Xiang Chen
1004704992
October 27, 2020

CSC420 Report

Part I: Theory (40 marks)

Q1) (10 marks)

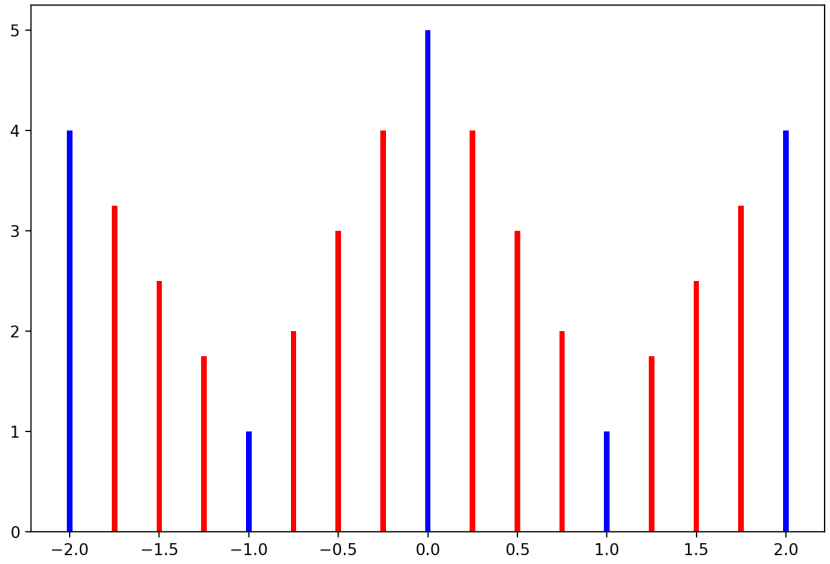
In this question, I use linear interpolation to upsample by a factor of 4, which is the following formula.

$$G(x) = \frac{x_2 - x}{x_2 - x_1} F(x_1) + \frac{x - x_1}{x_2 - x_1} F(x_2)$$

We have five points (-2.0,4), (-1.0,1),(0.0,5),(1.0,1),(2.0,4). We can simplify calculate the linear function of each line and put x = -1.75, -1.5, -1.25 to f_1, x = -0.75, -0.5, -0.25 to f_2, 0.25, 0.5, 0.75 to f_3, 1.25, 1.5, 1.75 to f_4 and plot the value.

$f_1 = -3x - 2$ $f_2 = 4x + 5$ $f_3 = -4x + 5$ $f_4 = 3x - 2$

Here is the graph generated by python.



Q2) (15 marks)

For this question, we first need to do some transformation for $sin^2(x)$.

We know $sin^2(x) = \frac{1}{2}(1 - cos(2x))$.

Then, we can have $sin^2(2F_a\pi t) = \frac{1}{2}(1 - cos(4F_a\pi t))$.

We can have $x(t) = \frac{1}{2}(1 - cos(4F_a\pi t)) + \frac{1}{7}cos(F_b\pi t) + \frac{1}{5}sin(5F_c\pi t)$.

We can remove the brackets.

Then, we have $x(t) = \frac{1}{2} - \frac{1}{2}cos(4F_a\pi t) + \frac{1}{7}cos(F_b\pi t) + \frac{1}{5}sin(5F_c\pi t)$.

The max frequency for x(t) is $max(\frac{4F_a\pi}{2\pi}, \frac{F_b\pi}{2\pi}, \frac{5F_a\pi}{2\pi})$.

By Nyqvist-Shannon sampling theorem, the sufficient sample-rate is $2 * max(2F_a, \frac{F_b}{2}, \frac{5F_c}{2})$.

Q3) (15 marks)

1.

We know N is a 2*2 Matrix. Therefore, we can use the formula for eigenvalue of 2*2 matrix, which is $\lambda_{\pm} = \frac{1}{2}[(a_{11} + a_{22}) \pm \sqrt{4a_{12}a_{21} - a_{12}a_{21}}]$ (1)(0.3.1).

We have $N = \begin{pmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{pmatrix}$, which $a_{11} = I_x^2$, $a_{12} = I_xI_y$, $a_{21} = I_xI_y$, $a_{11} = I_y^2$.

When we put them in to (0.3.1), we get $\lambda_1 = 0$ and $\lambda_2 = I_x^2 + I_y^2$.

Therefore, the second value of λ is $I_x^2 + I_y^2$.

2.

We know $N = V\begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}V^{-1}$, and $M = \sum_x \sum_y w(x,y) * (V\begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}V^{-1})$ from lecture.

By the property of linear system, we can make some form transition.

