Wensong Hu
EECS
University of Michigan

February 13, 2024

HW2 — Convolution and Feature Detection

Complete code is attached in appendix

1 Patches

1.1 Task 1: Image Patches

1.1.1 image_patches() and result

```
def image_patches(image, patch_size=(16, 16)):
    """
    Given an input image and patch_size,
    return the corresponding image patches made
    by dividing up the image into patch_size sections.

    Input- image: H x W
           patch_size: a scalar tuple M, N
    Output- results: a list of images of size M x N
    """
    # TODO: Use slicing to complete the function
    output = []
    H, W = image.shape
    h, w = patch_size
    num_h = H // h
    num_w = W // w

    for i in range(num_h):
        for j in range(num_w):
            patch = image[i * h : (i + 1) * h, i * w : (i + 1) * w]
            patch_mean = np.mean(patch)
            patch_std = np.std(patch)
            patch = (patch - patch_mean) / patch_std
            patch = np.nan_to_num(patch, nan=0.0, posinf=0.0, neginf=0.0)
            output.append(patch)
    # import pdb; pdb.set_trace()
    return output
```

Result see figure1.

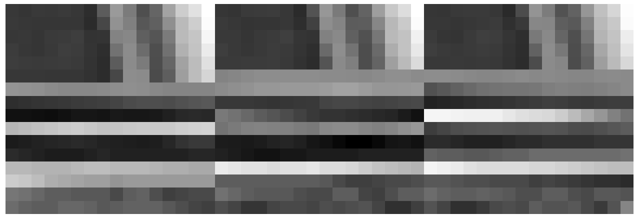

Figure 1: Image patches

1.1.2 Why normalized?

Normalization makes the measurement of similarity between image patches more robust to variations in lighting or illumination, focusing the comparison on the intrinsic patterns and textures. It ensures that the dot product similarity measure reflects the actual content similarity, rather than being affected by extrinsic factors like lighting conditions.

1.1.3 Discuss patches in early CV

Normalized patches, with zero mean and unit variance, are excellent for enhancing the robustness of matching or recognizing objects under varying conditions such as illumination changes, as they focus on the structure and texture rather than the absolute brightness. However, these patches might not be as effective when dealing with changes in an object's pose, scale, or significant variations in viewpoint, because normalization does not inherently account for geometric transformations, which could lead to significant differences in the appearance of patches extracted from these varied conditions.

2 Image Filtering

2.1 Task 2: Convolution and Gaussian Filter

2.1.1 Prove equivalency

We knows that, for two Gaussian filters $G_y \in \mathbb{R}^{k\times 1}$ and $G_x \in \mathbb{R}^{k\times 1}$:

$$G_x * G_y = G_x G_y = G \tag{1-1}$$

So we have:

$$\begin{aligned} G * X &= (G_x G_y) * X && \text{(1-2)} \\ &= (G_x * G_y) * x && \text{(1-3)} \\ &= G_x * (G_y * X) && \text{(1-4)} \end{aligned}$$

This proves that convolution by a 3D Gaussian filter is equivalent to sequentially applying a vertical and horizontal Gaussian filter.

2.1.2 convolve()

```python
def convolve(image, kernel):
    """
    Return the convolution result: image * kernel.
    Reminder to implement convolution and not cross-correlation!
    Caution: Please use zero-padding.

    Input- image: H x W
           kernel: h x w
    Output- convolve: H x W
    """
    output = np.zeros_like(image)
    if len(kernel.shape) == 2:
        kernel = kernel[ : :-1 , : :-1]
    elif len(kernel.shape) == 1:
        kernel = kernel[ : :-1]

    H, W = image.shape
    h, w = kernel.shape

    padding = [h//2, w//2]
    padded_image = np.zeros((H + 2 * padding[0], W + 2 * padding[1]), dtype=image.dtype)
    padded_image[padding[0] : H + padding[0], padding[1] : W + padding[1]] = image

    for y in range(H):
        for x in range(W):
            patch = padded_image[y : y + h , x : x + w]
            output[y, x] = np.sum(patch * kernel)

    return output
```

2.1.3 Result and Discuss

Result see figure 2.

Figure 2: Result of Gaussian Filtering

Gaussian filtering smooths an image by blurring and reducing its high-frequency components, mitigating noise and details.

2.1.4 Discussion of normalization of kernel

Ensure the filter sum up to 1 to ensure that the overall brightness of the image is preserved after filtering. If the filter sums to more than 1, it could artificially increase the intensity of the image (making it brighter), while if it sums to less than 1, it could decrease the intensity (making it darker). This preservation of brightness is important for maintaining the natural appearance of the image.

2.1.5 Derive convolution kernels for derivatives

Consider the image as a function $I : \mathbb{R}^2 \to \mathbb{R}$. We define the discrete derivatives in the horizontal (x) and vertical (y) directions as follows:

$$I_x(x, y) = [I(x+1, y) - I(x-1, y)] \approx 2\frac{\partial I}{\partial x}$$
$$I_y(x, y) = [I(x, y+1) - I(x, y-1)] \approx 2\frac{\partial I}{\partial y}$$

To represent these derivatives as convolutions, we need to define the kernels k_x and k_y that capture the discrete difference operation. The convolution operation for the horizontal derivative can be written as:

$$I_x = I * k_x$$

where k_x is a row vector that subtracts adjacent pixel values along the x-direction. Since convolution is a weighted sum, we assign weights that reflect the discrete difference operation. Thus, we define k_x as:

$$k_x = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

Similarly, for the vertical derivative, the convolution is:

$$I_y = I * k_y$$

where k_y is a column vector that subtracts adjacent pixel values along the y-direction. Accordingly, k_y is defined as:

$$k_y = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

Sometimes a factor of $\frac{1}{2}$ might be included in both kernels to account for the fact that we are approximating the derivative by the difference between the pixels that are two units apart (hence, the derivative is twice the value of the difference). However, the constant factor can be adjusted depending on the specific implementation and scale used in the image processing.

2.1.6 edge_detection()

```python
def edge_detection(image):
    """
    Return Ix, Iy and the gradient magnitude of the input image

    Input- image: H x W
    Output- Ix, Iy, grad_magnitude: H x W
    """
    # TODO: Fix kx, ky
    kx = np.array([-1, 0, 1]).reshape(1, 3)  # 1 x 3
    ky = np.array([-1, 0, 1]).reshape(3, 1)  # 3 x 1

    Ix = convolve(image, kx)
    Iy = convolve(image, ky)

    # TODO: Use Ix, Iy to calculate grad_magnitude
    grad_magnitude = np.sqrt(Ix ** 2 + Iy ** 2)

    return Ix, Iy, grad_magnitude
```

2.1.7 Result and Discussion

Result of edge detection for original image shown figure3 Result of edge detection for Gaussian filtered image shown figure4

Figure 3: Edge detection for original image

Figure 4: Edge detection for Gaussian filtering image

Although the images having tiny difference, we can still see that the edge detection on Gaussian filtered image has less edges than original one, which complies more with our goal. For example, the wrinkles on the sleeve are less detected so that the edge of human and object is less noisy. However, this may also remove some edges we desire to preserve.
Smoothing the figure before the edge detection is benificial since the smoothing remove the noise that will cause smaller derivative. The removal of high frequency noise will make the edge detection more clear.

