

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

```
1 def image_patches(image, patch_size=(16, 16)):  
2     """  
3     Given an input image and patch_size,  
4     return the corresponding image patches made  
5     by dividing up the image into patch_size sections.  
6  
7     Input- image: H x W  
8           patch_size: a scalar tuple M, N  
9     Output- results: a list of images of size M x N  
10    """  
11    # TODO: Use slicing to complete the function  
12    output = []  
13    H, W = image.shape  
14    h, w = patch_size  
15    num_h = H // h  
16    num_w = W // w  
17  
18    for i in range(num_h):  
19        for j in range(num_w):  
20            patch = image[i * h : (i + 1) * h, i * w : (i + 1) * w]  
21            patch_mean = np.mean(patch)  
22            patch_std = np.std(patch)  
23            patch = (patch - patch_mean) / patch_std  
24            patch = np.nan_to_num(patch, nan=0.0, posinf=0.0, neginf=0.0)  
25            output.append(patch)  
26    # import pdb; pdb.set_trace()  
27    return output
```

Result see figure1.

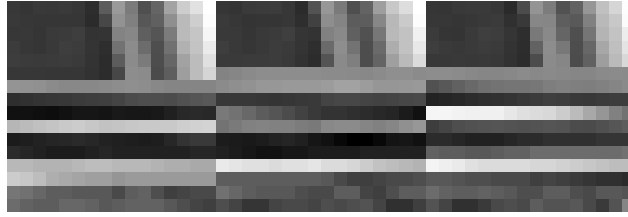


Figure 1: Image patches

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

$$G_x * G_y = G_x G_y = G \quad (1-1)$$

So we have:

$$G * X = (G_x G_y) * X \quad (1-2)$$

$$= (G_x * G_y) * x \quad (1-3)$$

$$= G_x * (G_y * X) \quad (1-4)$$

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

2.1.2 convolve()

```
1 def convolve(image, kernel):
2     """
3     Return the convolution result: image * kernel.
4     Reminder to implement convolution and not cross-correlation!
5     Caution: Please use zero-padding.
6
7     Input- image: H x W
8           kernel: h x w
9     Output- convolve: H x W
10    """
11    output = np.zeros_like(image)
12    if len(kernel.shape) == 2:
13        kernel = kernel[ : :-1 , : :-1]
14    elif len(kernel.shape) == 1:
15        kernel = kernel[ : :-1]
16
17    H, W = image.shape
18    h, w = kernel.shape
19
20    padding = [h//2, w//2]
21    padded_image = np.zeros((H + 2 * padding[0], W + 2 * padding[1]),
22                             dtype=image.dtype)
23    padded_image[padding[0] : H + padding[0], padding[1] : W + padding[1]]
24    = image
25
26    for y in range(H):
27        for x in range(W):
28            patch = padded_image[y : y + h , x : x + w]
29            output[y, x] = np.sum(patch * kernel)
```

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 $I : \mathbb{R}^2 \rightarrow \mathbb{R}$. 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:

$$k_x = [-1 \quad 0 \quad 1]$$

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()

```
1 def edge_detection(image):
2     """
3     Return Ix, Iy and the gradient magnitude of the input image
4
5     Input- image: H x W
6     Output- Ix, Iy, grad_magnitude: H x W
7     """
8     # TODO: Fix kx, ky
9     kx = np.array([-1, 0, 1]).reshape(1, 3) # 1 x 3
10    ky = np.array([-1, 0, 1]).reshape(3, 1) # 3 x 1
11
12    Ix = convolve(image, kx)
13    Iy = convolve(image, ky)
14
15    # TODO: Use Ix, Iy to calculate grad_magnitude
16    grad_magnitude = np.sqrt(Ix ** 2 + Iy ** 2)
17
18    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 beneficial 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

```

1 def bilateral_filter(image, window_size, sigma_d, sigma_r):
2     """
3     Return filtered image using a bilateral filter
4
5     Input-  image: H x W
6             window_size: (h, w)
7             sigma_d: sigma for the spatial kernel
8             sigma_r: sigma for the range kernel
9     Output- output: filtered image
10    """
11    # TODO: complete the bilateral filtering, assuming spatial and range
12    kernels are gaussian
13    H, W = image.shape
14    h, w = window_size
15    output = np.zeros_like(image, dtype=image.dtype)
16
17    padding = [h//2, w//2]
18    padded_image = np.zeros((H + 2 * padding[0], W + 2 * padding[1]),
19                             dtype=image.dtype)
20    padded_image[padding[0] : H + padding[0], padding[1] : W + padding[1]]
21    = image
22
23    range_x = np.arange(-int(w / 2), int(w / 2) + 1)
24    range_y = np.arange(-int(h / 2), int(h / 2) + 1)
25    mesh_x, mesh_y = np.meshgrid(range_x, range_y)
26    dis_mat = mesh_x **2 + mesh_y **2

```

```

24     # pdb.set_trace()
25
26     for y in range(H):
27         for x in range(W):
28             term1 = - dis_mat / (2 * sigma_d ** 2)
29             # pdb.set_trace()
30             image_in_kernel = padded_image[y : y + h, x : x + w]
31             term2 = - ( np.linalg.norm((image[y, x] - image_in_kernel),
32 keepdims=True) ** 2 / (2 * sigma_r ** 2))
33             w_ij = np.exp(term1 + term2)
34             output[y, x] = (image_in_kernel * w_ij).sum() / w_ij.sum()
35             # pdb.set_trace()
36     return output

```

Result see figure5.



Figure 5: Bilateral filtered image

2.2 Task 3: Sobel Operator

2.2.1 Show relation 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:

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

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()