Then, we can get $M = \sum_x \sum_y V\begin{pmatrix} w(x,y) * \lambda_1 & 0 \\ 0 & w(x,y) * \lambda_2 \end{pmatrix}V^{-1}$.

Since we have assumed N is a positive semi-definite, we can know $\lambda_1 \geq 0$ and $\lambda_2 \geq 0$.

Since $w(x, y) > 0$ (Gaussian Matrix), we know $V \begin{pmatrix} w(x,y) * \lambda_1 & 0 \\ 0 & w(x,y) * \lambda_2 \end{pmatrix} V^{-1}$ is also positive semi-definite.

Therefore, we can know M is also positive semi-definite since it is a sum of many matrixes which are positive semi-definite.

We can follow the intuition to prove the matrix of sum of many matrixes which are positive semi-definite is also positive semi-definite. Let assume any two matrix X and Y and they are positive semi-definite.

We can have $V^T X V > 0$ and $V^T Y V > 0$, then we can have $V^T X V + V^T Y V > 0$, which is $V^T (X + Y) V > 0$.

QED

(1)https://mathworld.wolfram.com/Eigenvalue.html#:~:text=Eigenvalues%20are%20a%20special%20set,144).

I HAVE ADDED MORE DETAILED COMMENTS IN RESIZE.PY & CORNER_DETECTION.PY AFTER I FINISHED THE REPORT. PLEASE SEE THEM.

Part II: Image Resizing with Seam Carving (75 marks)

1.

```
def Sobel_Operation(src):
# This function is only for grey scale image. #
# Please use getGreyImge first, if input is a RGB 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]])
    x_length, y_length = src.shape
    res = np.empty((x_length,y_length), dtype=float)
    conv_x = cv.filter2D(src,-1,sobel_x)
    conv_y = cv.filter2D(src,-1,sobel_y)
    for i in range(x_length):
        for j in range(y_length):
            res[i][j] = math.sqrt(conv_x[i][j]**2+conv_y[i][j]**2)
    return res

```

2. (45 marks)

```

def cutColoum(src):
    i_length, j_length = src.shape
    # Array for saving the minimum energy from bottom to up (Dynamic Programming).
    backtrack = np.zeros((i_length, j_length), dtype=float)
    # Recod the direction. We want to use the direction to trace the minimum energy path.
    directions = np.zeros((i_length, j_length), dtype=int)
    # Get the gradients(energy map).
    gradients = Sobel_Operation(src)
    # Initilize the last row of backtrack. It should be the same as the last row of our energy map.
    for j in range(j_length): backtrack[i_length-1][j] = gradients[i_length-1][j]
    # From bottom to top.
    for i in reversed(range(i_length-1)):
        for j in range(j_length):
            # case 1: j=0, we only have two choice up and up-right.
            if not j:
                # Get the index for the minimum value.
                index = np.argmin(backtrack[i+1, j:j + 2])
                # Get the minmium path from this point to the bottom.
                backtrack[i][j] = gradients[i][j] + backtrack[i+1][j+index]
                # Trace the direction.
                directions[i][j] = index
            # case 2: j=length-1, we only have two choices up-left and up.
            elif j == j_length - 1:
                # Get the index for the minimum value.
                index = np.argmin(backtrack[i+1, j-1:j+1])
                # Get the minmium path from this point to the bottom.
                backtrack[i][j] = gradients[i][j] + backtrack[i+1][j-1+index]
                # Trace the direction.
                directions[i][j] = index - 1
            # case 3: normal j, we have three choices up-left, up and up-right.
            else:
                # Get the index for the minimum value.
                index = np.argmin(backtrack[i+1, j-1:j + 2])
                # Get the minmium path from this point to the bottom.
                backtrack[i][j] = gradients[i][j] + backtrack[i+1][j-1+index]
                # Trace the direction.
                directions[i][j] = index - 1
    # Boolean matrix to record which one will be removed or kept.
    binary_matrix = np.ones((i_length, j_length), dtype=int)
    # Index for the minimum value of the first row.
    min_j = np.argmin(backtrack[0])
    # Trace the path of minimum value and set position of boolean matrix to 0.
    for i in range(i_length):
        binary_matrix[i][min_j] = 0
        min_j = min_j + directions[i][min_j]
    return binary_matrix
```

