself, the environment of the scene and the imaging characteristics of the imaging equipment. To some extent, SIFT algorithm can solve:

- Rotation, scaling, translation (RST) of target
- Image affine / projection transformation (viewpoint)
- illumination

1.3 SIFT Algorithm

In general, SIFT algorithm can be decomposed into four steps:

1. Constructing a Scale Space and keypoint detection: To make sure that features are scale-independent and extracting keypoints
2. Keypoint Localization: Identifying the suitable features or keypoints
3. Orientation Assignment: Ensure the keypoints are rotation invariant
4. Keypoint Descriptor: Assign a unique fingerprint to each keypoint

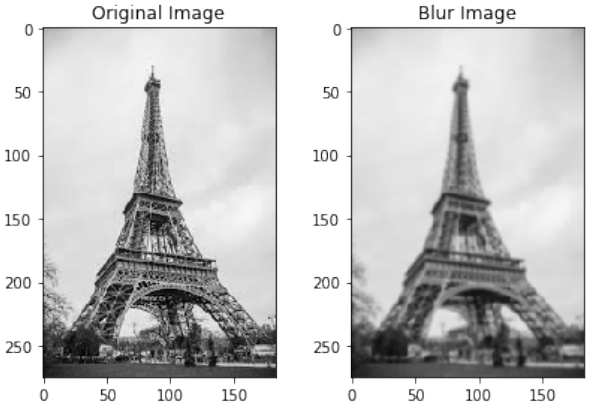
We will explain these steps in detail.

1.3.1 Constructing the Scale Space

We need to identify the most distinct features in a given image while ignoring any noise. Additionally, we need to ensure that the features are not scale-dependent. These are critical concepts, we will explain them one-by-one.

We use the Gaussian Blurring technique to reduce the noise in an image.

So, for every pixel in an image, the Gaussian Blur calculates a value based on its neighboring pixels. in the next page is an example of image before and after applying the Gaussian Blur. As you can see, the texture and minor details are removed from the image and only the relevant information like the shape and edges remain:

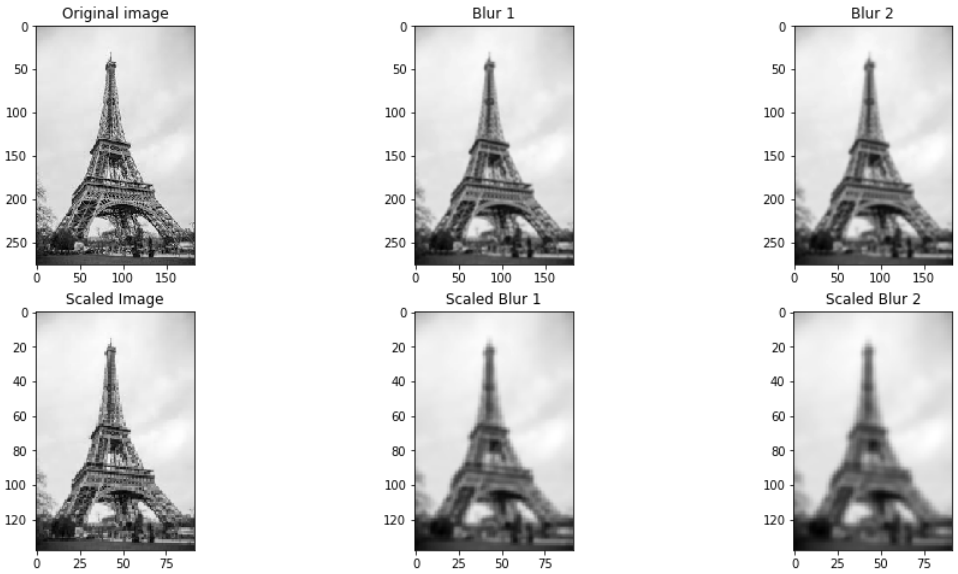


Gaussian Blur successfully removed the noise from the images and we have highlighted the important features of the image. Now, we need to ensure that these features must not be scale-dependent. This means we will be searching for these features on multiple scales, by creating a 'scale space'.

