# CSC420 - Assignment 2

## 1. Implement Seam Carving

My implementation consists of two tables

- Cumulative Energy Table
  - Look up table used for the dynamic programming algorithm
  - Stores the cumulative energy of the shortest (least expensive) path from the top row to the bottom row
- Seam Path Table
  - Table that stores the position of the previous pixel in the shortest energy path along a seam

The Cumulative Energy at a given pixel is calculated as:

$$E(i,j) = G(i,j) + \min\{E(i-1,j-1), E(i-1,j), E(i-1,j+1)\}$$

where E is the cumulative energy to point (i,j) and G is the gradient at that point

The Seam Path Table entry at a given pixel is the position of the previously traversed pixel with the lowest cumulative energy. It is calculated as:

$$P(i,j) = \underset{(x,\,y)}{\arg\min}\{E(x,y)\} \text{ , where } x = i-1, \quad y \in \{j-1,\ j,\ j+1\}$$

**Seam Carving Example #1**
The below image is seam carved 80 times to produce the final image.

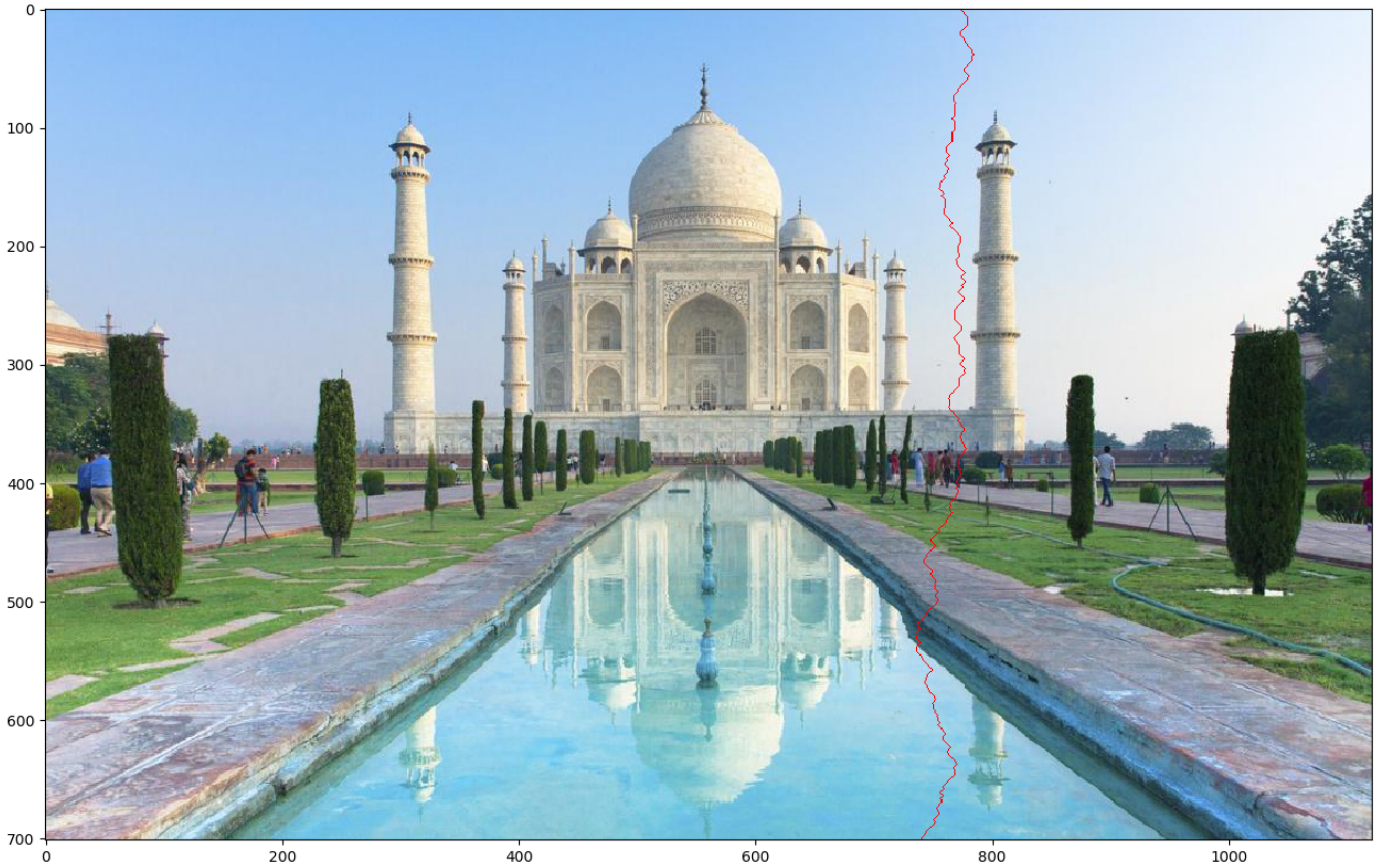

Original Image


80th Seam Removed


Final Seam Carved Image

**Seam Carving Example #2**
The below image is seam carved 80 times to produce the final image.


Original Image




80th Seam Removed


Final Seam Carved Image

# 2. Implement Harris Corner Detector

The implementation is straightforward and directly based off the lecture slides.

- My implementation uses the following parameters
  - gaussian window filter with standard deviation = 1, size = 5
  - alpha parameter = 0.04
  - threshold = 0.5 and 0.1

Key Algorithm Steps:

1) Create gaussian window filter (given a window size and standard deviation)
2) Find image gradient terms (I_xx, I_xy, I_yy)
3) Compute elements of matrix M (M_11, M_12, M_21, M_22)
4) Compute Cornerness (R = (M_11 * M_22 - M_12 * M_21) - alpha * (M_11 + M_22)**2) for each point
5) Threshold the Cornerness
6) Non-Maximum Supression