```
1 def sobel_operator(image):
2     """
3     Return Gx, Gy, and the gradient magnitude.
4
5     Input- image: H x W
6     Output- Gx, Gy, grad_magnitude: H x W
7     """
8     # TODO: Use convolve() to complete the function
9     Gx, Gy, grad_magnitude = None, None, None
10    S_x = np.array([[1, 0, -1],
11                    [2, 0, -2],
12                    [1, 0, -1],])
13
14    S_y = np.array([[1, 2, 1],
15                    [0, 0, 0],
16                    [-1, -2, -1],])
17
18    Gx = convolve(image, S_x)
19    Gy = convolve(image, S_y)
20    grad_magnitude = np.sqrt(Gx ** 2 + Gy ** 2)
21
22    return Gx, Gy, grad_magnitude
```

Results see figure 6, 7, and 8.

2.2.3 Result

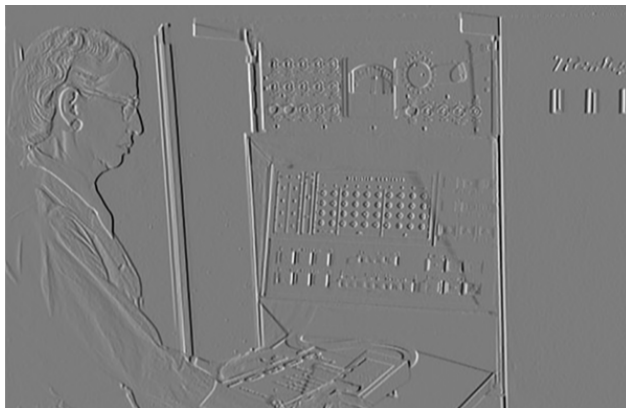


Figure 6: Sobel filter: $I * S_x$

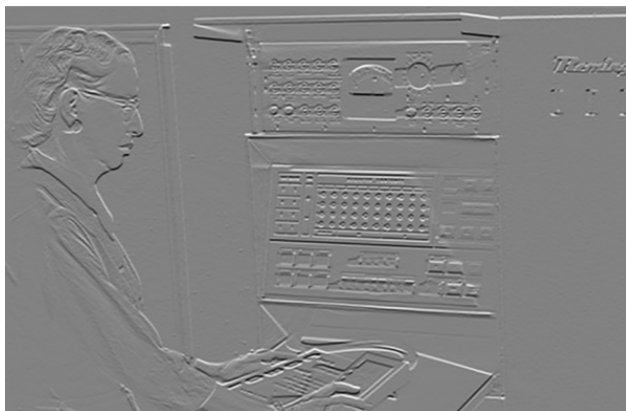


Figure 7: Sobel filter: $I * S_y$



Figure 8: Sobel filter: gradient magnitude

2.3 Task 4: LoG Filter

Result see figure 9, 10.

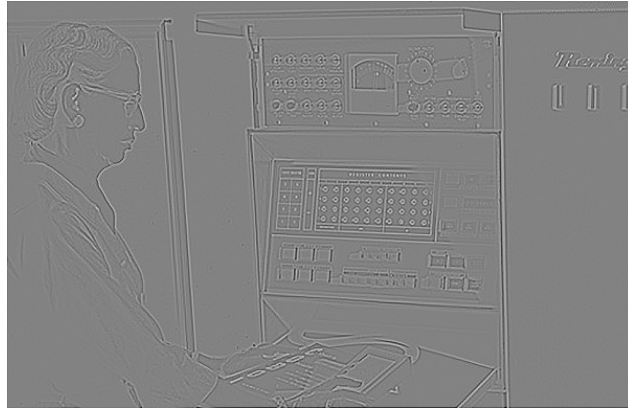


Figure 9: LoG filter 1



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:

$$(A * G_1) - (B * G_2) = (A - B) * G \quad (1-5)$$

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.

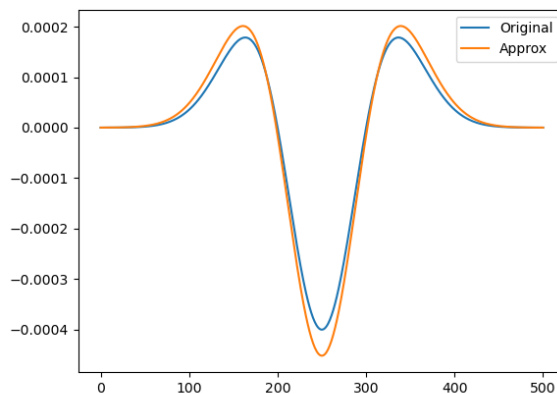


Figure 11: LoG and DoG

2.4 Task 5: Who's That Filter?

2.4.1 What is the filters?

The result is:

```

1  filter1 = np.ones((3, 3))
2  filter2 = np.ones((3, 3)) * 1/9
3  filter3 = np.array([[0, 0, 0],
4                      [1, 0, 0],
5                      [0, 0, 0]])
6  filter4 = np.array([[-1, 0, 1],
7                      [-1, 0, 1],
8                      [-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()

```
1 def corner_score(image, u=5, v=5, window_size=(5, 5)):
2     """
3     Given an input image, x_offset, y_offset, and window_size,
4     return the function E(u,v) for window size W
5     corner detector score for that pixel.
6     Use zero-padding to handle window values outside of the image.
7
8     Input- image: H x W
9           u: a scalar for x offset
10          v: a scalar for y offset
11          window_size: a tuple for window size
12
13     Output- results: a image of size H x W
14     """
15     output = np.zeros_like(image)
16     H, W = image.shape
17     h, w = window_size
18
19     shifted_image = np.roll(image, (u, v), axis=(1, 0))
20
21     padding = (h//2, w//2)
22
23     padded_image = np.zeros((H + 2 * padding[0], W + 2 * padding[1]),
24                             dtype=image.dtype)
25     padded_image[padding[0] : padding[0] + H, padding[1] : padding[1] + W]
26     = image
27     padded_shifted_image = np.zeros((H + 2 * padding[0], W + 2 * padding
28     [1]), dtype=image.dtype)
29     padded_shifted_image[padding[0] : padding[0] + H, padding[1] : padding
30     [1] + W] = shifted_image
31
32     for y in range(H):
33         for x in range(W):
34             e = np.sum((padded_shifted_image[y : y + h, x : x + w] -
35             padded_image[y : y + h, x : x + h]) ** 2)
36             output[y, x] = e
37
38     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 $O(HW hw)$, 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()