2.1.8 bilateral_filter() and result

```
def bilateral_filter(image, window_size, sigma_d, sigma_r):
    """
    Return filtered image using a bilateral filter

    Input-  image: H x W
            window_size: (h, w)
            sigma_d: sigma for the spatial kernel
            sigma_r: sigma for the range kernel
    Output- output: filtered image
    """
    # TODO: complete the bilateral filtering, assuming spatial and range
    kernels are gaussian
    H, W = image.shape
    h, w = window_size
    output = np.zeros_like(image, dtype=image.dtype)

    padding = [h//2, w//2]
    padded_image = np.zeros((H + 2 * padding[0], W + 2 * padding[1]),
    dtype=image.dtype)
    padded_image[padding[0] : H + padding[0], padding[1] : W + padding[1]]
    = image

    range_x = np.arange(-int(w / 2), int(w / 2) + 1)
    range_y = np.arange(-int(h / 2), int(h / 2) + 1)
    mesh_x, mesh_y = np.meshgrid(range_x, range_y)
    dis_mat = mesh_x **2 + mesh_y **2
```

```
    # pdb.set_trace()

    for y in range(H):
        for x in range(W):
            term1 = - dis_mat / (2 * sigma_d ** 2)
            # pdb.set_trace()
            image_in_kernel = padded_image[y : y + h, x : x + w]
            term2 = - ( np.linalg.norm((image[y, x] -  image_in_kernel), keepdims=True) ** 2 / (2 * sigma_r ** 2))
            w_ij = np.exp(term1 + term2)
            output[y, x] = (image_in_kernel * w_ij).sum() / w_ij.sum()
            # pdb.set_trace()

    return output
```

Result see figure5.

Figure 5: Bilateral filtered image

2.2 Task 3: Sobel Operator

2.2.1 Show relateion between Sobel and Gaussian kernel

The Sobel filter for the x-direction is given by:

$$S_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

And the Gaussian kernel G_s is given by:

$$G_s = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

We want to prove that:

$$(I * G_s) * k_x = I * (G_s * k_x) = I * S_x \implies (G_s * k_x) = S_x$$

where k_x is the horizontal filter that we need to derive.
We assume k_x to be of the form:

$$k_x = \begin{bmatrix} a & b & c \end{bmatrix}$$

Upon solving, first assume $k_x = \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$ based on the structure of the Sobel filter. This gives us:

$$beginalign]G_s * k_x = \begin{bmatrix} 2 & 0 & -2 \\ 4 & 0 & -4 \\ 2 & 0 & -2 \end{bmatrix}$$

where, the padding is zero padding.
After doing some normalization, the result matches the Sobel S_x filter exactly. Therefore, we conclude that applying the Sobel filter S_x to image I after Gaussian filtering with G_s is a analog to taking the horizontal derivative of the Gaussian-filtered image.

2.2.2 sobel_operator()

```python
def sobel_operator(image):
    """
    Return Gx, Gy, and the gradient magnitude.

    Input- image: H x W
    Output- Gx, Gy, grad_magnitude: H x W
    """
    # TODO: Use convolve() to complete the function
    Gx, Gy, grad_magnitude = None, None, None
    S_x = np.array([[1, 0, -1],
                    [2, 0, -2],
                    [1, 0, -1],])

    S_y = np.array([[1, 2, 1],
                    [0, 0, 0],
                    [-1, -2, -1],])

    Gx = convolve(image, S_x)
    Gy = convolve(image, S_y)
    grad_magnitude = np.sqrt(Gx ** 2  + Gy ** 2)

    return Gx, Gy, grad_magnitude
```

Results see figure 6, 7, and 8.

2.2.3 Result

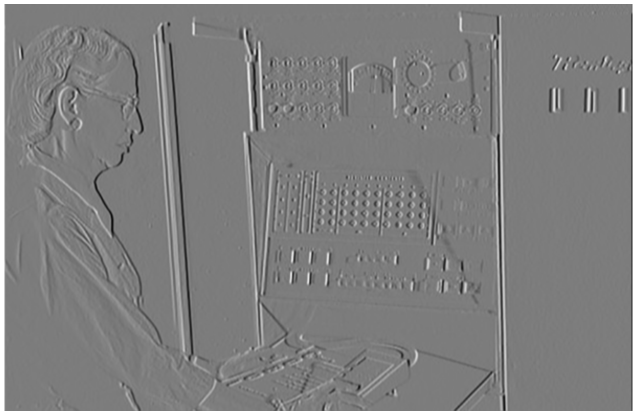

Figure 6: Sobel filter: I * Sx

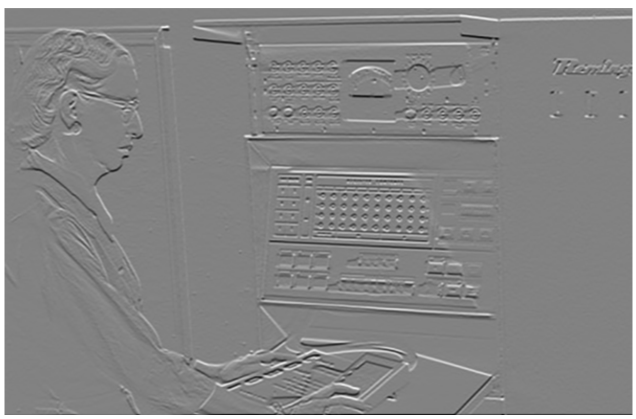

Figure 7: Sobel filter: I * Sy

Figure 8: Sobel filter: gradient magnitude

2.3 Task 4: LoG Filter

Result see figure 9, 10.

