COMPSCI 670: Computer Vision Fall 22

Homework 3

Prachi Jain

1 Short answers

1.1

Lucas-Kanade Optical Flow
To solve the aperture problem, we can pretend that the pixel's neighbors have the same values (u, v), we can create a system of equations as shown below:

$$\begin{bmatrix} I_x(p1) & I_y(p1) \\ I_x(p2) & I_y(p2) \\ \vdots & \vdots \\ I_x(pn) & I_y(pn) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} I_t(p1) \\ I_t(p2) \\ \vdots \\ I_t(pn) \end{bmatrix}$$

We can represent the above equation in terms: $Ad = b$ and can solve d by

$$(A^T A)d = A^T b$$

Hence:

$$\begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_y I_x & \sum I_y I_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix}$$

This is solvable when, $A^T A$ is invertible, is not too small die to noise(its eigenvalues λ_1 and λ_2 should not be too small) and $\lambda_1 \lambda_2$ should not be too large. It is called second moment matrix.

$$\begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_y I_x & \sum I_y I_y \end{bmatrix} = C = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

Here, C(1, 1) is sum over a small region, the hypothetical corner. C(1, 2) is the gradient with respect to x, times gradient with respect to y.

1. If either λ_1 or λ_2 is close to 0, then it is not a corner but a flat region.
2. If λ_1 or λ_2 are big, corners are there and this is where LUCAS-KANADE OPTICAL WORKS.
3. Corners are regions with two different directions of gradient (at least).
4. Aperture problem disappears at corners. Because at corners, 1st order approximation fails. Thus Systems will be "well-conditioned" at corners as they are "solvable" there.

1.2

Motion Field is the projection of the 3D relative vectors onto the 2D image plane.
Motion Field is what we want to determine and optical flow is what we estimate
For the Motion Field, we will derive an equation relating to 3D scene velocity induced by camera motion and its 2D motion flow field
Let's say at time t, a point's 3D position is at P.
At time t + 1, a point's 3D position will be at $R * P + T$, where R is relative rotation of that point compared to the same point at time t and T is the translation or displacement. Thus, the net 3D displacement will be

$$RP + T - P$$

We can make some general assumptions about the velocity and flow of a 3D point by saying that the displacement from time t to time t + 1 will be infinitesimal and thus, the rotation angles will be small.
Euler Angle Transformation says

$$R = \begin{bmatrix} 1 & -\psi & \theta \\ \psi & 1 & -\phi \\ -\theta & \phi & 1 \end{bmatrix} = I + \begin{bmatrix} 0 & -\psi & \theta \\ \psi & 0 & -\phi \\ -\theta & \phi & 0 \end{bmatrix} = I + S$$

where ψ is rotation about $Z - axis = \theta_z$; θ is rotation about $Y - axis = \theta_y$ and ϕ is rotation about $X - axis = \theta_x$.

Therefore displacement is equal to

$$(I + S)P + T - P = SP + T \text{ and } SP = [\theta_x, \theta_y, \theta_z]^T P$$

In limit, displacement becomes velocity:

$$V = T + \omega P$$

$$\text{For } T = [T_x, T_y, T_z]; \omega = [\omega_x, \omega_y, \omega_z]; P = [X, Y, Z]$$

$$V_x = -T_x + -\omega_y Z + \omega_z Y$$

$$V_y = -T_y + -\omega_z X + \omega_x Z$$

$$V_z = -T_z + -\omega_x Y + \omega_y X$$

The above equations are for relative velocity of a 3D point P with respect to camera.

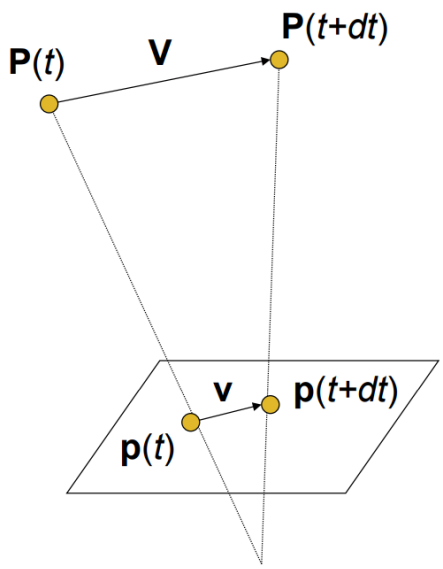


Figure 1: A 2D representation of p on an image plane of a 3D point P, while P is under motion in world space

The equations for relative velocity of a 3D point P can be thought as vectors of small magnitude of the scene depicted in above figure.
In perspective projection,

$$p = \frac{fP}{Z}$$

Taking derivative with respect to time,

$$\frac{dp}{dt} = v = \frac{d\frac{fP}{Z}}{dt}$$

$$\frac{dp}{dt} = v = \frac{f}{Z^2}\left[\frac{dP}{dt}Z - P\frac{dZ}{dt}\right]$$

$$\frac{dp}{dt} = v = \frac{f}{Z^2}\left[V \cdot Z - P \cdot V_z\right]$$

$$p = \frac{fP}{Z} \& P = \frac{pZ}{f}$$

$$\boxed{v = f\frac{V}{Z} - p\frac{V_z}{Z}}$$

v is for 2D projection so $v_z = 0$
Substituting values of V_x, V_y, V_z and v we get:

$$\boxed{v_x = \frac{T_z x - T_x f}{Z} - \omega_y f + \omega_z y + \frac{\omega_x xy}{f} - \frac{\omega_y x^2}{f}}$$

$$\boxed{v_y = \frac{T_z y - T_y f}{Z} + \omega_x f - \omega_z x - \frac{\omega_y xy}{f} + \frac{\omega_x y^2}{f}}$$

These are the final equations to convert a 3D motion in a world space to 2D points in a image plane with relative depth.
In both the above equations, the first terms are translational components and others are rotational components. It is apparent that for rotational components - "angle" is independent of the depth Z of a 3D point. But, the whole vector length depends on depth Z as the translation part has the depth component in it.
Hence it is shown that the direction of optical flow of a point in space is independent of its depth, although the magnitude of optical flow depends on depth.

1.3

In 2004, D.Lowe, University of British Columbia, came up with a new algorithm SIFT, in his paper, Distinctive Image Features from Scale-Invariant Keypoints, which extracts key points and computes its descriptors.
As the name suggests, SIFT has the property of scale invariance. The scale of an image landmark is its (rough) diameter in the image. It is denoted by sigma, which is measured in pixels. Scale invariance can detect similar landmarks even if their scale is different, thus making SIFT better than Harris. (Harris is not scale-invariant, a corner may become an edge if the scale changes.)

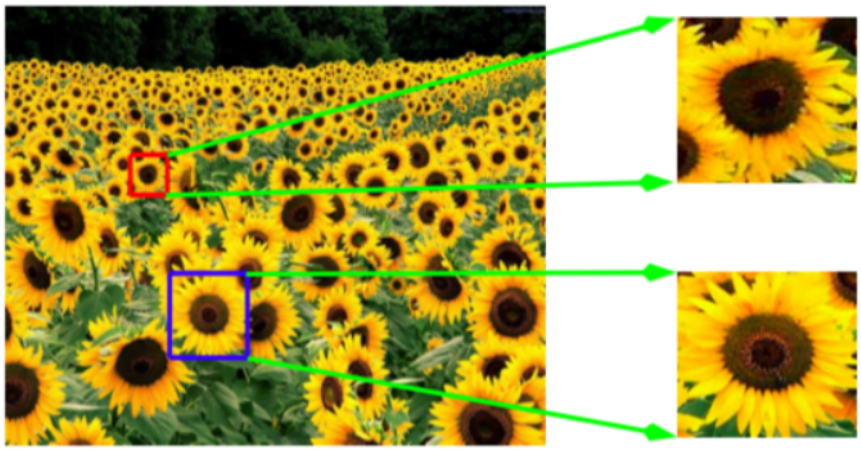


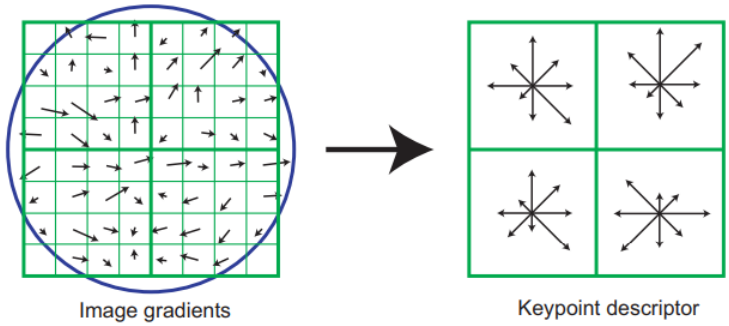
Figure 2: Scale invariance Illustration

The following are the significant stages of computation used to generate the set of image features:
1. Scale-space extrema detection: The first stage of computation searches over all scales

and image locations. It is implemented efficiently using a difference-of-Gaussian function to identify potential interest points invariant to scale and orientation.
2. Keypoint localization: At each candidate location, a detailed model is fit to determine location and scale. Keypoints are selected based on measures of their stability.
3. Orientation assignment: One or more orientations are assigned to each keypoint location based on local image gradient directions. All future operations are performed on image data that has been transformed relative to each feature's given orientation, scale, and location, thereby providing invariance to these transformations.
4. Keypoint descriptor: The local image gradients are measured at the selected scale in the region around each keypoint. These are transformed into a representation that allows for significant levels of local shape distortion and change in illumination.
This approach has been named the Scale Invariant Feature Transform (SIFT), as it transforms image data into scale-invariant coordinates relative to local features.

1.3.1 a.) SIFT descriptor achieves invariance to small deformations:

1. When the SIFT algorithm creates the keypoint descriptor, a 16x16 neighborhood around the keypoint is taken. It is divided into 16 sub-blocks of 4x4 size. For each sub-block, eight bin orientation histogram is created. So a total of 128 bin values are available.



A keypoint descriptor is created by first computing the gradient magnitude and orientation at each image sample point in a region around the keypoint location, as shown on the left. These are weighted by a Gaussian window, indicated by the overlaid circle. These samples are then accumulated into orientation histograms summarizing the contents over 4x4 subregions, as shown on the right, with the length of each arrow corresponding to the sum of the gradient magnitudes near that direction within the region. This figure shows a 2x2 descriptor array computed from an 8x8 set of samples.

Figure 3: Keypoint Descriptor

2. A Gaussian weighting function with sigma equal to one-half the width of the descriptor window is used to assign a weight to the magnitude of each sample point. This is illustrated with a circular window, although, of course, the weight falls off smoothly. The purpose of this Gaussian window is to avoid sudden changes(deformations) in the descriptor with small changes in the position of the window and to give less emphasis to gradients far from the center of the descriptor, as these are most affected by misregistration errors (deformations).
3. The keypoint descriptor allows a significant shift in gradient positions by creating orientation histograms over 4x4 sample regions. A gradient sample can shift up to 4 sample positions while still contributing to the same histogram, thereby achieving the objective of allowing for larger local positional shifts. (large deformations).
4. It is essential to avoid all boundary affects in which the descriptor abruptly changes as a

sample shifts smoothly from being within one histogram to another or from one orientation to another. Therefore, trilinear interpolation is used to distribute the value of each gradient sample into adjacent histogram bins.
5. Finally, the feature vector is modified to reduce the effects of illumination changes which also create deformations. First, the vector is normalized to unit length. A change in image contrast in which each pixel value is multiplied by a constant will multiply gradients by the same constant so that this contrast change will be canceled by vector normalization. A brightness change in which a constant is added to each image pixel will not affect the gradient values, as they are computed from pixel differences. Therefore, the descriptor is invariant to affine changes in illumination.
6. However, non-linear illumination changes can also occur due to camera saturation or illumination changes that affect 3D surfaces with differing orientations by different amounts. These effects can cause a large change in relative magnitudes for some gradients but are less likely to affect the gradient orientations. Therefore, reducing the influence of large gradient magnitudes by thresholding the unit feature vector values to be no larger than 0.2 and then renormalizing to unit length is implemented. This means that matching the magnitudes for large gradients is no longer as important and that the distribution of orientations has a greater emphasis.

1.3.2 (b) SIFT descriptor achieves invariance to orientation:

1After Scale-space Extrema Detection and Keypoint Localization, SIFT Algorithm assigns consistent orientation to each keypoint based on local image gradient directions. Thus, SIFT can be represented relative to this orientation and therefore achieve invariance to image rotation in the following way:
1. The scale of the keypoint is used to select the Gaussian smoothed image, L, with the closest scale so that all computations are performed in a scale-invariant manner.
2. For each image sample, L(x, y), at this scale, the gradient magnitude, $m(x, y)$, and orientation, $\theta(x, y)$, is precomputed using pixel differences:

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

$$\theta(x,y) = \tan^{-1}((L(x,y+1) - L(x,y-1))/(L(x+1,y) - L(x-1,y)))$$

3. Depending on the scale, a neighborhood is taken around the keypoint location, and the gradient magnitude and direction are calculated in that region.
4. An orientation histogram with 36 bins covering 360 degrees range of orientations is created (Each sample added to the histogram is weighted by gradient magnitude and gaussian-weighted circular window with equal to 1.5 times the scale of the key point).
5. Peaks in the orientation histogram correspond to the dominant directions of local gradients. The highest peak in the histogram is detected, and then any other local peak within 80% of the highest peak is also used to create a key point with that orientation.
6. Therefore, for locations with multiple peaks of similar magnitude, multiple key points will be created at the same location and scale but at different orientations. Only about 15% of points are assigned multiple orientations, which contributes significantly to matching stability.
7. Finally, a parabola is fit to the three histogram values closest to each peak to interpolate the peak position for better accuracy.

1.4

Implementation:

```
# -*- coding: utf-8 -*-
"""Copy of Copy of 1_4.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1WxDXGPv8r0DWzRoZ3G0pCWRnG1xMQU4X
"""

import numpy as np
import matplotlib.pyplot as plt
import os
import cv2
from scipy import ndimage

image1 = np.ones((300,300,3))
image2 = np.ones((300,300,3))

center_cordinates = (150,150)
radius = 100
color = (0, 0, 250)
thickness = -1
image1 = cv2.circle(image1, center_cordinates, radius, color, thickness)
image2 = cv2.circle(image2, center_cordinates, radius, color, thickness)

plt.imshow(image1)
plt.title('Circle with radius: 100 and center: (150,150)')
plt.show()

plt.imshow(image2)
plt.title('Circle with radius: 100 and center: (150,150)')
plt.show()

#scale-normalized Laplacian
num_figures = 200
fig, axes = plt.subplots(20,10, figsize=(80,160))
nlsigma = []
sigma = 100/np.sqrt(2)
sigmas = []
for i in range(20):
    for j in range(10):
        response = np.power(sigma, 2)*ndimage.gaussian_laplace(image1, sigma)
        nlsigma.append(max(response[150,150,0],response[150,150,1],response
    [150,150,2]))
        axes[i][j].imshow(response)
        axes[i][j].set_title("Normalised Laplacian Sigma = {}".format(round(
    sigma,2)))
        sigmas.append(sigma)
        sigma = sigma/np.sqrt(2)

#Gaussian
num_figures = 200
fig, axes = plt.subplots(20,10, figsize=(80,160))
gsigma =[]
sigma = 100/np.sqrt(2)
```

```
sigmas = []
for i in range(20):
    for j in range(10):
        response = ndimage.gaussian_filter(image2, sigma)
        gsigma.append(max(response[150,150,0],response[150,150,1],response
    [150,150,2]))
        axes[i][j].imshow(response)
        axes[i][j].set_title("Gaussian: Sigma = {}".format(round(sigma,2)))
        sigmas.append(sigma)
        sigma = sigma/np.sqrt(2)

plt.plot(sigmas, nlsigma, label = "Scale-Normalized Laplacian Response")
plt.plot(sigmas, gsigma, label = "Gaussian Response")
plt.xlabel('sigma')
plt.ylabel('Responses')
plt.title('Responses as a function of    at circle center')
plt.grid(which='major')
plt.grid(which='minor')
plt.minorticks_on()
plt.legend()
plt.show()
```

From the below Responses of filters plot, it is observed Gaussian filter has a continuous decreasing sigma trend with a small peak in the begining. No the Gaussian filter is not scale selective. There is no Gaussian response which has a maximum (or minimum) at a scale that is characteristic of the scale feature. scale. The scale normalized Laplacian response has a continuous decreasing sigma trend as well with a much higher(3X) peak in the begining. We can observe circles centered at each strong local maxima in scale normalized Laplacian response which makes it scale selective.

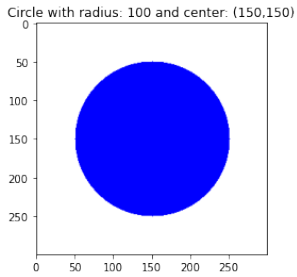


Figure 4: Picture with the radius of the circle

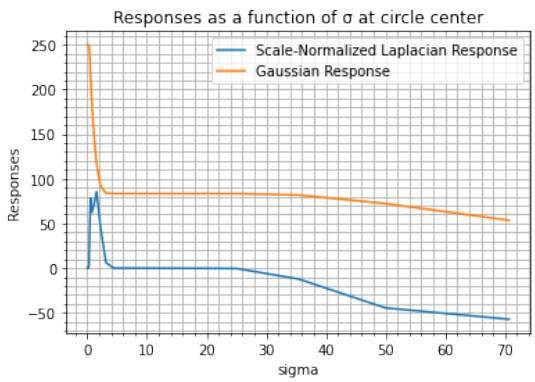


Figure 5: Responses of filters as a function of sigma at the center of an image of a circle

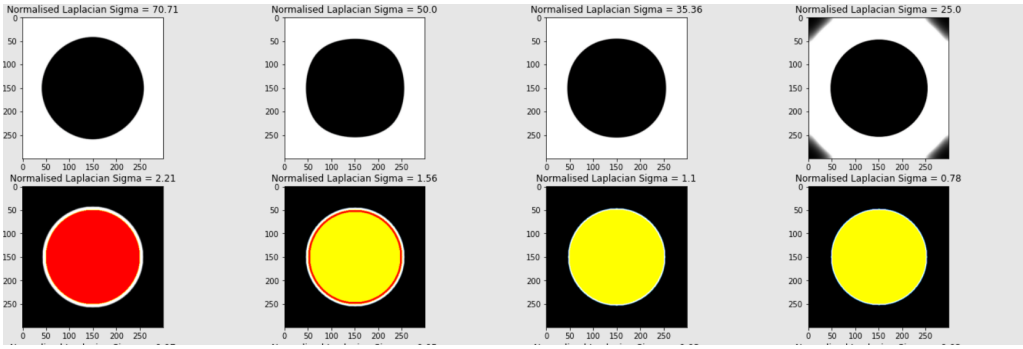


Figure 6: Sample Outputs of the scale normalized Laplacian filter responses

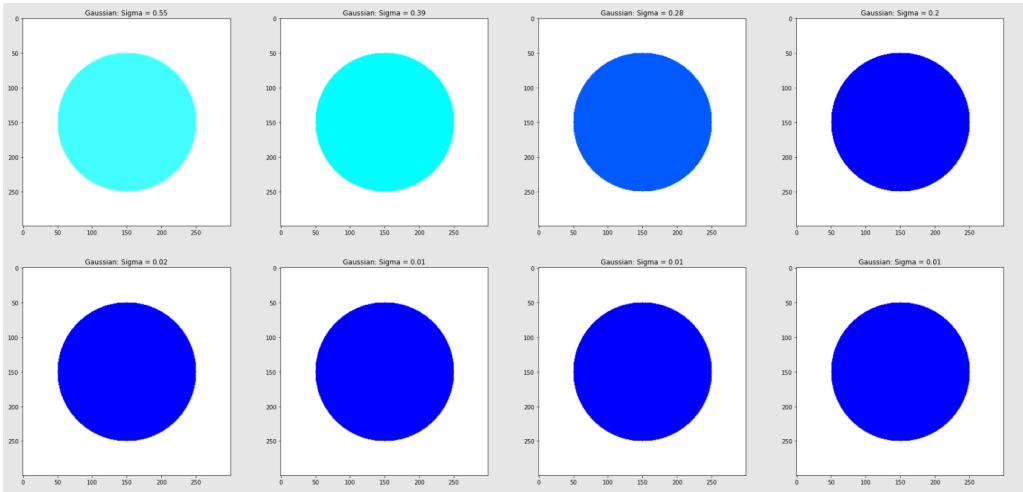


Figure 7: Sample Outputs of the Gaussian filter responses

1.5

1.5.1 RANSAC

Given Parameters:

- **Number of sampled points:** n (minimum number needed to fit the model) = 2
- **Inlier ratio i (i=#inliers/#datapoints):** 0.75

Choosing Parameters:

- **Fraction of points that are outliers:** p
- **Fraction of points that are inliers:** 1- p
- **Probability of Success:** s
- **Probability of Failure:** s-1
- **Number of Trails:** T

Choose T so that, with probability p, at least one random sample set is free from outliers
If p is the fraction of points that are outliers, the probability that choosing one point yields an

inlier will be:

$$1 - p$$

Now, the probability of choosing n inliers in a row (case when sample only contains inliers) will be:

$$(1 - p)^n$$

For our case, we have n = 2:

$$(1 - p)^2$$

So, the probability that one or more points in the sample were outliers (sample is contaminated) will be

$$1 - (1 - p)^2$$

And the probability that one or more points in the samplewere outliers (sample is contaminated) EVEN AFTER T iterations will be

$$(1 - (1 - p)^2)^T$$

From above equation, we can determine that the probability that RANSAC will terminate without finding the correct solution after T iterations will be,

$$P(\ fail\ T\ times) = select\ at\ least\ one\ outlier\ in\ all\ Ttrials$$

Iterations T are needed to find the correct solution with probability > 0:99 will be

$$s = 1 - (1 - (1 - p)^2)^T$$
$$0.99 = 1 - (1 - (0.75)^2)^T$$
$$T = \frac{log(1 - 0.99)}{log(1 - (0.75)^2)}$$
$$T = \frac{log(0.01)}{log(1 - 0.5625)}$$
$$T = \frac{log(0.01)}{log(0.4375)}$$
$$T = 5.570690151$$
$$T \simeq 6$$

Thus, we need 6 iterations or trials to find the correct Solution which would be good trade off between accuracy(99%) and time consumption.

1.5.2 Bruteforce Search

The bruteforce technique involves testing all lines that pass through a pair of points to find the best one. Assuming no 3 points are collinear, for N points, testing all such lines would reqire the following number of iterations :

$$\binom{N}{2} = \binom{100}{2} = 4950 \; where \; 100 \; is \; given \; number \; of \; points$$

1.5.3 Comparision

Although brute force provides better(perfect) accuracy than the RANSAC, it is computationally expensive with time complexity of $O(n^2)$ while RANSAC's time complexity can be viewed as approximately $O(logN)$ or around that which is basically much less than the bruteforce technique.

2 Depth from disparity

2.1 Compute depth image

We used the following implementation of Sum of Squared Differences(SSD) and Normalized Correlation as patch similarity measure functions:
Normalized Correlation
Find patch of the first image whose counterpart we want to find in second image, on the same epipolar line
Find patch of the second image , it will be each patch on the epipolar line of the second image width = width of the image/ length of the epipolar line
Then divide the sum of the difference between two patches by the square roots of the variances of each patch

```
#imported libraries
import numpy as np

def depthFromStereo(img1, img2, ws):
    #We take mean of all three channels of the first and second images to get a grayscale approximation of the images
    img1 = (img1[:,:,0]+img1[:,:,1]+img1[:,:,2])/3
    img2 = (img2[:,:,0]+img2[:,:,1]+img2[:,:,2])/3

    height1, width1 = np.shape(img1)
    height2, width2 = np.shape(img2)

    #Initialize depth mean to all zeros
    depth = np.zeros((height1,width1))
    difference = set()

    padSize = ws//2
    padHeight, padWidth = ws//2, ws//2

    #Handling even window sizes
    if ws % 2 ==0:
        padWidth -= 1

```

```
    pad1 = np.pad(img1, ((padSize, padSize), (padSize, padSize)), mode='constant', constant_values=(0,0))
    pad2 = np.pad(img2, ((padSize, padSize), (padSize, padSize)), mode='constant', constant_values=(0,0))

    metric = 0

    if metric == 0:
    #SSD
        #Traverse through all Epipolar Lines
        for i in range(height1):
            pX = i + padSize
            patch = [np.zeros((ws, ws))] * width2
            #For each pixel on epipolar lines
            for j in range(width2):
                pY = j + padSize
                #Patch Box
                patch[j] = pad2[pX - padHeight:pX+padHeight, pY-padWidth:pY + padWidth]

            for k in range(width1):
                x, dX = k, k
                pv = k + padSize
                patch1 = pad1[pX - padHeight:pX+ padHeight, pv-padWidth:pv + padWidth]
                minScore = float('inf')
                #Match the image patches on an epipolar line
                for j in range(k):
                    patch2 = patch[j]
                    SSD = float('inf')
                    SSD = np.sum(np.multiply(patch1 - patch2, patch1 - patch2))
                    if SSD <= minScore:
                        dX = j
                        minScore = SSD
                try:
                    disparity = abs(x - dX)
                    depth[i,k] = min(1/disparity, 1)
                    difference.add(disparity)
                except:
                    depth[i,k] = 1
                    difference.add(0)
    elif metric == 1:
    #Normalized correlation
        for i in range(height1):
            pX = i + padSize
            patch = [np.zeros((ws,ws))]* width2
            for j in range(width2):
                pY = j + padSize
                patch[j] = pad2[pX-padHeight:pX+padHeight, pY-padWidth:pY+padWidth]

            for k in range(width1):
                x, dX = k, k
                pv = k + padSize
                patch1 = pad1[pX-padHeight:pX+padHeight, pv-padWidth:pv+padWidth]

                maxScore = -float('inf')
```

```
                for j in range(width2):
                    patch2 = patch[j]
                    norm_corr = - float('inf')
                    norm_corr = (1 / width1) * (np.sum(patch1 - np.mean(patch2))
    / np.sqrt(np.var(patch1) * np.var(patch2)))
                    if norm_corr > maxScore:
                        dX = j
                        maxScore = norm_corr
                try:
                    disparity = abs(x - dX)
                    depth[i,k] = min(1/disparity, 1)
                    difference.add(disparity)
                except:
                    depth[i,k] = 1
                    difference.add(0)

    return depth
```

```
# This code is part of:
#
#   CMPSCI 670: Computer Vision, Fall 2020
#   University of Massachusetts, Amherst
#   Instructor: Subhransu Maji
#
#   Mini-project 2

import matplotlib.pyplot as plt
from utils import imread
from depthFromStereo import depthFromStereo
import os

read_path = "../data/disparity/"

testExamples = ["cones", "teddy"]
for exampleIndex in range(len(testExamples)):
    im_name1 = "{}_im2.png".format(testExamples[exampleIndex])
    im_name2 = "{}_im6.png".format(testExamples[exampleIndex])

    #Read test images
    img1 = imread(os.path.join(read_path, im_name1))
    img2 = imread(os.path.join(read_path, im_name2))
    ws = 100
    #Compute depth
    depth = depthFromStereo(img1, img2, ws)

    #Show result
    plt.imshow(depth, cmap="jet")
    plt.show()
    save_path = "../../outputs/disparity/"
    save_file = im_name1+""
    if not os.path.isdir(save_path):
        os.makedirs(save_path)
    plt.savefig('{}_SSDws{}.png'.format(save_file, ws), cmap="jet")
```

2.2 Disparity: Results

Here, we will try different patch similarity measure and window sizes to show their impact on resultant depth images.