```

1 def harris_detector(image, window_size=(5, 5)):
2     """
3     Given an input image, calculate the Harris Detector score for all
4     pixels
5     You can use same-padding for intensity (or 0-padding for derivatives)
6     to handle window values outside of the image.
7
8     Input- image: H x W
9     Output- results: a image of size H x W
10    """
11    # compute the derivatives
12    kx = np.array([-1, 0, 1]).reshape(1, 3)
13    ky = np.array([-1, 0, 1]).reshape(3, 1)
14    Ix = scipy.ndimage.convolve(image, kx, mode='constant', cval=0)
15    Iy = scipy.ndimage.convolve(image, ky, mode='constant', cval=0)
16
17    Ixx = Ix ** 2
18    Iyy = Iy ** 2
19    Ixy = Ix * Iy
20
21    # For each image location, construct the structure tensor and
22    # calculate
23    # the Harris response
24    M = np.zeros((3, image.shape[0], image.shape[1]))
25
26    # for y in range(image.shape[0]):
27    #     for x in range(image.shape[1]):
28    #         # import pdb; pdb.set_trace()

```

```

27 kernel = np.ones(window_size)
28 M[0] = scipy.ndimage.convolve(Ixx, kernel, mode='constant', cval=0)
29 M[1] = scipy.ndimage.convolve(Ixy, kernel, mode='constant', cval=0)
30 M[2] = scipy.ndimage.convolve(Iyy, kernel, mode='constant', cval=0)
31
32 alpha = 0.05
33
34 response = M[0] * M[2] - M[1] ** 2 - alpha * (M[0] + M[2]) ** 2
35
36 return response

```

3.2.2 Result

Result see figure 16

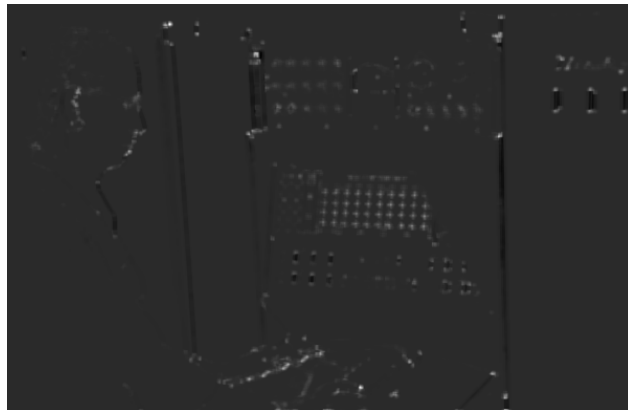


Figure 16: Harris response

4 Blob Detection

4.1 Task 8: Single-Scale Blob Detection

4.1.1 gaussian_filter()

```

1 def gaussian_filter(image, sigma):
2     """
3     Given an image, apply a Gaussian filter with the input kernel size
4     and standard deviation
5
6     Input
7     image: image of size HxW
8     sigma: scalar standard deviation of Gaussian Kernel
9
10    Output
11    Gaussian filtered image of size HxW
12    """
13    H, W = image.shape
14    # -- good heuristic way of setting kernel size

```

```

15     kernel_size = int(2 * np.ceil(2 * sigma) + 1)
16     # Ensure that the kernel size isn't too big and is odd
17     kernel_size = min(kernel_size, min(H, W) // 2)
18     if kernel_size % 2 == 0:
19         kernel_size = kernel_size + 1
20     # TODO implement gaussian filtering of size kernel_size x kernel_size
21     # Similar to Corner detection, use scipy's convolution function.
22     # Again, be consistent with the settings (mode = 'reflect').
23
24     # create gaussian kernel
25     ax = np.linspace(-(kernel_size - 1) / 2., (kernel_size - 1) / 2.,
26 kernel_size)
27     xx, yy = np.meshgrid(ax, ax)
28     kernel = np.exp(-0.5 * (np.square(xx) + np.square(yy)) / np.square(
29 sigma))
30     kernel /= np.sum(kernel)
31
32     output = scipy.ndimage.convolve(image, kernel, mode='reflect')
33     return output

```

4.1.2 Result

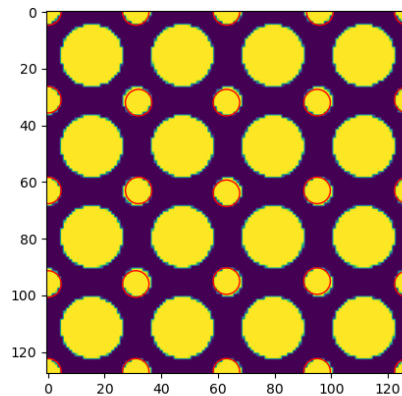


Figure 17: Single-scale blob detection - small

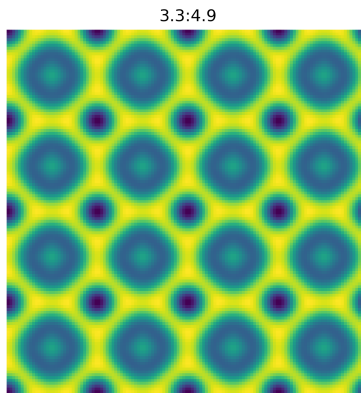


Figure 18: Single-scale blob detection DoG - small

The parameter used for small blob:

```
1 k = 5
2 sigma_1 = 3.7
3 sigma_2 = k * sigma_1
```

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

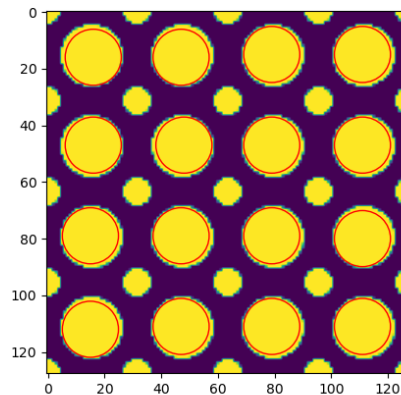


Figure 19: Single-scale blob detection - large

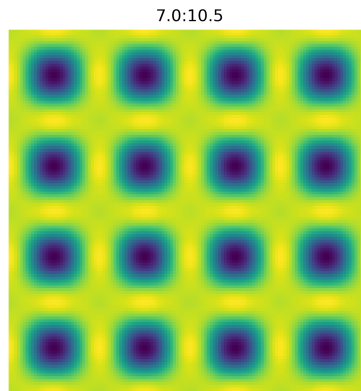


Figure 20: Single-scale blob detection DoG - large

The parameter used for large blob:

```

1 k = 1.9
2 sigma_1 = 3.7
3 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:

```

1  #"Detecting cell1 -- 008cell"
2  k = 3
3  sigma_1 = 3.7
4  sigma_2 = k * sigma_1
5
6  #"Detecting cell2 -- 004cell"
7  k = 3
8  sigma_1 = 3.4
9  sigma_2 = k * sigma_1
10
11  #"Detecting cell3 -- 005cell"
12  k = 5
13  sigma_1 = 3.7
14  sigma_2 = k * sigma_1
15
16  #"Detecting cell4 -- 006cell"
17  k = 1.9
18  sigma_1 = 3.7
19  sigma_2 = k * sigma_1

```

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

4.2.2 Plots and Discussion

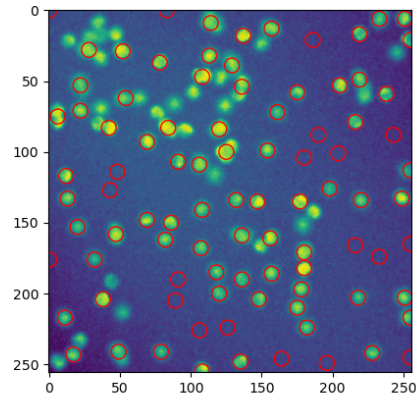


Figure 21: Blob detection - cell1

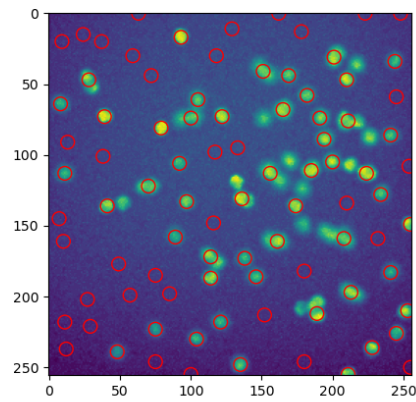


Figure 22: Blob detection - cell2

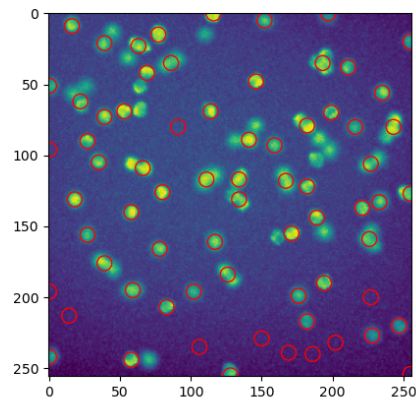


Figure 23: Blob detection - cell3

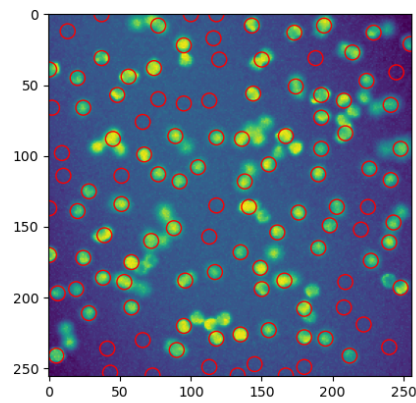


Figure 24: Blob detection - cell4

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

```
1 import os
2
3 import numpy as np
4
5 from common import read_img, save_img
6
7 import pdb
8 import cv2
9 import matplotlib.pyplot as plt
10
11
12 def image_patches(image, patch_size=(16, 16)):
13     """
14     Given an input image and patch_size,
15     return the corresponding image patches made
16     by dividing up the image into patch_size sections.
17
18     Input- image: H x W
19           patch_size: a scalar tuple M, N
20     Output- results: a list of images of size M x N
21     """
22     # TODO: Use slicing to complete the function
23     output = []
24     H, W = image.shape
25     h, w = patch_size
26     num_h = H // h
27     num_w = W // w
28
29     for i in range(num_h):
30         for j in range(num_w):
31             patch = image[i * h : (i + 1) * h, i * w : (i + 1) * w]
32             patch_mean = np.mean(patch)
33             patch_std = np.std(patch)
34             patch = (patch - patch_mean) / patch_std
35             patch = np.nan_to_num(patch, nan=0.0, posinf=0.0, neginf=0.0)
36             output.append(patch)
37     # import pdb; pdb.set_trace()
38     return output
39
40
41 def convolve(image, kernel):
42     """
43     Return the convolution result: image * kernel.
44     Reminder to implement convolution and not cross-correlation!
45     Caution: Please use zero-padding.
46
47     Input- image: H x W
48           kernel: h x w
49     Output- convolve: H x W
50     """
```

```

51     output = np.zeros_like(image)
52     if len(kernel.shape) == 2:
53         kernel = kernel[ : :-1 , : :-1]
54     elif len(kernel.shape) == 1:
55         kernel = kernel[ : :-1]
56
57     H, W = image.shape
58     h, w = kernel.shape
59
60     padding = [h//2, w//2]
61     padded_image = np.zeros((H + 2 * padding[0], W + 2 * padding[1]),
62 dtype=image.dtype)
63     padded_image[padding[0] : H + padding[0], padding[1] : W + padding[1]]
64     = image
65
66     for y in range(H):
67         for x in range(W):
68             patch = padded_image[y : y + h , x : x + w]
69             output[y, x] = np.sum(patch * kernel)
70
71     return output
72
73 def edge_detection(image):
74     """
75     Return Ix, Iy and the gradient magnitude of the input image
76
77     Input- image: H x W
78     Output- Ix, Iy, grad_magnitude: H x W
79     """
80     # TODO: Fix kx, ky
81     kx = np.array([-1, 0, 1]).reshape(1, 3) # 1 x 3
82     ky = np.array([-1, 0, 1]).reshape(3, 1) # 3 x 1
83
84     Ix = convolve(image, kx)
85     Iy = convolve(image, ky)
86
87     # TODO: Use Ix, Iy to calculate grad_magnitude
88     grad_magnitude = np.sqrt(Ix ** 2 + Iy ** 2)
89
90     return Ix, Iy, grad_magnitude
91
92 def sobel_operator(image):
93     """
94     Return Gx, Gy, and the gradient magnitude.
95
96     Input- image: H x W
97     Output- Gx, Gy, grad_magnitude: H x W
98     """
99     # TODO: Use convolve() to complete the function
100     Gx, Gy, grad_magnitude = None, None, None
101     S_x = np.array([[1, 0, -1],
102                     [2, 0, -2],