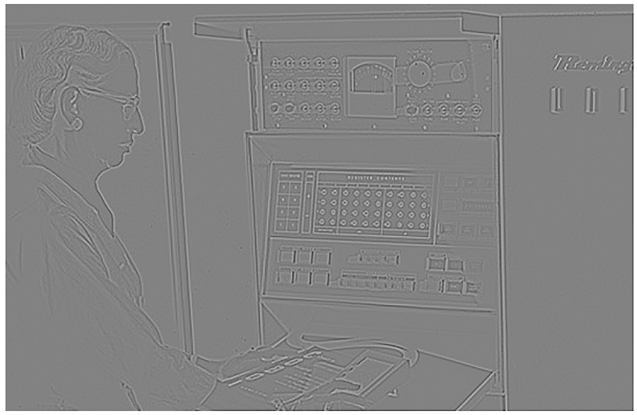

Figure 9: LoG filter 1

Figure 10: LoG filter 2

The figure given by the LoG filter 2 is more clear than LoG filter 1. The reason for this is that the blob size of the original figure have higher alignment with the LoG filter 2 which has a larger covariance value, in other word, the original figure has higher response to LoG filter 2.

Yes, these filters can detect edges, but the filter covariance need to be manually tuned to find the best response. They can also used for blob detection to detect patterns in figure.

2.3.1 Approximate LoG

The Laplace of Gaussian (LoG) of image f can be written as

$$\nabla^2(f * g) = f * \nabla^2 g$$

That is, the Laplace of the image smoothed by a Gaussian kernel is identical to the image convolved with the Laplace of the Gaussian kernel. This convolution can be further expanded, in the 2D case, as

$$f * \nabla^2 g = f * \left(\frac{\partial^2}{\partial x^2} g + \frac{\partial^2}{\partial y^2} g \right) = f * \frac{\partial^2}{\partial x^2} g + f * \frac{\partial^2}{\partial y^2} g$$

The approximation of the LoG by the DoG can be understood by considering the Taylor series expansion of $G(x, k\sigma)$ around $G(x, \sigma)$. This expansion involves the second derivative of the Gaussian function, which is what the LoG operator essentially captures.
Convolution is a linear operation:

$$(A * G_1) - (B * G_2) = (A - B) * G \tag{1-5}$$

for functions G_1, G_2, and G, and constants A and B.
Thus, the convolution of the image with the DoG function approximates the convolution of the image with the LoG function due to the linearity of convolution and the properties of Gaussian functions and their derivatives.
See figure 11.

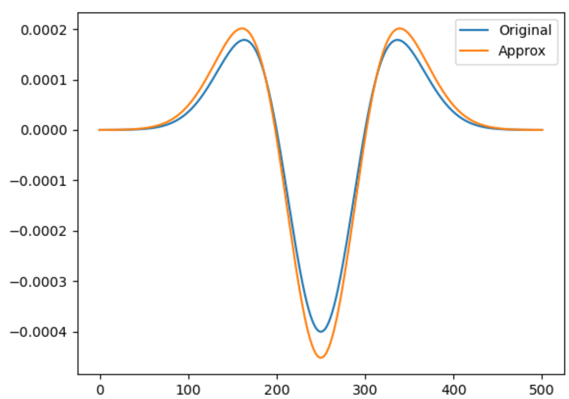

Figure 11: LoG and DoG

2.4 Task 5: Who's That Filter?

2.4.1 What is the filters?

The result is:

```
    filter1 = np.ones((3, 3))
    filter2 = np.ones((3, 3)) * 1/9
    filter3 = np.array([[0, 0, 0],
                        [1, 0, 0],
                        [0, 0, 0]])
    filter4 = np.array([[-1, 0, 1],
                        [-1, 0, 1],
                        [-1, 0, 1]])
```

2.4.2 Function of filter 1

Filter 1 does a blurring to the image. It is different from filter 2 since it is not normalized, but kernel filter 2 is normalized.

3 Feature Extraction

3.1 Task 6: Coner Score

3.1.1 corner_score()

```python
def corner_score(image, u=5, v=5, window_size=(5, 5)):
    """
    Given an input image, x_offset, y_offset, and window_size,
    return the function E(u,v) for window size W
    corner detector score for that pixel.
    Use zero-padding to handle window values outside of the image.

    Input- image: H x W
           u: a scalar for x offset
           v: a scalar for y offset
           window_size: a tuple for window size

    Output- results: a image of size H x W
    """
    output = np.zeros_like(image)
    H, W = image.shape
    h, w = window_size

    shifted_image = np.roll(image, (u, v), axis=(1, 0))

    padding = (h//2, w//2)

    padded_image = np.zeros((H + 2 * padding[0], W + 2 * padding[1]), dtype=image.dtype)
    padded_image[padding[0] : padding[0] + H, padding[1] : padding[1] + W] = image
    padded_shifted_image = np.zeros((H + 2 * padding[0], W + 2 * padding[1]), dtype=image.dtype)
    padded_shifted_image[padding[0] : padding[0] + H, padding[1] : padding[1] + W] = shifted_image

    for y in range(H):
        for x in range(W):
            e = np.sum((padded_shifted_image[y : y + h, x : x + w] - padded_image[y : y + h , x : x + h]) ** 2)
            output[y, x] = e

    return output
```

3.1.2 Result

Result see figure 12 to 15

Figure 12: Corner score with (u, v) = (0, 5)

Figure 13: Corner score with (u, v) = (0, -5)

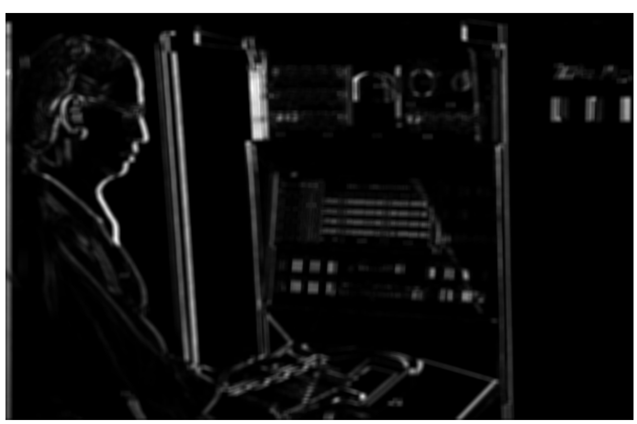

Figure 14: Corner score with (u, v) = (5, 0)

Figure 15: Corner score with (u, v) = (-5, 0)

3.1.3 Discuss

If a figure has size $H*W$, and the kernel used is $h*w$, the time complexity will be $O(HWhw)$, which is a large high order term for the computer back to the 80s.