2.2.1 Outputs

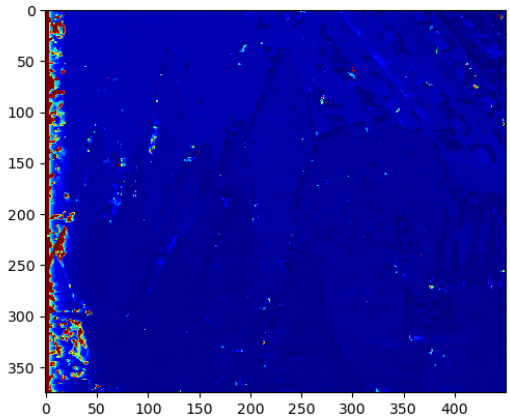


Figure 8: Depth Map with Normalized Correlation patch similarity measure function for cones with window size 5

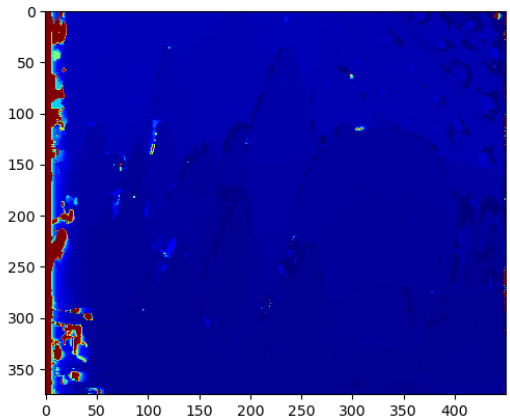


Figure 9: Depth Map with Normalized Correlation patch similarity measure function for cones with window size 10

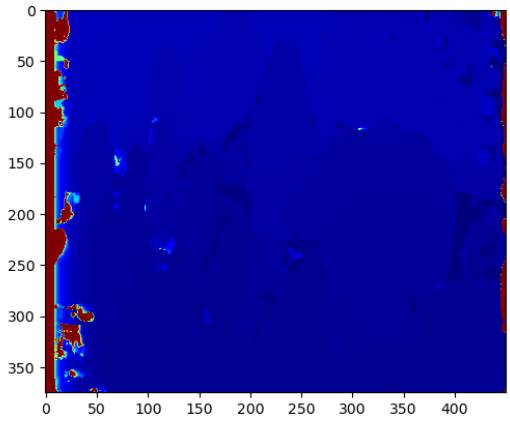


Figure 10: Depth Map with Normalized Correlation patch similarity measure function for cones with window size 15

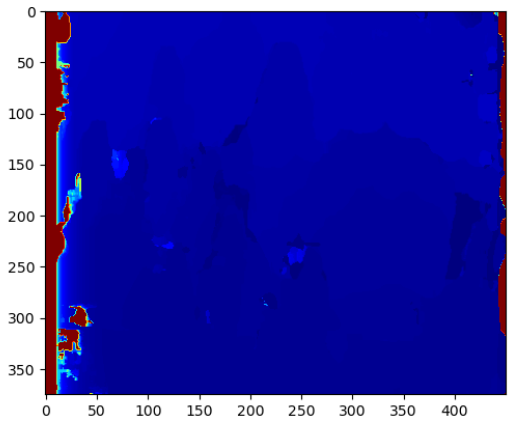


Figure 11: Depth Map with Normalized Correlation patch similarity measure function for cones with window size 20

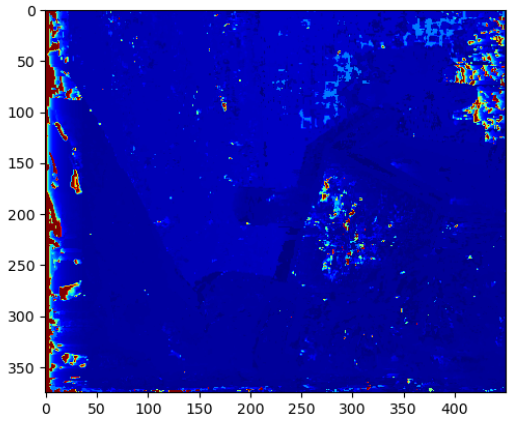


Figure 12: Depth Map with Normalized Correlation patch similarity measure function for teddy with window size 5

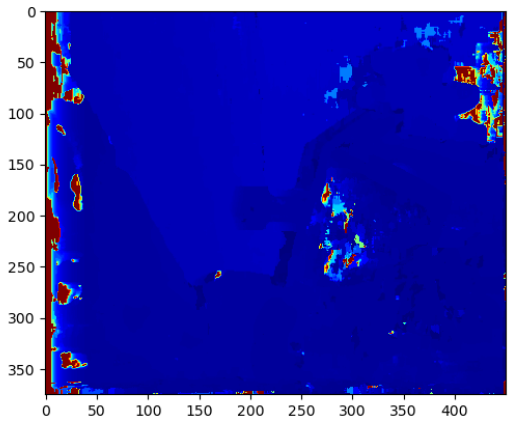


Figure 13: Depth Map with Normalized Correlation patch similarity measure function for teddy with window size 10

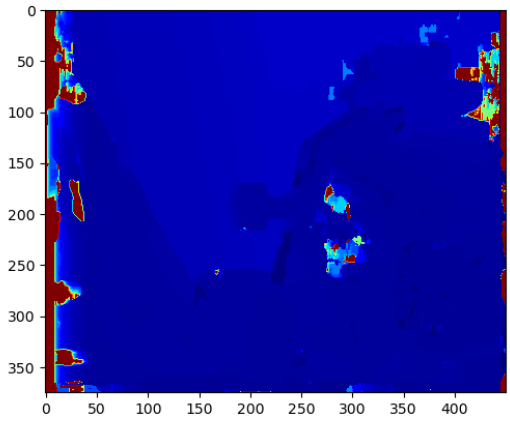


Figure 14: Depth Map with Normalized Correlation patch similarity measure function for teddy with window size 15

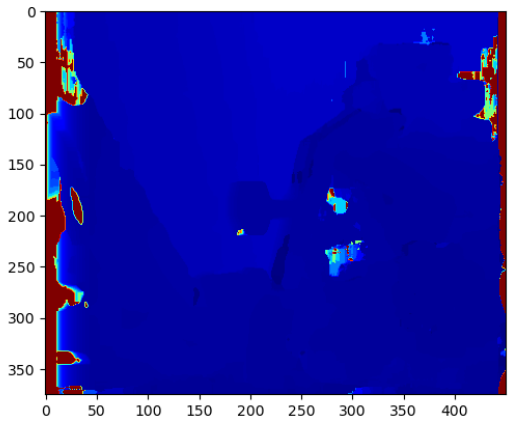


Figure 15: Depth Map with Normalized Correlation patch similarity measure function for teddy with window size 20

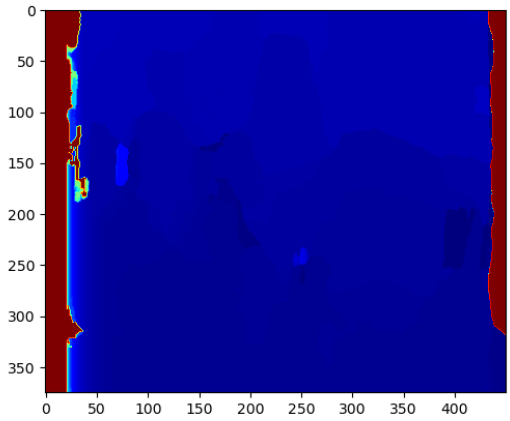


Figure 16: Depth Map with SSD patch similarity measure function for cones with window size 5

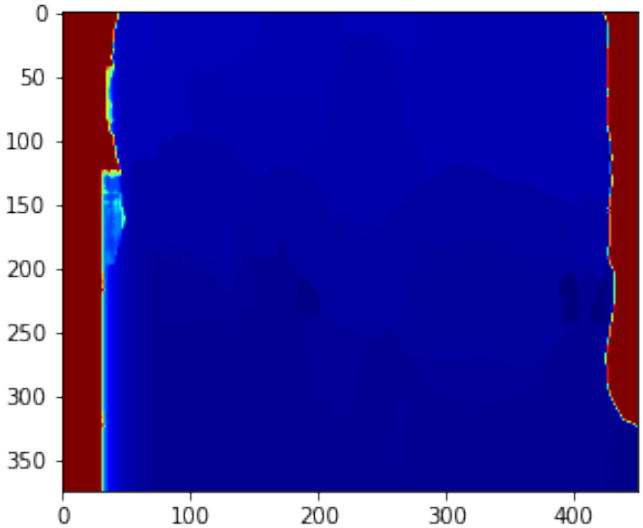


Figure 17: Depth Map with SSD patch similarity measure function for cones with window size 10

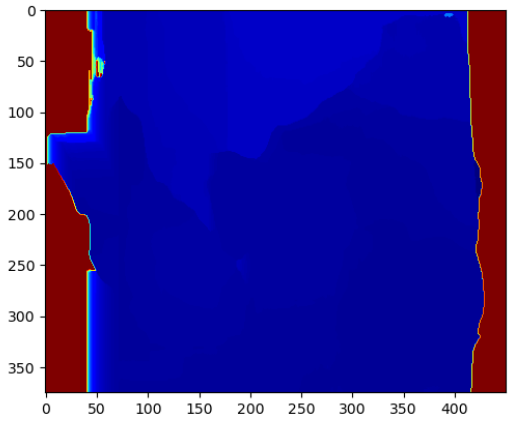


Figure 18: Depth Map with SSD patch similarity measure function for cones with window size 15

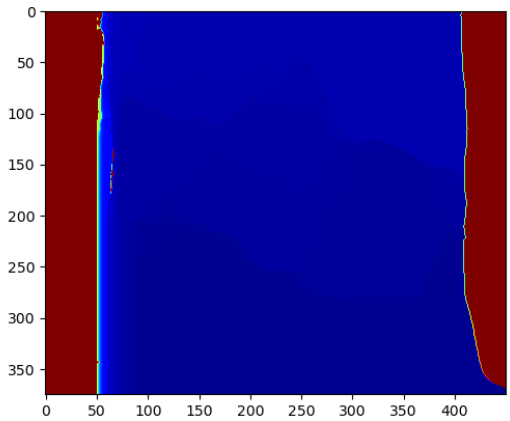


Figure 19: Depth Map with SSD patch similarity measure function for cones with window size 20

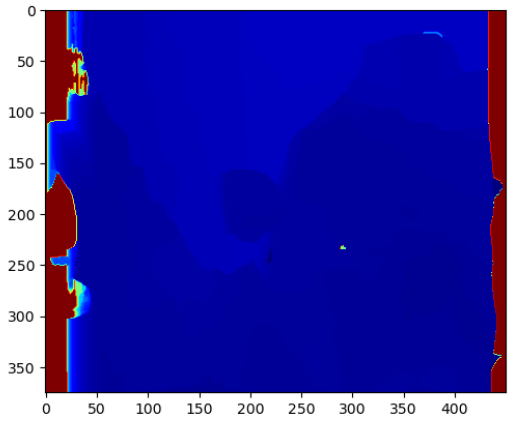


Figure 20: Depth Map with SSD patch similarity measure function for teddy with window size 5

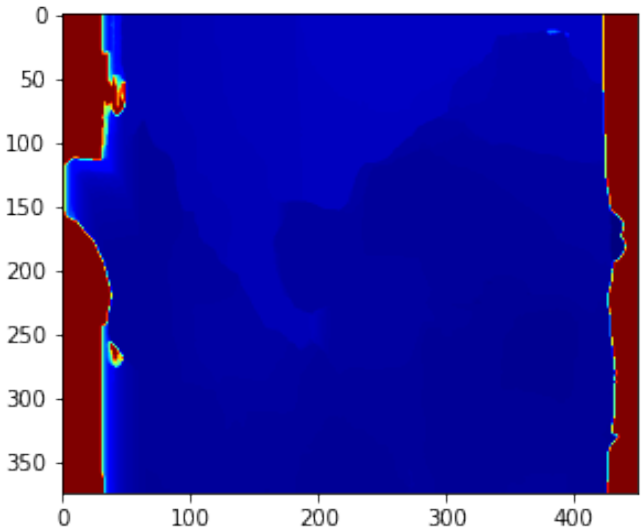


Figure 21: Depth Map with SSD patch similarity measure function for teddy with window size 10

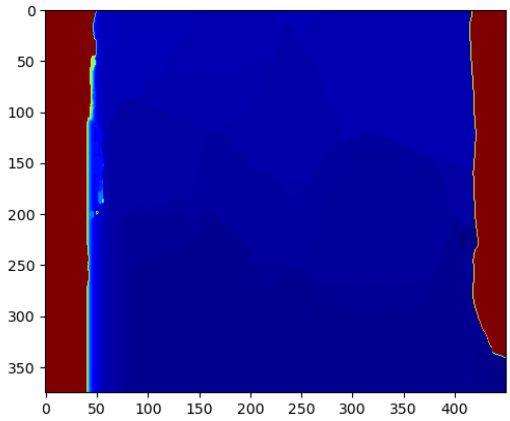


Figure 22: Depth Map with SSD patch similarity measure function for teddy with window size 15

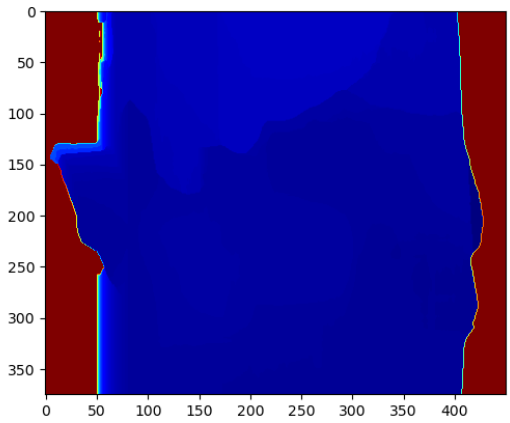


Figure 23: Depth Map with SSD patch similarity measure function for teddy with window size 20

2.2.2 Discussion

As the patch size increases, the result loses it's "sharpness" or the clarity round the edges. Ideal patch size should be somewhere from 5 to 10.
But with small patch sizes, there's lot of noise in the depth map.
Normalized Correlation gives better result than SSD.
Normalized Correlation is preferred over SSD because it is invariant to affine (translation + scaling) intensity changes while SSD is not invariant. Which might result in highly fluctuating and incorrect descriptors between two similar images.

2.2.3 What worked well and where it failed

Patch size is small e.g., 5 to 10 pixels, this will lead to more computation as the resolution goes higher. Hence its an accuracy-time and space trade-off.

Also, with bigger patches, the artifact effect is more apparent on the boundaries. Therefore, this approach doesn't work as good on the boundaries with bigger patches as for the core part of the images.
Different patch similarity measure functions certainly makes a difference, Normalized Correlation is a go-to method for any kind of error measurement, but application specific cost measurements work better.
Too small of a patch size will lead to matching a patch with totally unrelated, different part of the image because as the patch size goes smaller, the chances are higher of two pixel patterns looking very similar.
The boundary artifact grows bigger as the distance between the two cameras increase.
Normalized Correlation outputs were developed in hours and costed a lot of runtime

2.3 Discussion: Advantages and disadvantages of Disparity Stereo over the Photometric Stereo

1. Photometric Stereo needs light falling on the object from many different Azimuthal and Elevation angles to estimate depth with passable accuracy. Disparity Stereo technique does not require a change in lighting conditions at all.
2. Comparatively, Disparity Stereo can work well with a small number of sample images, while Photometric Stereo needs a relatively higher number of images to estimate depth.
3. Only small movements in camera position is acceptable for Disparity Stereo, bigger/ wider distance between two cameras might result in totally wrong and illegible output
4. Changing the lighting in Disparity Stereo for one camera can affect the change of color of the same object, which will be an obstacle in trying to recognize the patch with the most minor disparity.
5. Disparity Stereo is strictly based on epipolar geometry. If the cameras don't have the same height base, the whole concept of Stereo, which we applied won't work.
6. Disparity Stereo doesn't depict continuous depth like photometric Stereo which does using different integration techniques. It just separate objects by attributing them to different "planes," but the actual smoothness of depth is not there.

2.4 Implementation of Second Order Statistics:

```
# This code is part of:
#
#   CMPSCI 670: Computer Vision, Fall 2020
#   University of Massachusetts, Amherst
#   Instructor: Subhransu Maji
#
#   Homework 3

import numpy as np
import skimage.io as sio
import os
from skimage.color import rgb2gray
import matplotlib.pyplot as plt

#Imported librarires
from matplotlib.colors import LinearSegmentedColormap
from scipy import ndimage
from scipy.ndimage import gaussian_filter
```

```python

def second_order_statistics(ix, iy, ws):
    height, width = np.shape(ix)
    R = np.zeros((height, width))

    k = 0.03

    IXIX = gaussian_filter(np.multiply(ix, ix),(ws-1)//2)
    IXIY = gaussian_filter(np.multiply(ix, iy),(ws-1)//2)
    IYIY = gaussian_filter(np.multiply(iy, iy),(ws-1)//2)

    #For each pixel(u,v)
    for u in range(height):
        for v in range(width):
            SecondMoment = np.zeros((2,2))
            a = IXIX[u, v]
            b = IXIY[u,v]
            c = b
            d = IYIY[u, v]
            SecondMoment[0, 0] = a
            SecondMoment[0, 1] = b
            SecondMoment[1, 0] = c
            SecondMoment[1, 1] = d
            #Compute quantity R
            lambda1 = a * d
            lambda2 =  b * c
            sumlamda12 = a + d
            R[u, v] = (lambda1 - lambda2) - k *(np.power(sumlamda12, 2))

    return R

color_list = ['white','darkblue']
cmap = LinearSegmentedColormap.from_list("",color_list)
files = ["cones_im2.png","teddy_im2.png"]

for image_file in files:
    # read image
    image = sio.imread(image_file)
    #convert image to gray
    image = rgb2gray(image)
    #compute differentiations along x and y axis respectively
    # x-diff
    #--------- add your code here ------------------#
    kernelX = np.array([[0, 0, 0], [-1, 0, 1], [0, 0, 0]], np.float32)
    Ix = ndimage.convolve(image, kernelX)

    # y-diff
    #--------- add your code here ------------------#
    kernelY = np.array([[0, -1, 0], [0, 0, 0], [0, 1, 0]], np.float32)
    Iy = ndimage.convolve(image, kernelY)

    #set window size
    #--------- modify this accordingly ------------------#
    for ws in range(5, 11, 5):
        heatMapImg = second_order_statistics(Ix, Iy, ws)
        plt.imshow(heatMapImg, cmap =cmap)
        plt.title("{} Heat Map with radius:{}".format(image_file, ws))
        plt.show()
```

```
        plt.savefig("{}_{}HeatMap.png".format(image_file, ws), cmap =cmap)
```

2.5 Visualization of R

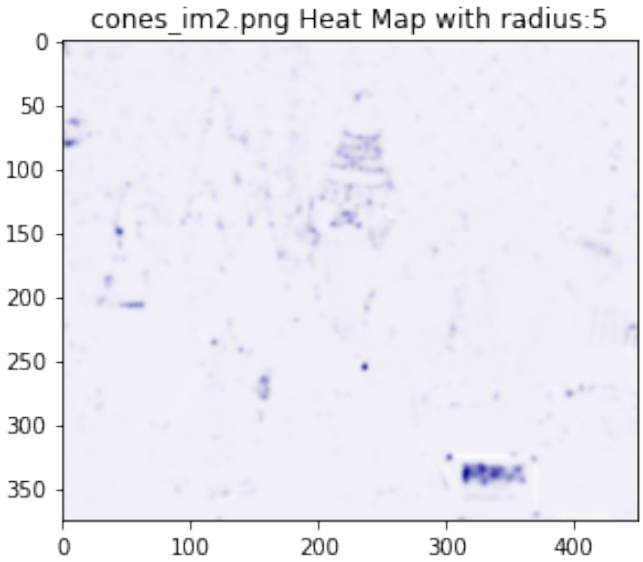


Figure 24: Information Patches for cones_im2 with window size 5

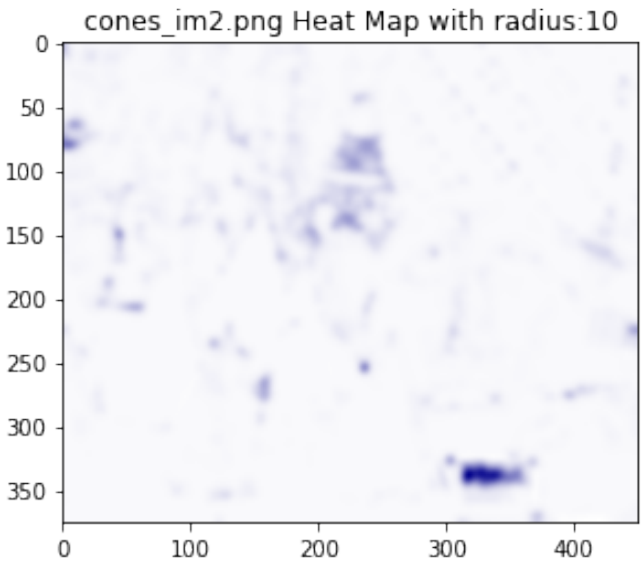


Figure 25: Information Patches for cones_im2 with window size 10

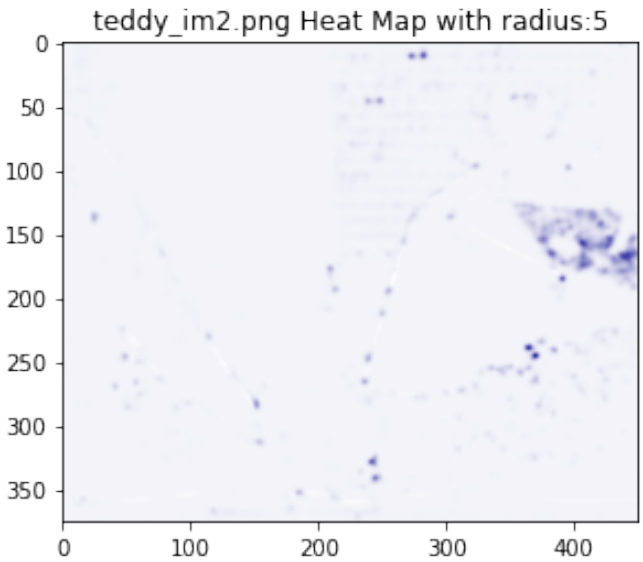


Figure 26: Information Patches for teddy_im2 with window size 5

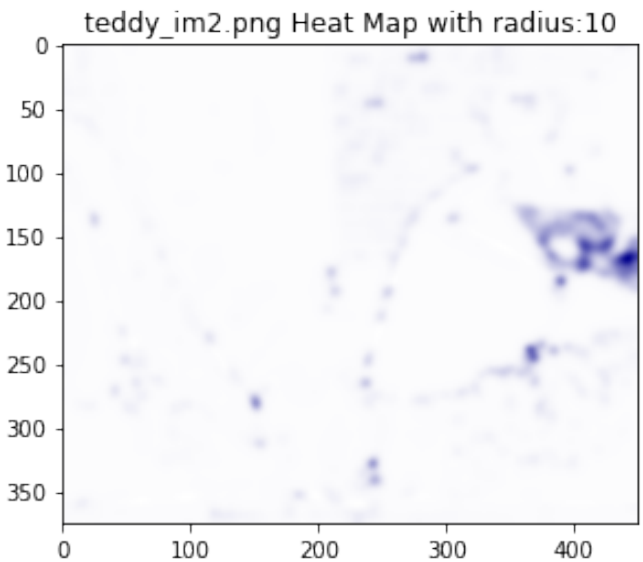


Figure 27: Information Patches for teddy_im2 with window size 10

2.6 Discussion what does R indicate

Following are QUANTITATIVE ASSERTIONS about R

- When R is large -
  For example R(u; v) for pixel at (u; v) is large, it would be because both λ_1 and λ_2 are large, making the first term of equation of R large, because second term would be multiplied by 0.03, so the subtraction will still keep R relatively very large because both eigenvalues are large, then we can say that it indicates a corner.

- When R is around near the mean or median of all the R values -
  For example R(u; v) for pixel at (u; v) is moderate, it would be because one of λ_1 and λ_2 is

large, while the other one is small, making the subtraction of two terms of equation of R depend on how large one λ is relative to the other one, the more the difference between the two eigenvalues, the sharper/ more accurate the edge depiction, then we can say that it indicates an edge.

- If $\lambda_1 \gg \lambda_2$
  then the edge will be vertical, else the edge will be horizontal.
- If R is small,
  then both λ_1 and λ_2 would be small, we can say that it indicates a flat surface.