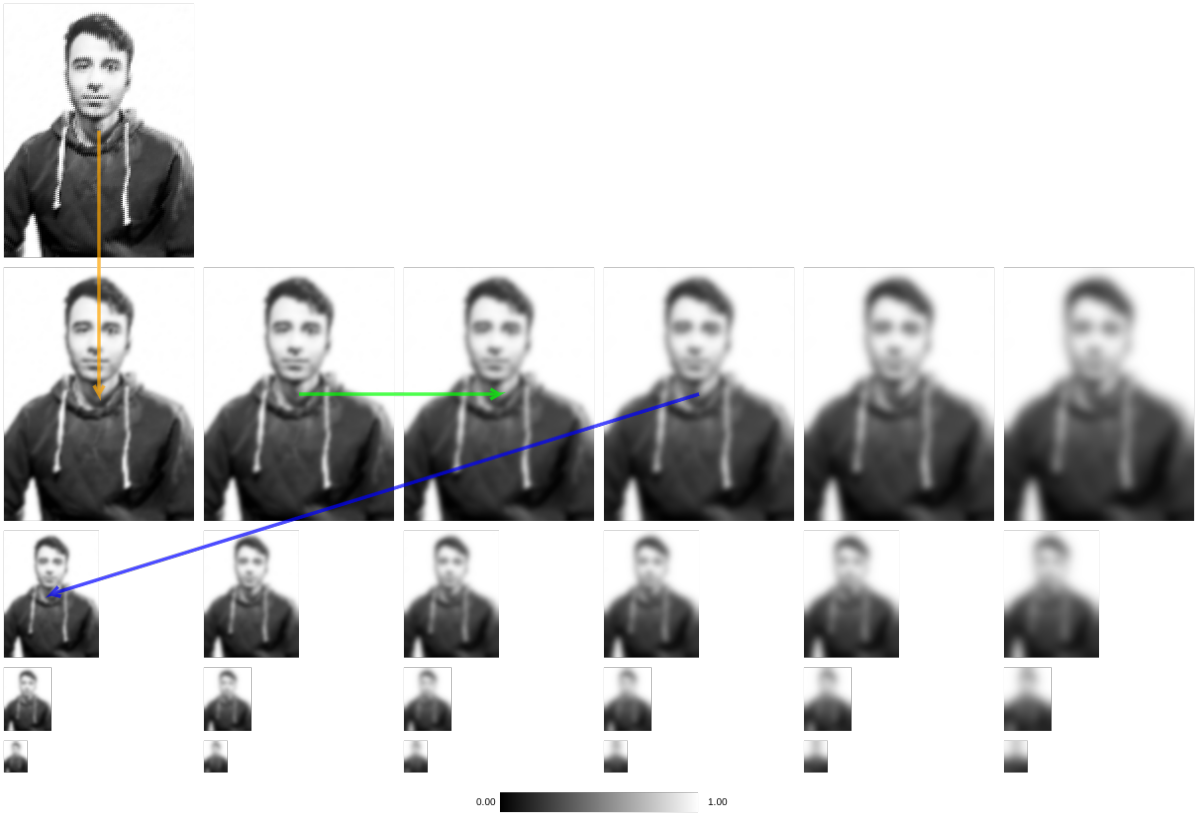
Scale space is a collection of images having different scales, generated from a single image.

Hence, these blur images are created for multiple scales. To create a new set of images of different scales, we will take the original image and reduce the scale by half. For each new image, we will create blur versions as we saw above.

Here is an example to understand it in a better manner. We have the original image of size (275, 183) and a scaled image of dimension (138, 92). For both the images, two blur images are created:



as suggested by the main paper The ideal number of octaves should be four, and for each octave, the number of blur images should be five.in the next page there is an example of creating scale space with my own photo!



The picture is subsequently blurred using a Gaussian convolution. That's indicated by the orange arrow.

What follows is a sequence of further convolutions with increasing standard deviation. Each picture further to the right is the result of convoluting its left neighbor, as indicated by the green arrow.

Finally, the antepenultimate picture of each row is downsampled - see the blue arrow. This starts another row of convolutions. Now we have constructed a scale space.

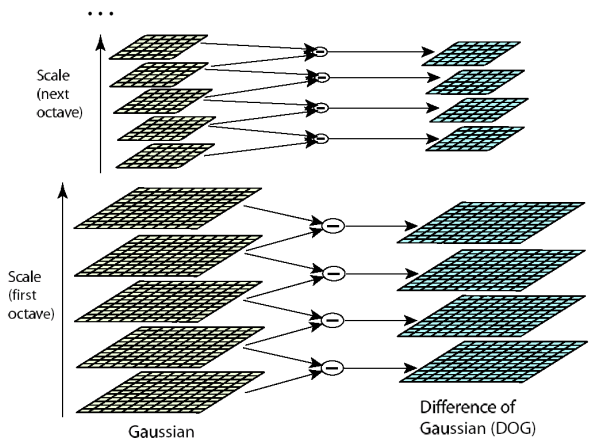
1.3.2 Difference of Gaussian

Now for the next step. Let's imagine for a moment that each octave of our scale space were a continuous space with three dimensions: the x and y coordinates of the pixels and the standard deviation of the convolution. In an ideal world, we would now want to compute the Laplacian of the scale-space function which assigns gray values to each element of this space. The extrema of the Laplacian would then be candidates for the key points our algorithm is looking for. But as we have to work in a discrete approximation of this continuous space, we'll instead use a technique called difference of Gaussians.
The response of a derivative of Gaussian filter to a perfect step edge decreases as σ increases. To keep response the same(scale-invariant), must multiply Gaussian derivative by σ. Laplacian is the second Gaussian derivative, so it must be multiplied by σ^2.