3.2 Task 7: Harris Corner Detector

3.2.1 harris_detector()

```
def harris_detector(image, window_size=(5, 5)):
    """
    Given an input image, calculate the Harris Detector score for all pixels
    You can use same-padding for intensity (or 0-padding for derivatives)
    to handle window values outside of the image.

    Input- image: H x W
    Output- results: a image of size H x W
    """
    # compute the derivatives
    kx = np.array([-1, 0, 1]).reshape(1, 3)
    ky = np.array([-1, 0, 1]).reshape(3, 1)
    Ix = scipy.ndimage.convolve(image, kx, mode='constant', cval=0)
    Iy = scipy.ndimage.convolve(image, ky, mode='constant', cval=0)

    Ixx = Ix ** 2
    Iyy = Iy ** 2
    Ixy = Ix * Iy

    # For each image location, construct the structure tensor and calculate
    # the Harris response
    M = np.zeros((3, image.shape[0], image.shape[1]))

    # for y in range(image.shape[0]):
    #     for x in range(image.shape[1]):
    #         # import pdb; pdb.set_trace()
```

```
    kernel = np.ones(window_size)
    M[0] = scipy.ndimage.convolve(Ixx, kernel, mode='constant', cval=0)
    M[1] = scipy.ndimage.convolve(Ixy, kernel, mode='constant', cval=0)
    M[2] = scipy.ndimage.convolve(Iyy, kernel, mode='constant', cval=0)

    alpha = 0.05

    response = M[0] * M[2] - M[1] ** 2 - alpha * (M[0] + [2]) ** 2

    return response
```

3.2.2 Result

Result see figure 16

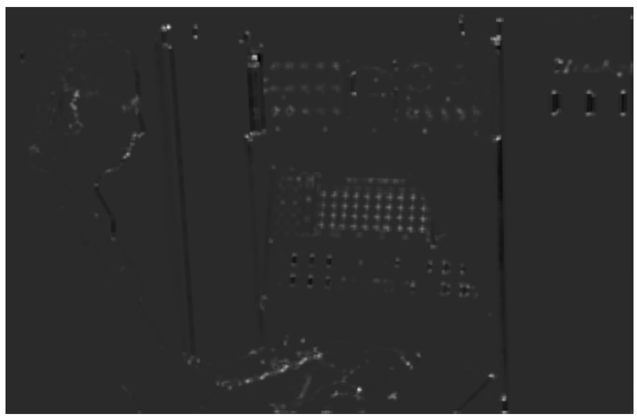

Figure 16: Harris response

4 Blob Detection

4.1 Task 8: Single-Scale Blob Detection

4.1.1 gaussian_filter()

```
def gaussian_filter(image, sigma):
    """
    Given an image, apply a Gaussian filter with the input kernel size
    and standard deviation

    Input
      image: image of size HxW
      sigma: scalar standard deviation of Gaussian Kernel

    Output
      Gaussian filtered image of size HxW
    """
    H, W = image.shape
    # -- good heuristic way of setting kernel size
```

```python
    kernel_size = int(2 * np.ceil(2 * sigma) + 1)
    # Ensure that the kernel size isn't too big and is odd
    kernel_size = min(kernel_size, min(H, W) // 2)
    if kernel_size % 2 == 0:
        kernel_size = kernel_size + 1
    # TODO implement gaussian filtering of size kernel_size x kernel_size
    # Similar to Corner detection, use scipy's convolution function.
    # Again, be consistent with the settings (mode = 'reflect').

    # create gaussian kernel
    ax = np.linspace(-(kernel_size - 1) / 2., (kernel_size - 1) / 2., kernel_size)
    xx, yy = np.meshgrid(ax, ax)
    kernel = np.exp(-0.5 * (np.square(xx) + np.square(yy)) / np.square(sigma))
    kernel /= np.sum(kernel)

    output = scipy.ndimage.convolve(image, kernel, mode='reflect')
    return output
```

4.1.2 Result

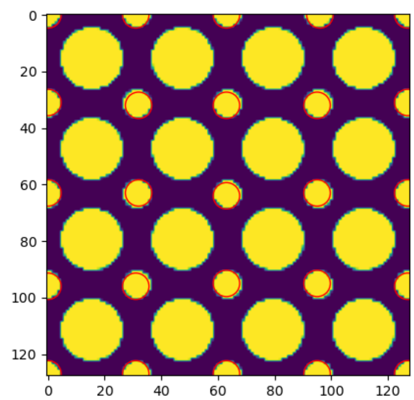

Figure 17: Single-scale blob detection - small

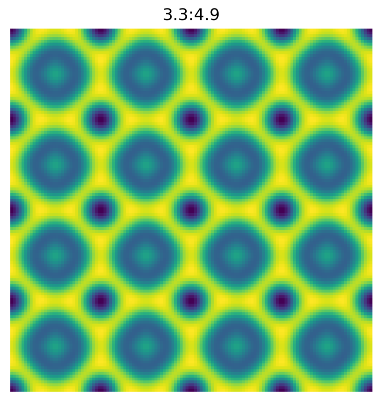

Figure 18: Single-scale blob detection DoG - small

The parameter used for small blob:

```
    k = 5
    sigma_1 = 3.7
    sigma_2 = k * sigma_1
```

There are 25 maxima observed for small blob. No false peak.

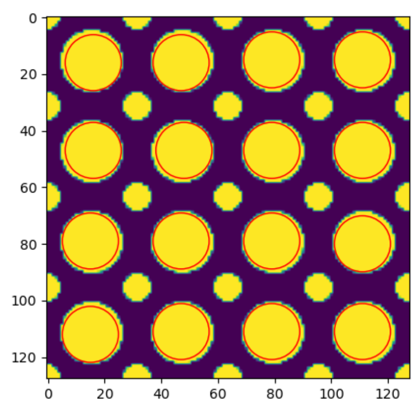

Figure 19: Single-scale blob detection - large

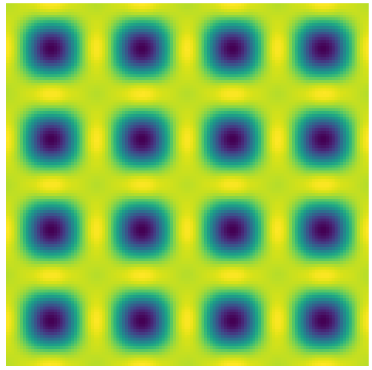

Figure 20: Single-scale blob detection DoG - large

The parameter used for large blob:

```
    k = 1.9
    sigma_1 = 3.7
    sigma_2 = k * sigma_1
```

There are 17 maxima observed for large blob. No false peak.

4.2 Task 9: Cell Counting

4.2.1 Parameters and Results

Parameters for them are:

```
    #"Detecting cell1 -- 008cell"
    k = 3
    sigma_1 = 3.7
    sigma_2 = k * sigma_1

    #"Detecting cell2 -- 004cell"
    k = 3
    sigma_1 = 3.4
    sigma_2 = k * sigma_1

    #"Detecting cell3 -- 005cell"
    k = 5
    sigma_1 = 3.7
    sigma_2 = k * sigma_1

    #"Detecting cell4 -- 006cell"
    k = 1.9
    sigma_1 = 3.7
    sigma_2 = k * sigma_1
```

The numbers of blobs are 102, 95, 105 and 158 respectively.