Referring to Figures included in Visualization of R subsection, we make the following observations:

- For both the cones and teddy images, when we vary window radius from 5 to 10,
  It shows that the information indicator doesn't capture much details about the object because as the neighboring area increases, the pixel values make eigenvectors negate the total sum. Even if there's a large λ, the gradient and sum total will hide it's impact, resulting in edges fading into a flat region.
- In cones image for radius 5,
  We can see the outline of the cones, fisherbrand safety matches label and the face mask mug edges in the background.
- While for radius of 10, in cones image,
  We can only decipher a very little part of edges at cones and fisherbrand safety matches label edges.
- Also, the cones original image has many tiny artifacts in it, making it very congested and noisy when we try to produce an information patch.
- In teddy image, for radius 5
  We can see 3 edge outlines (laid down house) and some minor details captured very well about the house front. As this image appears to include flat regions towards left and central sides, most of the part is empty white, depicting very small R values.
- For radius 10 in teddy image,
  We can see that fine edges fade into flat region. It only shows on the right side (from the center) of the image which is highly contrasting from it's background.

3 Scale-space blob detection

3.1 Implementation of *detectblobs.py*

Here we localize circular patches(blobs) in a given image using a Laplacian of Gaussian filter and by doing scale-space analysis.
By increasing the sigma value we tend to miss out on the finer features while too low sigma value leads to detecting too many local features. Thus, choosing the optimum sigma value depends on the application and the image at hand.
Also,number of detected blobs decrease if we increase the thresholding factor.

Approach: Iteratively filter the image with kernel of increasing size
1.Build a Laplacian scale space, starting with some initial scale and going for n iterations:

1.a. Filter image with scale-normalized Laplacian at current scale.
1.b. Save the square of Laplacian response for current level of scale space.
1.c Increase scale by a factor k.
2. Perform non-maximum suppression in scale space.
3. Display resulting circles at their characteristic scales for points above a threshold.

```python
# This code is part of:
#
#   CMPSCI 670: Computer Vision
#   University of Massachusetts, Amherst
#   Instructor: Subhransu Maji
#

import numpy as np
import cv2
import scipy
from skimage.color import rgb2gray

#Imported libraries
from scipy.ndimage.filters import rank_filter
from itertools import chain
import time
from skimage import transform

def  LoGFilter(initSigma, filterSize):
    #Laplacian of Gaussian Filter
    sigma = initSigma ** 2

    idRange = np.linspace(-(filterSize - 1) / 2., (filterSize - 1) / 2., filterSize)
    x, y = np.meshgrid(idRange, idRange)

    kernel = np.exp(-(np.square(x) + np.square(y)) / (2. * sigma))
    kernel[kernel < np.finfo(float).eps * np.amax(kernel)] = 0
    kSum = np.sum(kernel)
    if kSum != 0:
        kernel /= np.sum(kernel)

    tmp = np.multiply(kernel, np.square(x) + np.square(y) - 2 * sigma) / (sigma ** 2)
    kernel = tmp - np.sum(tmp) / (filterSize ** 2)
    return kernel

def detectBlobs(im, param=None):
    # Input:
    #   IM - input image
    #
    # Ouput:
    #   BLOBS - n x 5 array with blob in each row in (x, y, radius, angle, score)

    #Convert image to grayscale and then to double such that minimum and maximum intensity are 0 and 1 respectively
    im = rgb2gray(im[:,:,:])
    im = cv2.normalize(im.astype('float'), None, 0.0, 1.0, cv2.NORM_MINMAX)

    #Compute scale space representation
    initSigma = 1.6 #Initial Scale
```

```
    factor = 1.25 #np.sqrt(2) #Factor by which scale is mulitplied each time
    threshold =  0.0001
    finalSigma = np.power(factor, 15) #Final Scale

    #Dynamically decide iteration levels from initial scale, last scale and multiplication factor
    n = 12 #int(np.ceil((np.log(finalSigma) - np.log(initSigma))/np.log(factor))) #Iterations of Laplacian Scale Space

    sigmas = [initSigma] * n
    sigmas[0] = initSigma

    height, width = np.shape(im)
    scaleSpace = np.empty((height, width, n))

    method = 0
    ######################################################################
    # Compute blob response (Squared Laplacian response in scale-space) #
    ######################################################################

    if method == 0:
        #Increase filter size, keep image same
        #Slower version
        for i in range(n):
            #Generate the Laplacian of Gaussian for Scale Level
            filterSize = int(round(6 * sigmas[i])) #Compute filter kernel size from previous sigma
            # Ensure right - odd filter size
            if filterSize % 2 == 0:
                filterSize += 1
            LoG = np.power(sigmas[i],2) * LoGFilter(sigmas[i], filterSize) # Obtain filter with normalization and apply convolution.
            nextFilter= cv2.filter2D(im, cv2.CV_32F, LoG) #Create a replica of previous filter
            scaleSpace[:,:, i] = np.square(nextFilter) #Store squared filter at each level
            #Update sigma for next level
            if (i+1 < n):
                sigmas[i+1] = sigmas[i] * factor

    elif method == 1:
        #Downsampling image - Faster version
        # Ensure odd filter size
        filterSize = int(round(6 * sigmas[0]))
        # Ensure right - odd filter size
        if filterSize % 2 == 0:
            filterSize += 1
        LoG = LoGFilter(sigmas[0], filterSize )

        # Scale the image.
        for i in range(n):
            # Down scale
            scaledH = int(height * ((1/factor) ** i))
            scaledW = int(width * ((1/factor) ** i))
            scaledIm = transform.rescale(im, (1/factor) ** i, order=3)
            nextFilter= cv2.filter2D(scaledIm, cv2.CV_32F, LoG)**2
            # Upscale
            scaleSpace[:,:, i] = transform.resize(nextFilter, (height, width),
```

```
order=3)

    elif method == 2:
        #Difference-of-Gaussian pyramid
        # Ensure odd filter size.
        filterSize = int(round(6 * sigmas[0]))

        # Obtain filter (no normalization needed).
        DoG = LoGFilter(sigmas[0], filterSize) - LoGFilter(sigmas[0]*1.25, filterSize )

        # Scale the image.
        for i in range(n):
            #Down scale
            scaledH = int(height * ((1/factor) ** i))
            scaledW = int(width * ((1/factor) ** i))
            scaledIm = transform.rescale(im, (1/factor) ** i, order=3)
            nextFilter= cv2.filter2D(scaledIm, cv2.CV_32F, DoG)**2
            # Upscale
            scaleSpace[:,:, i] = transform.resize(nextFilter, (height, width), order=3)

    ##################################################
    # Perform Non-Maximum Suppression in Scale Space  #
    ##################################################
    maxScaleSpace = np.empty((height, width, n)) #Non-Maximum Suppression scale space for each level

    #Non-Maximum Suppression in each 2D slice
    #A point is Local Maximum if it is higher than all its neighbors (each point has 26 neighbors in 3D)
    for level in range(n):
        maxScaleSpace[:,:,level] = rank_filter(scaleSpace[:,:, level], -1, (3,3)) #Determining Local Maximum point using rank filter

    layerScaleSpace = np.zeros((maxScaleSpace.shape)) #Non-Maximum Suppression scale space for all levels

    #Compute non-max suppression across all layers
    for i in range(height):
        for j in range(width):
            maxValue = np.amax(maxScaleSpace[i,j,:])
            maxIndex = np.argmax(maxScaleSpace[i,j,:])
            layerScaleSpace[i,j,maxIndex] = maxValue

    #Zero Out All Positions that are not Local Maxima of the score (if the Value if not Greater than all its Neighbors)
    for level in range(n): # check
        layerScaleSpace[:,:,level] = np.where((layerScaleSpace[:,:,level] == scaleSpace[:,:,level]), layerScaleSpace[:,:,level], 0)

    #Mask filter to eliminate boundaries noises
    #check
    mask = np.zeros((height, width, n))
    for i in range(n):
        boundary = int(np.ceil(sigmas[i] * np.sqrt(2)))
        mask[boundary+1:height-boundary, boundary+1:width-boundary] = 1

```

```
    centerX = []
    centerY = []
    radii = []
    score = []

    #Set a Threshold on Squared Laplacian Response above which to report Region Detections
    for level in range(n):#check >=
        layerScaleSpace[:,:,level] = np.where((layerScaleSpace[:,:,level] > threshold) & (mask[:,:,level] == 1), layerScaleSpace[:,:,level], 0)

        #Obtain X and Y center coordinates for local maxima
        centerX.append(list(np.where(layerScaleSpace[:,:, level] != 0)[0]))
        centerY.append(list(np.where(layerScaleSpace[:,:,level] != 0)[1]))
        radius = np.sqrt(2) * sigmas[level] * np.ones(len(centerX[level])) # Compute radius
        radii.append(list(radius))
        score.append(layerScaleSpace[np.where(layerScaleSpace[:,:, level] != 0)[0],np.where(layerScaleSpace[:,:, level] != 0)[1],level])

    centerX = list(chain.from_iterable(centerX))
    centerY = list(chain.from_iterable(centerY))
    radii = list(chain.from_iterable(radii))
    score = list(chain.from_iterable(score))

    numBlob = len(centerX)
    blobs = []
    angle = np.zeros(numBlob)
    print(numBlob)
    if (numBlob > 0):
        for i in range(numBlob):
            blobs.append([centerY[i], centerX[i], radii[i], angle[i], score[i]])

    return np.array(blobs)
```

3.2 Results of given test images

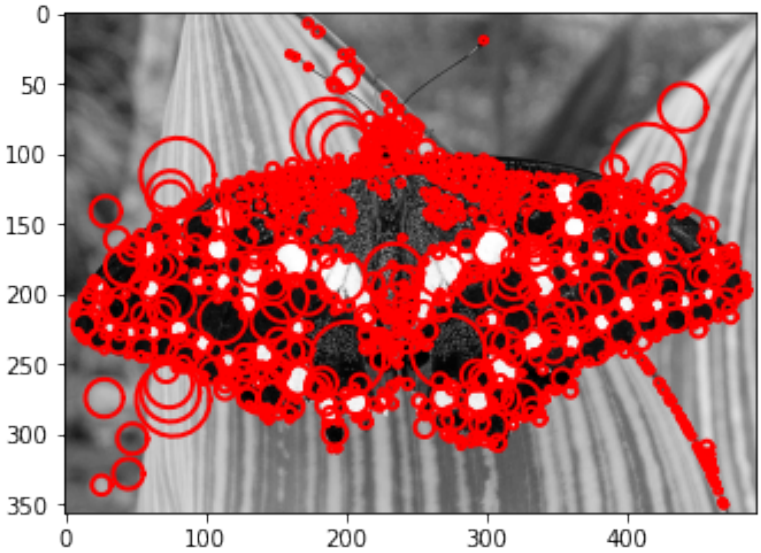


Figure 28: butterfly with number of blobs:1776, Initial Sigma =1.6 threshold = 0.0001

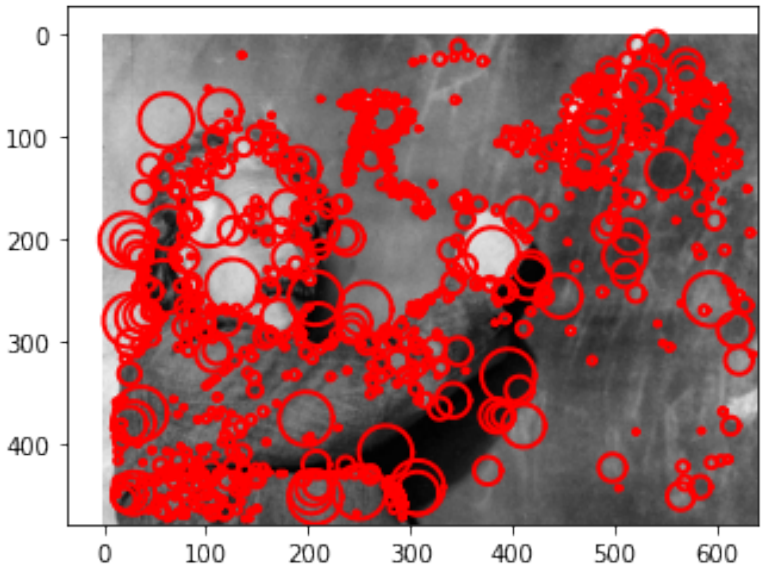


Figure 29: einstein with number of blobs:2636, Initial Sigma =1.6 threshold = 0.0001

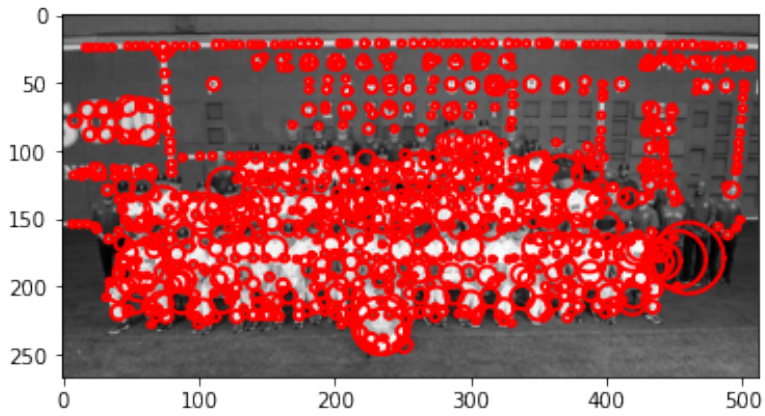


Figure 30: faces with number of blobs:1970, Initial Sigma =1.6 threshold = 0.0001

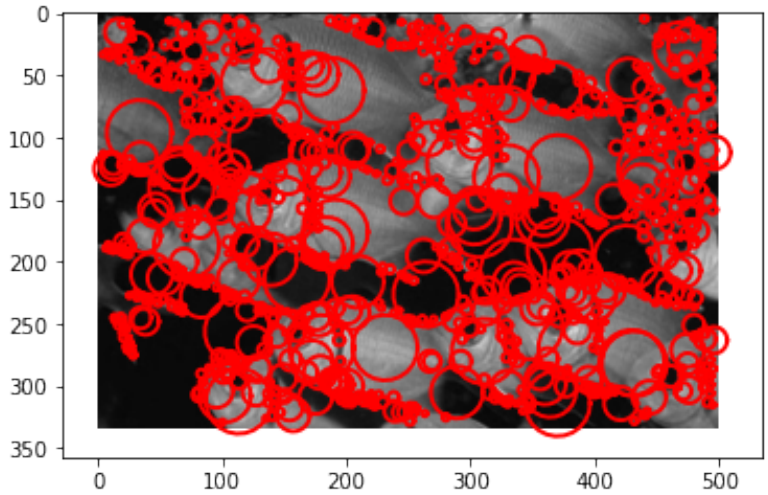


Figure 31: fishes with number of blobs:1163, Initial Sigma =1.6 threshold = 0.0001

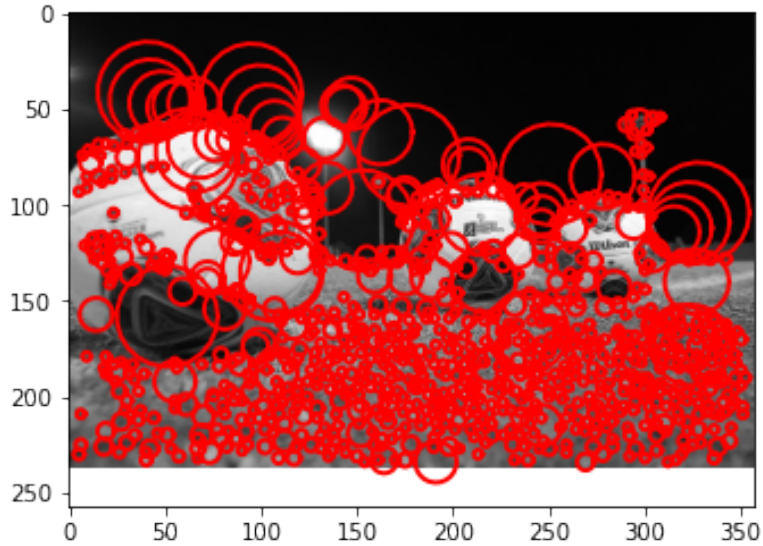


Figure 32: football with number of blobs:1073, Initial Sigma =1.6 threshold = 0.0001

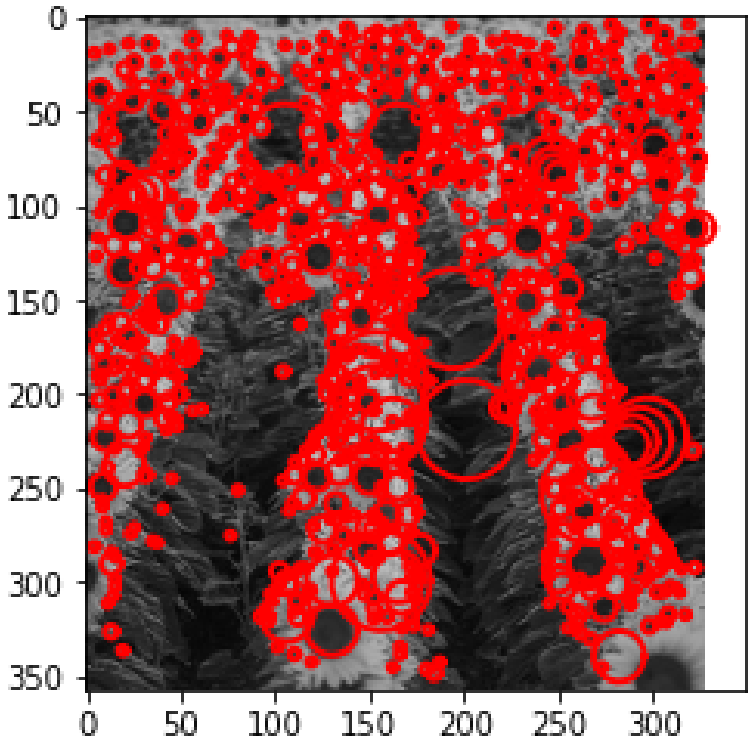


Figure 33: sunflowers with number of blobs:2083, Initial Sigma =1.6 threshold = 0.0001

4 Image stitching

4.1 Detect keypoints

Following settings of sigma = 1.3, threshold =0.05, n = 12 and factor = 1.25 were set in *detectblobs2.py*
Implementation of *detectblobs2.py*

```
# This code is part of:
#
#   CMPSCI 670: Computer Vision
#   University of Massachusetts, Amherst
#   Instructor: Subhransu Maji
#

import numpy as np
import cv2
import scipy
from skimage.color import rgb2gray

#Imported libraries
from scipy.ndimage.filters import rank_filter
from itertools import chain

def  LoGFilter(initSigma, filterSize):
    #Laplacian of Gaussian Filter
    sigma = initSigma ** 2

    # Ensure right - odd filter size
```

```
    if filterSize % 2 == 0:
        filterSize += 1

    idRange = np.linspace(-(filterSize - 1) / 2., (filterSize - 1) / 2., filterSize)
    x, y = np.meshgrid(idRange, idRange)

    kernel = np.exp(-(np.square(x) + np.square(y)) / (2. * sigma))
    kernel[kernel < np.finfo(float).eps * np.amax(kernel)] = 0
    kSum = np.sum(kernel)
    if kSum != 0:
        kernel /= np.sum(kernel)

    tmp = np.multiply(kernel, np.square(x) + np.square(y) - 2 * sigma) / (sigma ** 2)
    kernel = tmp - np.sum(tmp) / (filterSize ** 2)
    return kernel

def detectBlobs(im, param=None):
    # Input:
    #   IM - input image
    #
    # Ouput:
    #   BLOBS - n x 5 array with blob in each row in (x, y, radius, angle, score)

    #Convert image to grayscale and then to double such that minimum and maximum intensity are 0 and 1 respectively
    im = rgb2gray(im[:,:,:])
    im = cv2.normalize(im.astype('float'), None, 0.0, 1.0, cv2.NORM_MINMAX)

    #Compute scale space representation
    initSigma = 1.3 #Initial Scale
    factor = 1.25 #np.sqrt(2) #Factor by which scale is mulitplied each time
    threshold =  0.005
    finalSigma = np.power(factor, 16) #Final Scale

    #Dynamically decide iteration levels from initial scale, last scale and multiplication factor
    n = 12 #int(np.ceil((np.log(finalSigma) - np.log(initSigma))/np.log(factor))) #Iterations of Laplacian Scale Space

    sigmas = np.empty(n)
    sigmas[0] = initSigma

    height, width = np.shape(im)
    scaleSpace = np.empty((height, width, n))

    method = 0
    ######################################################################
    # Compute blob response (Squared Laplacian response in scale-space) #
    ######################################################################

    if method == 0:
        #Increase filter size, keep image same
        for i in range(n):
            #Generate the Laplacian of Gaussian for Scale Level
```

```
            filterSize = int(round(6 * sigmas[i])) #Compute filter kernel size from previous sigma
            LoG = np.power(sigmas[i],2) * LoGFilter(sigmas[i], filterSize) #Obtain filter with normalization and apply convolution.
            nextFilter= cv2.filter2D(im, cv2.CV_32F, LoG) #Create a replica of previous filter
            scaleSpace[:,:, i] = np.square(nextFilter) #Store squared filter at each level

            #Update sigma for next level
            if (i+1 < n):
                sigmas[i+1] = sigmas[i] * factor

    ##################################################
    # Perform Non-Maximum Suppression in Scale Space  #
    ##################################################
    maxScaleSpace = np.empty((height, width, n)) #Non-Maximum Suppression scale space for each level

    #Non-Maximum Suppression in each 2D slice
    #A point is Local Maximum if it is higher than all its neighbors (each point has 26 neighbors in 3D)
    for level in range(n):
        maxScaleSpace[:,:,level] = rank_filter(scaleSpace[:,:, level], -1, (3,3)) #Determining Local Maximum point using rank filter

    layerScaleSpace = np.zeros((maxScaleSpace.shape)) #Non-Maximum Suppression scale space for all levels

    #Compute non-max suppression across all layers
    for i in range(height):
        for j in range(width):
            maxValue = np.amax(maxScaleSpace[i,j,:])
            maxIndex = np.argmax(maxScaleSpace[i,j,:])
            layerScaleSpace[i,j,maxIndex] = maxValue

    #Zero Out All Positions that are not Local Maxima of the score (if the Value if not Greater than all its Neighbors)
    for level in range(n):
        layerScaleSpace[:,:,level] = np.where((layerScaleSpace[:,:,level] == scaleSpace[:,:,level]), layerScaleSpace[:,:,level], 0)

    #Mask filter to eliminate boundaries noises
    #check
    mask = np.zeros((height, width, n))
    for i in range(n):
        boundary = int(np.ceil(sigmas[i] * np.sqrt(2)))
        mask[boundary+1:height-boundary, boundary+1:width-boundary] = 1

    centerX = []
    centerY = []
    radii = []
    score = []

    #Set a Threshold on Squared Laplacian Response above which to report Region Detections
    for level in range(n):
```

```
        layerScaleSpace[:,:,level] = np.where((layerScaleSpace[:,:,level] > threshold) & (mask[:,:,level] == 1), layerScaleSpace[:,:,level], 0)

        #Obtain X and Y center coordinates for local maxima
        centerX.append(list(np.where(layerScaleSpace[:,:, level] != 0)[0]))
        centerY.append(list(np.where(layerScaleSpace[:,:,level] != 0)[1]))
        radius = np.sqrt(2) * sigmas[level] * np.ones(len(centerX[level])) #np.sqrt(2) * np.power(factor, level-1) * initSigma * np.ones(len(centerX[level])) # Compute radius
        radii.append(list(radius))
        score.append(layerScaleSpace[np.where(layerScaleSpace[:,:, level] != 0)[0],np.where(layerScaleSpace[:,:, level] != 0)[1],level])

    centerX = list(chain.from_iterable(centerX))
    centerY = list(chain.from_iterable(centerY))
    radii = list(chain.from_iterable(radii))
    score = list(chain.from_iterable(score))

    numBlob = len(centerX)
    blobs = []
    angle = np.zeros(numBlob)
    if (numBlob > 0):
        for i in range(numBlob):
            blobs.append([centerY[i], centerX[i], radii[i], angle[i], score[i]])

    return np.array(blobs)
```

Implementation of *drawtblobs2.py*

```
import numpy as np
import matplotlib.pyplot as plt
from skimage.color import rgb2gray

# DRAWBLOBS overlays the image with blobs as circles
#   DRAWBLOBS(IM, BLOBS, THRESHOLD) overalys IM converted to a grayscale
#   image with BLOBS that are above a THRESHOLD. If THRESHOLD is not
#   specified it is set to 0.
#
# Input:
#   IM - the image (if rgb image is provided it is converted to grayscale)
#   BLOBS - n x 5 matrix with each row is a blob (x, y, radius, angle, score)
#   THRESHOLD - only blobs above this are shown (default = 0)
#
# This code is part of:
#
#   CMPSCI 670: Computer Vision
#   University of Massachusetts, Amherst
#   Instructor: Subhransu Maji

def drawBlobs(im, blobs, title):
    nmax = blobs.shape[0]
    nmax = min(nmax, blobs.shape[0])

    if im.shape[2] > 1:
        im = rgb2gray(im)

    plt.figure()
    plt.imshow(im, cmap="gray")
```

```
    if nmax < 1:
        return

    order = np.argsort(-blobs[:, 4])
    theta = np.linspace(0, 2*np.pi, 24)
    for i in range(nmax):
        r = blobs[order[i], 2]
        r = 1
        plt.plot(blobs[order[i], 0] + r*np.cos(theta),
                 blobs[order[i], 1] + r*np.sin(theta),
                 'r-', linewidth=2)
    plt.title(title)
    plt.savefig("{}.png".format(title))
    plt.show()
```

4.2 Feature extraction

SIFT Feature Descriptors transforms an image into a large collection of feature vectors, each of which is invariant to image translation, scaling, and rotation, partially invariant to illumination changes and robust to local geometric distortion.
In this method:
1. We divide each blob in 4x4 grid.
2. Inside each sub-patch, compute histogram of gradient orientations (8 reference angles).
3. We normalize the vector to a unit length.
4. Resulting descriptor will be 4x4x8 = 128 dimensions for each blob.
5. This step is already implemented; at the end of this step, we will get SIFT features for both of our images. *Intermediate Results:*
SIFT Features