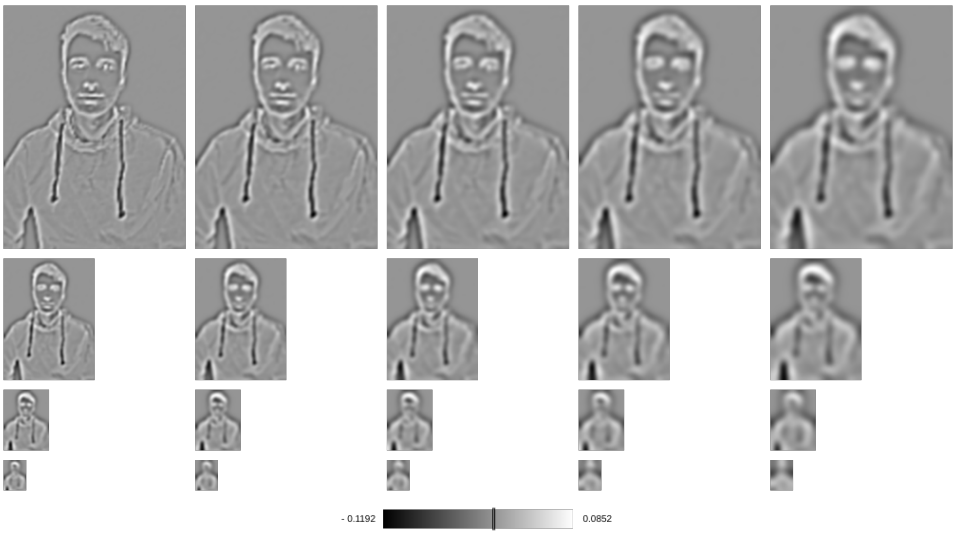
$$G(x, y, k\sigma) - G(x, y, \sigma) \approx (k-1)\sigma^2\nabla^2 G.$$

This shows that when the DoG function has scales differing by a constant factor it already incorporates the σ^2 scale normalization required for the scale-invariant Laplacian. The factor $k-1$ in the equation is a constant over all scales and therefore does not influence extrema location.

DoG creates another set of images, for each octave, by subtracting every image from the previous image in the same scale. in the next page is a visual explanation of how DoG is implemented:



Now in our example, For each pair of horizontally adjacent pictures we compute the differences of the individual pixels.



We have enhanced features for each of these images. Now that we have a new set of images, we are going to use this to find the important keypoints.

1.3.3 Keypoint Localization

Once the images have been created, the next step is to find the important keypoints from the image that can be used for feature matching. The idea is to find the local maxima and minima for the images.

The discrete extrema of these difference images will now be good approximations for the actual extrema we talked about above. A discrete maximum in our case is a pixel whose gray value is larger than those of all of its 26 neighbor pixels; and a discrete minimum is of course defined in an analogous way. Here we count as "neighbors" the eight adjacent pixels in the same picture, the corresponding two pixels in the adjacent pictures in the same octave, and finally their neighbors in the same picture.

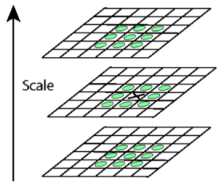
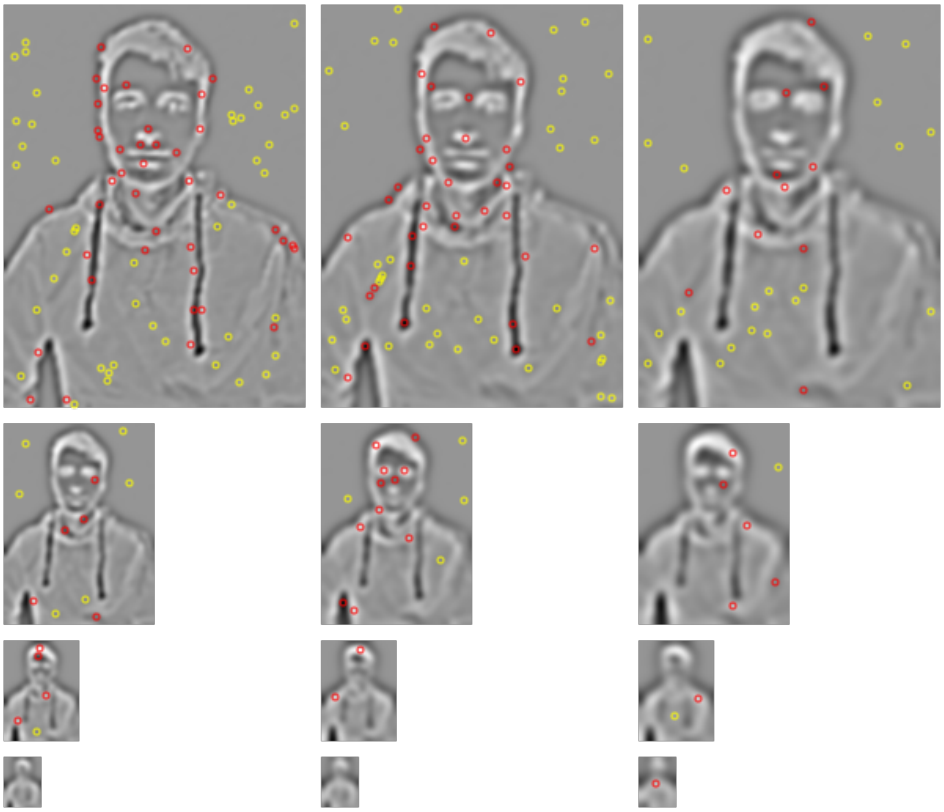