4.2.2 Plots and Discussion

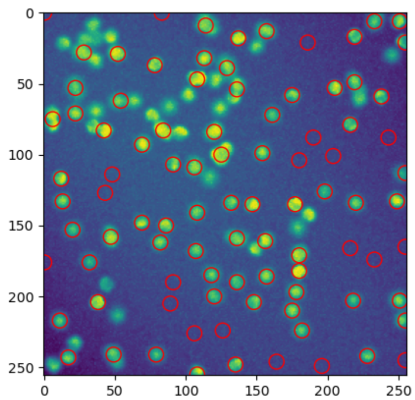

Figure 21: Blob detection - cell1

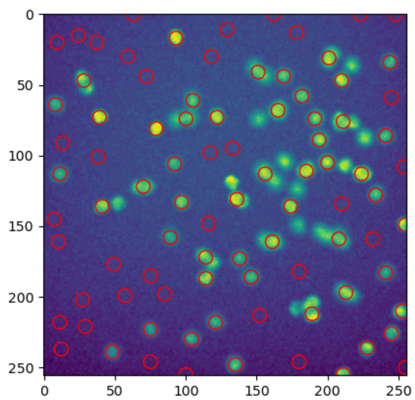

Figure 22: Blob detection - cell2

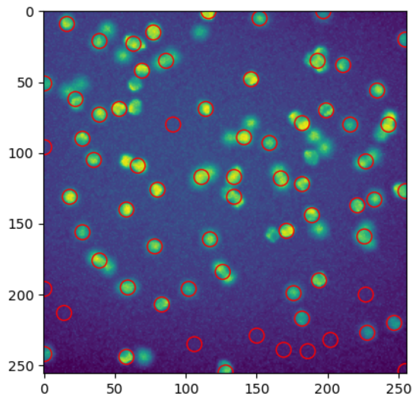

Figure 23: Blob detection - cell3

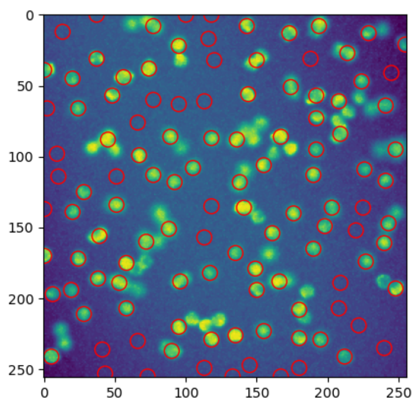

Figure 24: Blob detection - cell4

Increasing sigma will allow more cells be detected, but also increase the probability of false detection. Increasing k will make the false detection less, but correct detection will also be eliminate.
Iterating or using gradient to find the best sigma and k might be a way to make this cell detection more universal applicable.