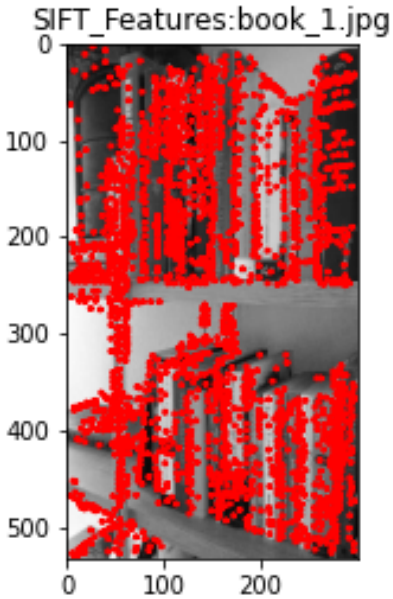


Figure 34: SIFT Features : book_1.jpg

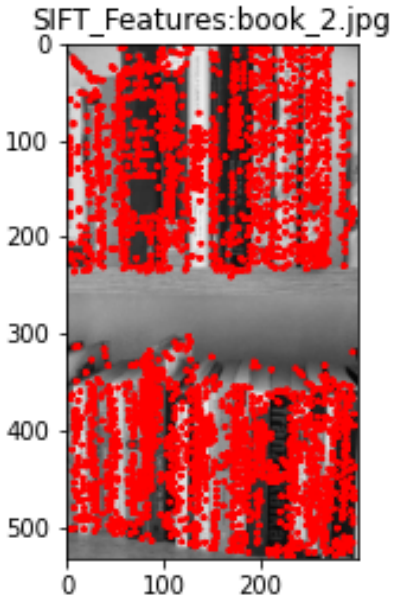


Figure 35: SIFT Features : book_2.jpg

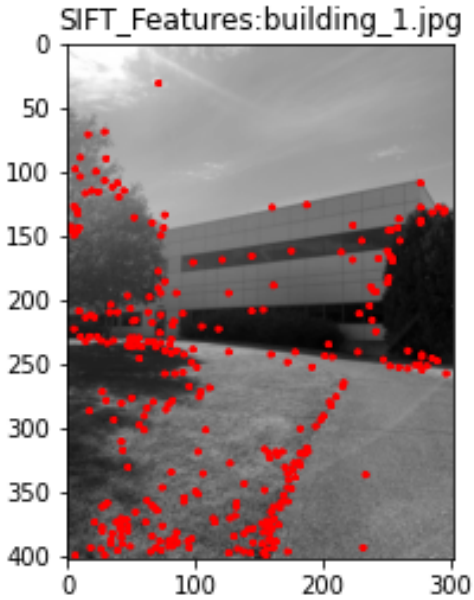


Figure 36: SIFT Features : building_1.jpg

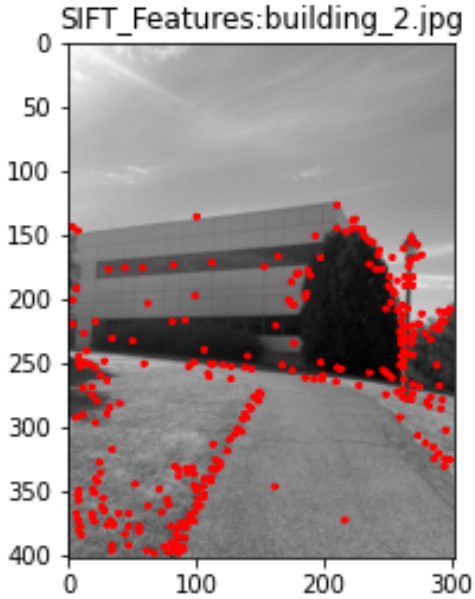


Figure 37: SIFT Features : building_2.jpg

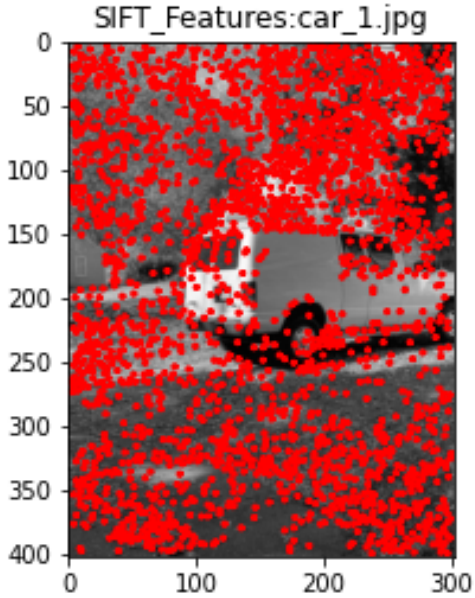


Figure 38: SIFT Features : car_1.jpg

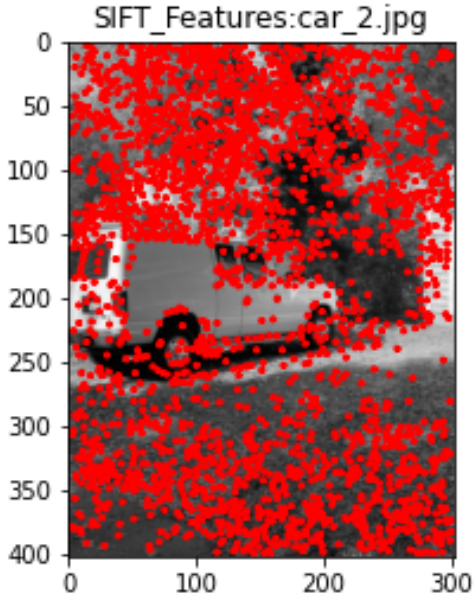


Figure 39: SIFT Features : car_2.jpg

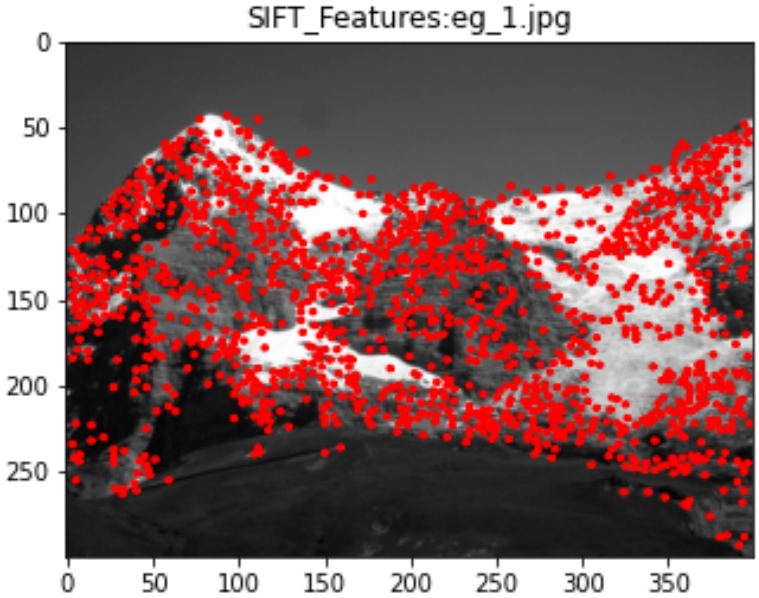


Figure 40: SIFT Features : eg_1.jpg

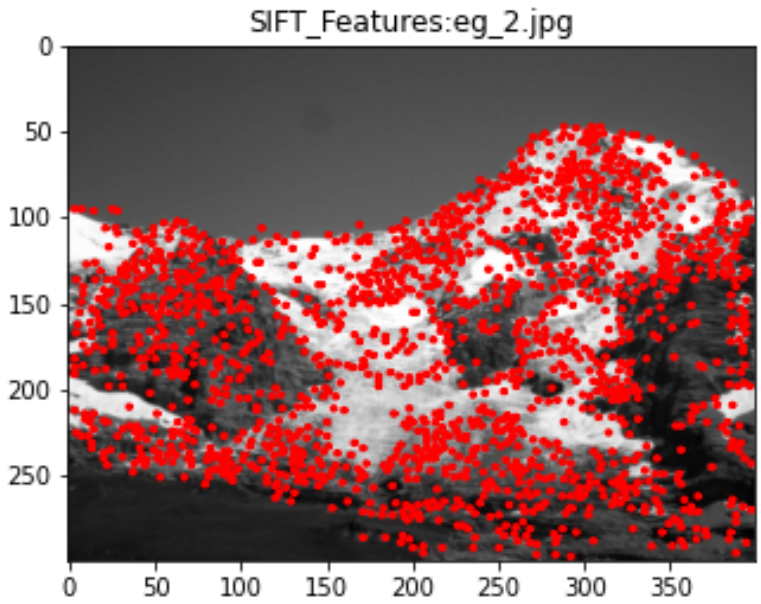


Figure 41: SIFT Features : eg_2.jpg

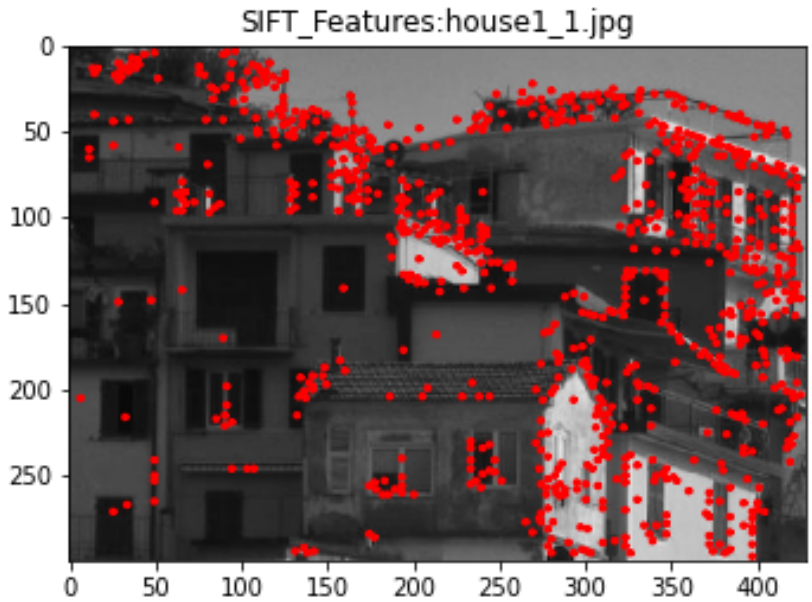


Figure 42: SIFT Features : house1_1.jpg

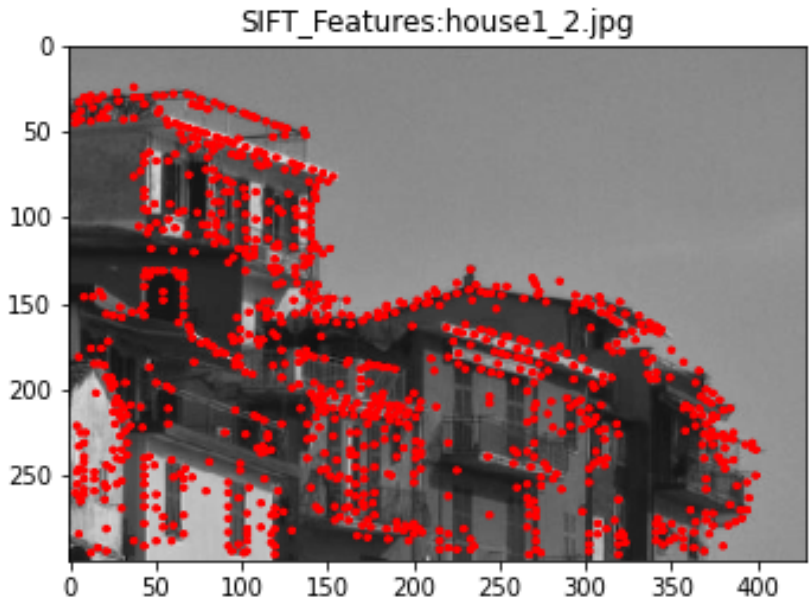


Figure 43: SIFT Features : house1_2.jpg

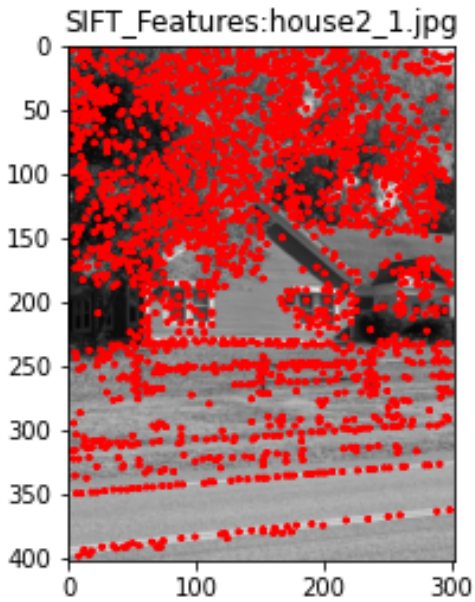


Figure 44: SIFT Features : house2_1.jpg

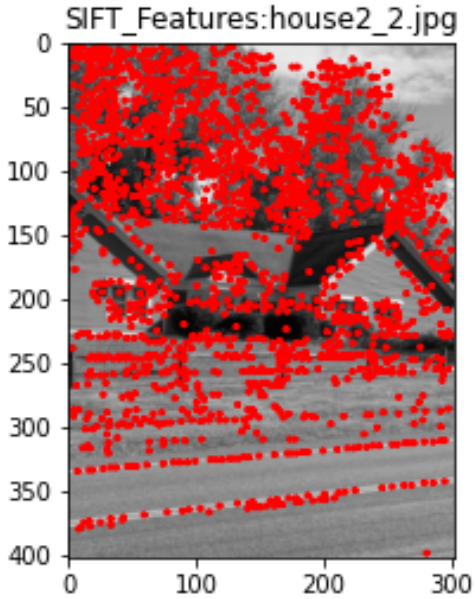


Figure 45: SIFT Features : house2_2.jpg

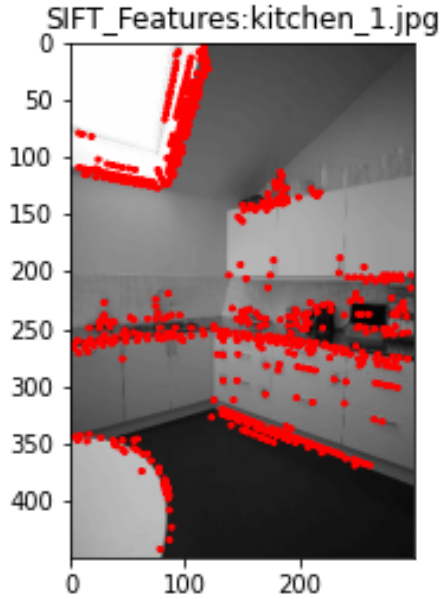


Figure 46: SIFT Features : kitchen_1.jpg

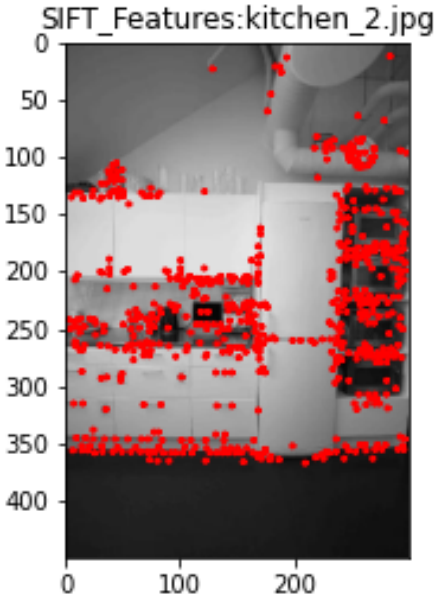


Figure 47: SIFT Features : kitchen_2.jpg

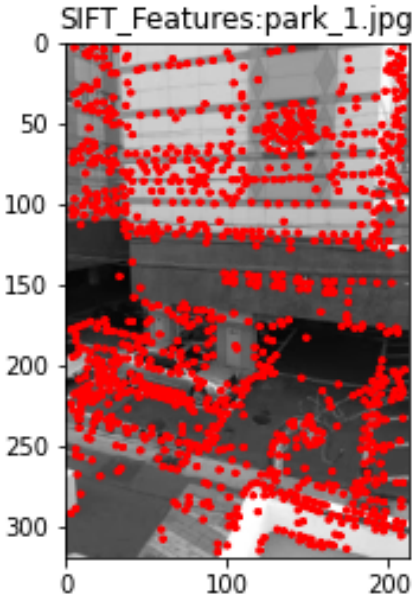


Figure 48: SIFT Features : park_1.jpg

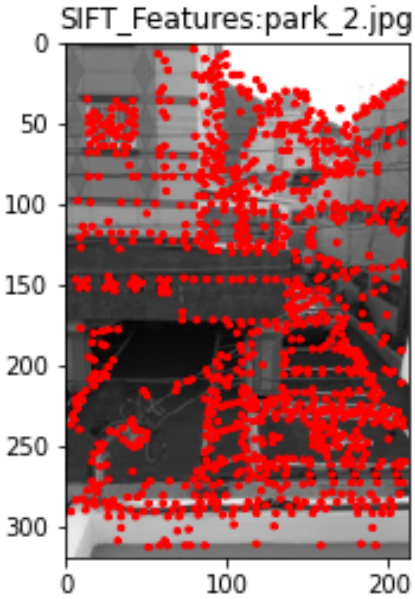


Figure 49: SIFT Features : park_2.jpg

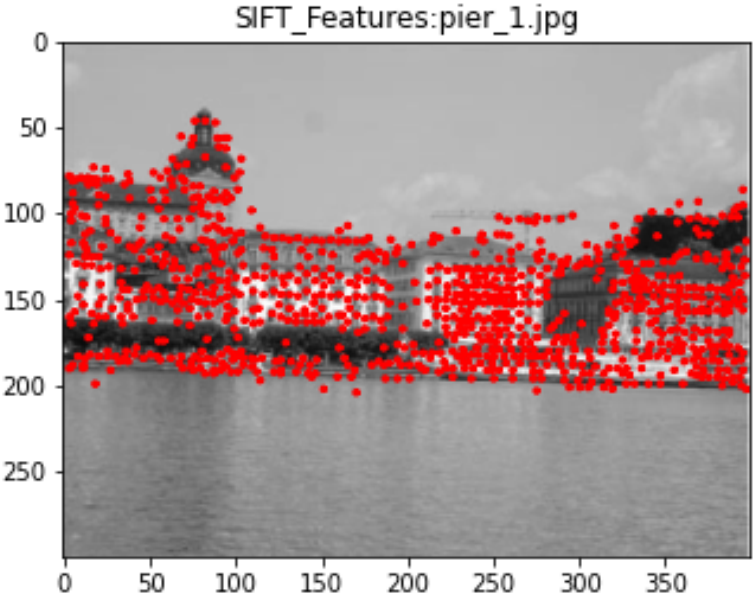


Figure 50: SIFT Features : pier_1.jpg

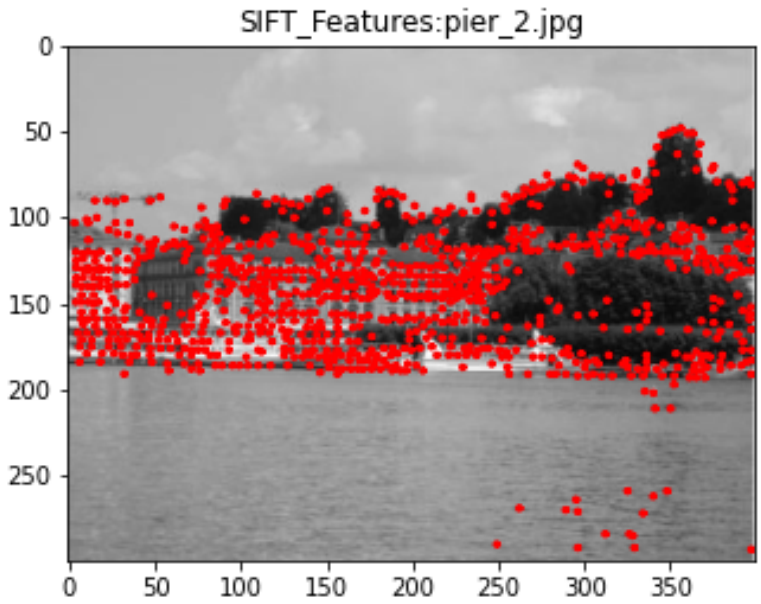


Figure 51: SIFT Features : pier_2.jpg

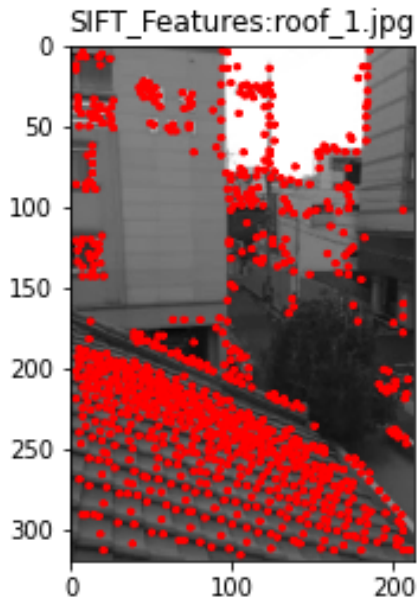


Figure 52: SIFT Features : roof_1.jpg

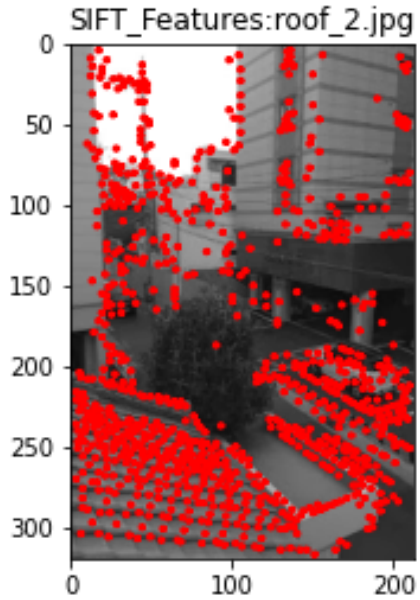


Figure 53: SIFT Features : roof_2.jpg

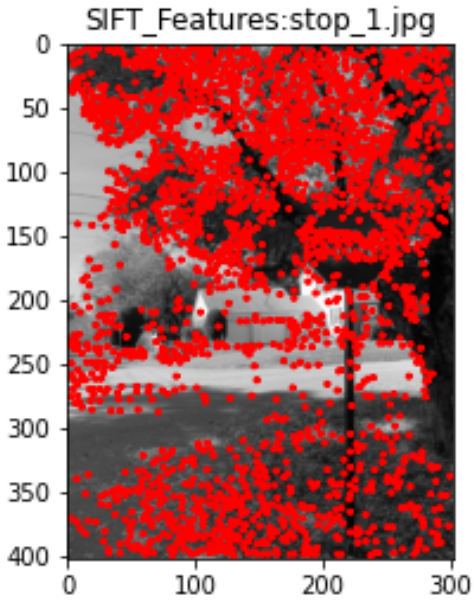


Figure 54: SIFT Features : stop_1.jpg

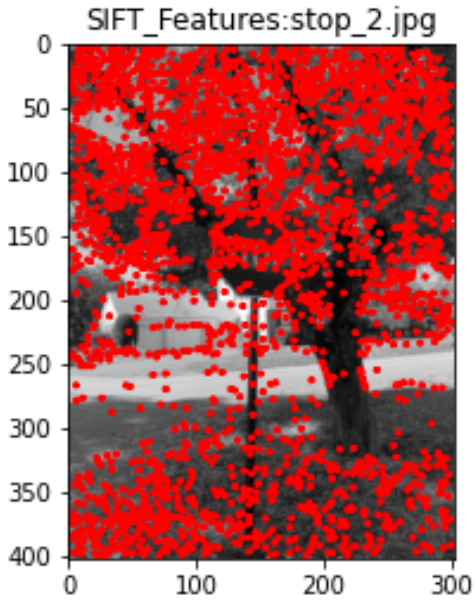


Figure 55: SIFT Features : stop_2.jpg

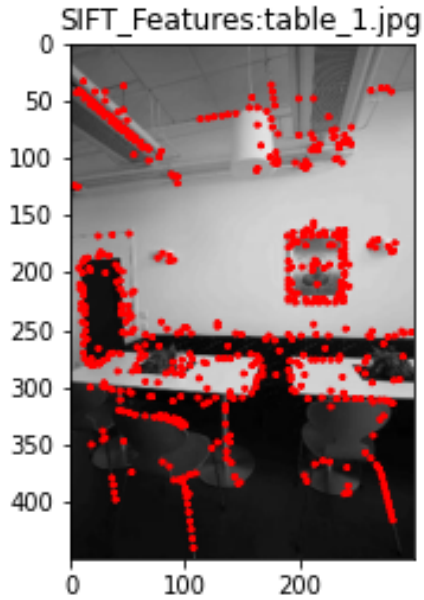


Figure 56: SIFT Features : table_1.jpg

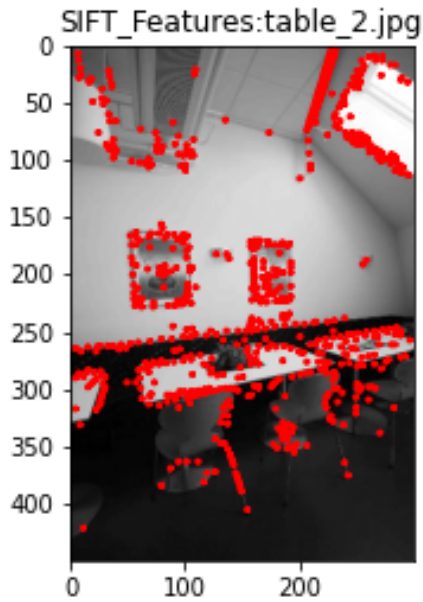


Figure 57: SIFT Features : table_2.jpg

4.3 Computing matches

Here, we compute matches between the two sets of SIFT features.
1. We find the best match of each feature f1 from image1 to f2 of image2 using the smallest sum-of-squared-differences.
2. If we find a match for f1(i) where $i \in$ [0,1,...N], from f2[0,1,...,M] we will say matches(i) is the closest feature in f2 to the ith feature f1(:,i).
3. All other entries will be matches i = -1, which indicates no matches.
4. We initialize matches() to all minus ones and then loop through all SIFT features of image2, for each feature of image1, to find the pairs where SSD is the least and store that f2 for i^{th} feature in f1.
5. There is a shortcoming of this method where we must deal with ambiguous putative matches. We can compare distance of NEAREST NEIGHBOR to that of SECOND NEAREST NEIGHBOR.
6. The ratio of closest distance to second-closest distance will be HIGH FOR FEATURES THAT ARE NOT DISTINCTIVE. Threshold of 0.8 provides a good separation.
7. We set matches(i)=0 if i^{th} feature f1(:,i) fails on the ratio test.
Implementation of *computeMatches.py*

```
import numpy as np
from scipy.spatial.distance import cdist

# This code is part of:
#
#   CMPSCI 670: Computer Vision, Fall 2019
#   University of Massachusetts, Amherst
#   Instructor: Subhransu Maji
#
def computeMatches(f1, f2):
    """ Match two sets of SIFT features f1 and f2 """
    N, d = np.shape(f1)
```

```
    M, d = np.shape(f2)

    matches = np.ones((N, )) * -1

    method = 0
    if method == 0:
        print("Computing matching using SSD")
        SSD = cdist(f1, f2, 'seuclidean')
        #For each feature from image 1, find its nearest match image 2 and store indices of lowest/minima SSD feature
        matches = np.argmin(SSD, axis=1)

    elif method == 1:
        print("Computing matching using Ratio:")
        distanceSSD = cdist(f1, f2, 'seuclidean')
        #For each Descriptor in the First Image, Select its Match to Second Image
        bestMatch = list(np.argmin( distanceSSD, axis=1))
        for x in range(N):
        #For each Descriptor in the First Image, Select its Match to Second Imag
            bestId = bestMatch[x]
            bestValue = distanceSSD[x, bestId] #Most likely nearest neighbor match

            #Next most likely match after best one
            secondBestMatch = list(distanceSSD[x,:])
            secondBestMatch[bestId] = float('inf')
            secondBestValue = min(secondBestMatch)
           # After determining best and second best match, compute ratio
            if bestId != -1 and bestValue/secondBestValue <= 0.8:
                matches[x] = bestId
            else:
                matches[x] = -1
    return matches
```