Figure 1: how to find exterma?

The extrema we've found are marked below. (Some are marked with yellow circles. These are indeed extrema, but their absolute values are so small that we'll discard them before proceeding. The algorithm assumes that it's likely these extrema exist only due to image noise.)



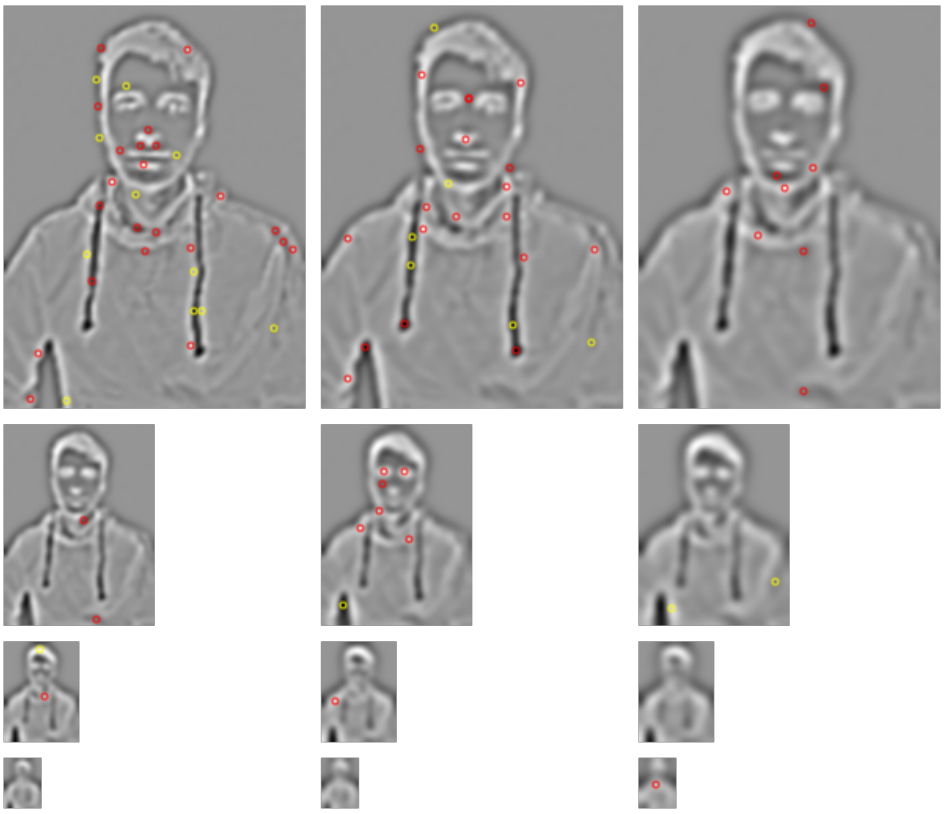
We now have potential keypoints that represent the images and are scale-invariant. We will apply the last check over the selected keypoints to ensure that these are the most accurate keypoints to represent the image.

The extrema we've found so far will of course have discrete coordinates. We now try to refine these coordinates. This is done (for each extremum) by approximating the quadratic Taylor expansion of the scale-space function and computing its extrema. This is an iterative process and either we are able to refine the location or we give up after a couple of steps and discard the point.

Now that we have better ("continuous") coordinates, we also do a bit more. We try to identify (and discard) key point candidates which lie on edges. These aren't good key points as they are invariant to translations parallel to the edge direction. Edge extrema are found by comparing the principal curvatures of the scale-space function (or rather its projection onto the picture plane) at the corresponding locations.

(This is done with the help of the trace and the determinant of the Hessian.