5 Appendix

5.0.1 filters.py

```
import os

import numpy as np

from common import read_img, save_img

import pdb
import cv2
import matplotlib.pyplot as plt


def image_patches(image, patch_size=(16, 16)):
    """
    Given an input image and patch_size,
    return the corresponding image patches made
    by dividing up the image into patch_size sections.

    Input- image: H x W
           patch_size: a scalar tuple M, N
    Output- results: a list of images of size M x N
    """
    # TODO: Use slicing to complete the function
    output = []
    H, W = image.shape
    h, w = patch_size
    num_h = H // h
    num_w = W // w

    for i in range(num_h):
        for j in range(num_w):
            patch = image[i * h : (i + 1) * h, i * w : (i + 1) * w]
            patch_mean = np.mean(patch)
            patch_std = np.std(patch)
            patch = (patch - patch_mean) / patch_std
            patch = np.nan_to_num(patch, nan=0.0, posinf=0.0, neginf=0.0)
            output.append(patch)
    # import pdb; pdb.set_trace()
    return output


def convolve(image, kernel):
    """
    Return the convolution result: image * kernel.
    Reminder to implement convolution and not cross-correlation!
    Caution: Please use zero-padding.

    Input- image: H x W
           kernel: h x w
    Output- convolve: H x W
    """
```

```python
    output = np.zeros_like(image)
    if len(kernel.shape) == 2:
        kernel = kernel[::-1, ::-1]
    elif len(kernel.shape) == 1:
        kernel = kernel[::-1]

    H, W = image.shape
    h, w = kernel.shape

    padding = [h//2, w//2]
    padded_image = np.zeros((H + 2 * padding[0], W + 2 * padding[1]), dtype=image.dtype)
    padded_image[padding[0] : H + padding[0], padding[1] : W + padding[1]] = image

    for y in range(H):
        for x in range(W):
            patch = padded_image[y : y + h, x : x + w]
            output[y, x] = np.sum(patch * kernel)

    return output


def edge_detection(image):
    """
    Return Ix, Iy and the gradient magnitude of the input image

    Input- image: H x W
    Output- Ix, Iy, grad_magnitude: H x W
    """
    # TODO: Fix kx, ky
    kx = np.array([-1, 0, 1]).reshape(1, 3)  # 1 x 3
    ky = np.array([-1, 0, 1]).reshape(3, 1)  # 3 x 1

    Ix = convolve(image, kx)
    Iy = convolve(image, ky)

    # TODO: Use Ix, Iy to calculate grad_magnitude
    grad_magnitude = np.sqrt(Ix ** 2 + Iy ** 2)

    return Ix, Iy, grad_magnitude


def sobel_operator(image):
    """
    Return Gx, Gy, and the gradient magnitude.

    Input- image: H x W
    Output- Gx, Gy, grad_magnitude: H x W
    """
    # TODO: Use convolve() to complete the function
    Gx, Gy, grad_magnitude = None, None, None
    S_x = np.array([[1, 0, -1],
                    [2, 0, -2],
```

```
                    [1, 0, -1],])

    S_y = np.array([[1, 2, 1],
                    [0, 0, 0],
                    [-1, -2, -1],])

    Gx = convolve(image, S_x)
    Gy = convolve(image, S_y)
    grad_magnitude = np.sqrt(Gx ** 2  + Gy ** 2)

    return Gx, Gy, grad_magnitude

def bilateral_filter(image, window_size, sigma_d, sigma_r):
    """
    Return filtered image using a bilateral filter

    Input-  image: H x W
            window_size: (h, w)
            sigma_d: sigma for the spatial kernel
            sigma_r: sigma for the range kernel
    Output- output: filtered image
    """
    # TODO: complete the bilateral filtering, assuming spatial and range kernels are gaussian
    H, W = image.shape
    h, w = window_size
    output = np.zeros_like(image, dtype=image.dtype)

    padding = [h//2, w//2]
    padded_image = np.zeros((H + 2 * padding[0], W + 2 * padding[1]), dtype=image.dtype)
    padded_image[padding[0] : H + padding[0], padding[1] : W + padding[1]] = image

    range_x = np.arange(-int(w / 2), int(w / 2) + 1)
    range_y = np.arange(-int(h / 2), int(h / 2) + 1)
    mesh_x, mesh_y = np.meshgrid(range_x, range_y)
    dis_mat = mesh_x **2 + mesh_y **2
    # pdb.set_trace()

    for y in range(H):
        for x in range(W):
            term1 = - dis_mat / (2 * sigma_d ** 2)
            # pdb.set_trace()
            image_in_kernel = padded_image[y : y + h, x : x + w]
            term2 = - ( np.linalg.norm((image[y, x] -  image_in_kernel), keepdims=True) ** 2 / (2 * sigma_r ** 2))
            w_ij = np.exp(term1 + term2)
            output[y, x] = (image_in_kernel * w_ij).sum() / w_ij.sum()
            # pdb.set_trace()

    return output
```

```python
def main():
    # The main function
    img = read_img('./grace_hopper.png')
    """ Image Patches """
    if not os.path.exists("./image_patches"):
        os.makedirs("./image_patches")

    # -- TODO Task 1: Image Patches --
    # (a)
    # First complete image_patches()
    patches = image_patches(img)
    # Now choose any three patches and save them
    # chosen_patches should have those patches stacked vertically/horizontally
    idxs = [np.random.randint(0, len(patches)) for _ in range(3)]
    # print(idxs)
    chosen_patches = np.array([patches[i] for i in idxs])
    chosen_patches = chosen_patches.reshape(16, -1)
    # import pdb; pdb.set_trace()
    save_img(chosen_patches, "./image_patches/q1_patch.png")

    # (b), (c): No code

    """ Convolution and Gaussian Filter """
    if not os.path.exists("./gaussian_filter"):
        os.makedirs("./gaussian_filter")

    # -- TODO Task 2: Convolution and Gaussian Filter --
    # (a): No code

    # (b): Complete convolve()

    # (c)
    # Calculate the Gaussian kernel described in the question.
    # There is tolerance for the kernel.
    kernel_size = 3
    kernel_sigma = 0.572
    # kernel_sigma = 2
    kernel_range = np.arange(-int(kernel_size / 2), int(kernel_size / 2) + 1)
    kernel_x, kernel_y = np.meshgrid(kernel_range, kernel_range)
    kernel_gaussian = np.exp(- (kernel_x ** 2 + kernel_y ** 2) / (2 * kernel_sigma ** 2))
    kernel_gaussian /= kernel_gaussian.sum()
    # print(kernel_gaussian.sum())
    # pdb.set_trace()
    filtered_gaussian = convolve(img, kernel_gaussian)
    save_img(filtered_gaussian, "./gaussian_filter/q2_gaussian.png")

    # (d), (e): No code

    # (f): Complete edge_detection()

    # (g)
```

```
    # Use edge_detection() to detect edges
    # for the orignal and gaussian filtered images.
    _, _, edge_detect = edge_detection(img)
    save_img(edge_detect, "./gaussian_filter/q3_edge.png")
    _, _, edge_with_gaussian = edge_detection(filtered_gaussian)
    save_img(edge_with_gaussian, "./gaussian_filter/q3_edge_gaussian.png")

    print("Gaussian Filter is done. ")

    # (h) complete biliateral_filter()
    if not os.path.exists("./bilateral"):
        os.makedirs("./bilateral")

    image_bilataral_filtered = bilateral_filter(img, (5, 5), 3, 75)
    img_cv2 = cv2.imread('./grace_hopper.png')
    image_bilataral_filtered_cv2 = cv2.bilateralFilter(img_cv2, 5, 75, 3)
    save_img(image_bilataral_filtered, "./bilateral/bilateral_output.png")
    save_img(image_bilataral_filtered_cv2, "./bilateral/
bilateral_output_cv2.png")

    # -- TODO Task 3: Sobel Operator --
    if not os.path.exists("./sobel_operator"):
        os.makedirs("./sobel_operator")

    # (a): No code

    # (b): Complete sobel_operator()

    # (c)
    Gx, Gy, edge_sobel = sobel_operator(img)
    save_img(Gx, "./sobel_operator/q2_Gx.png")
    save_img(Gy, "./sobel_operator/q2_Gy.png")
    save_img(edge_sobel, "./sobel_operator/q2_edge_sobel.png")

    print("Sobel Operator is done. ")

    # -- TODO Task 4: LoG Filter --
    if not os.path.exists("./log_filter"):
        os.makedirs("./log_filter")

    # (a)
    kernel_LoG1 = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]])
    kernel_LoG2 = np.array([[0, 0, 3, 2, 2, 2, 3, 0, 0],
                            [0, 2, 3, 5, 5, 5, 3, 2, 0],
                            [3, 3, 5, 3, 0, 3, 5, 3, 3],
                            [2, 5, 3, -12, -23, -12, 3, 5, 2],
                            [2, 5, 0, -23, -40, -23, 0, 5, 2],
                            [2, 5, 3, -12, -23, -12, 3, 5, 2],
                            [3, 3, 5, 3, 0, 3, 5, 3, 3],
                            [0, 2, 3, 5, 5, 5, 3, 2, 0],
                            [0, 0, 3, 2, 2, 2, 3, 0, 0]])
    filtered_LoG1 = convolve(img, kernel_LoG1)
    filtered_LoG2 = convolve(img, kernel_LoG2)
    # Use convolve() to convolve img with kernel_LOG1 and kernel_LOG2
```

```
    save_img(filtered_LoG1, "./log_filter/q1_LoG1.png")
    save_img(filtered_LoG2, "./log_filter/q1_LoG2.png")

    # (b)
    # Follow instructions in pdf to approximate LoG with a DoG
    data = np.load('log1d.npz')
    plt.figure(1)
    plt.plot(data['log50'])
    plt.plot(data['gauss53'] - data['gauss50'])
    plt.legend(['Original', 'Approx'])
    plt.show()
    print("LoG Filter is done. ")


if __name__ == "__main__":
    main()
```