```

```

103         [1, 0, -1],])
104
105     S_y = np.array([[1, 2, 1],
106                    [0, 0, 0],
107                    [-1, -2, -1],])
108
109     Gx = convolve(image, S_x)
110     Gy = convolve(image, S_y)
111     grad_magnitude = np.sqrt(Gx ** 2 + Gy ** 2)
112
113     return Gx, Gy, grad_magnitude
114
115 def bilateral_filter(image, window_size, sigma_d, sigma_r):
116     """
117     Return filtered image using a bilateral filter
118
119     Input-   image: H x W
120             window_size: (h, w)
121             sigma_d: sigma for the spatial kernel
122             sigma_r: sigma for the range kernel
123     Output-  output: filtered image
124     """
125     # TODO: complete the bilateral filtering, assuming spatial and range
126     kernels are gaussian
127     H, W = image.shape
128     h, w = window_size
129     output = np.zeros_like(image, dtype=image.dtype)
130
131     padding = [h//2, w//2]
132     padded_image = np.zeros((H + 2 * padding[0], W + 2 * padding[1]),
133                             dtype=image.dtype)
134     padded_image[padding[0] : H + padding[0], padding[1] : W + padding[1]]
135     = image
136
137     range_x = np.arange(-int(w / 2), int(w / 2) + 1)
138     range_y = np.arange(-int(h / 2), int(h / 2) + 1)
139     mesh_x, mesh_y = np.meshgrid(range_x, range_y)
140     dis_mat = mesh_x **2 + mesh_y **2
141     # pdb.set_trace()
142
143     for y in range(H):
144         for x in range(W):
145             term1 = - dis_mat / (2 * sigma_d ** 2)
146             # pdb.set_trace()
147             image_in_kernel = padded_image[y : y + h, x : x + w]
148             term2 = - ( np.linalg.norm((image[y, x] - image_in_kernel),
149                                     keepdims=True) ** 2 / (2 * sigma_r ** 2))
150             w_ij = np.exp(term1 + term2)
151             output[y, x] = (image_in_kernel * w_ij).sum() / w_ij.sum()
152             # pdb.set_trace()
153
154     return output

```

```

153 def main():
154     # The main function
155     img = read_img('./grace_hopper.png')
156     """ Image Patches """
157     if not os.path.exists("./image_patches"):
158         os.makedirs("./image_patches")
159
160     # -- TODO Task 1: Image Patches --
161     # (a)
162     # First complete image_patches()
163     patches = image_patches(img)
164     # Now choose any three patches and save them
165     # chosen_patches should have those patches stacked vertically/
    horizontally
166     idxs = [np.random.randint(0, len(patches)) for _ in range(3)]
167     # print(idxs)
168     chosen_patches = np.array([patches[i] for i in idxs])
169     chosen_patches = chosen_patches.reshape(16, -1)
170     # import pdb; pdb.set_trace()
171     save_img(chosen_patches, "./image_patches/q1_patch.png")
172
173     # (b), (c): No code
174
175     """ Convolution and Gaussian Filter """
176     if not os.path.exists("./gaussian_filter"):
177         os.makedirs("./gaussian_filter")
178
179     # -- TODO Task 2: Convolution and Gaussian Filter --
180     # (a): No code
181
182     # (b): Complete convolve()
183
184     # (c)
185     # Calculate the Gaussian kernel described in the question.
186     # There is tolerance for the kernel.
187     kernel_size = 3
188     kernel_sigma = 0.572
189     # kernel_sigma = 2
190     kernel_range = np.arange(-int(kernel_size / 2), int(kernel_size / 2) +
    1)
191     kernel_x, kernel_y = np.meshgrid(kernel_range, kernel_range)
192     kernel_gaussian = np.exp(- (kernel_x ** 2 + kernel_y ** 2) / (2 *
    kernel_sigma ** 2))
193     kernel_gaussian /= kernel_gaussian.sum()
194     # print(kernel_gaussian.sum())
195     # pdb.set_trace()
196     filtered_gaussian = convolve(img, kernel_gaussian)
197     save_img(filtered_gaussian, "./gaussian_filter/q2_gaussian.png")
198
199     # (d), (e): No code
200
201     # (f): Complete edge_detection()
202
203     # (g)