**Harris Corner Example #1**


Cornerness Threshold = 0.5

**Harris Corner Example #2**


Cornerness Threshold = 0.1

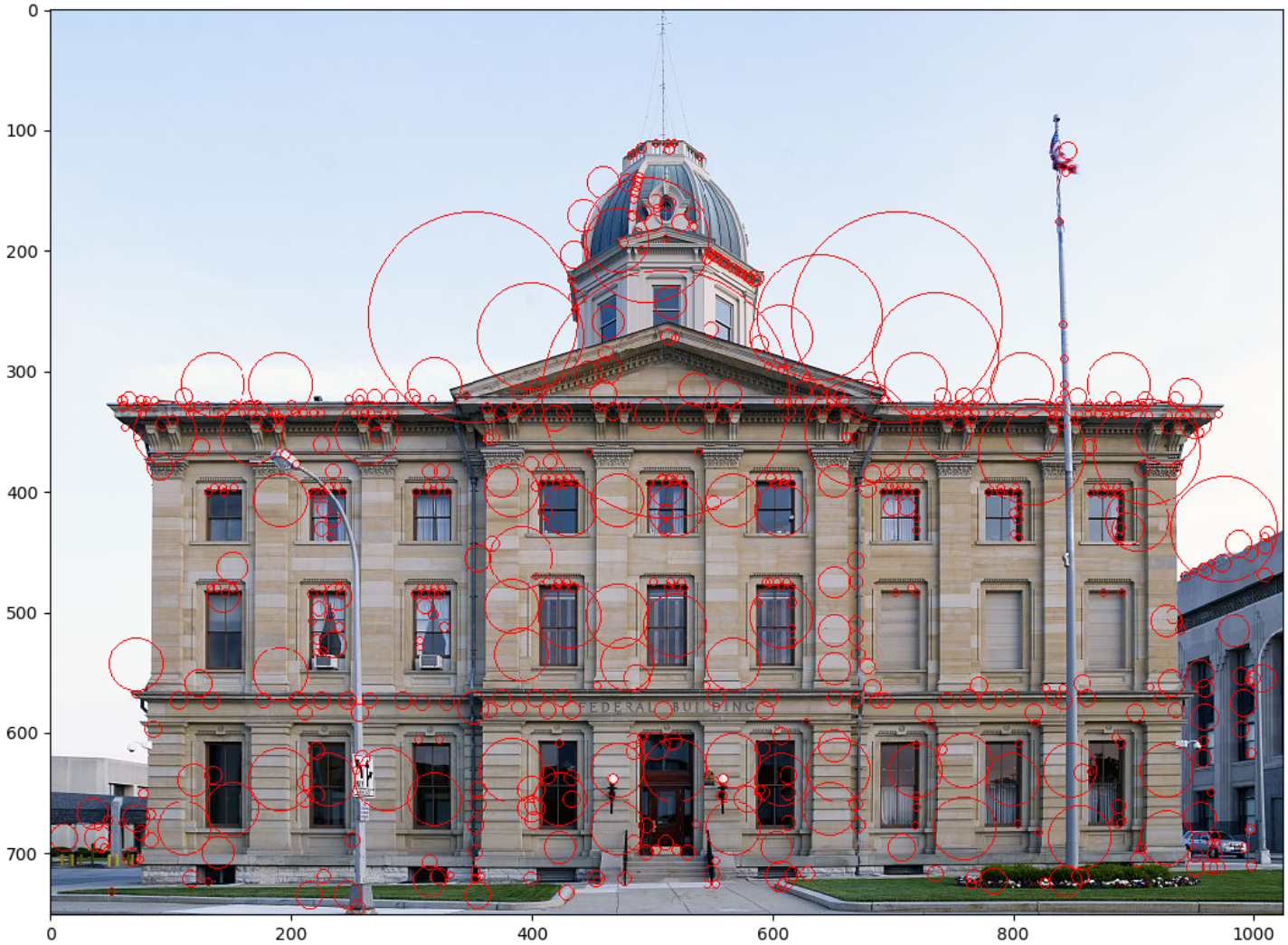
# 3. Implement SIFT Detector

Most of the implementation is described in the lecture slides. However, there are two concepts that need explanation:

- Determining the location of a keypoint by upscaling the position of maxima at higher octaves
  - The pyramid_gaussian function takes every **odd** numbered index and discards the even numbered indices
    - We need to upscale by $2i + 1$ for each octave (n>1)
    - A formula for cumulative upscaling at the nth octave is $2^{n-1}i + 2^{n-1} - 1$
      - this formula is off by 1 pixel for n=1. Since 1 pixel is inconsequential I disregarded this case when upscaling
- Determining the radius of a blob/circle detected given the scale and octave of the maxima
  - Given a scale and octave we can calculate the equivalent standard deviation of the DoG/Laplacian.
  - The maxima of the Laplacian occurs when the zeros of the Laplacian are aligned with the circle
    - Therefore maximum response occurs at $r = \sigma\sqrt{2}$