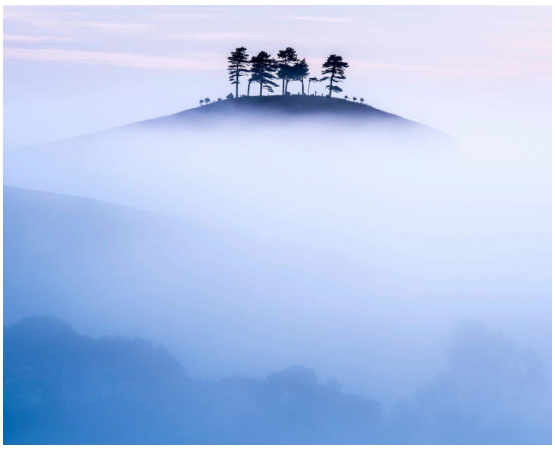
3. (10 marks)

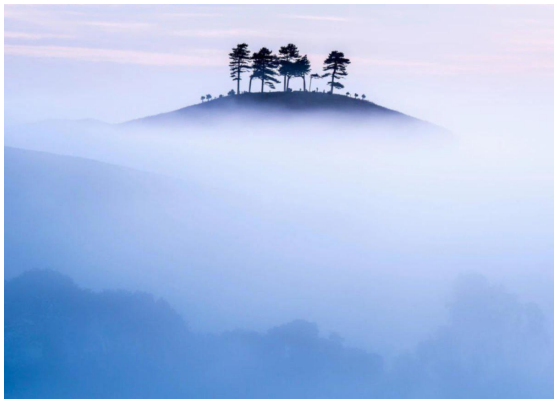
```
def mainCol(src,new_col):
    row, col, _ = src.shape
    new_image = src.copy()
    if(new_col>=col): return src
    for k in range(col-new_col):
        binary_matrix = cutColoum(getGreyImge(new_image))
        i_length,j_length = binary_matrix.shape
        buff = new_image.copy()
        new_image = np.zeros((i_length,j_length-1,_),dtype=np.uint8)
        for i in range(i_length):
            new_j = 0
            for j in range(j_length):
                if binary_matrix[i][j]:
                    new_image[i][new_j][0] = buff[i][j][0]
                    new_image[i][new_j][1] = buff[i][j][1]
                    new_image[i][new_j][2] = buff[i][j][2]
                    new_j += 1
    return new_image
```

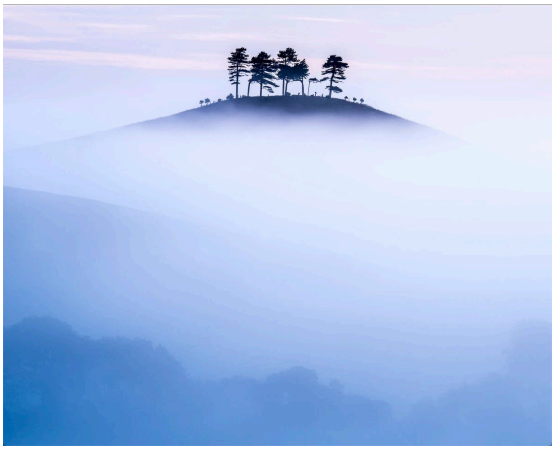
4. (10 marks)

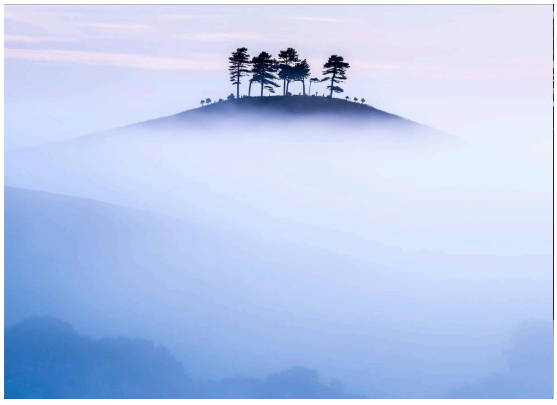
```

def cutRow(src):
    i_length, j_length = src.shape
    rotate = np.zeros((j_length, i_length), dtype=float)
    for i in range(i_length):
        for j in range(j_length):
            rotate[j_length-j-1][i] = src[i][j]
    return cutColoum(rotate)
def mainRow(src,new_row):
# This funciton take src of RGB image.
    # Get r, c and dimension.
    row, col, _ = src.shape
    # Make a copy, since we need to loop n times on new image.
    new_image = src.copy()
    # If the parameter is invalid, return src.
    if(new_row>=row): return src
    # Cut coloums row-new_row times.
    for k in range(row-new_row):
        binary_matrix = cutRow(getGreyImge(new_image))
        j_length,i_length = binary_matrix.shape
        rotate_binary_matrix = np.zeros((i_length, j_length), dtype=int)
        # Rotate the binary matrix.
        for i in range(i_length):
            for j in range(j_length):
                rotate_binary_matrix[i][j] = binary_matrix[j_length-j-1][i]
        buff = new_image.copy()
        new_image = np.zeros((i_length-1,j_length,_),dtype=np.uint8)
        # Set new value to new_image based on binary matrix.
        for j in range(j_length):
            new_i = 0
            for i in range(i_length):
                if rotate_binary_matrix[i][j]:
                    new_image[new_i][j][0] = buff[i][j][0]
                    new_image[new_i][j][1] = buff[i][j][1]
                    new_image[new_i][j][2] = buff[i][j][2]
                    new_i += 1
    return new_image
```