The remaining key points are shown below. (As we now have better estimates regarding their position, we can also discard some more low-contrast points. These are again marked with yellow color.)



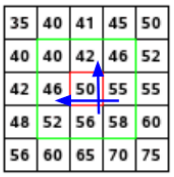
You might have the impression that there are some "new" points which weren't among the extrema further above. But these will be points which moved from one scale picture to another one. (For example, if a point was originally in the middle, i.e. if its scale value had been 2, the refined value could now be 2.57. That would mean it'd now appear on the right as the nearest integer would now be 3.)

1.3.4 Orientation Assignment

At this stage, we have a set of stable keypoints for the images. We will now assign an orientation to each of these keypoints so that they are invariant to rotation. We can again divide this step into two smaller steps:

1. Calculate the magnitude and orientation
2. Create a histogram for magnitude and orientation

Consider the sample image shown below:



Let's say we want to find the magnitude and orientation for the pixel value in red. For this, we will calculate the gradients in x and y directions by taking the difference between 55 46 and 56 42. This comes out to be Gx = 9 and Gy = 14 respectively.
Once we have the gradients, we can find the magnitude and orientation. The magnitude represents the intensity of the pixel and the orientation gives the direction for the same.

For each pixel in a square-shaped patch around the key point, we approximate the gradient using finite differences. Recall that the gradient points in the direction of the greatest increase and its magnitude is the slope in that direction. The intervall from 0 to 360 degrees is divided into a fixed number of bins (36 by default) and the value of the bin the gradient's direction belongs to is incremented by the gradient's magnitude after it has been multiplied with a Gaussian weight. The latter is done to reduce the contribution of more distant pixels.

This histogram would peak at some point. The bin at which we see the peak will be the orientation for the keypoint. Additionally, if there is another significant peak, then another keypoint is generated with the magnitude and scale the same as the keypoint used to generate the histogram. And the angle or orientation will be equal to the new bin that has the peak. Effectively at this point, we can say that there can be a small increase in the number of keypoints.

Key points near the image border which don't have enough neighboring pixels to compute a reference orientation are discarded. Key points without a dominating orientation are also discarded. On the other hand, key points with more than one dominating orientation might appear more than once in the next steps, namely once per orientation.

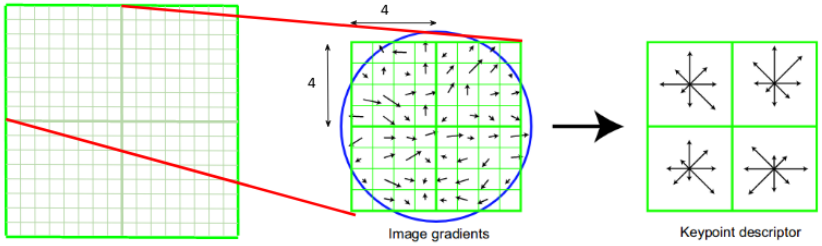
1.3.5 Keypoint Descriptor

This is the final step for SIFT. So far, we have stable keypoints that are scale-invariant and rotation invariant. In this section, we will use the neighboring pixels, their orientations, and magnitude, to generate a unique fingerprint for this keypoint called a 'descriptor'.

Additionally, since we use the surrounding pixels, the descriptors will be partially invariant to illumination or brightness of the images.

This step is pretty similar to the one above. We will again compute a histogram for the distribution of the directions of the gradients in a neighborhood of each key point. The difference is that this time the neighborhood is a circle and the coordinate system is rotated to match the reference orientation. Also,

the full truth is that we not only compute one, but rather sixteen histograms(with bins equal to 8). Each histogram corresponds to a point near the center of the new coordinate system and the contribution of each gradient from within the circle-shaped neighborhood is distributed over these histograms according to proximity.



So, what do we have now? We have a potentially large set of descriptors. Practical experience has shown that these descriptors can often be used to identify objects in images even if they are depicted with different illumination, from a different perspective, or in a different size compared to a reference image. Why does this work? Here are some reasons:

- Key points are extracted at different scales and blur levels and all subsequent computations are performed within the scale space framework. This will make the descriptors invariant to image scaling and small changes in perspective.
- Computation relative to a reference orientation is supposed to make the descriptors robust against rotation.
- Likewise, the descriptor information is stored relative to the key point position and thus invariant against translations.
- Many potential key points are discarded if they are deemed unstable or hard to locate precisely. The remaining key points should thus be relatively immune to image noise.
- The histograms are normalized at the end which means the descriptors will not store the magnitudes of the gradients, only their relations to each other. This should make the descriptors invariant against global, uniform illumination changes.
- The histogram values are also thresholded to reduce the influence of large gradients. This will make the information partly immune to local, non-uniform changes in illumination.