Intermediate Results:
Here we show the output of showMatches(im1, im2, c1, c2, matches)

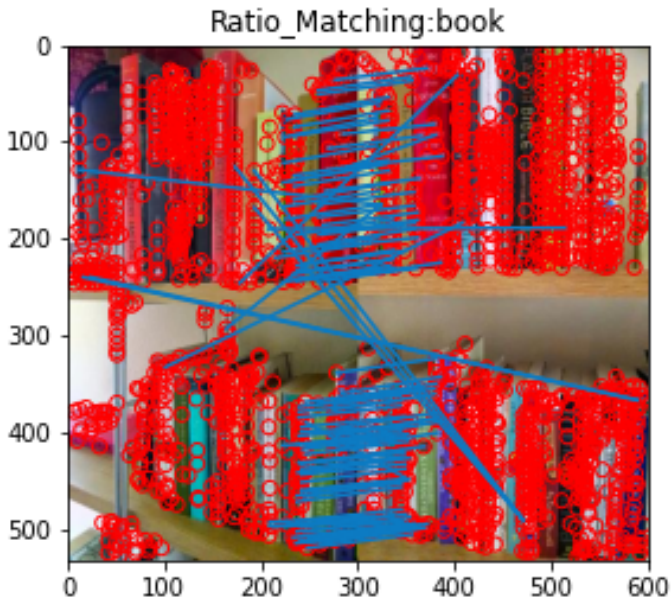


Figure 58: Ratio_Matching : book

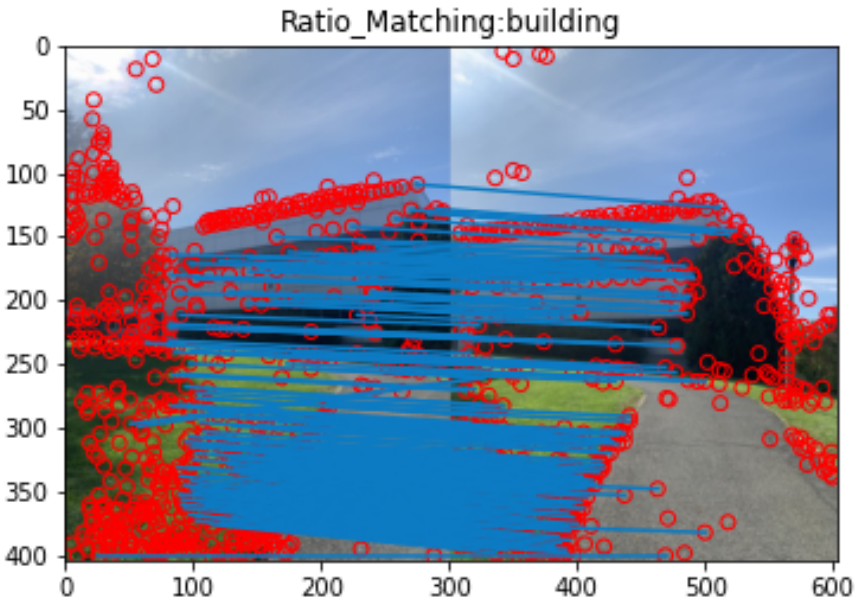


Figure 59: Ratio_Matching : building

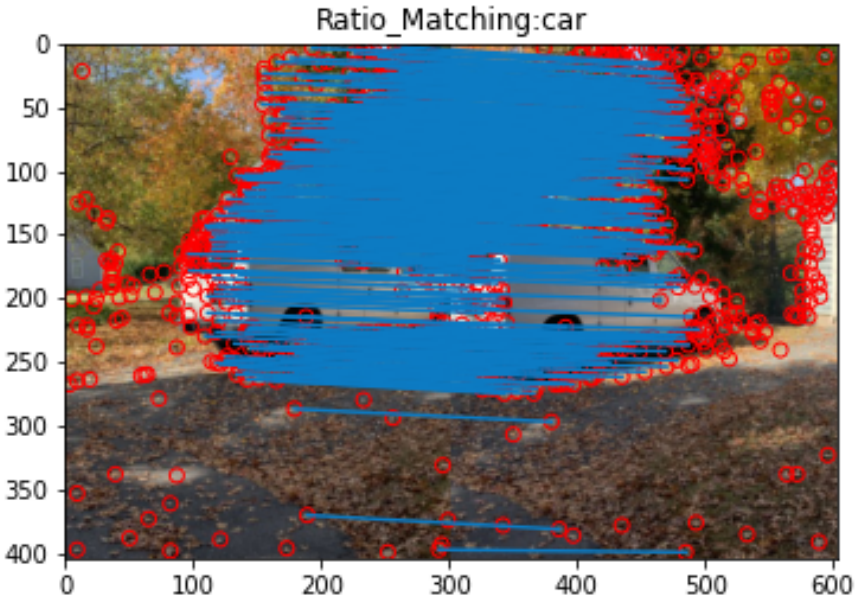


Figure 60: Ratio_Matching : car

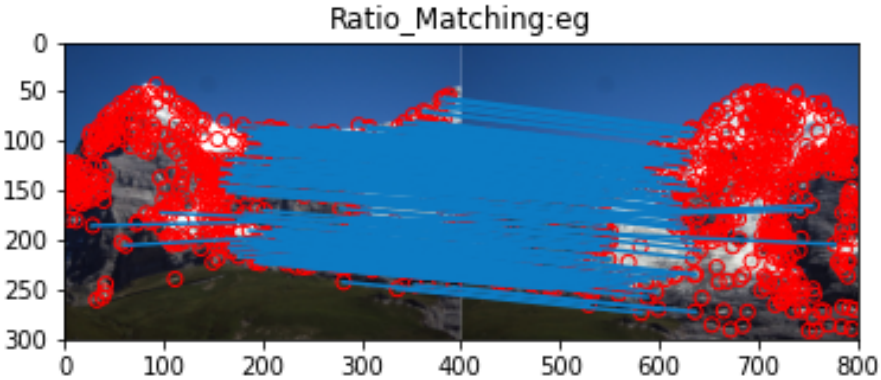


Figure 61: Ratio_Matching : eg

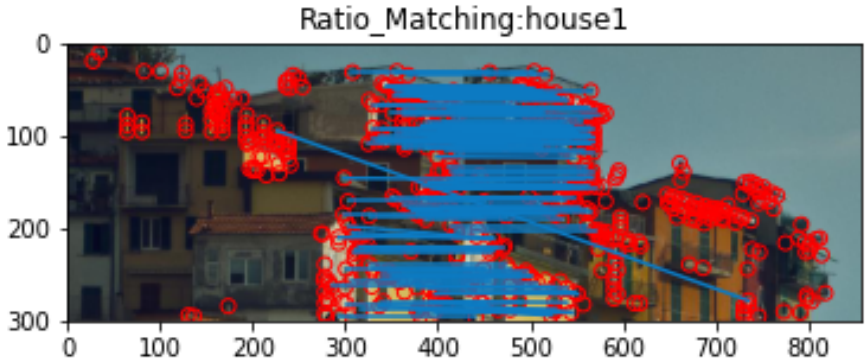


Figure 62: Ratio_Matching : house1

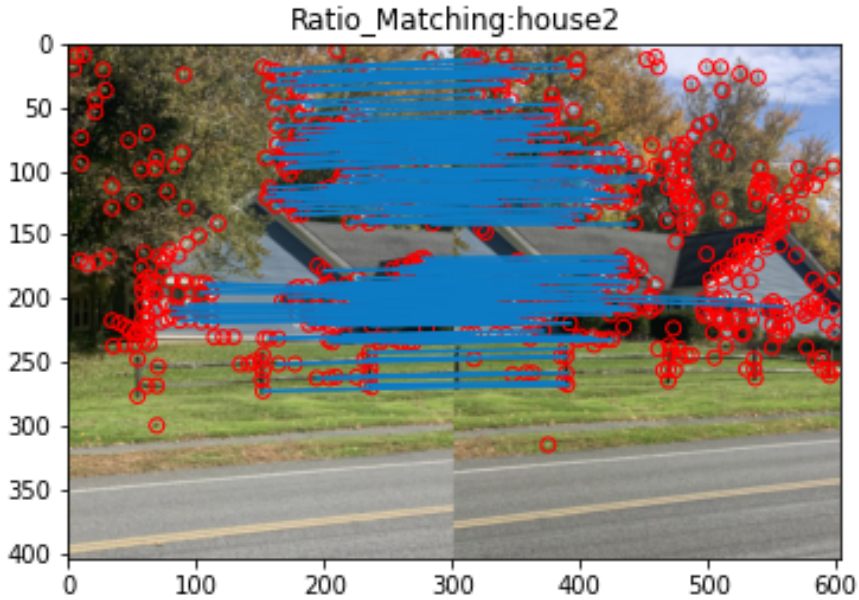


Figure 63: Ratio_Matching : house2

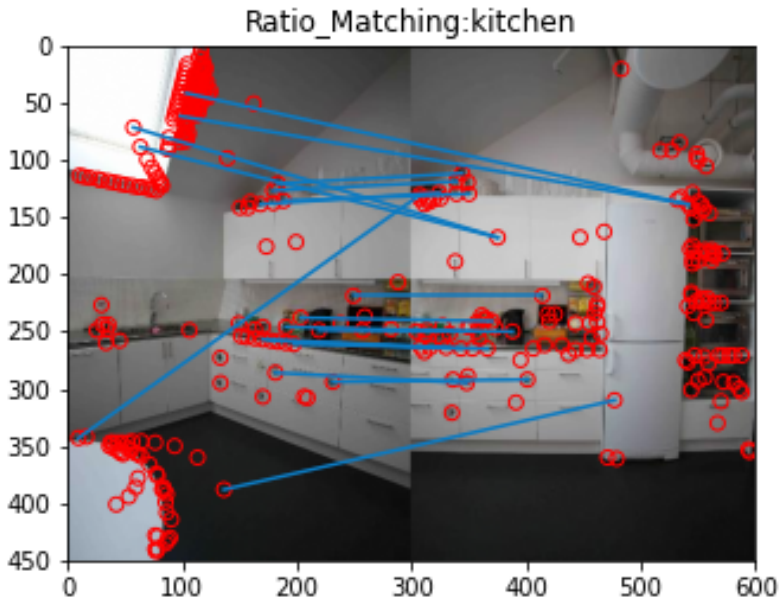


Figure 64: Ratio_Matching : kitchen

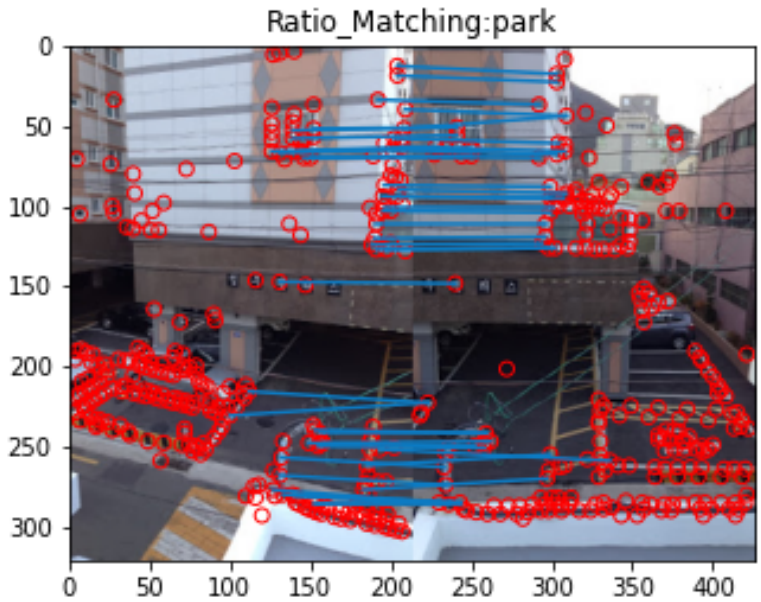


Figure 65: Ratio_Matching : park

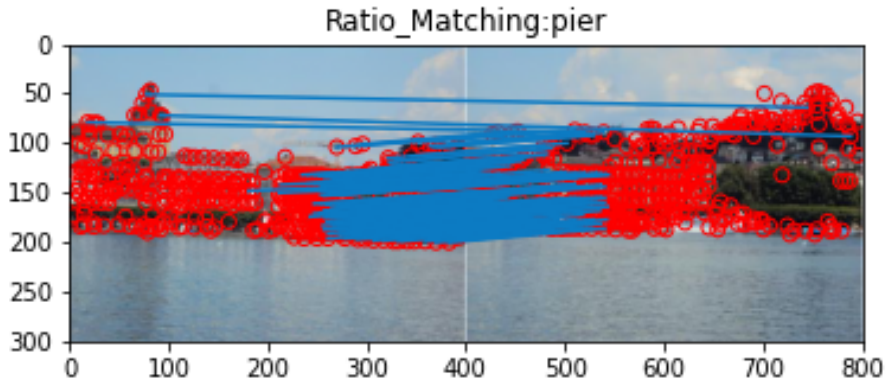


Figure 66: Ratio_Matching : pier

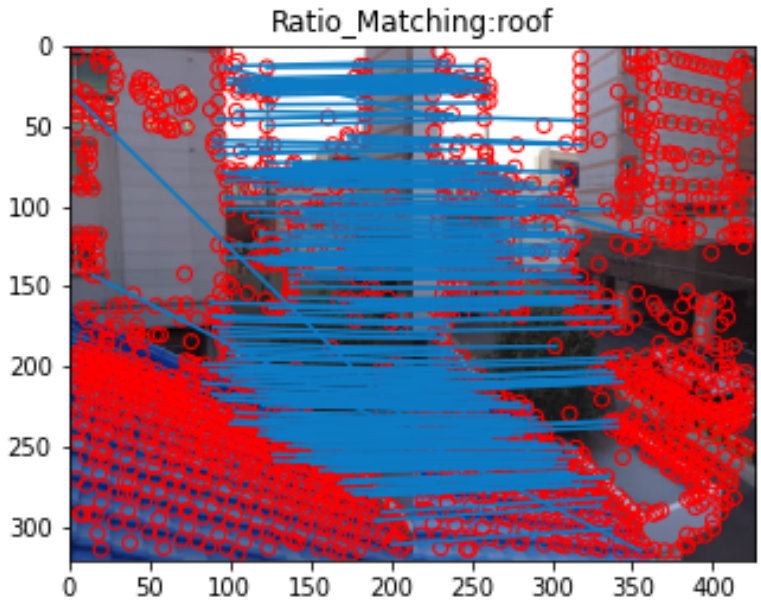


Figure 67: Ratio_Matching : roof

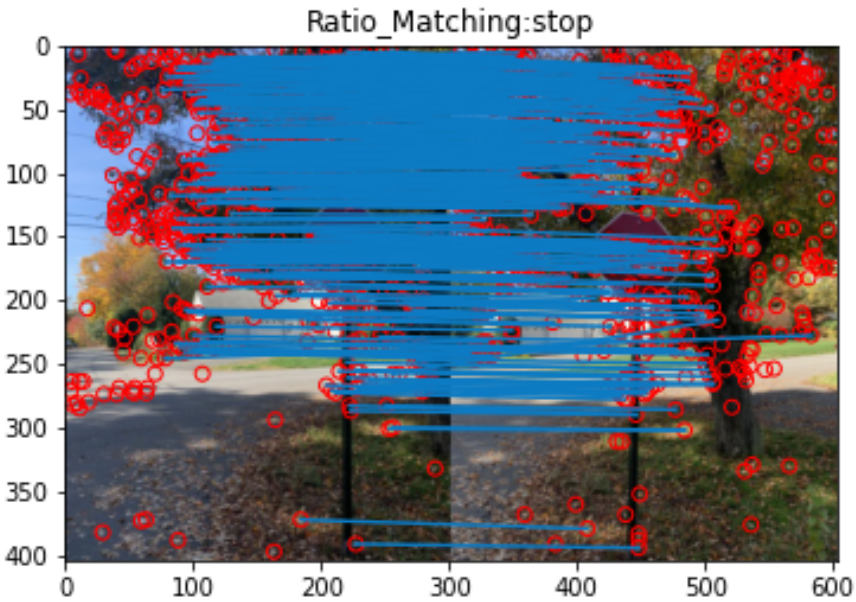


Figure 68: Ratio_Matching : stop

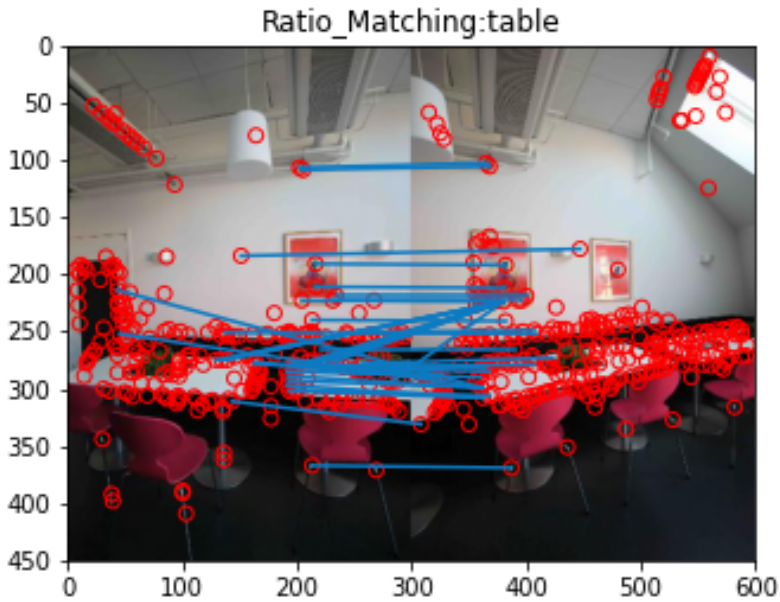


Figure 69: Ratio_Matching : table

4.4 Estimating affine transformation using RANSAC

:
Using RANSAC, we will estimate the AFFINE TRANSLATION that agrees with most matches (inliers).
An affine transformation is any transformation that preserves collinearity (i.e., all points

lying on a line initially still lie on a line after transformation) and ratios of distances (e.g., the midpoint of a line segment remains the midpoint after transformation).

We have been given blobs for image1 and image2 the matches of SIFT features between them. So first task will be to find all the non-empty matches from the matches list, because some of them are assigned to 0, which are of no use in this context. We also store the original indices of the feature comparison entries. We arrange the base and transform points according to the matrix forms such that

AFFINE = $\frac{BASEPOINTS}{TRANSFORMEDPOINTS}$.

RANSAC pseudocode:
1. Randomly select minimal subset of points.
2. Compute Affine Transformation for the base and transformed point matrices.
3. Reshape the Affine Transform and the Base Point matrices so they can be multiplied to get the Transformed Point matrix HYPOTHESIZING THE MODEL.
4. Calculate the error between actual transformed points and the ones which we computed by getting an Affine Transform COMPUTING THE ERROR FUNCTION.
5. If the inlier count is less than the 1 threshold size; it counts as a valid model.
6. And if the inlier count is more than the previous best, the current Affne Transform is the best now. Store the best transform, the SSD error and the input and transformed points that produce that transform.
7. Iterate this procedure arbitrary times.
I have used six pairs of points to estimate the this *Intermediate Results:*
Here we show the output of showMatches(im1, im2, c1, c2, matches) for RANSAC
Implementation of *ransac.py*

```
import numpy as np
import cv2
import time
import random

# This code is part of:
#
#   CMPSCI 670: Computer Vision, Fall 2019
#   University of Massachusetts, Amherst
#   Instructor: Subhransu Maji
#


def calaculateError(Expected, Actual):
    xE, yE = Expected[0], Expected[1]
    xT, yT = Actual[0] , Actual[1]
    error = np.sqrt((xE-xT)**2 + (yE-yT)**2)
    return error

def ransac(matches, blobs1, blobs2):
    iterations = 1000
    N = len(matches)

    #Get all the non-empty feature matches
    features = list(np.where(matches!=-1)[0])

```

```
    #Reset max inliers counter
    maxInliers = 0
    bestAffineT = None
    inlinePts = []

    threshold = 1

    #Store all features and locations in inputs for 1st image and in bases for 2nd image
    for iter in range(iterations):
        bases = random.sample(features, 6)
        inputs = []

        for id in bases:
            inputs.append(int(matches[id]))

        coordX = []
        coordY = []
        coordT = []

        for i in range(6):
            coordX.append(blobs1[bases[i]][0])
            coordY.append(blobs1[bases[i]][1])
            coordT.append(blobs2[inputs[i]][0])
            coordT.append(blobs2[inputs[i]][1])

        #Affine Transform for the current iteration
        B = np.array(coordT).T

        A1 = np.array([[coordX[0], coordY[0], 0,0,1,0],[0,0,coordX[0], coordY[0],0,1]])
        A2 = np.array([[coordX[1], coordY[1], 0,0,1,0],[0,0,coordX[1], coordY[1],0,1]])
        A3 = np.array([[coordX[2], coordY[2], 0,0,1,0],[0,0,coordX[2], coordY[2],0,1]])
        A4 = np.array([[coordX[3], coordY[3], 0,0,1,0],[0,0,coordX[3], coordY[3],0,1]])
        A5 = np.array([[coordX[4], coordY[4], 0,0,1,0],[0,0,coordX[4], coordY[4],0,1]])
        A6 = np.array([[coordX[5], coordY[5], 0,0,1,0],[0,0,coordX[5], coordY[5],0,1]])
        A = np.concatenate((A1, A2, A3, A4, A5, A6))

        leastSquares = np.linalg.lstsq(A, B, rcond=None)

        lstqSoln = leastSquares[0]
        residual = leastSquares[1]
        rank = leastSquares[2]
        SingularCoefficent = leastSquares[3]

        M = np.array([[lstqSoln[0], lstqSoln[1]], [lstqSoln[2], lstqSoln[3]]])
        T = np.array([[lstqSoln[4]], lstqSoln[5]])

        #Find and add inliers
        numInline = 0
        inline = []

        for f in features:
```

```
            x, y = blobs1[f][0], blobs1[f][1]
            xT, yT = blobs2[int(matches[f])][0], blobs2[int(matches[f])][1]
            xE = lstqSoln[0] * x + lstqSoln[1] * y + lstqSoln[4]
            yE = lstqSoln[2] * x + lstqSoln[3] * y + lstqSoln[5]
            error = calaculateError([xE, yE],[xT, yT])

            if error < threshold:
                numInline+=1
                inline.append(f)

        #Find the best model by inlier count and error values
        if numInline > maxInliers:
            maxInliers = numInline
            inlinePts = inline
            bestAffineT = np.array([[lstqSoln[0],lstqSoln[1],lstqSoln[4]],[lstqSoln[2],lstqSoln[3],lstqSoln[5]]])

    print("Maximum Inliers: ", maxInliers)
    print( "Msize: " , len(features))
    print("Affine Transformation: ")
    print(bestAffineT)
    return inlinePts, bestAffineT
```