5.0.2 filtersmon.py

```
import numpy as np
import scipy.signal
import matplotlib.pyplot as plt


def conv(I, f):
    """Apply same-sized convolution with a filter with zero-padding"""
    # Note that this is convolution! This is filtering but with f
   [::-1,::-1]
    return scipy.signal.convolve2d(
        I, f, mode='same', boundary='fill', fillvalue=0.0)


def nnUpsample(I, factor):
    """Nearest neighbor upsample an image by the given factor"""
    return np.kron(I, np.ones((factor, factor)))

# -- TODO Task 5: Who's That Filter? --
# (a): Fill in the filters to get the data to match

filter0 = np.diag([0, 1, 0])
filter1 = np.ones((3, 3))
filter2 = np.ones((3, 3)) * 1/9
filter3 = np.array([[0, 0, 0],
                    [1, 0, 0],
                    [0, 0, 0]])
filter4 = np.array([[-1, 0, 1],
                    [-1, 0, 1],
                    [-1, 0, 1]])

# (b): No code

filters = [filter0, filter1, filter2, filter3, filter4]

```

```python
np.random.seed(442)
data = (plt.imread("filtermon/442.png").astype(float)
        [:, :, 0] < 0.5).astype(float)


plt.figure()
plt.imshow(nnUpsample(data, 10))
plt.colorbar()
plt.savefig("input.png")


for fi, f in enumerate(filters):
    c = conv(data, f)
    sol = np.load("filtermon/output_%d.npy" % fi)
    plt.imsave(f"./filtermon/output_{fi}.png", sol)
    matches = False

    if np.allclose(c, sol, rtol=1e-2, atol=1e-5):
        print("Filter %d matches" % fi)
        matches = True
    else:
        print("Filter %d doesn't match" % fi)

    plt.figure()
    fig, axs = plt.subplots(1, 2)

    im = axs[0].imshow(nnUpsample(c, 10))
    axs[0].set_title("Yours (%s)" % ("Match!" if matches else "No Match"))
    plt.colorbar(im, ax=axs[0])

    im = axs[1].imshow(nnUpsample(sol, 10))
    axs[1].set_title("Target")
    plt.colorbar(im, ax=axs[1])

    plt.tight_layout()
    plt.savefig("comparison_%d.pdf" % (fi))
```

5.0.3 corners.py

```python
import os

import numpy as np
import scipy.ndimage
# Use scipy.ndimage.convolve() for convolution.
# Use zero padding (Set mode = 'constant'). Refer docs for further info.

from common import read_img, save_img


def corner_score(image, u=5, v=5, window_size=(5, 5)):
    """
    Given an input image, x_offset, y_offset, and window_size,
    return the function E(u,v) for window size W
```

```
    corner detector score for that pixel.
    Use zero-padding to handle window values outside of the image.

    Input- image: H x W
           u: a scalar for x offset
           v: a scalar for y offset
           window_size: a tuple for window size

    Output- results: a image of size H x W
    """
    output = np.zeros_like(image)
    H, W = image.shape
    h, w = window_size

    shifted_image = np.roll(image, (u, v), axis=(1, 0))

    padding = (h//2, w//2)

    padded_image = np.zeros((H + 2 * padding[0], W + 2 * padding[1]), dtype=image.dtype)
    padded_image[padding[0] : padding[0] + H, padding[1] : padding[1] + W] = image
    padded_shifted_image = np.zeros((H + 2 * padding[0], W + 2 * padding[1]), dtype=image.dtype)
    padded_shifted_image[padding[0] : padding[0] + H, padding[1] : padding[1] + W] = shifted_image

    for y in range(H):
        for x in range(W):
            e = np.sum((padded_shifted_image[y : y + h, x : x + w] - padded_image[y : y + h , x : x + h]) ** 2)
            output[y, x] = e

    return output


def harris_detector(image, window_size=(5, 5)):
    """
    Given an input image, calculate the Harris Detector score for all pixels
    You can use same-padding for intensity (or 0-padding for derivatives)
    to handle window values outside of the image.

    Input- image: H x W
    Output- results: a image of size H x W
    """
    # compute the derivatives
    kx = np.array([-1, 0, 1]).reshape(1, 3)
    ky = np.array([-1, 0, 1]).reshape(3, 1)
    Ix = scipy.ndimage.convolve(image, kx, mode='constant', cval=0)
    Iy = scipy.ndimage.convolve(image, ky, mode='constant', cval=0)

    Ixx = Ix ** 2
    Iyy = Iy ** 2
```

```python
    Ixy = Ix * Iy

    # For each image location, construct the structure tensor and calculate
    # the Harris response
    M = np.zeros((3, image.shape[0], image.shape[1]))

    # for y in range(image.shape[0]):
    #     for x in range(image.shape[1]):
    #         # import pdb; pdb.set_trace()
    kernel = np.ones(window_size)
    M[0] = scipy.ndimage.convolve(Ixx, kernel, mode='constant', cval=0)
    M[1] = scipy.ndimage.convolve(Ixy, kernel, mode='constant', cval=0)
    M[2] = scipy.ndimage.convolve(Iyy, kernel, mode='constant', cval=0)

    alpha = 0.05

    response = M[0] * M[2] - M[1] ** 2 - alpha * (M[0] + [2]) ** 2

    return response


def main():
    img = read_img('./grace_hopper.png')

    # Feature Detection
    if not os.path.exists("./feature_detection"):
        os.makedirs("./feature_detection")

    # -- TODO Task 6: Corner Score --
    # (a): Complete corner_score()

    # (b)
    # Define offsets and window size and calulcate corner score
    W = (5, 5)
    tuples = ((0, 5), (0, -5), (5, 0), (-5, 0))
    for i, (u, v) in enumerate(tuples):
        score = corner_score(img, u, v, W)
        save_img(score, f"./feature_detection/corner_score_{i}.png")

    # (c): No Code

    # -- TODO Task 7: Harris Corner Detector --
    # (a): Complete harris_detector()

    # (b)
    harris_corners = harris_detector(img)
    save_img(harris_corners, "./feature_detection/harris_response.png")


if __name__ == "__main__":
    main()
```