2 Tracking Soccer Ball

First, we have to do some real-time preprocessing on frames. for example, we can use thresholding based on color to separate grass(since it's always green and we can find color ranges with trial and error) and turn it to black so our algorithm will do better in detecting the ball.

then we should take a photo from the ball that we want to track and extract features of this photo with SIFT algorithm. we also can take several photos from the ball(but not many because it will increase computation time) in different Conditions (Although SIFT is invariant to several properties but it's worth trying). and we also have to extract features using SIFT from all these photos that are training photos.

now for each frame, we have to extract SIFT features and match these features with features that we have extracted from training photos. we can use different matching algorithms for example we can calculate distances between each pair of features.

now we have to specify a threshold for distances in the matching process and we keep only good matches. then if there was a certain number of good matches (this number can be selected with trial and error) in a certain neighborhood (this neighborhood depends on the angle and distance of the camera from the playground and we know size of the ball) between this frame and any of the training photos we will consider those keypoints or that specific neighborhood as the ball.

3 Object tracking

for all tasks in this section first, I took an image as a training image for the detector. then detector finds key points and descriptor calculates descriptors for these key points and a matcher tries to match descriptors in the current frame and training image.

3.1 SIFT

in this section i used SIFT as detector and also descriptor. here is the code for initializing parameters.

```
#initialize parameters
#minimum match points to accept detection
MIN_MATCH_COUNT=15

#creating SIFT object
detector=cv2.xfeatures2d.SIFT_create()

#initializing matcher, we will use histogram based metric and
# 2 nearest neighbor method
FLANN_INDEX_KDITREE=0
flannParam=dict(algorithm=FLANN_INDEX_KDITREE,tree=5)
flann=cv2.FlannBasedMatcher(flannParam,{})

# a list for keeping last 3 frames
images = []
```

Listing 1: initializing parameters

i used a minimum value for number of matches for deciding if we have a detection in current frame at all or not.

Classical feature descriptors (SIFT, SURF, ...) are usually compared and matched using the Euclidean distance (or L2-norm). Since SIFT and SURF descriptors represent the histogram of oriented gradient (of the Haar wavelet response for SURF) in a neighborhood, alternatives of the Euclidean distance are histogram-based metrics so i usedFlannBasedMatcher for matching.

then i calculated key points and descriptors using SIFT for train image.

```
#reading train img
trainImg=cv2.imread("train.jpg",0)

#computing sift keypoints and descriptors
trainKP,trainDesc=detector.detectAndCompute(trainImg,None)
```

Listing 2: computing sift keypoints and descriptors of train img

then i will use a function that i wrote named center.
this function calculate sift key points and descriptors for an image then matches these features with key points of train image and detect the object and return center and border of object

```
def center(QueryImg  , detector , flann , trainDesc ):
    """this function calculate sift keypoints and discriptors
        for an image then matches these features with keypoints
        of train img and detect the object and return center and border
        of object

    Args:
        QueryImg ([2d np array]): [input img]
        detector ([CV2 detector]): [detctor that we want to use  (here we use SIFT)]
        flann ([cv2.FlannBasedMatcher]): [matcher for mathcing features]
        trainDesc ([cv2 descriptor]): [descriptors of train img]
```

```
12
13      Returns:
14          return 4 vars
15          first one tell us that we have detected object in img
16          seconde one is avg point of mathced keypoints
17          third one is border of detcted object
18          fourth on is avg point of vertixes of border
19      """
```

Listing 3: center function information

we can see center function code here:

```
1  def center(QueryImg  , detector , flann , trainDesc ):
2      #computing keypoints and descriptors for input img
3      queryKP,queryDesc=detector.detectAndCompute(QueryImg,None)
4
5      #matching calculated descriptors and train img descriptors
6      matches=flann.knnMatch(queryDesc,trainDesc,k=2)
7
8      # a list for keeping only good matches
9      goodMatch=[]
10
11     # The distance ratio between the two nearest matches of a considered
12     # keypoint is computed and it is a good match when this value is below
13     # a threshold
14     for m,n in matches:
15         if(m.distance<0.75*n.distance):
16             goodMatch.append(m)
17
18     #if we have matched enough keypoints for detection
19     if(len(goodMatch)>MIN_MATCH_COUNT):
20
21         tp=[] # keeping mathced points in train image
22         qp=[] # keeping mathced points in input image
23
24         goodMatch = sorted(goodMatch , key= lambda x: x.distance)
25
26         for m in goodMatch[:int(len(goodMatch)*.8)]:
27             tp.append(trainKP[m.trainIdx].pt)
28             qp.append(queryKP[m.queryIdx].pt)
29
30         tp,qp=np.float32((tp,qp))
31
32         #finding borders of detected object
33         H,status=cv2.findHomography(tp,qp,cv2.RANSAC,3.0)
34         h,w=trainImg.shape
35         trainBorder=np.float32([[[0,0],[0,h-1],[w-1,h-1],[w-1,0]]])
36         queryBorder=cv2.perspectiveTransform(trainBorder,H)
37
38         qp=np.float64(qp)
39         #avg of matched keypoints
40         KP_avg = np.int64(np.sum(qp , axis=0) / qp.shape[0])
41         #avg of vertixes of detected border
42         border_avg = np.int64(np.sum(queryBorder[0] , axis=0) / queryBorder[0].shape
       [0])
43         print("Match Found")
44
45         return True , KP_avg, queryBorder ,border_avg
46
47     else:
48         print("Not Enough match found-")
49         print(len(goodMatch),MIN_MATCH_COUNT)
50         return False , False
```