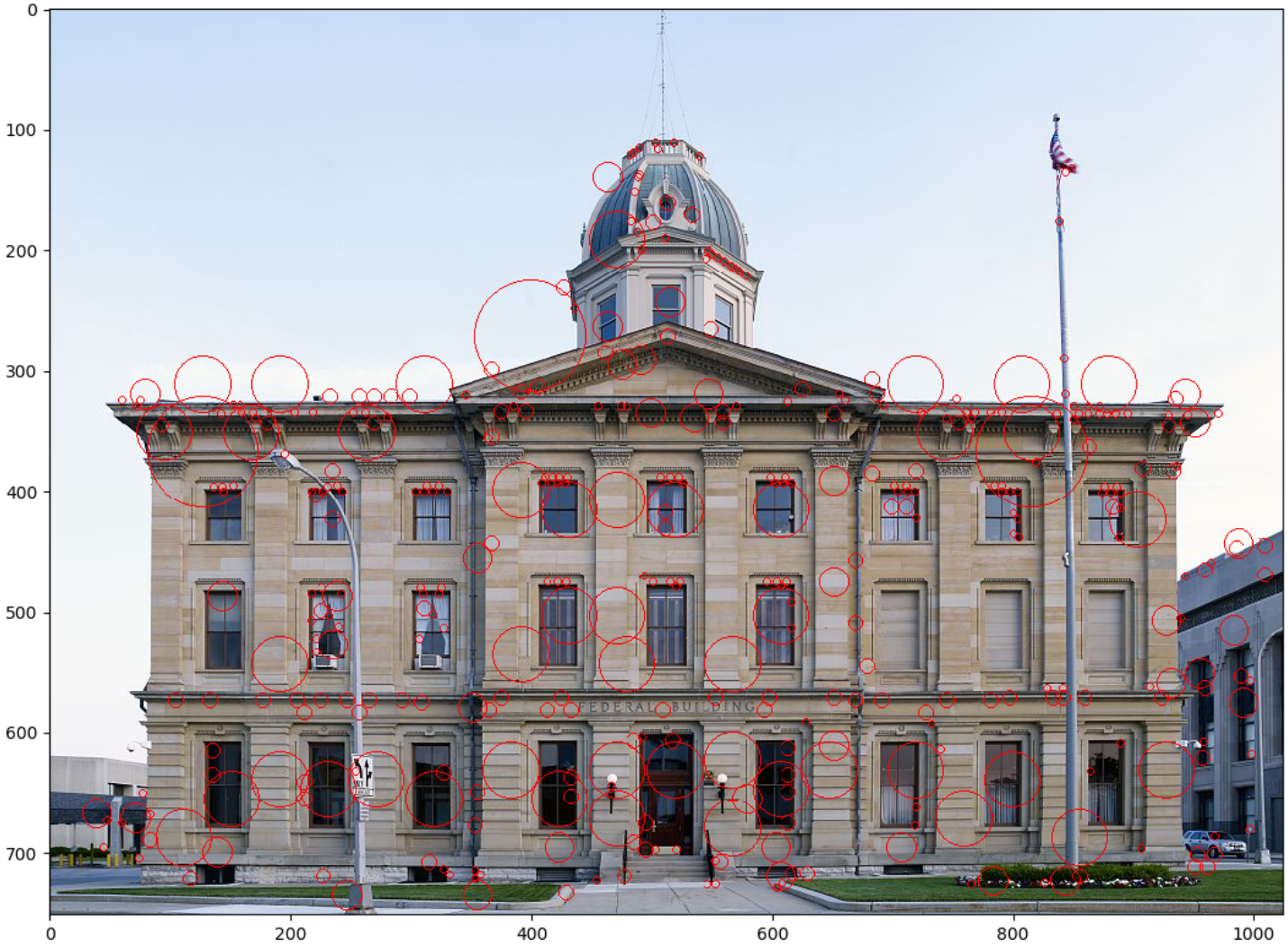
The parameter values I used in my implementation are as follows:
- standard deviation used in the scale space = 1.6
- DoG threshold = 0.1

**SIFT Detector Example #1**




Number of Scales per Octave = 5

**SIFT Detector Example #2**




Number of Scales per Octave = 4

# Appendix - Code

**gradient.py** (helper function)

```
import numpy as np
from scipy import ndimage

def gradient(image):

    sobel_x = np.array([
        [-1, 0, +1],
        [-2, 0, +2],
        [-1, 0, +1]
    ])
    sobel_y = np.array([
        [-1, -2, -1],
        [0, 0, 0],
        [+1, +2, +1]
    ])

    G_x = ndimage.correlate(image, sobel_x, mode='nearest')
    G_y = ndimage.correlate(image, sobel_y, mode='nearest')
    G = np.sqrt(G_x**2 + G_y**2)

    return G, G_x, G_y
```