5.0.4 blob_detection.py

```python
import os
import matplotlib.pyplot as plt
import numpy as np
import scipy.ndimage
# Use scipy.ndimage.convolve() for convolution.
# Use same padding (mode = 'reflect'). Refer docs for further info.

from common import (find_maxima, read_img, visualize_maxima,
                    visualize_scale_space)


def gaussian_filter(image, sigma):
    """
    Given an image, apply a Gaussian filter with the input kernel size
    and standard deviation

    Input
      image: image of size HxW
      sigma: scalar standard deviation of Gaussian Kernel

    Output
      Gaussian filtered image of size HxW
    """
    H, W = image.shape
    # -- good heuristic way of setting kernel size
    kernel_size = int(2 * np.ceil(2 * sigma) + 1)
    # Ensure that the kernel size isn't too big and is odd
    kernel_size = min(kernel_size, min(H, W) // 2)
    if kernel_size % 2 == 0:
        kernel_size = kernel_size + 1
    # TODO implement gaussian filtering of size kernel_size x kernel_size
    # Similar to Corner detection, use scipy's convolution function.
    # Again, be consistent with the settings (mode = 'reflect').

    # create gaussian kernel
    ax = np.linspace(-(kernel_size - 1) / 2., (kernel_size - 1) / 2.,
    kernel_size)
    xx, yy = np.meshgrid(ax, ax)
    kernel = np.exp(-0.5 * (np.square(xx) + np.square(yy)) / np.square(
    sigma))
    kernel /= np.sum(kernel)

    output = scipy.ndimage.convolve(image, kernel, mode='reflect')
    return output




def main():
    image = read_img('polka.png')
    # import pdb; pdb.set_trace()
    # Create directory for polka_detections
```

```python
    if not os.path.exists("./polka_detections"):
        os.makedirs("./polka_detections")

    # -- TODO Task 8: Single-scale Blob Detection --

    # (a), (b): Detecting Polka Dots
    # First, complete gaussian_filter()
    print("Detecting small polka dots")
    # -- Detect Small Circles
    k = 1.5
    sigma_1 = 3.3
    sigma_2 = k * sigma_1
    gauss_1 = gaussian_filter(image, sigma_1)  # to implement
    gauss_2 = gaussian_filter(image, sigma_2)  # to implement

    # calculate difference of gaussians
    DoG_small = gauss_2 - gauss_1  # to implement

    # visualize maxima
    maxima = find_maxima(DoG_small, k_xy=10)
    visualize_scale_space(DoG_small, sigma_1, sigma_2 / sigma_1,
                          './polka_detections/polka_small_DoG.png')
    visualize_maxima(image, maxima, sigma_1, sigma_2 / sigma_1,
                     './polka_detections/polka_small.png')
    plt.clf()


    # -- Detect Large Circles
    print("Detecting large polka dots")
    k = 1.5
    sigma_1 = 7
    sigma_2 = k * sigma_1
    gauss_1 = gaussian_filter(image, sigma_1)  # to implement
    gauss_2 = gaussian_filter(image, sigma_2)  # to implement

    # calculate difference of gaussians
    DoG_large = gauss_2 - gauss_1  # to implement

    # visualize maxima
    # Value of k_xy is a sugguestion; feel free to change it as you wish.
    maxima = find_maxima(DoG_large, k_xy=10)
    visualize_scale_space(DoG_large, sigma_1, sigma_2 / sigma_1,
                          './polka_detections/polka_large_DoG.png')
    visualize_maxima(image, maxima, sigma_1, sigma_2 / sigma_1,
                     './polka_detections/polka_large.png')
    plt.clf()

    # # # -- TODO Task 9: Cell Counting --
    print("Detecting cells")

    cell_1 = read_img("./cells/008cell.png")
    cell_2 = read_img("./cells/004cell.png")
    cell_3 = read_img("./cells/005cell.png")
    cell_4 = read_img("./cells/006cell.png")
```

```
    cells = [cell_1, cell_2, cell_3, cell_4]

    # Detect the cells in any four (or more) images from vgg_cells
    # Create directory for cell_detections
    if not os.path.exists("./cell_detections"):
        os.makedirs("./cell_detections")


    print("Detecting cell1")
    k = 3
    sigma_1 = 3.7
    sigma_2 = k * sigma_1
    gauss_1 = gaussian_filter(cell_1, sigma_1)  # to implement
    gauss_2 = gaussian_filter(cell_1, sigma_2)  # to implement

    # calculate difference of gaussians
    DoG_cell1 = gauss_2 - gauss_1  # to implement

    # visualize maxima
    maxima = find_maxima(DoG_cell1, k_xy=10)
    visualize_scale_space(DoG_cell1, sigma_1, sigma_2 / sigma_1,
                          './cell_detections/cell1_DoG.png')
    visualize_maxima(cell_1, maxima, sigma_1, sigma_2 / sigma_1,
                     './cell_detections/cell1.png')
    plt.clf()


    print("Detecting cell2")
    k = 3
    sigma_1 = 3.4
    sigma_2 = k * sigma_1
    gauss_1 = gaussian_filter(cell_2, sigma_1)  # to implement
    gauss_2 = gaussian_filter(cell_2, sigma_2)  # to implement

    # calculate difference of gaussians
    DoG_cell2 = gauss_2 - gauss_1  # to implement

    # visualize maxima
    maxima = find_maxima(DoG_cell2, k_xy=10)
    visualize_scale_space(DoG_cell2, sigma_1, sigma_2 / sigma_1,
                          './cell_detections/cell2_DoG.png')
    visualize_maxima(cell_2, maxima, sigma_1, sigma_2 / sigma_1,
                     './cell_detections/cell2.png')
    plt.clf()




    print("Detecting cell3")
    k = 5
    sigma_1 = 3.7
    sigma_2 = k * sigma_1
    gauss_1 = gaussian_filter(cell_3, sigma_1)  # to implement
    gauss_2 = gaussian_filter(cell_3, sigma_2)  # to implement
```

```python

    # calculate difference of gaussians
    DoG_cell3 = gauss_2 - gauss_1  # to implement

    # visualize maxima
    maxima = find_maxima(DoG_cell3, k_xy=10)
    visualize_scale_space(DoG_cell3, sigma_1, sigma_2 / sigma_1,
                          './cell_detections/cell3_DoG.png')
    visualize_maxima(cell_3, maxima, sigma_1, sigma_2 / sigma_1,
                     './cell_detections/cell3.png')
    plt.clf()




    print("Detecting cell4")
    k = 1.9
    sigma_1 = 3.7
    sigma_2 = k * sigma_1
    gauss_1 = gaussian_filter(cell_4, sigma_1)  # to implement
    gauss_2 = gaussian_filter(cell_4, sigma_2)  # to implement

    # calculate difference of gaussians
    DoG_cell4 = gauss_2 - gauss_1  # to implement

    # visualize maxima
    maxima = find_maxima(DoG_cell4, k_xy=10)
    visualize_scale_space(DoG_cell4, sigma_1, sigma_2 / sigma_1,
                          './cell_detections/cell4_DoG.png')
    visualize_maxima(cell_4, maxima, sigma_1, sigma_2 / sigma_1,
                     './cell_detections/cell4.png')
    plt.clf()



if __name__ == '__main__':
    main()
```