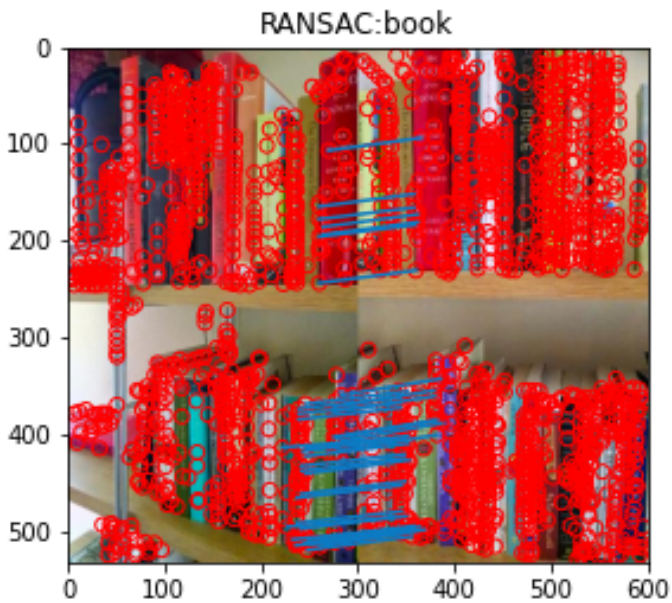


Figure 70: RANSAC: book

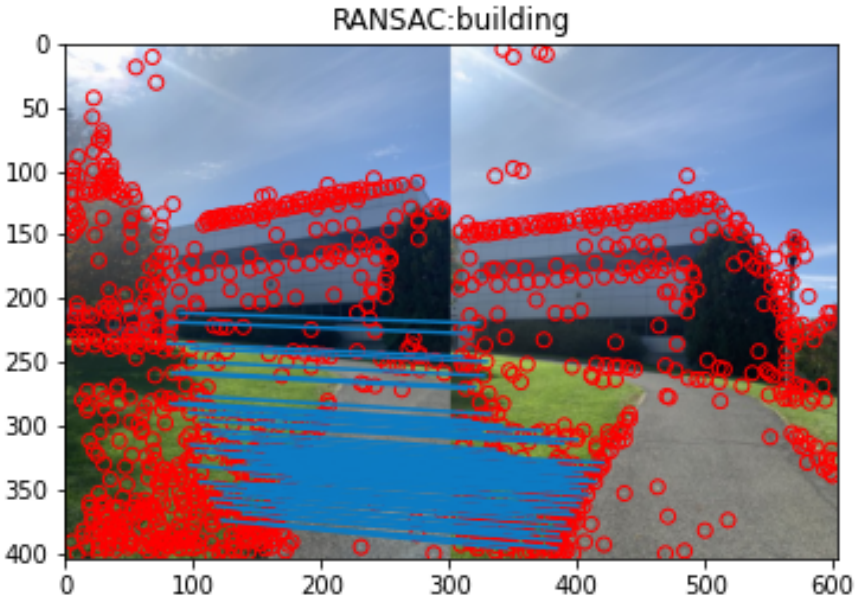


Figure 71: RANSAC: building

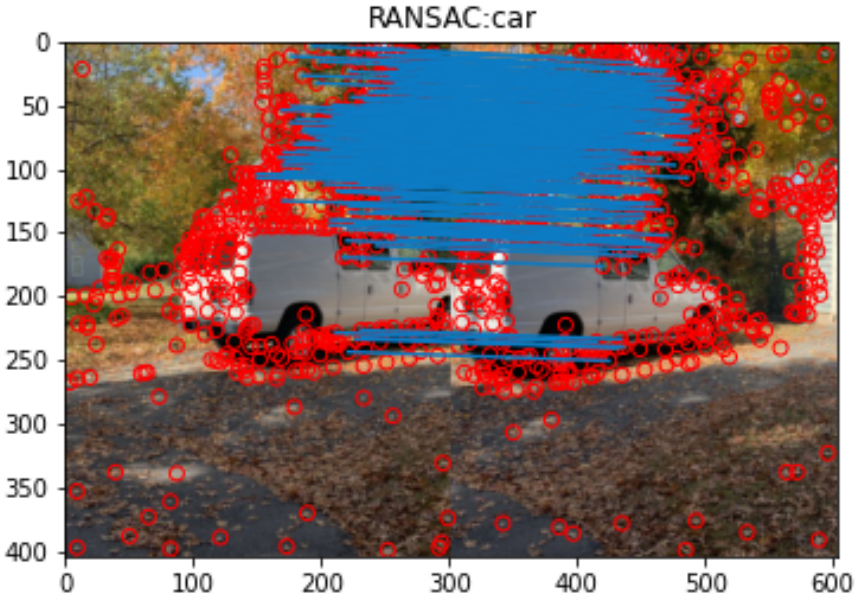


Figure 72: RANSAC: car

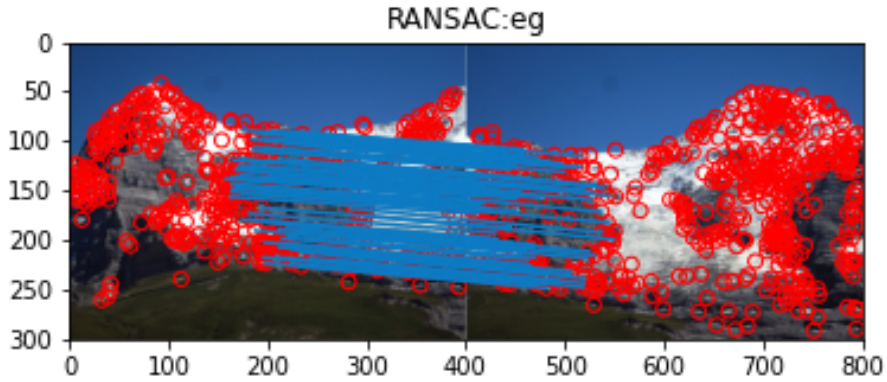


Figure 73: RANSAC: eg

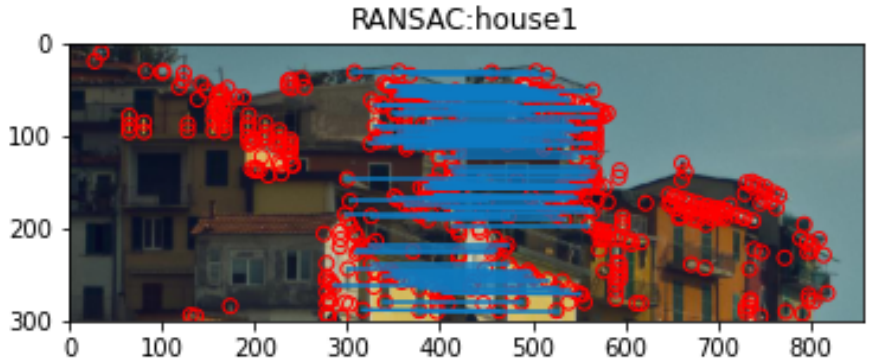


Figure 74: RANSAC: house1

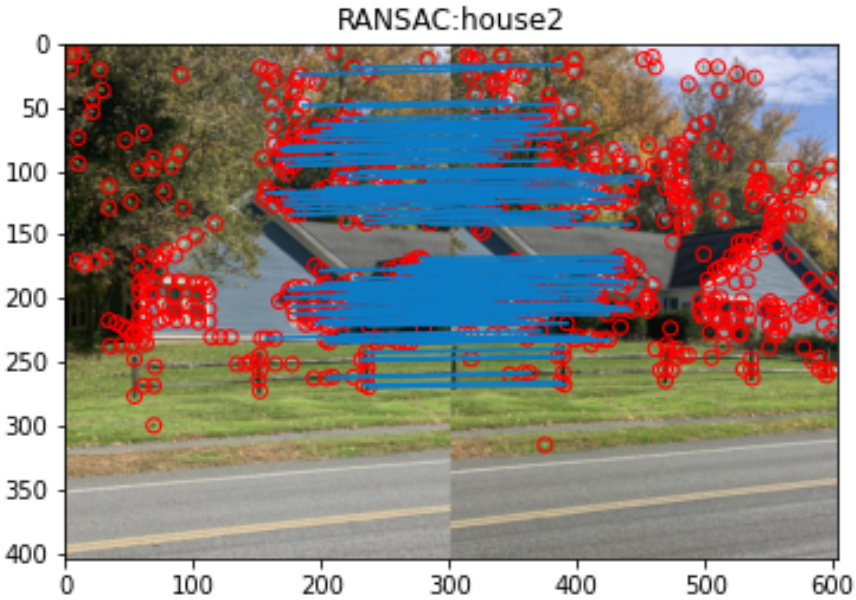


Figure 75: RANSAC: house2

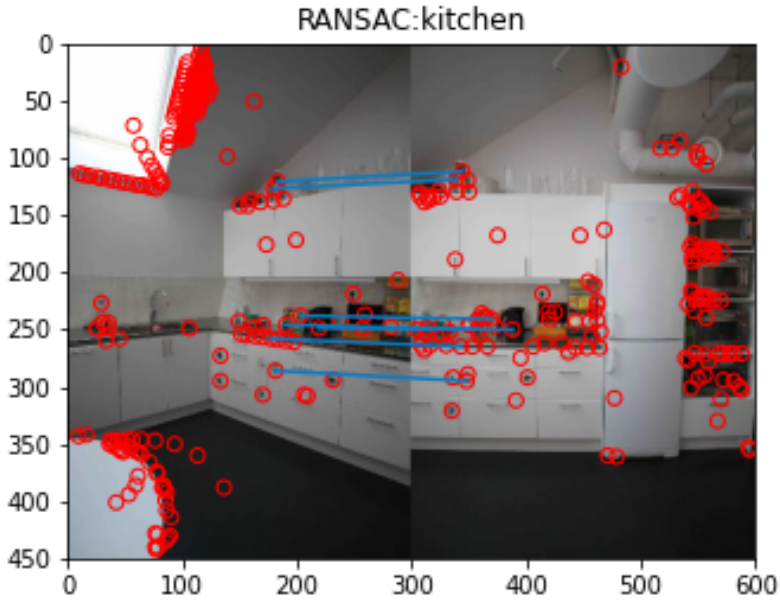


Figure 76: RANSAC: kitchen

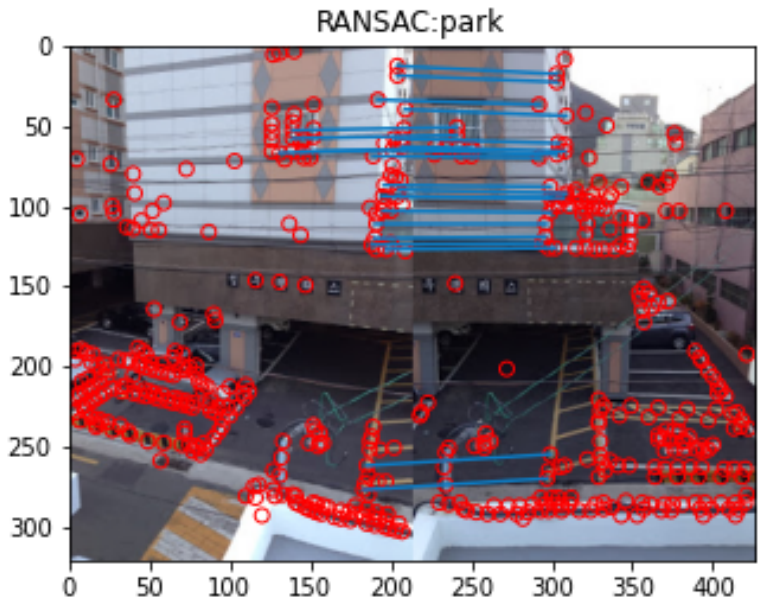


Figure 77: RANSAC: park

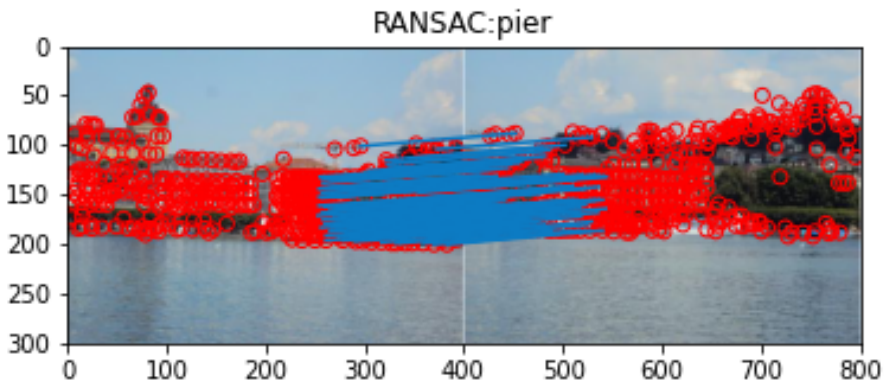


Figure 78: RANSAC: pier

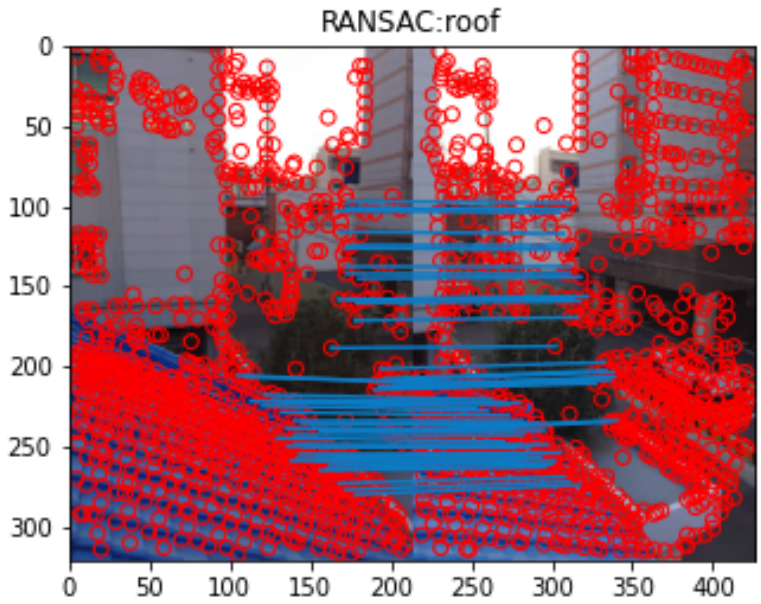


Figure 79: RANSAC: roof

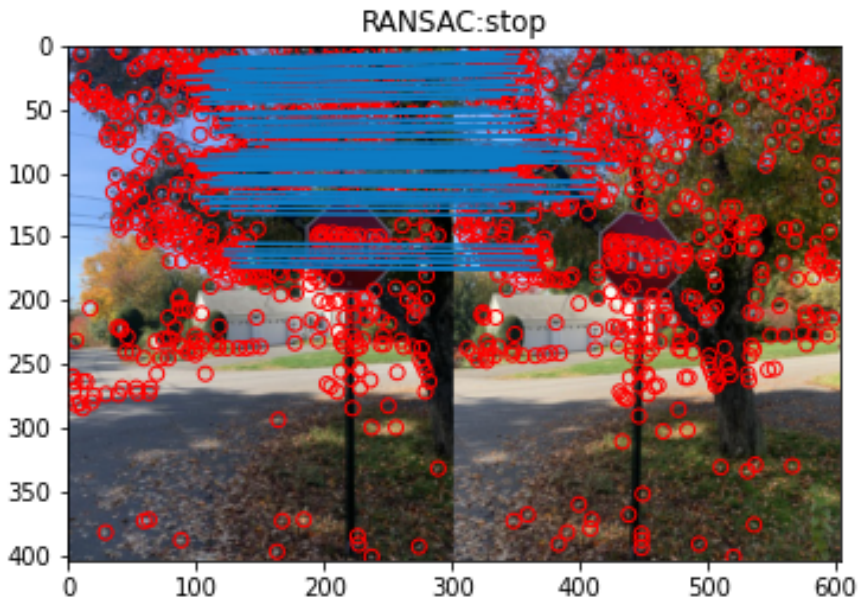


Figure 80: RANSAC: stop

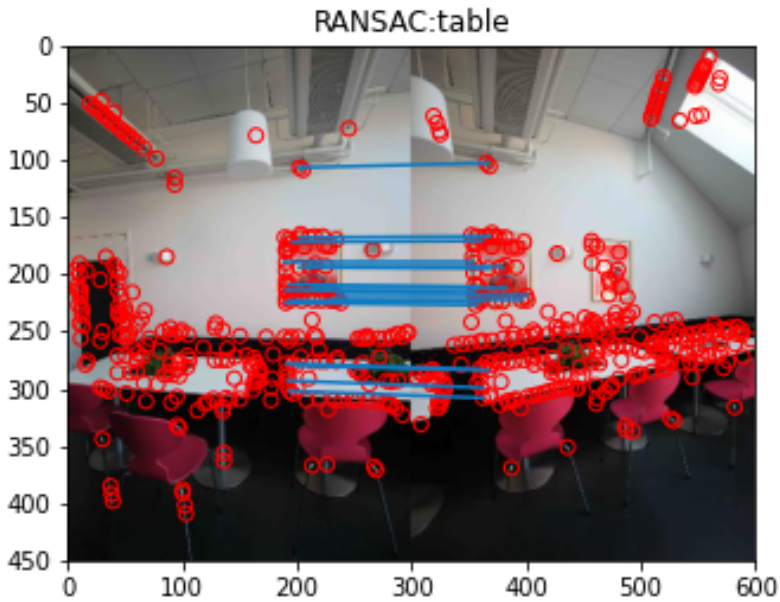


Figure 81: RANSAC: table

Estimated Maximum Inliers, MSize and Affine transformation for given test images are as follows:
1. 'stop'
Maximum Inliers: 75
Msize: 325
Affine Transformation:
$[[1.08502999e+00 \quad -2.42057204e-02 \quad -7.84957685e+01]$
$[9.67183257e-02 \quad 1.03451525e+00 \quad -1.62962322e+01]]$

2. 'car'
Maximum Inliers: 147
Msize: 322
Affine Transformation:
$[[9.75907895e-01 \quad 9.63936928e-03 \quad -9.48601035e+01]$
$[4.67499375e-02 \quad 9.81731710e-01 \quad 1.51218795e-01]]$

3. 'building'
Maximum Inliers: 10
Msize: 16
Affine Transformation:
$[[1.05319156e+00 \quad -5.44017900e-02 \quad -6.20549839e+01]$
$[-1.23040204e-03 \quad 1.03442818e+00 \quad 2.69313096e+00]]$

4. 'book'
Maximum Inliers: 35
Msize: 98
Affine Transformation:

$[[9.58470121e-01 \quad 2.89691851e-02 \quad -1.95928465e+02]$
$[-1.34075179e-01 \quad 9.88253567e-01 \quad 2.52309368e+01]]$

5.'house1'
Maximum Inliers: 99
Msize: 134
Affine Transformation:
$[[1.00104915e+00 \quad -2.43559316e-03 \quad -2.80596911e+02]$
$[-9.48173958e-04 \quad 9.99446206e-01 \quad 7.90062481e-01]]$

6. 'house2'
Maximum Inliers: 145
Msize: 211
Affine Transformation:
$[[1.00292600e+00 \quad -3.18514800e-02 \quad -1.40578380e+02]$
$[3.42841395e-02 \quad 1.00161781e+00 \quad -1.06336511e+01]]$

7. 'kitchen'
Maximum Inliers: 7
Msize: 19
Affine Transformation:
$[[1.08300776e+00 \quad 4.15450154e-02 \quad -1.59546706e+02]$
$[-1.05407388e-01 \quad 1.10873821e+00 \quad -2.27539246e+00]]$

8. 'park'
Maximum Inliers: 23
Msize: 47
Affine Transformation:
$[[9.79106162e-01 \quad 5.60250656e-02 \quad -1.10369994e+02]$
$[7.79082594e-02 \quad 9.61264955e-01 \quad -1.07670804e+01]]$

9. 'pier'
Maximum Inliers: 114
Msize: 168
Affine Transformation:
$[[9.95744606e-01 \quad 1.69017118e-02 \quad -2.44377061e+02]$
$[-7.94633918e-03 \quad 9.97793058e-01 \quad -9.76763788e+00]]$

10. 'roof'
Maximum Inliers: 21
Msize: 39
Affine Transformation:
$[[1.07638635e+00 \quad 3.31842404e-02 \quad -8.83539714e+01]$
$[4.96547713e-02 \quad 1.03970850e+00 \quad -9.91721303e+00]]$

11. 'table'
Maximum Inliers: 11
Msize: 39

Affine Transformation:
$[[9.57472461e-05 \quad 1.53460378e-01 \quad 3.77764937e+01]$
$[-1.01607611e-12 \quad 7.50518225e-13 \quad 2.24000000e+02]]$

12. 'eg'
Maximum Inliers: 91
Msize: 190
Affine Transformation:
$[[1.02534704e+00 \quad -4.70246855e-02 \quad -1.48634911e+02]$
$[4.66248629e-02 \quad 1.01825011e+00 \quad 7.24751619e+00]]$

Implementation of *evalStitching.py*

```
import numpy as np
import matplotlib.pyplot as plt
import os
from utils import imread
from utils import showMatches
from detectBlobs2 import detectBlobs
from drawBlobs2 import drawBlobs
from computeSift import compute_sift
from computeMatches import computeMatches
from ransac import ransac
from mergeImages import mergeImages
import random
import warnings
warnings.filterwarnings("ignore")

#Image directory
dataDir = os.path.join('..', 'data', 'stitching')

#Read input images
testExamples = ['stop', 'car', 'building', 'book', 'house1', 'house2', 'kitchen'
    , 'park', 'pier', 'roof', 'table','eg']

for exampleIndex in range(len(testExamples)):
    random.seed(1)
    imageName1 = '{}_1.jpg'.format(testExamples[exampleIndex])
    imageName2 = '{}_2.jpg'.format(testExamples[exampleIndex])

    im1 = imread(os.path.join(dataDir, imageName1))
    im2 = imread(os.path.join(dataDir, imageName2))

    #Detect keypoints
    blobs1 = detectBlobs(im1)
    blobs2 = detectBlobs(im2)

    #Compute SIFT features
    sift1 = compute_sift(im1, blobs1[:, 0:4])
    sift2 = compute_sift(im2, blobs2[:, 0:4])

    #Display SIFT features
    t0 = "SIFT_Features:"+imageName1
    t1 = "SIFT_Features:"+imageName2
    drawBlobs(im1, blobs1, t0)
    drawBlobs(im2, blobs2, t1)
```

```
    matches = computeMatches(sift1, sift2)
    t2 = "Ratio_Matching:"+str(testExamples[exampleIndex])
    showMatches(im1, im2, blobs1, blobs2, matches, t2)

    #Ransac to find correct matches and compute transformation
    inliers, transf = ransac(matches, blobs1, blobs2)

    goodMatches = np.full_like(matches, -1)
    goodMatches[inliers] = matches[inliers]

    t3 = "RANSAC:"+str(testExamples[exampleIndex])
    showMatches(im1, im2, blobs1, blobs2, goodMatches, title=t3)

    #Merge two images and display the output
    stitchIm = mergeImages(im2, im1, transf)
    plt.figure()
    plt.imshow(stitchIm)
    plt.title('Stitched Image: {}'.format(testExamples[exampleIndex]))
    plt.savefig('StitchedImage:{}.png'.format(testExamples[exampleIndex]))
    #files.download('StitchedImage:{}.png'.format(testExamples[exampleIndex]))
    plt.show()
```

Here we show the output of panoramic stitched images:

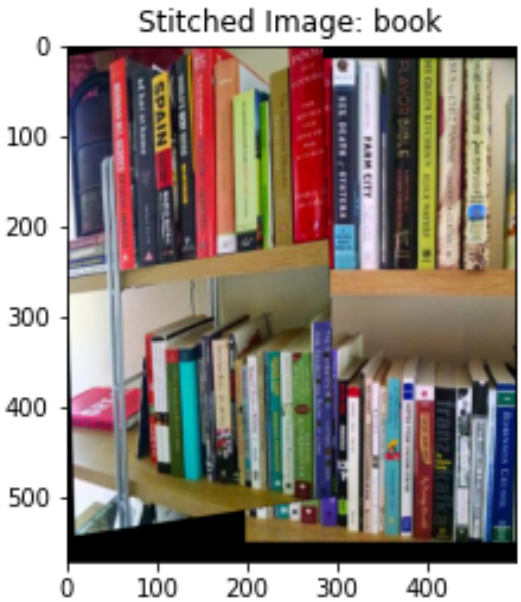


Figure 82: Stitched Image: book

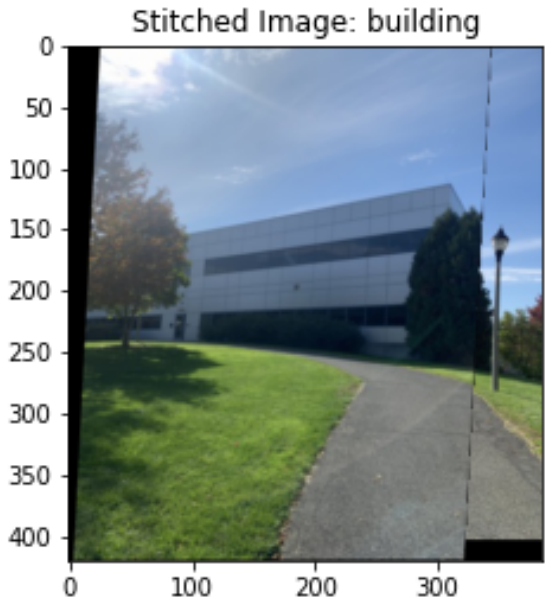


Figure 83: Stitched Image: building

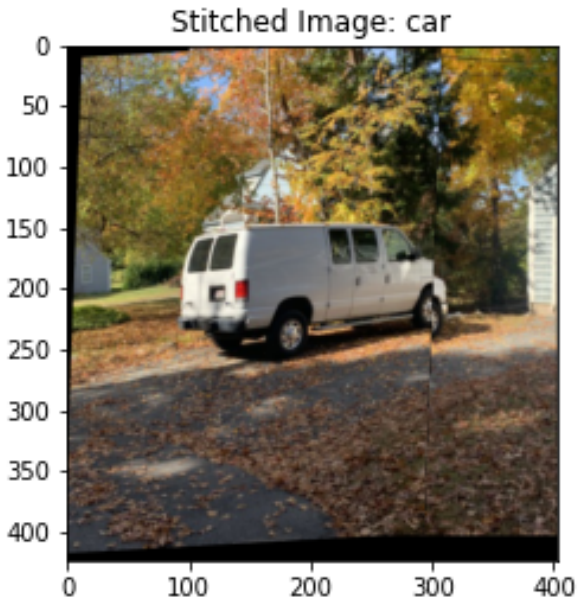


Figure 84: Stitched Image: car

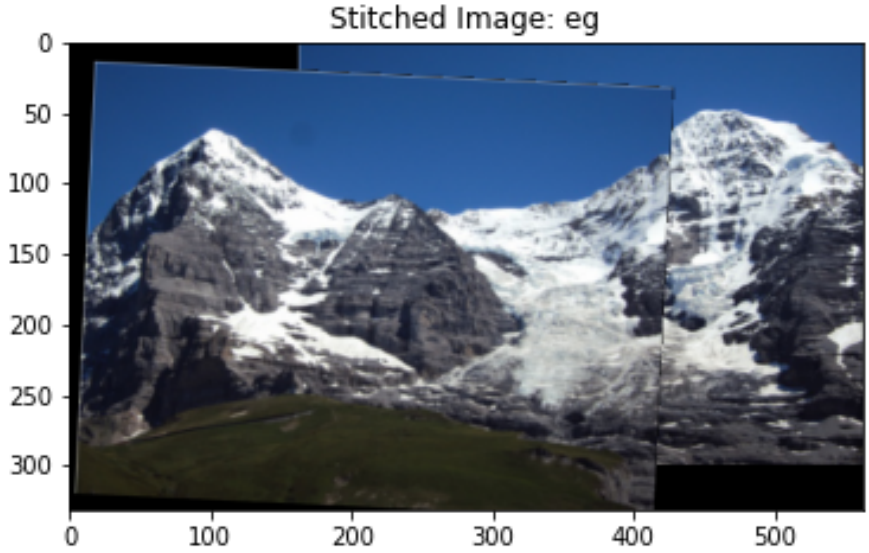


Figure 85: Stitched Image: eg

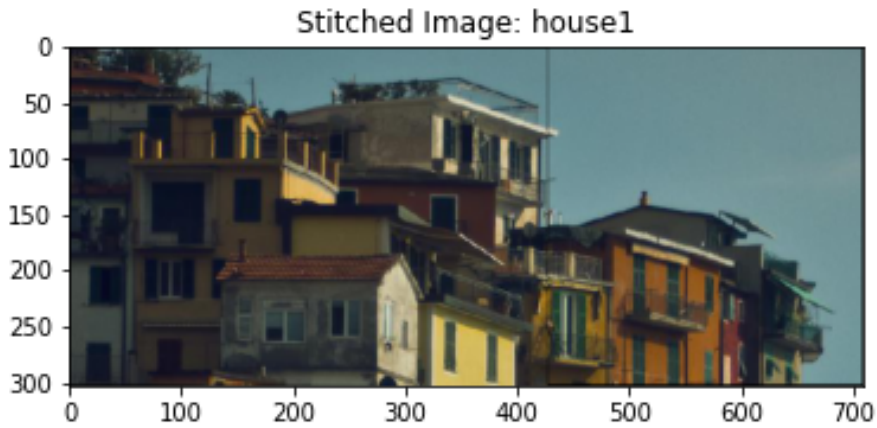


Figure 86: Stitched Image: house1

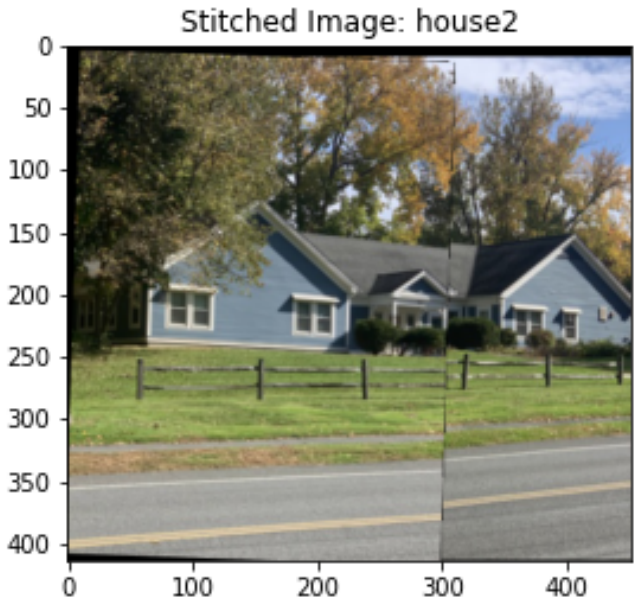


Figure 87: Stitched Image: house2

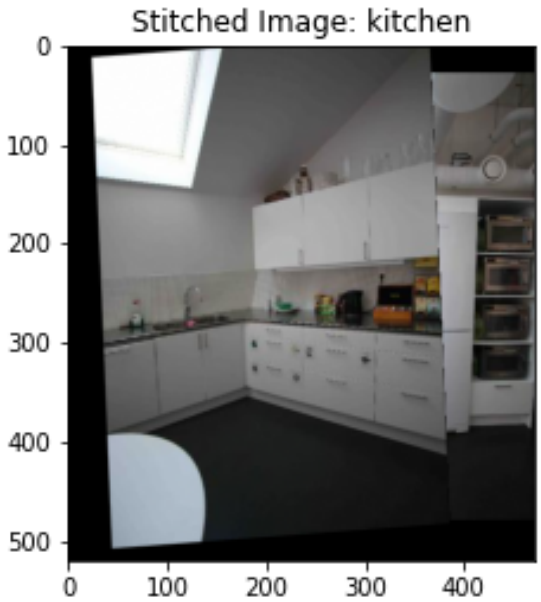


Figure 88: Stitched Image: kitchen

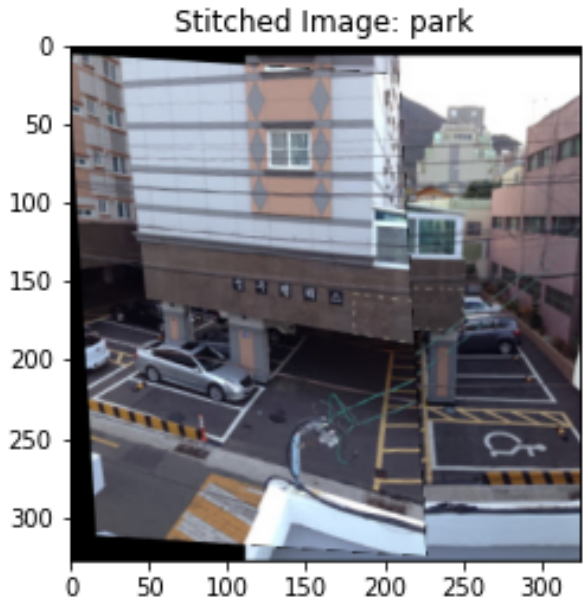


Figure 89: Stitched Image: park

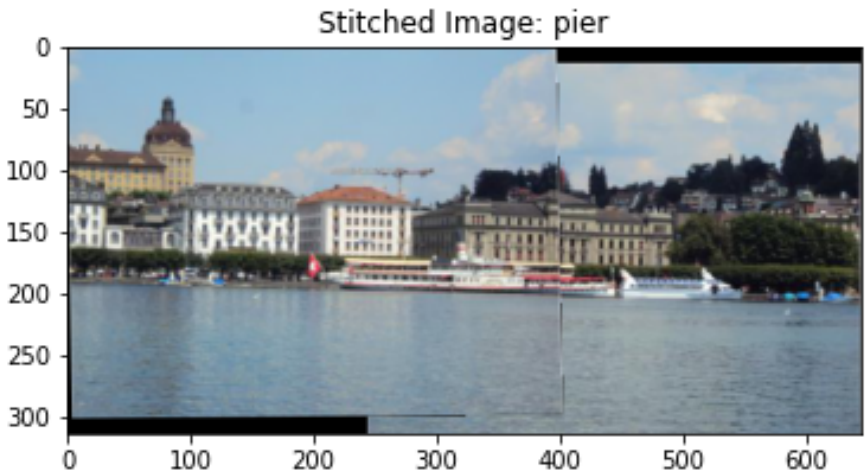


Figure 90: Stitched Image: pier

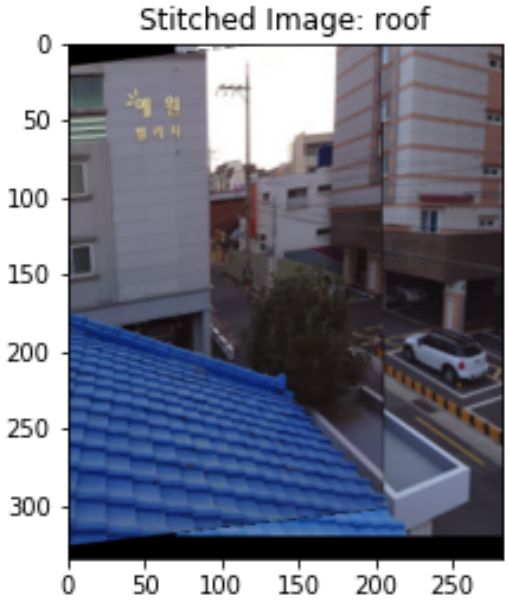


Figure 91: Stitched Image: roof

Figure 92: Stitched Image: stop

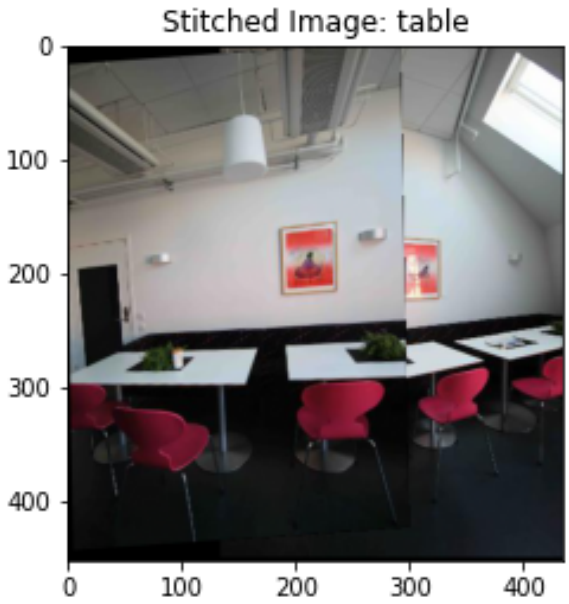


Figure 93: Stitched Image: table

5 Extra Credit

5.1 Different strategies to detect blobs at multiple scales - Downsample image

Iteratively downsample the image and filter it with the kernel of fixed size.
At each level, downsample(resize) the image by factor k.
Filter it with fixed sized LoG already made in the previous section.
Resize it to its original size.
Results prove downsampling image is faster than our original implemention. Implementation of *detectblobs.py*

```
# This code is part of:
#
#   CMPSCI 670: Computer Vision
#   University of Massachusetts, Amherst
#   Instructor: Subhransu Maji
#

import numpy as np
import cv2
import scipy
from skimage.color import rgb2gray

#Imported libraries
from scipy.ndimage.filters import rank_filter
from itertools import chain
import time
from skimage import transform

def  LoGFilter(initSigma, filterSize):
```

```
    #Laplacian of Gaussian Filter
    sigma = initSigma ** 2

    idRange = np.linspace(-(filterSize - 1) / 2., (filterSize - 1) / 2., filterSize)
    x, y = np.meshgrid(idRange, idRange)

    kernel = np.exp(-(np.square(x) + np.square(y)) / (2. * sigma))
    kernel[kernel < np.finfo(float).eps * np.amax(kernel)] = 0
    kSum = np.sum(kernel)
    if kSum != 0:
        kernel /= np.sum(kernel)

    tmp = np.multiply(kernel, np.square(x) + np.square(y) - 2 * sigma) / (sigma ** 2)
    kernel = tmp - np.sum(tmp) / (filterSize ** 2)
    return kernel

def detectBlobs(im, param=None):
    # Input:
    #   IM - input image
    #
    # Ouput:
    #   BLOBS - n x 5 array with blob in each row in (x, y, radius, angle, score)

    #Convert image to grayscale and then to double such that minimum and maximum intensity are 0 and 1 respectively
    im = rgb2gray(im[:,:,:])
    im = cv2.normalize(im.astype('float'), None, 0.0, 1.0, cv2.NORM_MINMAX)

    #Compute scale space representation
    initSigma = 1.6 #Initial Scale
    factor = 1.25 #np.sqrt(2) #Factor by which scale is mulitplied each time
    threshold =  0.0001
    finalSigma = np.power(factor, 15) #Final Scale

    #Dynamically decide iteration levels from initial scale, last scale and multiplication factor
    n = 12 #int(np.ceil((np.log(finalSigma) - np.log(initSigma))/np.log(factor))) #Iterations of Laplacian Scale Space

    sigmas = [initSigma] * n
    sigmas[0] = initSigma

    height, width = np.shape(im)
    scaleSpace = np.empty((height, width, n))

    method = 0
    ##################################################################
    # Compute blob response (Squared Laplacian response in scale-space) #
    ##################################################################

    if method == 0:
        #Increase filter size, keep image same
        #Slower version
        for i in range(n):
            #Generate the Laplacian of Gaussian for Scale Level
```

```
            filterSize = int(round(6 * sigmas[i])) #Compute filter kernel size from previous sigma
            # Ensure right - odd filter size
            if filterSize % 2 == 0:
                filterSize += 1
            LoG = np.power(sigmas[i],2) * LoGFilter(sigmas[i], filterSize) # Obtain filter with normalization and apply convolution.
            nextFilter= cv2.filter2D(im, cv2.CV_32F, LoG) #Create a replica of previous filter
            scaleSpace[:,:, i] = np.square(nextFilter) #Store squared filter at each level
            #Update sigma for next level
            if (i+1 < n):
                sigmas[i+1] = sigmas[i] * factor

    elif method == 1:
        #Downsampling image - Faster version
        # Ensure odd filter size
        filterSize = int(round(6 * sigmas[0]))
        # Ensure right - odd filter size
        if filterSize % 2 == 0:
            filterSize += 1
        LoG = LoGFilter(sigmas[0], filterSize )

        # Scale the image.
        for i in range(n):
            # Down scale
            scaledH = int(height * ((1/factor) ** i))
            scaledW = int(width * ((1/factor) ** i))
            scaledIm = transform.rescale(im, (1/factor) ** i, order=3)
            nextFilter= cv2.filter2D(scaledIm, cv2.CV_32F, LoG)**2
            # Upscale
            scaleSpace[:,:, i] = transform.resize(nextFilter, (height, width), order=3)

    elif method == 2:
        #Difference-of-Gaussian pyramid
        # Ensure odd filter size.
        filterSize = int(round(6 * sigmas[0]))

        # Obtain filter (no normalization needed).
        DoG = LoGFilter(sigmas[0], filterSize) - LoGFilter(sigmas[0]*1.25, filterSize )

        # Scale the image.
        for i in range(n):
            #Down scale
            scaledH = int(height * ((1/factor) ** i))
            scaledW = int(width * ((1/factor) ** i))
            scaledIm = transform.rescale(im, (1/factor) ** i, order=3)
            nextFilter= cv2.filter2D(scaledIm, cv2.CV_32F, DoG)**2
            # Upscale
            scaleSpace[:,:, i] = transform.resize(nextFilter, (height, width), order=3)

    ##################################################
    # Perform Non-Maximum Suppression in Scale Space  #
    ##################################################
```

```
    maxScaleSpace = np.empty((height, width, n)) #Non-Maximum Suppression scale space for each level

    #Non-Maximum Suppression in each 2D slice
    #A point is Local Maximum if it is higher than all its neighbors (each point has 26 neighbors in 3D)
    for level in range(n):
        maxScaleSpace[:,:,level] = rank_filter(scaleSpace[:,:, level], -1, (3,3)) #Determining Local Maximum point using rank filter

    layerScaleSpace = np.zeros((maxScaleSpace.shape)) #Non-Maximum Suppression scale space for all levels

    #Compute non-max suppression across all layers
    for i in range(height):
        for j in range(width):
            maxValue = np.amax(maxScaleSpace[i,j,:])
            maxIndex = np.argmax(maxScaleSpace[i,j,:])
            layerScaleSpace[i,j,maxIndex] = maxValue

    #Zero Out All Positions that are not Local Maxima of the score (if the Value if not Greater than all its Neighbors)
    for level in range(n): # check
        layerScaleSpace[:,:,level] = np.where((layerScaleSpace[:,:,level] == scaleSpace[:,:,level]), layerScaleSpace[:,:,level], 0)

    #Mask filter to eliminate boundaries noises
    #check
    mask = np.zeros((height, width, n))
    for i in range(n):
        boundary = int(np.ceil(sigmas[i] * np.sqrt(2)))
        mask[boundary+1:height-boundary, boundary+1:width-boundary] = 1

    centerX = []
    centerY = []
    radii = []
    score = []

    #Set a Threshold on Squared Laplacian Response above which to report Region Detections
    for level in range(n):#check >=
        layerScaleSpace[:,:,level] = np.where((layerScaleSpace[:,:,level] > threshold) & (mask[:,:,level] == 1), layerScaleSpace[:,:,level], 0)

        #Obtain X and Y center coordinates for local maxima
        centerX.append(list(np.where(layerScaleSpace[:,:, level] != 0)[0]))
        centerY.append(list(np.where(layerScaleSpace[:,:,level] != 0)[1]))
        radius = np.sqrt(2) * sigmas[level] * np.ones(len(centerX[level])) # Compute radius
        radii.append(list(radius))
        score.append(layerScaleSpace[np.where(layerScaleSpace[:,:, level] != 0)[0],np.where(layerScaleSpace[:,:, level] != 0)[1],level])

    centerX = list(chain.from_iterable(centerX))
    centerY = list(chain.from_iterable(centerY))
    radii = list(chain.from_iterable(radii))
    score = list(chain.from_iterable(score))

```

```
    numBlob = len(centerX)
    blobs = []
    angle = np.zeros(numBlob)
    print(numBlob)
    if (numBlob > 0):
        for i in range(numBlob):
            blobs.append([centerY[i], centerX[i], radii[i], angle[i], score[i]])

    return np.array(blobs)
```

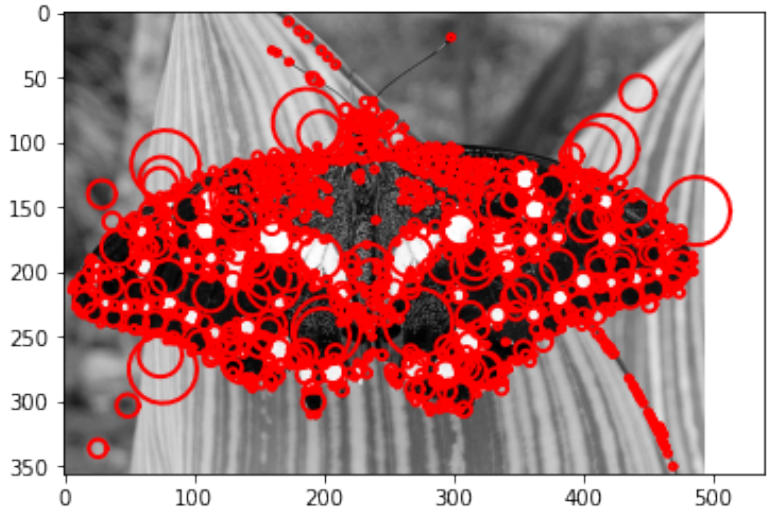


Figure 94: butterfly Sigma = 1.6 threshold = 0.0001

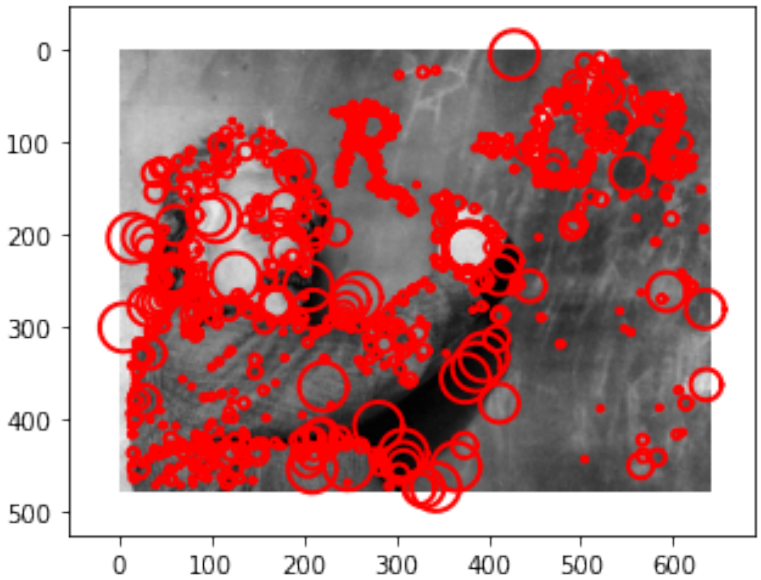


Figure 95: einstein Sigma = 1.6 threshold = 0.0001

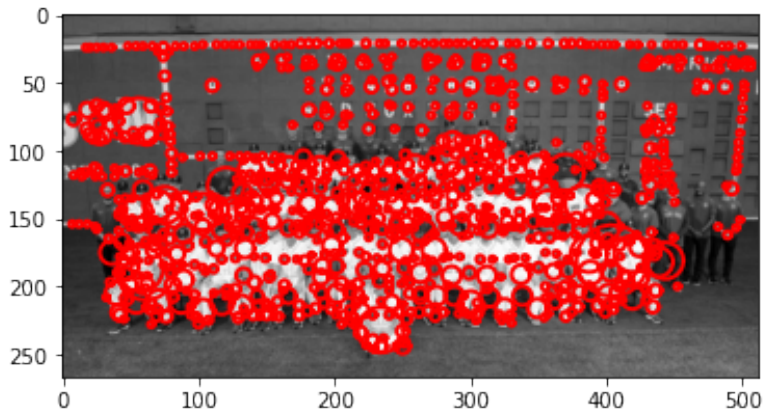


Figure 96: faces with, Sigma = 1.6 threshold = 0.0001

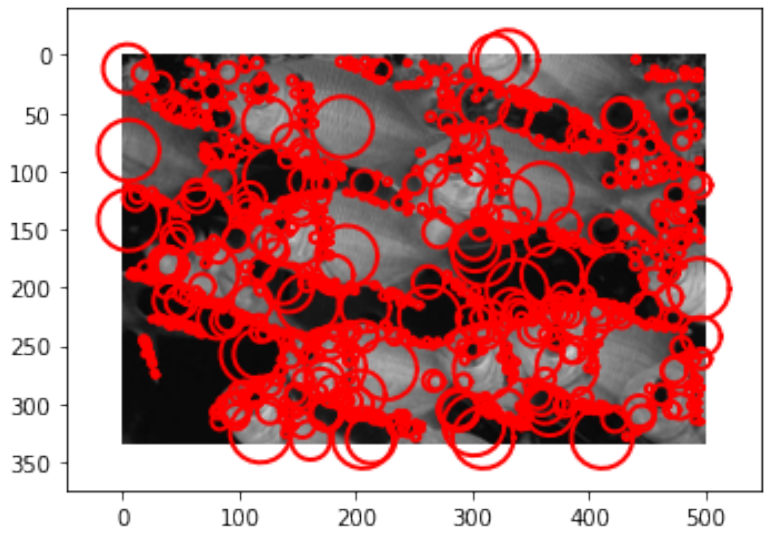


Figure 97: fishes with Sigma = 1.6 threshold = 0.0001

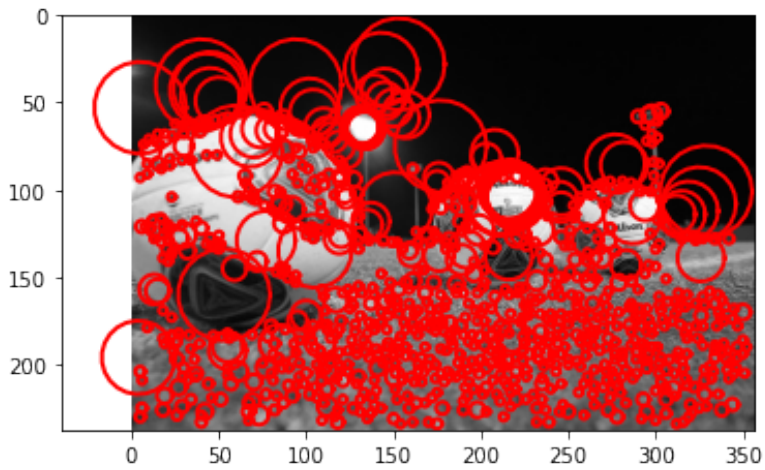


Figure 98: football with Sigma = 1.6 threshold = 0.0001

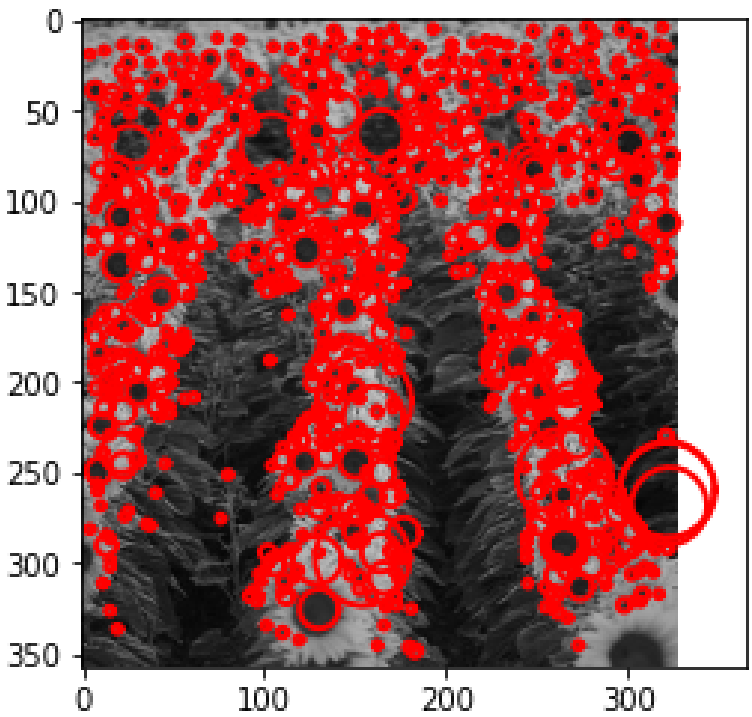


Figure 99: sunflowers with Sigma = 1.6 threshold = 0.0001

	LoG : Increasing Filter Size		LoG: Downsampling Image	
	Number of Blobs	Computing Blob Response (sec)	Number of Blobs	Computing Blob Response (sec)
butterfly	1776	1.932194471	1891	0.692731857
einstein	2615	3.325047731	2200	1.10229516
faces	1967	1.336702108	1975	0.545448065
fishes	1160	1.69015336	1262	0.637813568
football	1058	0.872504234	1146	0.346801043
sunflowers	2083	1.21902442	2189	0.441859007