```



```

204 # Use edge_detection() to detect edges
205 # for the original and gaussian filtered images.
206 _, _, edge_detect = edge_detection(img)
207 save_img(edge_detect, "./gaussian_filter/q3_edge.png")
208 _, _, edge_with_gaussian = edge_detection(filtered_gaussian)
209 save_img(edge_with_gaussian, "./gaussian_filter/q3_edge_gaussian.png")
210
211 print("Gaussian Filter is done. ")
212
213 # (h) complete biliateral_filter()
214 if not os.path.exists("./bilateral"):
215     os.makedirs("./bilateral")
216
217 image_bilataral_filtered = bilateral_filter(img, (5, 5), 3, 75)
218 img_cv2 = cv2.imread('./grace_hopper.png')
219 image_bilataral_filtered_cv2 = cv2.bilateralFilter(img_cv2, 5, 75, 3)
220 save_img(image_bilataral_filtered, "./bilateral/bilateral_output.png")
221 save_img(image_bilataral_filtered_cv2, "./bilateral/
bilateral_output_cv2.png")
222
223 # -- TODO Task 3: Sobel Operator --
224 if not os.path.exists("./sobel_operator"):
225     os.makedirs("./sobel_operator")
226
227 # (a): No code
228
229 # (b): Complete sobel_operator()
230
231 # (c)
232 Gx, Gy, edge_sobel = sobel_operator(img)
233 save_img(Gx, "./sobel_operator/q2_Gx.png")
234 save_img(Gy, "./sobel_operator/q2_Gy.png")
235 save_img(edge_sobel, "./sobel_operator/q2_edge_sobel.png")
236
237 print("Sobel Operator is done. ")
238
239 # -- TODO Task 4: LoG Filter --
240 if not os.path.exists("./log_filter"):
241     os.makedirs("./log_filter")
242
243 # (a)
244 kernel_Log1 = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]])
245 kernel_Log2 = np.array([[0, 0, 3, 2, 2, 2, 3, 0, 0],
246                          [0, 2, 3, 5, 5, 5, 3, 2, 0],
247                          [3, 3, 5, 3, 0, 3, 5, 3, 3],
248                          [2, 5, 3, -12, -23, -12, 3, 5, 2],
249                          [2, 5, 0, -23, -40, -23, 0, 5, 2],
250                          [2, 5, 3, -12, -23, -12, 3, 5, 2],
251                          [3, 3, 5, 3, 0, 3, 5, 3, 3],
252                          [0, 2, 3, 5, 5, 5, 3, 2, 0],
253                          [0, 0, 3, 2, 2, 2, 3, 0, 0]])
254 filtered_Log1 = convolve(img, kernel_Log1)
255 filtered_Log2 = convolve(img, kernel_Log2)
256 # Use convolve() to convolve img with kernel_LOG1 and kernel_LOG2

```

```

257     save_img(filtered_Log1, "./log_filter/q1_Log1.png")
258     save_img(filtered_Log2, "./log_filter/q1_Log2.png")
259
260     # (b)
261     # Follow instructions in pdf to approximate LoG with a DoG
262     data = np.load('log1d.npz')
263     plt.figure(1)
264     plt.plot(data['log50'])
265     plt.plot(data['gauss53'] - data['gauss50'])
266     plt.legend(['Original', 'Approx'])
267     plt.show()
268     print("LoG Filter is done. ")
269
270
271 if __name__ == "__main__":
272     main()

```

5.0.2 filtersmon.py

```

1  import numpy as np
2  import scipy.signal
3  import matplotlib.pyplot as plt
4
5
6  def conv(I, f):
7      """Apply same-sized convolution with a filter with zero-padding"""
8      # Note that this is convolution! This is filtering but with f
9      [::-1,::-1]
10     return scipy.signal.convolve2d(
11         I, f, mode='same', boundary='fill', fillvalue=0.0)
12
13 def nnUpsample(I, factor):
14     """Nearest neighbor upsample an image by the given factor"""
15     return np.kron(I, np.ones((factor, factor)))
16
17 # -- TODO Task 5: Who's That Filter? --
18 # (a): Fill in the filters to get the data to match
19
20 filter0 = np.diag([0, 1, 0])
21 filter1 = np.ones((3, 3))
22 filter2 = np.ones((3, 3)) * 1/9
23 filter3 = np.array([[0, 0, 0],
24                     [1, 0, 0],
25                     [0, 0, 0]])
26 filter4 = np.array([[ -1, 0, 1],
27                     [ -1, 0, 1],
28                     [ -1, 0, 1]])
29
30 # (b): No code
31
32 filters = [filter0, filter1, filter2, filter3, filter4]
33

```

```

34 np.random.seed(442)
35 data = (plt.imread("filtermon/442.png").astype(float)
36        [:, :, 0] < 0.5).astype(float)
37
38
39 plt.figure()
40 plt.imshow(nnUpsample(data, 10))
41 plt.colorbar()
42 plt.savefig("input.png")
43
44
45 for fi, f in enumerate(filters):
46     c = conv(data, f)
47     sol = np.load("filtermon/output_%d.npy" % fi)
48     plt.imsave(f"./filtermon/output_{fi}.png", sol)
49     matches = False
50
51     if np.allclose(c, sol, rtol=1e-2, atol=1e-5):
52         print("Filter %d matches" % fi)
53         matches = True
54     else:
55         print("Filter %d doesn't match" % fi)
56
57     plt.figure()
58     fig, axs = plt.subplots(1, 2)
59
60     im = axs[0].imshow(nnUpsample(c, 10))
61     axs[0].set_title("Yours (%s)" % ("Match!" if matches else "No Match"))
62     plt.colorbar(im, ax=axs[0])
63
64     im = axs[1].imshow(nnUpsample(sol, 10))
65     axs[1].set_title("Target")
66     plt.colorbar(im, ax=axs[1])
67
68     plt.tight_layout()
69     plt.savefig("comparison_%d.pdf" % (fi))

```

5.0.3 corners.py

```

1 import os
2
3 import numpy as np
4 import scipy.ndimage
5 # Use scipy.ndimage.convolve() for convolution.
6 # Use zero padding (Set mode = 'constant'). Refer docs for further info.
7
8 from common import read_img, save_img
9
10
11 def corner_score(image, u=5, v=5, window_size=(5, 5)):
12     """
13     Given an input image, x_offset, y_offset, and window_size,
14     return the function E(u,v) for window size W