---

## seam_carving.py

```
import numpy as np
from skimage import io
from skimage.color import rgb2gray
from scipy import ndimage
from gradient import *

def minimum_seam(image):
    # FIND IMAGE GRADIENT
    G, G_x, G_y = gradient(image)
    height, width = G.shape

    # GENERATE DP LOOKUP TABLE
    E = np.full(G.shape, np.inf)        # Energy Table
    E[0] = G[0]
    P = np.empty_like(G, dtype=object)  # Seam Path Table

    # POPULATE LOOK UP TABLES
    for i in range(1, height):
        for j in range(width):
            E_left = E[i-1,j-1] if j-1 >=0 else np.inf
            E_right = E[i-1,j+1] if j+1 < width else np.inf
            E_center = E[i-1,j]

            # UPDATE ENERGY TABLE
            E[i,j] = G[i,j] + min(E_left, E_center, E_right)

            # UPDATE PATH TABLE
```

```
            if E_left == min(E_left, E_center, E_right):
                P[i,j] = (i-1, j-1)
            elif E_center == min(E_left, E_center, E_right):
                P[i,j] = (i-1, j)
            elif E_right == min(E_left, E_center, E_right):
                P[i,j] = (i-1, j+1)

    # FIND MINIMUM ENERGY PATH
    min_i = height-1
    min_j = np.argmin(E[height-1])
    seam_root = (min_i, min_j)

    # RETURN SEAM PATH AND SEAM ROOT INDEX
    return P, seam_root


def seam_carving(image):
    image_gray = rgb2gray(image)
    P, seam_index = minimum_seam(image_gray)

    carve_image = np.zeros_like(image[:,:-1,:])
    seam_image = np.copy(image)

    while True:
        # Get index of the pixel on the seam
        i, j = seam_index[0], seam_index[1]

        # Draw Seam Line in Red
        seam_image[i,j] = (255,0,0)

        # Remove pixel
        carve_image[i,:,0] = np.delete(image[i,:,0], j)
        carve_image[i,:,1] = np.delete(image[i,:,1], j)
        carve_image[i,:,2] = np.delete(image[i,:,2], j)

        if i != 0:  # Move to next pixel on seam path
            seam_index = P[i,j]
        else:       # Exit loop after entire seam is traversed
            break

    return seam_image, carve_image


def main():
    image = io.imread('images/water.jpg', as_gray=False)

    carve_image = np.copy(image)
    for i in range(80):
        seam_image, carve_image = seam_carving(carve_image)
        print(carve_image.shape)

    io.imshow(image)
    io.show()

    io.imshow(seam_image)
    io.show()

    io.imshow(carve_image)
    io.show()
```

## harris_corner_detector.py

```
import numpy as np
from skimage import io
from skimage import draw
from skimage.color import rgb2gray
from scipy import ndimage
from gradient import *

# RETURNS CORNERNESS MAP OF ONLY LOCAL MAXIMUM
def find_local_maximum(R):
    height, width = R.shape
    Corners = np.zeros_like(R)

    for i in range(height):
        for j in range(width):
            thresh = max(
                R[i-1,j-1] if i-1>=0 and j-1>=0 else 0,
                R[i-1,j]   if i-1>=0 else 0,
                R[i-1,j+1] if i-1>=0 and j+1<width else 0,
                R[i,j-1]   if j-1>=0 else 0,
                R[i,j+1]   if j+1<width else 0,
                R[i+1,j-1] if i+1<height and j-1>=0 else 0,
                R[i+1,j]   if i+1<height else 0,
                R[i+1,j+1] if i+1<height and j+1<width else 0
            )
            if R[i,j] >= thresh:
                Corners[i,j] = R[i,j]
    return Corners

def harris_corner_detector(image, window_size, stddev, thresh):
    image_gray = rgb2gray(image)

    # CREATE GAUSSIAN WINDOW FILTER
    if window_size%2==0: window_size=window_size+1
    k = int((window_size-1)/2)
    window = np.zeros((window_size, window_size))
    window[k, k] = 1
    window = ndimage.gaussian_filter(window, sigma=stddev)

    # FIND IMAGE GRADIENT TERMS
    G, I_x, I_y = gradient(image_gray)
    I_xx = I_x * I_x
    I_yy = I_y * I_y
    I_xy = I_x * I_y

    # COMPUTE M
    M_11 = ndimage.correlate(I_xx, window, mode='nearest')
    M_12 = M_21 = ndimage.correlate(I_xy, window, mode='nearest')
    M_22 = ndimage.correlate(I_yy, window, mode='nearest')

    # COMPUTE CORNERNESS R
    alpha = 0.04
    R = (M_11 * M_22 - M_12 * M_21) - alpha * (M_11 + M_22)**2

    # THRESHOLD R
    threshold = thresh
    R[R<threshold] = 0
```

```
    # NON-LOCAL-MAXIMUM SUPRESSION
    Corners = find_local_maximum(R)

    return R, Corners

def main():
    image = io.imread('images/building.jpg', as_gray=False)
    height, width, depth = image.shape

    R, Corners = harris_corner_detector(image, 5, stddev=1, thresh=0.5)

    # DRAW CORNERS ONTO IMAGE
    for i in range(height):
        for j in range(width):
            if Corners[i,j] > 0:
                rr, cc = draw.circle(i,j, 2, image.shape)
                image[rr,cc,:] = (255,0,0)

   io.imshow(image)
    io.show()
```