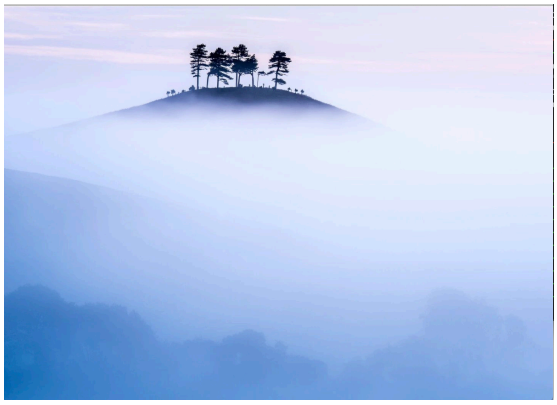
5. (10 marks)











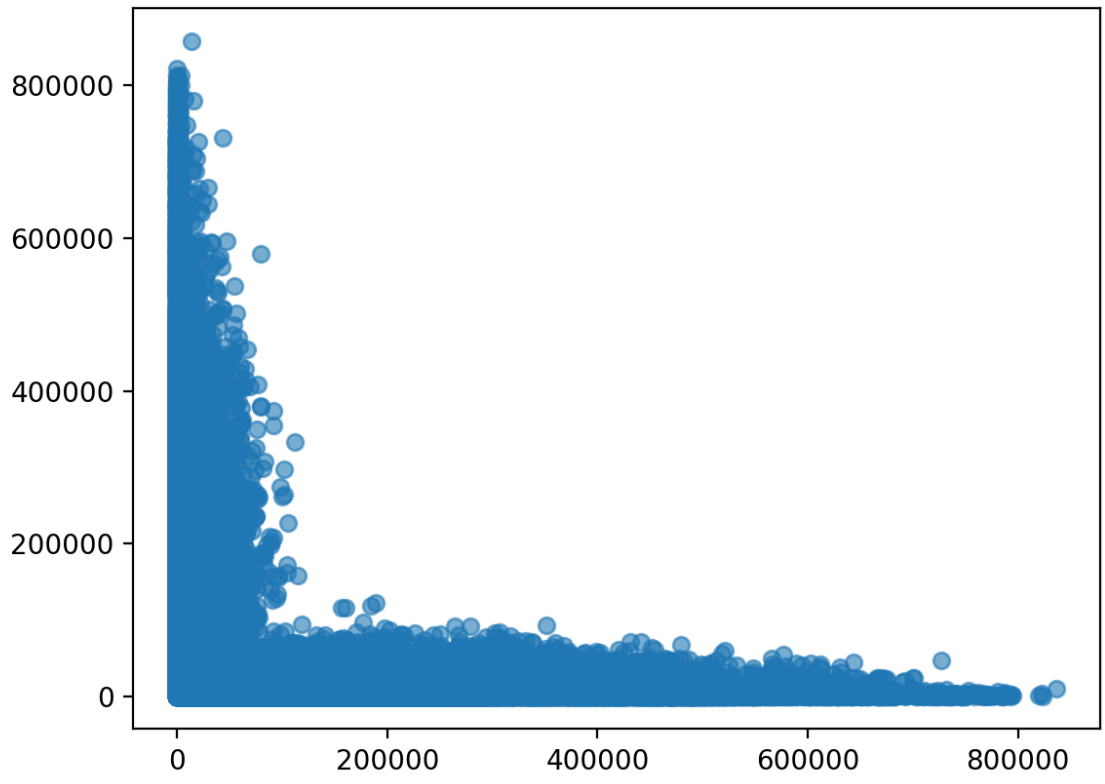
Part III: Corner Detection (35 marks)

1.