```

```

15     corner detector score for that pixel.
16     Use zero-padding to handle window values outside of the image.
17
18     Input- image: H x W
19           u: a scalar for x offset
20           v: a scalar for y offset
21           window_size: a tuple for window size
22
23     Output- results: a image of size H x W
24     """
25     output = np.zeros_like(image)
26     H, W = image.shape
27     h, w = window_size
28
29     shifted_image = np.roll(image, (u, v), axis=(1, 0))
30
31     padding = (h//2, w//2)
32
33     padded_image = np.zeros((H + 2 * padding[0], W + 2 * padding[1]),
34 dtype=image.dtype)
35     padded_image[padding[0] : padding[0] + H, padding[1] : padding[1] + W]
36     = image
37     padded_shifted_image = np.zeros((H + 2 * padding[0], W + 2 * padding
38 [1]), dtype=image.dtype)
39     padded_shifted_image[padding[0] : padding[0] + H, padding[1] : padding
40 [1] + W] = shifted_image
41
42     for y in range(H):
43         for x in range(W):
44             e = np.sum((padded_shifted_image[y : y + h, x : x + w] -
45 padded_image[y : y + h, x : x + h]) ** 2)
46             output[y, x] = e
47
48     return output
49
50 def harris_detector(image, window_size=(5, 5)):
51     """
52     Given an input image, calculate the Harris Detector score for all
53     pixels
54     You can use same-padding for intensity (or 0-padding for derivatives)
55     to handle window values outside of the image.
56
57     Input- image: H x W
58     Output- results: a image of size H x W
59     """
60     # compute the derivatives
61     kx = np.array([-1, 0, 1]).reshape(1, 3)
62     ky = np.array([-1, 0, 1]).reshape(3, 1)
63     Ix = scipy.ndimage.convolve(image, kx, mode='constant', cval=0)
64     Iy = scipy.ndimage.convolve(image, ky, mode='constant', cval=0)
65
66     Ixx = Ix ** 2
67     Iyy = Iy ** 2

```

```

63     Ixy = Ix * Iy
64
65     # For each image location, construct the structure tensor and
66     # calculate
67     # the Harris response
68     M = np.zeros((3, image.shape[0], image.shape[1]))
69
70     # for y in range(image.shape[0]):
71     #     for x in range(image.shape[1]):
72     #         # import pdb; pdb.set_trace()
73     kernel = np.ones(window_size)
74     M[0] = scipy.ndimage.convolve(Ixx, kernel, mode='constant', cval=0)
75     M[1] = scipy.ndimage.convolve(Ixy, kernel, mode='constant', cval=0)
76     M[2] = scipy.ndimage.convolve(Iyy, kernel, mode='constant', cval=0)
77
78     alpha = 0.05
79
80     response = M[0] * M[2] - M[1] ** 2 - alpha * (M[0] + M[2]) ** 2
81
82     return response
83
84 def main():
85     img = read_img('./grace_hopper.png')
86
87     # Feature Detection
88     if not os.path.exists("./feature_detection"):
89         os.makedirs("./feature_detection")
90
91     # -- TODO Task 6: Corner Score --
92     # (a): Complete corner_score()
93
94     # (b)
95     # Define offsets and window size and calculate corner score
96     W = (5, 5)
97     tuples = ((0, 5), (0, -5), (5, 0), (-5, 0))
98     for i, (u, v) in enumerate(tuples):
99         score = corner_score(img, u, v, W)
100         save_img(score, f"./feature_detection/corner_score_{i}.png")
101
102     # (c): No Code
103
104     # -- TODO Task 7: Harris Corner Detector --
105     # (a): Complete harris_detector()
106
107     # (b)
108     harris_corners = harris_detector(img)
109     save_img(harris_corners, "./feature_detection/harris_response.png")
110
111
112 if __name__ == "__main__":
113     main()