---

## SIFT_keypoint_detector.py

```
import numpy as np
from skimage import io
from skimage import draw
from skimage.color import rgb2gray
from skimage.transform import pyramid_gaussian
from scipy import ndimage
from math import ceil

# n_scales: number of scales in the octave
# sigma: base standard devation for each octave
def create_scale_space(octave, sigma, n_scales):
    shape = (octave.shape[0], octave.shape[1], n_scales)
    I = np.zeros(shape)
    k = 2**(1./(n_scales-1))

    for i in range(n_scales):
        stddev = (k**i) * sigma
        I[:,:,i] = ndimage.gaussian_filter(octave, sigma=stddev)

    return I

def difference_of_gaussians(I):
    shape = (I.shape[0], I.shape[1], I.shape[2]-1)
    D = np.zeros(shape)

    for i in range(1, I.shape[2]):
        D[:,:,i-1] = I[:,:,i] - I[:,:,i-1]

    return D
```

```
def check_local_extrema(D, i, j, k):
    target = D[i,j,k]
    for x in [i-1, i, i+1]:
        for y in [j-1, j, j+1]:
            for z in [k-1, k, k+1]:
                if D[x,y,z] > target:
                    return False
    return True

def find_extrema(D, thresh):
    n_scales = D.shape[2]
    height, width = D.shape[:2]
    extrema = []

    for k in range(1, n_scales-1):
        for i in range(1, height-1):
            for j in range(1, width-1):
                isLocalExtrema = check_local_extrema(D, i, j, k)
                # If a point is a local extrema add it to the list
                if isLocalExtrema:
                    # Threshold the point
                    if D[i,j,k] > thresh:
                        extrema.append((i,j,k))

    return extrema

def find_sift_keypoints(local_extrema, octave_num, sigma, n_scales):
    k = 2**(1./(n_scales-1))
    keypoints = []

    for extrema in local_extrema:
        # Upsample the location and scale
        scaling_factor = (2 ** (octave_num-1))

        # i,j coordinate up-scaling
        location = (extrema[0] * scaling_factor + scaling_factor - 1,
                    extrema[1] * scaling_factor + scaling_factor - 1)

        # determining the scale at the keypoint
        scale = (k ** extrema[2]) * sigma * scaling_factor
        sift_keypoint = (location, scale)

        keypoints.append(sift_keypoint)

    return keypoints

# stddev:   standard deviation used for gaussian smoothing of the
#           first scale in each octave
# n_scales: number of scales per octave
# thresh:   threshold for the DoG extrema
def sift_keypoint_detector(image, stddev, n_scales, thresh):
    image_gray = rgb2gray(image)
    sift_keypoints = []
    oct_num = 1

    # COMPUTE GAUSSIAN PYRAMIDS
    pyramid = tuple(pyramid_gaussian(image_gray, multichannel=False))

    # iterate over each octave
```

```
    for octave in pyramid:

        # CREATE SCALE SPACE
        I = create_scale_space(octave, sigma=stddev, n_scales=n_scales)

        # COMPUTE DIFFERENCE OF GAUSSIANS
        D = difference_of_gaussians(I)

        # Modify D to make dealing with DoG easier
        D = D**2            # Square to deal with only maxima
        D_max = D.max()     # Scale to 1
        D = D/D_max

        # LOCATE LOCAL EXTREMA
        local_extrema = find_extrema(D, thresh=thresh)

        # STORE LOCAL EXTREMA POSITION AND SCALE IN RETURN LIST
        keypoints = find_sift_keypoints(local_extrema, oct_num, stddev, n_scales)
        sift_keypoints.extend(keypoints)

        oct_num = oct_num + 1

    return sift_keypoints



def main():
    image = io.imread('images/building.jpg', as_gray=False)

    keypoints = sift_keypoint_detector(image,
                                       stddev=1.6,
                                       n_scales=5,
                                       thresh=0.1)

    # Plot detected SIFT keypoints
    for point in keypoints:
        (i,j) = point[0]
        stddev = point[1]
        radius = ceil(2**(0.5) * stddev)

        rr, cc = draw.circle_perimeter(i,j, radius, shape=image.shape)
        image[rr,cc,:] = (255,0,0)


    io.imshow(image)
    io.show()
```