Figure 100: Elapsed Times for each process using both techniques: Increasing Kernel Size and Down sampling the Image which proves that down sampling the image is faster

5.2 Difference-of-Gaussian pyramid

Difference-of-Gaussian pyramid is implemented by finding the difference between Gaussian Laplacian filters of scaled transformed images (with inputs sigma and sigma*1.25 respectively) Implementation of *detectblobs.py*

```
# This code is part of:
#
#   CMPSCI 670: Computer Vision
#   University of Massachusetts, Amherst
#   Instructor: Subhransu Maji
#

import numpy as np
import cv2
```

```
import scipy
from skimage.color import rgb2gray

#Imported libraries
from scipy.ndimage.filters import rank_filter
from itertools import chain
import time
from skimage import transform

def  LoGFilter(initSigma, filterSize):
    #Laplacian of Gaussian Filter
    sigma = initSigma ** 2

    idRange = np.linspace(-(filterSize - 1) / 2., (filterSize - 1) / 2., filterSize)
    x, y = np.meshgrid(idRange, idRange)

    kernel = np.exp(-(np.square(x) + np.square(y)) / (2. * sigma))
    kernel[kernel < np.finfo(float).eps * np.amax(kernel)] = 0
    kSum = np.sum(kernel)
    if kSum != 0:
        kernel /= np.sum(kernel)

    tmp = np.multiply(kernel, np.square(x) + np.square(y) - 2 * sigma) / (sigma ** 2)
    kernel = tmp - np.sum(tmp) / (filterSize ** 2)
    return kernel

def detectBlobs(im, param=None):
    # Input:
    #   IM - input image
    #
    # Ouput:
    #   BLOBS - n x 5 array with blob in each row in (x, y, radius, angle, score)

    #Convert image to grayscale and then to double such that minimum and maximum intensity are 0 and 1 respectively
    im = rgb2gray(im[:,:,:])
    im = cv2.normalize(im.astype('float'), None, 0.0, 1.0, cv2.NORM_MINMAX)

    #Compute scale space representation
    initSigma = 1.6 #Initial Scale
    factor = 1.25 #np.sqrt(2) #Factor by which scale is mulitplied each time
    threshold =  0.0001
    finalSigma = np.power(factor, 15) #Final Scale

    #Dynamically decide iteration levels from initial scale, last scale and multiplication factor
    n = 12 #int(np.ceil((np.log(finalSigma) - np.log(initSigma))/np.log(factor)) ) #Iterations of Laplacian Scale Space

    sigmas = [initSigma] * n
    sigmas[0] = initSigma

    height, width = np.shape(im)
    scaleSpace = np.empty((height, width, n))

```

```
    method = 0
    ##################################################################
    # Compute blob response (Squared Laplacian response in scale-space) #
    ##################################################################

    if method == 0:
        #Increase filter size, keep image same
        #Slower version
        for i in range(n):
            #Generate the Laplacian of Gaussian for Scale Level
            filterSize = int(round(6 * sigmas[i])) #Compute filter kernel size from previous sigma
            # Ensure right - odd filter size
            if filterSize % 2 == 0:
                filterSize += 1
            LoG = np.power(sigmas[i],2) * LoGFilter(sigmas[i], filterSize) # Obtain filter with normalization and apply convolution.
            nextFilter= cv2.filter2D(im, cv2.CV_32F, LoG) #Create a replica of previous filter
            scaleSpace[:,:, i] = np.square(nextFilter) #Store squared filter at each level
            #Update sigma for next level
            if (i+1 < n):
                sigmas[i+1] = sigmas[i] * factor

    elif method == 1:
        #Downsampling image - Faster version
        # Ensure odd filter size
        filterSize = int(round(6 * sigmas[0]))
        # Ensure right - odd filter size
        if filterSize % 2 == 0:
            filterSize += 1
        LoG = LoGFilter(sigmas[0], filterSize )

        # Scale the image.
        for i in range(n):
            # Down scale
            scaledH = int(height * ((1/factor) ** i))
            scaledW = int(width * ((1/factor) ** i))
            scaledIm = transform.rescale(im, (1/factor) ** i, order=3)
            nextFilter= cv2.filter2D(scaledIm, cv2.CV_32F, LoG)**2
            # Upscale
            scaleSpace[:,:, i] = transform.resize(nextFilter, (height, width), order=3)

    elif method == 2:
        #Difference-of-Gaussian pyramid
        # Ensure odd filter size.
        filterSize = int(round(6 * sigmas[0]))

        # Obtain filter (no normalization needed).
        DoG = LoGFilter(sigmas[0], filterSize) - LoGFilter(sigmas[0]*1.25, filterSize )

        # Scale the image.
        for i in range(n):
            #Down scale
            scaledH = int(height * ((1/factor) ** i))
```

```
            scaledW = int(width * ((1/factor) ** i))
            scaledIm = transform.rescale(im, (1/factor) ** i, order=3)
            nextFilter= cv2.filter2D(scaledIm, cv2.CV_32F, DoG)**2
            # Upscale
            scaleSpace[:,:, i] = transform.resize(nextFilter, (height, width), order=3)

    ##################################################
    # Perform Non-Maximum Suppression in Scale Space  #
    ##################################################
    maxScaleSpace = np.empty((height, width, n)) #Non-Maximum Suppression scale space for each level

    #Non-Maximum Suppression in each 2D slice
    #A point is Local Maximum if it is higher than all its neighbors (each point has 26 neighbors in 3D)
    for level in range(n):
        maxScaleSpace[:,:,level] = rank_filter(scaleSpace[:,:, level], -1, (3,3)) #Determining Local Maximum point using rank filter

    layerScaleSpace = np.zeros((maxScaleSpace.shape)) #Non-Maximum Suppression scale space for all levels

    #Compute non-max suppression across all layers
    for i in range(height):
        for j in range(width):
            maxValue = np.amax(maxScaleSpace[i,j,:])
            maxIndex = np.argmax(maxScaleSpace[i,j,:])
            layerScaleSpace[i,j,maxIndex] = maxValue

    #Zero Out All Positions that are not Local Maxima of the score (if the Value if not Greater than all its Neighbors)
    for level in range(n): # check
        layerScaleSpace[:,:,level] = np.where((layerScaleSpace[:,:,level] == scaleSpace[:,:,level]), layerScaleSpace[:,:,level], 0)

    #Mask filter to eliminate boundaries noises
    #check
    mask = np.zeros((height, width, n))
    for i in range(n):
        boundary = int(np.ceil(sigmas[i] * np.sqrt(2)))
        mask[boundary+1:height-boundary, boundary+1:width-boundary] = 1

    centerX = []
    centerY = []
    radii = []
    score = []

    #Set a Threshold on Squared Laplacian Response above which to report Region Detections
    for level in range(n):#check >=
        layerScaleSpace[:,:,level] = np.where((layerScaleSpace[:,:,level] > threshold) & (mask[:,:,level] == 1), layerScaleSpace[:,:,level], 0)

        #Obtain X and Y center coordinates for local maxima
        centerX.append(list(np.where(layerScaleSpace[:,:, level] != 0)[0]))
        centerY.append(list(np.where(layerScaleSpace[:,:,level] != 0)[1]))
```

```
        radius = np.sqrt(2) * sigmas[level] * np.ones(len(centerX[level])) # Compute radius
        radii.append(list(radius))
        score.append(layerScaleSpace[np.where(layerScaleSpace[:,:, level] != 0)[0],np.where(layerScaleSpace[:,:, level] != 0)[1],level])

    centerX = list(chain.from_iterable(centerX))
    centerY = list(chain.from_iterable(centerY))
    radii = list(chain.from_iterable(radii))
    score = list(chain.from_iterable(score))

    numBlob = len(centerX)
    blobs = []
    angle = np.zeros(numBlob)
    print(numBlob)
    if (numBlob > 0):
        for i in range(numBlob):
            blobs.append([centerY[i], centerX[i], radii[i], angle[i], score[i]])

    return np.array(blobs)
```

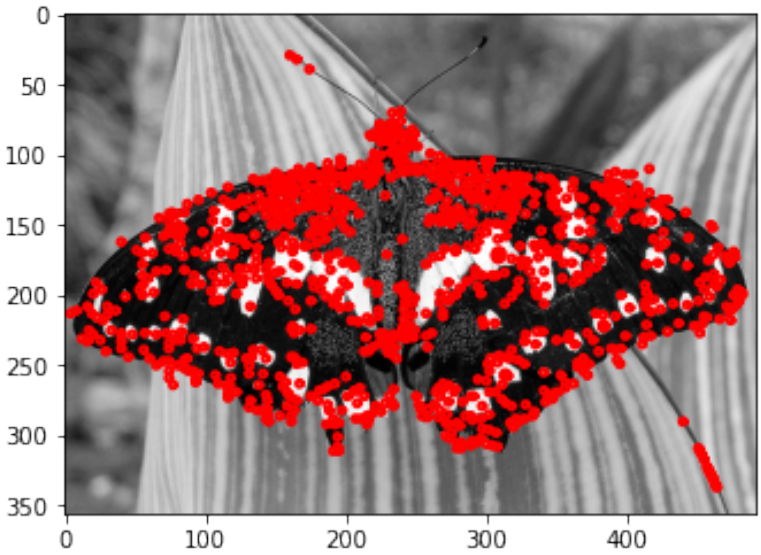


Figure 101: butterfly with Sigma = 1.6 threshold = 0.0001

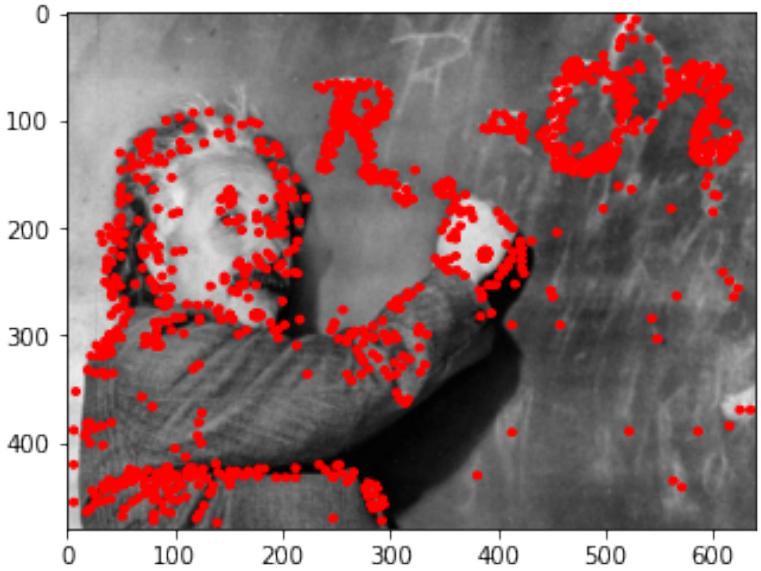


Figure 102: einstein with Sigma = 1.6 threshold = 0.0001

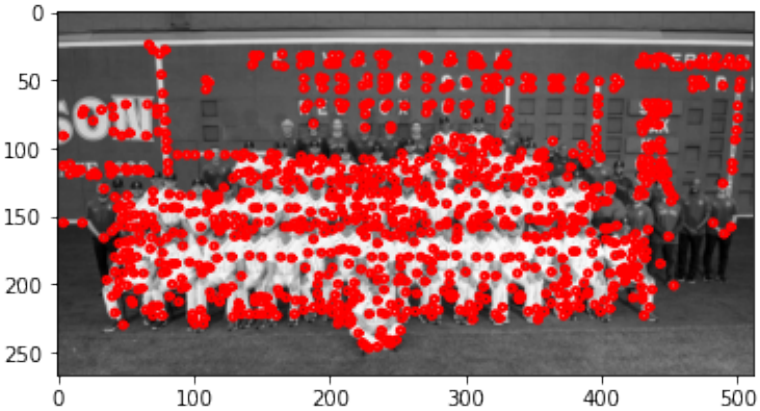


Figure 103: faces with Sigma = 1.6 threshold = 0.0001

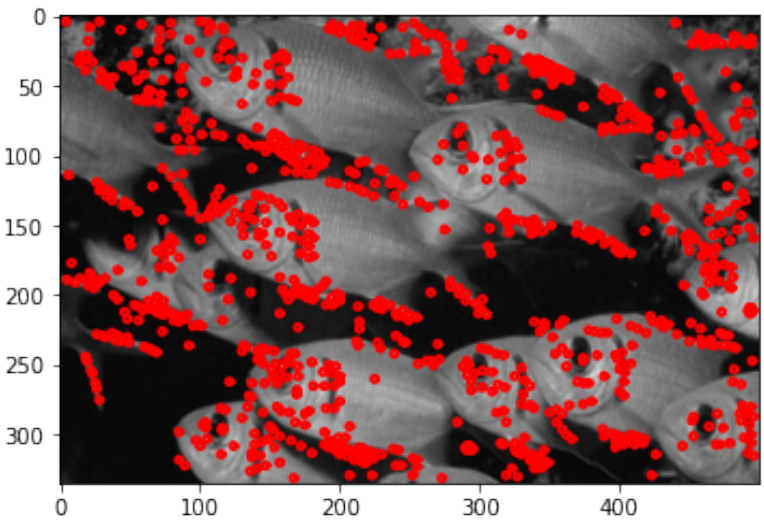


Figure 104: fishes with Sigma = 1.6 threshold = 0.0001

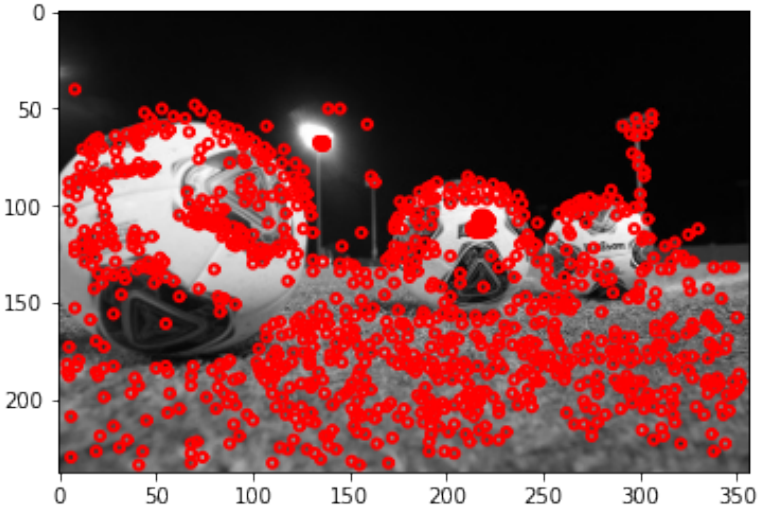


Figure 105: football with Sigma = 1.6 threshold = 0.0001

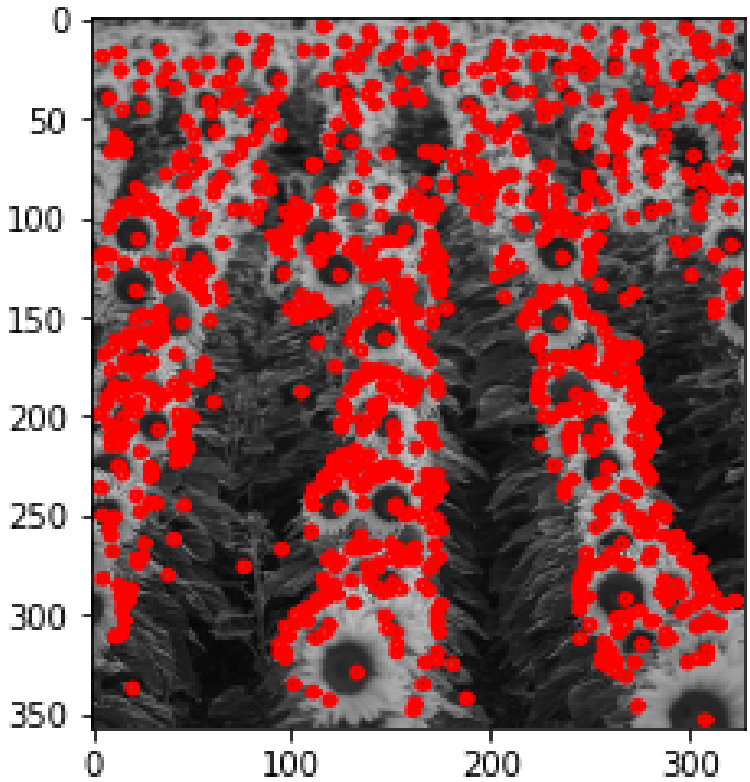


Figure 106: sunflowers with Sigma = 1.6 threshold = 0.0001

	LoG : Increasing Filter Size		LoG: Downsampling Image		Difference of Gaussian Pyramid	
	Number of Blobs	Computing Blob Response (sec)	Number of Blobs	Computing Blob Response (sec)	Number of Blobs	Computing Blob Response (sec)
butterfly	1776	1.932194471	1891	0.692731857	2313	2.113242388
einstein	2615	3.325047731	2200	1.10229516	1600	3.342083216
faces	1967	1.336702108	1975	0.545448065	2513	1.482041359
fishes	1160	1.69015336	1262	0.637813568	1543	2.037741661
football	1058	0.872504234	1146	0.346801043	1199	0.981491804
sunflowers	2083	1.21902442	2189	0.441859007	2934	1.23356986

Figure 107: Elapsed Times for each process using all techniques for difference of Gaussian pyramid

5.3 Harris Corner Detection

1. Calculate image x and y derivatives.
2. Derivate again the previous values to obtain the second derivative.
3. For each pixel, sum the last step obtained derivatives. Here we are making a 1 pixel sift of the windows over the image.
4. For each pixel and using the sums of the previous step, define H matrix.
5. Calculate the response of the detector
6. Use a threshold value in order to exclude some of the detections.

Implementation of *harris.py*

```
from pylab import *
from scipy import signal
from scipy import *
import numpy as np
from PIL import Image
```

```python

def gauss_derivative_kernels(size, sizey=None):
    """ returns x and y derivatives of a 2D
        gauss kernel array for convolutions """
    size = int(size)
    if not sizey:
        sizey = size
    else:
        sizey = int(sizey)
    y, x = mgrid[-size:size+1, -sizey:sizey+1]
    #x and y derivatives of a 2D gaussian with standard dev half of size
    # (ignore scale factor)
    gx = - x * exp(-(x**2/float((0.5*size)**2)+y**2/float((0.5*sizey)**2)))
    gy = - y * exp(-(x**2/float((0.5*size)**2)+y**2/float((0.5*sizey)**2)))
    return gx,gy

def gauss_kernel(size, sizey = None):
    """ Returns a normalized 2D gauss kernel array for convolutions """
    size = int(size)
    if not sizey:
        sizey = size
    else:
        sizey = int(sizey)
    x, y = mgrid[-size:size+1, -sizey:sizey+1]
    g = exp(-(x**2/float(size)+y**2/float(sizey)))
    return g / g.sum()

def compute_harris_response(im):
    """ compute the Harris corner detector response function
        for each pixel in the image"""
    #derivatives
    gx,gy = gauss_derivative_kernels(3)
    imx = signal.convolve(im,gx, mode='same')
    imy = signal.convolve(im,gy, mode='same')
    #kernel for blurring
    gauss = gauss_kernel(3)
    #compute components of the structure tensor
    Wxx = signal.convolve(imx*imx,gauss, mode='same')
    Wxy = signal.convolve(imx*imy,gauss, mode='same')
    Wyy = signal.convolve(imy*imy,gauss, mode='same')
    #determinant and trace
    Wdet = Wxx*Wyy - Wxy**2
    Wtr = Wxx + Wyy
    return Wdet / Wtr

def get_harris_points(harrisim, min_distance=10, threshold=0.1):
    """ return corners from a Harris response image
        min_distance is the minimum nbr of pixels separating
        corners and image boundary"""
    #find top corner candidates above a threshold
    corner_threshold = max(harrisim.ravel()) * threshold
    harrisim_t = (harrisim > corner_threshold) * 1
    #get coordinates of candidates
    candidates = harrisim_t.nonzero()
    coords = [ (candidates[0][c],candidates[1][c]) for c in range(len(candidates
    [0]))]
    #...and their values
    candidate_values = [harrisim[c[0]][c[1]] for c in coords]
```

```
    #sort candidates
    index = argsort(candidate_values)
    #store allowed point locations in array
    allowed_locations = zeros(harrisim.shape)
    allowed_locations[min_distance:-min_distance,min_distance:-min_distance] = 1
    #select the best points taking min_distance into account
    filtered_coords = []
    for i in index:
        if allowed_locations[coords[i][0]][coords[i][1]] == 1:
            filtered_coords.append(coords[i])
            allowed_locations[(coords[i][0]-min_distance):(coords[i][0]+min_distance),
                (coords[i][1]-min_distance):(coords[i][1]+min_distance)] = 0
    return filtered_coords

def plot_harris_points(image, filtered_coords):
    """ plots corners found in image"""
    figure()
    gray()
    imshow(image)
    plot([p[1] for p in filtered_coords],[p[0] for p in filtered_coords],'r*')
    axis('off')
    show()

images = ['butterfly.jpg','einstein.jpg','faces.jpg','fishes.jpg','football.jpg','sunflowers.jpg']
for imageName in images:
    im = np.array(Image.open(imageName).convert("L"))
    harrisim = compute_harris_response(im)
    filtered_coords = get_harris_points(harrisim,min_distance=10, threshold=0.1)
    plot_harris_points(im, filtered_coords)
```

Figure 108: butterfly

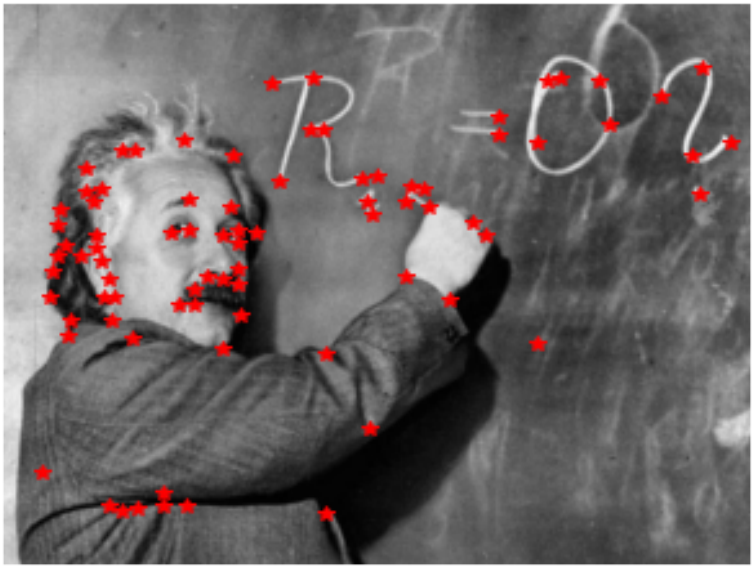


Figure 109: einstein

Figure 110: faces

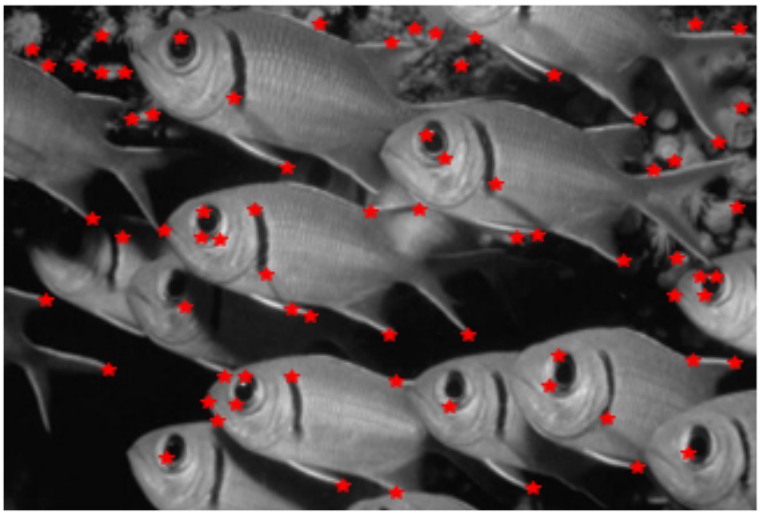


Figure 111: fishes

Figure 112: football

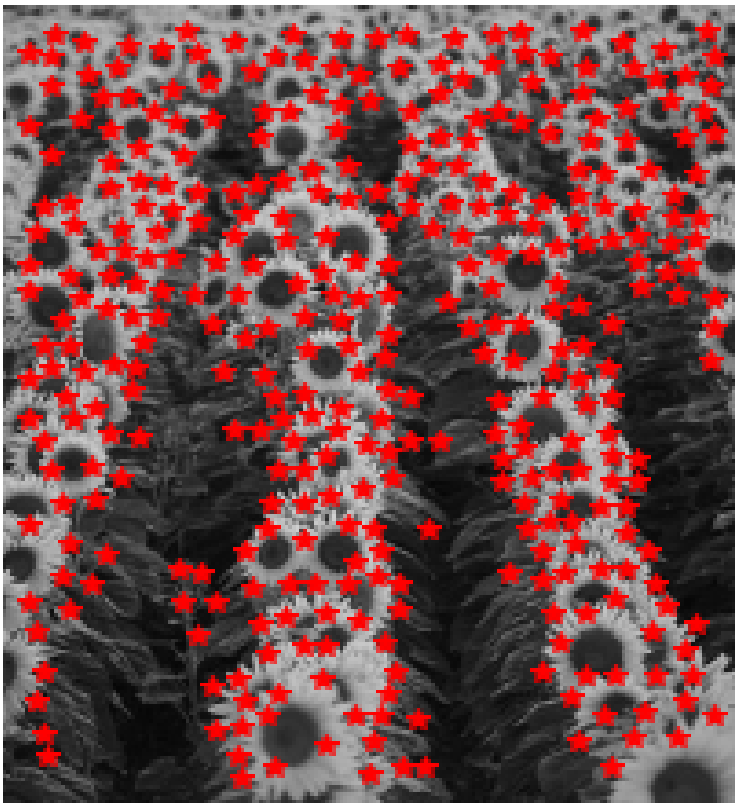


Figure 113: sunflowers