Listing 4: center function code

as we can see first i found key points and descriptors for input image, then i used the matcher to match descriptors of input and training image.

then The distance ratio between the two the nearest matches of a considered key point is computed, and it is a good match when this value is below a threshold. Indeed, this ratio allows helping to discriminate between ambiguous matches (distance ratio between the two nearest neighbors is close to one) and well discriminated matches.

next, I checked if we have enough matches for detecting target object. then I saved 80 percent of the best matches based on their distances from their corresponding matches.

then i used two methods for calculating a point to follow for tracking target. first, I took an average of the points corresponding to the matches that I have saved and in the second method i calculated border of the target object and then took an average on vertices of this border.

now in a loop i will read each frame from webcam and save last 3 frames and perform center function on them.

```
while True:
    #reading frames
    ret, QueryImgBGR=cam.read()
    QueryImgBGR_copy = QueryImgBGR.copy()
    QueryImg=cv2.cvtColor(QueryImgBGR,cv2.COLOR_BGR2GRAY) #turning frame to gray

    if ret:

        images.append(QueryImg)

    if len(images) > 2: #skiping till we have seen more than 2 frames

        #center function will detect the object in passed frame
        #and retrun center and border of detected object
        #we will perform this function in last 3 frames

        C0 = center(images[2]  , detector , flann , trainDesc)
        C1 = center(images[1]  , detector , flann , trainDesc)
        C2 = center(images[0]  , detector , flann , trainDesc)


        if C0[0] and C1[0]: #if we have detected object in last 2 frames based on num
     of matches
            #drawing an arrow(green) from avg of keypoints of prevoius frame to
            #avg of keypoints of current frame
            cv2.arrowedLine(QueryImgBGR, C1[1], C0[1], (0,255,0), 2)
            cv2.arrowedLine(QueryImgBGR_copy, C1[3], C0[3], (0,255,0), 2)

        if C2[0] and C1[0]:
            #drawing an arrow(purple) from avg of keypoints of -2 frame to
            #avg of prevoius of current frame
            cv2.arrowedLine(QueryImgBGR, C2[1], C1[1], (255,0,255), 2)
            cv2.arrowedLine(QueryImgBGR_copy, C2[3], C1[3], (255,0,255), 2)
            #drawing border for detected object
            cv2.polylines(QueryImgBGR,[np.int32(C2[2])],True,(0,0,255),3)
            cv2.polylines(QueryImgBGR_copy,[np.int32(C2[2])],True,(0,0,255),3)

        images.pop(0)
```

Listing 5: main loop

as we in last 3 frames, for each frame i perform center function and if we have detections i draw 2 arrows. and also i draw the border.

results of this section are 2 videos named sift poly and sift avg. in the first video i drew arrows based on average of vertices of object border and in the second i drew arrows based on average of matches key points.

3.2 SIFT + FREAK

the process is like last section. in this section i used SIFT as detector and freak as descriptor. here is the code for initializing parameters.

```
#########################################################################
#initialize parameters
#minimum match points to accept detection
MIN_MATCH_COUNT=80

#creating SIFT object
detector=cv2.cv2.xfeatures2d.SIFT_create()

#creating freak object
freak = cv2.xfeatures2d.FREAK_create()

#initializing matcher, hence freak is a binary descriptor
#we will use hamming dist
bf = cv2.BFMatcher(cv2.NORM_HAMMING2, crossCheck=True)

# a list for keeping last 3 frames
images = []
```

Listing 6: initializing parameters

here i used a brute force matcher because freak is a binary descriptor and It's fast to calculate distances, i also used hamming distance based on same reason.

then i calculated key points using SIFT and descriptors using freak for train image.

```
#reading train img
trainImg=cv2.imread("train.jpg",0)

#computing sift keypoints
kp = detector.detect(trainImg,None)

#computing freak keypoints and descriptors
trainKP,trainDesc= freak.compute(trainImg,kp)
```

Listing 7: computing sift keypoints and freak descriptors of train img

the rest of the code is like sift section except that i used brute force matcher for matching in center function.

results of this section are 2 videos named freak poly and freak avg. in the first video i drew arrows based on average of vertices of object border and in the second i drew arrows based on average of matches key points.based on results we can see that freak is not so invariant to the scale.