```

5.0.4 blob_detection.py

```
1 import os
2 import matplotlib.pyplot as plt
3 import numpy as np
4 import scipy.ndimage
5 # Use scipy.ndimage.convolve() for convolution.
6 # Use same padding (mode = 'reflect'). Refer docs for further info.
7
8 from common import (find_maxima, read_img, visualize_maxima,
9                     visualize_scale_space)
10
11
12 def gaussian_filter(image, sigma):
13     """
14     Given an image, apply a Gaussian filter with the input kernel size
15     and standard deviation
16
17     Input
18         image: image of size HxW
19         sigma: scalar standard deviation of Gaussian Kernel
20
21     Output
22         Gaussian filtered image of size HxW
23     """
24     H, W = image.shape
25     # -- good heuristic way of setting kernel size
26     kernel_size = int(2 * np.ceil(2 * sigma) + 1)
27     # Ensure that the kernel size isn't too big and is odd
28     kernel_size = min(kernel_size, min(H, W) // 2)
29     if kernel_size % 2 == 0:
30         kernel_size = kernel_size + 1
31     # TODO implement gaussian filtering of size kernel_size x kernel_size
32     # Similar to Corner detection, use scipy's convolution function.
33     # Again, be consistent with the settings (mode = 'reflect').
34
35     # create gaussian kernel
36     ax = np.linspace(-(kernel_size - 1) / 2., (kernel_size - 1) / 2.,
37                     kernel_size)
38     xx, yy = np.meshgrid(ax, ax)
39     kernel = np.exp(-0.5 * (np.square(xx) + np.square(yy)) / np.square(
40         sigma))
41     kernel /= np.sum(kernel)
42
43     output = scipy.ndimage.convolve(image, kernel, mode='reflect')
44     return output
45
46
47 def main():
48     image = read_img('polka.png')
49     # import pdb; pdb.set_trace()
50     # Create directory for polka_detections
```

```

51 if not os.path.exists("./polka_detections"):
52     os.makedirs("./polka_detections")
53
54 # -- TODO Task 8: Single-scale Blob Detection --
55
56 # (a), (b): Detecting Polka Dots
57 # First, complete gaussian_filter()
58 print("Detecting small polka dots")
59 # -- Detect Small Circles
60 k = 1.5
61 sigma_1 = 3.3
62 sigma_2 = k * sigma_1
63 gauss_1 = gaussian_filter(image, sigma_1) # to implement
64 gauss_2 = gaussian_filter(image, sigma_2) # to implement
65
66 # calculate difference of gaussians
67 DoG_small = gauss_2 - gauss_1 # to implement
68
69 # visualize maxima
70 maxima = find_maxima(DoG_small, k_xy=10)
71 visualize_scale_space(DoG_small, sigma_1, sigma_2 / sigma_1,
72                       './polka_detections/polka_small_DoG.png')
73 visualize_maxima(image, maxima, sigma_1, sigma_2 / sigma_1,
74                  './polka_detections/polka_small.png')
75 plt.clf()
76
77
78 # -- Detect Large Circles
79 print("Detecting large polka dots")
80 k = 1.5
81 sigma_1 = 7
82 sigma_2 = k * sigma_1
83 gauss_1 = gaussian_filter(image, sigma_1) # to implement
84 gauss_2 = gaussian_filter(image, sigma_2) # to implement
85
86 # calculate difference of gaussians
87 DoG_large = gauss_2 - gauss_1 # to implement
88
89 # visualize maxima
90 # Value of k_xy is a suggestion; feel free to change it as you wish.
91 maxima = find_maxima(DoG_large, k_xy=10)
92 visualize_scale_space(DoG_large, sigma_1, sigma_2 / sigma_1,
93                       './polka_detections/polka_large_DoG.png')
94 visualize_maxima(image, maxima, sigma_1, sigma_2 / sigma_1,
95                  './polka_detections/polka_large.png')
96 plt.clf()
97
98 # # # -- TODO Task 9: Cell Counting --
99 print("Detecting cells")
100
101 cell_1 = read_img("./cells/008cell.png")
102 cell_2 = read_img("./cells/004cell.png")
103 cell_3 = read_img("./cells/005cell.png")
104 cell_4 = read_img("./cells/006cell.png")

```

```

105     cells = [cell_1, cell_2, cell_3, cell_4]
106
107     # Detect the cells in any four (or more) images from vgg_cells
108     # Create directory for cell_detections
109     if not os.path.exists("./cell_detections"):
110         os.makedirs("./cell_detections")
111
112
113     print("Detecting cell1")
114     k = 3
115     sigma_1 = 3.7
116     sigma_2 = k * sigma_1
117     gauss_1 = gaussian_filter(cell_1, sigma_1) # to implement
118     gauss_2 = gaussian_filter(cell_1, sigma_2) # to implement
119
120     # calculate difference of gaussians
121     DoG_cell1 = gauss_2 - gauss_1 # to implement
122
123     # visualize maxima
124     maxima = find_maxima(DoG_cell1, k_xy=10)
125     visualize_scale_space(DoG_cell1, sigma_1, sigma_2 / sigma_1,
126                          './cell_detections/cell1_DoG.png')
127     visualize_maxima(cell_1, maxima, sigma_1, sigma_2 / sigma_1,
128                     './cell_detections/cell1.png')
129     plt.clf()
130
131
132     print("Detecting cell2")
133     k = 3
134     sigma_1 = 3.4
135     sigma_2 = k * sigma_1
136     gauss_1 = gaussian_filter(cell_2, sigma_1) # to implement
137     gauss_2 = gaussian_filter(cell_2, sigma_2) # to implement
138
139     # calculate difference of gaussians
140     DoG_cell2 = gauss_2 - gauss_1 # to implement
141
142     # visualize maxima
143     maxima = find_maxima(DoG_cell2, k_xy=10)
144     visualize_scale_space(DoG_cell2, sigma_1, sigma_2 / sigma_1,
145                          './cell_detections/cell2_DoG.png')
146     visualize_maxima(cell_2, maxima, sigma_1, sigma_2 / sigma_1,
147                     './cell_detections/cell2.png')
148     plt.clf()
149
150
151
152
153     print("Detecting cell3")
154     k = 5
155     sigma_1 = 3.7
156     sigma_2 = k * sigma_1
157     gauss_1 = gaussian_filter(cell_3, sigma_1) # to implement
158     gauss_2 = gaussian_filter(cell_3, sigma_2) # to implement

```



```

159
160 # calculate difference of gaussians
161 DoG_cell3 = gauss_2 - gauss_1 # to implement
162
163 # visualize maxima
164 maxima = find_maxima(DoG_cell3, k_xy=10)
165 visualize_scale_space(DoG_cell3, sigma_1, sigma_2 / sigma_1,
166                       './cell_detections/cell3_DoG.png')
167 visualize_maxima(cell_3, maxima, sigma_1, sigma_2 / sigma_1,
168                 './cell_detections/cell3.png')
169 plt.clf()
170
171
172
173
174 print("Detecting cell4")
175 k = 1.9
176 sigma_1 = 3.7
177 sigma_2 = k * sigma_1
178 gauss_1 = gaussian_filter(cell_4, sigma_1) # to implement
179 gauss_2 = gaussian_filter(cell_4, sigma_2) # to implement
180
181 # calculate difference of gaussians
182 DoG_cell4 = gauss_2 - gauss_1 # to implement
183
184 # visualize maxima
185 maxima = find_maxima(DoG_cell4, k_xy=10)
186 visualize_scale_space(DoG_cell4, sigma_1, sigma_2 / sigma_1,
187                       './cell_detections/cell4_DoG.png')
188 visualize_maxima(cell_4, maxima, sigma_1, sigma_2 / sigma_1,
189                 './cell_detections/cell4.png')
190 plt.clf()
191
192
193
194 if __name__ == '__main__':
195     main()

```

Submitted by Wensong Hu on February 13, 2024.