3.3 SIFT + Kalman

in this section i used the process explained in SIFT section for detecting the center of the target object. for this i used average of vertices of object border that i detected. then for each frame i passed the center point of the object in the current frame as the position of the object to a kalman filter for predicting next state of the object. in fact we use kalman filter here for predicting next position of object based on prevoius positions. here is the code for kalman filter calss:

```
#kalman class that we will use for kalman filter
class KalmanFilter:
    #initializing kalman filter and its parameters
    kf = cv2.KalmanFilter(4, 2)
    kf.measurementMatrix = np.array([[1, 0, 0, 0], [0, 1, 0, 0]], np.float32)
    kf.transitionMatrix = np.array([[1, 0, 1, 0], [0, 1, 0, 1], [0, 0, 1, 0], [0, 0,
    0, 1]], np.float32)
```

```
7
8
9       def predict(self, coordX, coordY):
10          ''' This function estimates the position of the object'''
11          measured = np.array([[np.float32(coordX)], [np.float32(coordY)]])
12          self.kf.correct(measured)
13          predicted = self.kf.predict()
14          x, y = int(predicted[0]), int(predicted[1])
15          return x, y
```

Listing 8: kalman filter

and here is center function for this section:

```
1  def center(QueryImg  , detector , flann , trainDesc , predict = None):
2      """this function calculate sift keypoints and discriptors
3          for an image then matches these features with keypoints
4          of train img and detect the object and return center and border
5          of object,  it also predicts the next position of object
6          based on the current positon and a kalman filter, i should note that
7          kalman filter uses all previous points to predict
8
9      Args:
10         QueryImg ([2d np array]): [input img]
11         detector ([CV2 detector]): [detctor that we want to use  (here we use SIFT)]
12         flann ([cv2.FlannBasedMatcher]): [matcher for mathcing features]
13         trainDesc ([cv2 descriptor]): [descriptors of train img]
14         predict ([none or a 2*1 np array]): position of last predicted point by
       kalman
15
16     Returns:
17         return 4 vars
18         first one tell us that we have detected object in img
19         seconde one is avg point of mathced keypoints
20         third one is predicted position
21         fourth one is the border of detcted object
22     """
23
24     #computing keypoints and descriptors for input img
25     queryKP,queryDesc=detector.detectAndCompute(QueryImg,None)
26
27     #matching calculated descriptors and train img descriptors
28     matches=flann.knnMatch(queryDesc,trainDesc,k=2)
29
30     # a list for keeping only good matches
31     goodMatch=[]
32
33     # The distance ratio between the two nearest matches of a considered
34     # keypoint is computed and it is a good match when this value is below
35     # a threshold
36     for m,n in matches:
37         if(m.distance<0.75*n.distance):
38             goodMatch.append(m)
39
40     #if we have matched enough keypoints for detection
41     if(len(goodMatch)>MIN_MATCH_COUNT):
42
43         goodMatch = sorted(goodMatch , key= lambda x: x.distance)
44
45         tp=[] # keeping mathced points in train image
46         qp=[] # keeping mathced points in input image
47
48         for m in goodMatch[:int(len(goodMatch)*.8)]:
49             tp.append(trainKP[m.trainIdx].pt)
50             qp.append(queryKP[m.queryIdx].pt)
51
52         tp,qp=np.float32((tp,qp))
53
54         #finding borders of detected object
55         H,status=cv2.findHomography(tp,qp,cv2.RANSAC,3.0)
56         h,w=trainImg.shape
```

```
57          trainBorder=np.float32([[[0,0],[0,h-1],[w-1,h-1],[w-1,0]]])
58          queryBorder=cv2.perspectiveTransform(trainBorder,H)
59
60          qp = queryBorder[0]
61
62          #avg of vertixes of detected border
63          detected_center = np.int64(np.sum(queryBorder[0] , axis=0) / queryBorder[0].
        shape[0])
64          print("Match Found")
65
66          #predicting next position with kalman based on current position
67          predicted = kf.predict(detected_center[0] , detected_center[1])
68
69          return True ,detected_center ,predicted , queryBorder
70
71      else:
72
73          print("Not Enough match found-")
74          print(len(goodMatch),MIN_MATCH_COUNT)
75
76          if predict:
77
78              #predicting next position with kalman based on prevoius prediction
79              predict = kf.predict(predict[0] , predict[1])
80          return False , predict
```

Listing 9: center function

the main point in this function is if we have detected object in the input frame we will use center of the object for predicting next state of the object using kalman filter and if not we will use previous prediction for predicting next state.

i have shown kalman predictions with a red circle.

result of this section is a video named KalmanFilter.