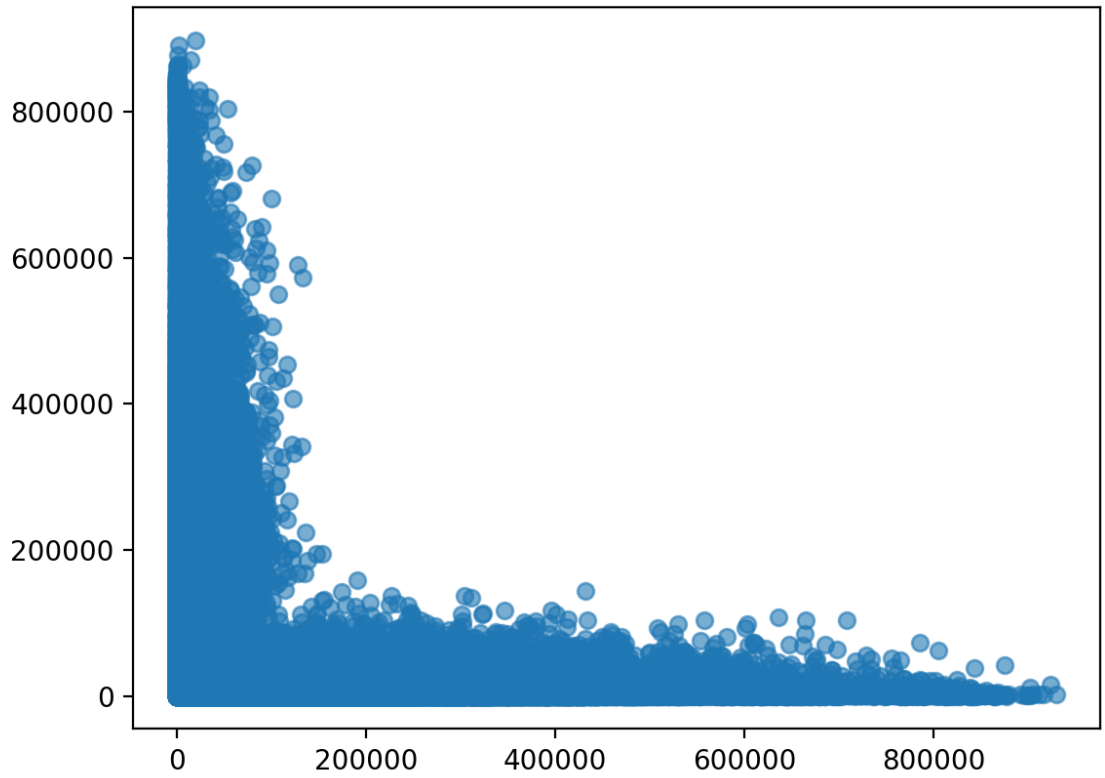
```
28
29 def eigenvalues(src,kernel_size):
30     sobel_x = np.array([[-1,0,1],[-2,0,2],[-1,0,1]])
31     sobel_y = np.array([[-1,-2,-1],[0,0,0],[1,2,1]])
32     grad_x = cv.filter2D(src,-1,sobel_x)
33     grad_y = cv.filter2D(src,-1,sobel_y)
34     I_xy = grad_x * grad_y
35     I_x = grad_x * grad_x
36     I_y = grad_y * grad_y
37     window = Gaussian_Blur(kernel_size/6,kernel_size)
38     I_x = cv.filter2D(I_x,-1,window)
39     I_y = cv.filter2D(I_y,-1,window)
40     I_xy = cv.filter2D(I_xy,-1,window)
41     x_length, y_length = src.shape
42     eigenvalues_1 = np.zeros((x_length,y_length),dtype=complex)
43     eigenvalues_2 = np.zeros((x_length,y_length),dtype=complex)
44     for i in range(x_length):
45         for j in range(y_length):
46             M = np.array([[I_x[i][j],I_xy[i][j]],[I_xy[i][j],I_y[i][j]]])
47             eigvals, eigvecs = la.eig(M)
48             # If you don't want me to use library, here is the alternative way to get eigenvalue of 2*2 Matricx
49             # eigenvalues_1[i][j] = (1/2) * (M[0][0]+M[1][1] - math.sqrt(4*M[0][1]*M[1][0]+(M[0][0]-M[1][1])**2))
50             # eigenvalues_2[i][j] = (1/2) * (M[0][0]+M[1][1] + math.sqrt(4*M[0][1]*M[1][0]+(M[0][0]-M[1][1])**2))
51             eigenvalues_1[i][j] = eigvals[0]
52             eigenvalues_2[i][j] = eigvals[1]
53     return eigenvalues_1, eigenvalues_2
```

2.

This is the scatterplot for I_1(ex4.jpg).



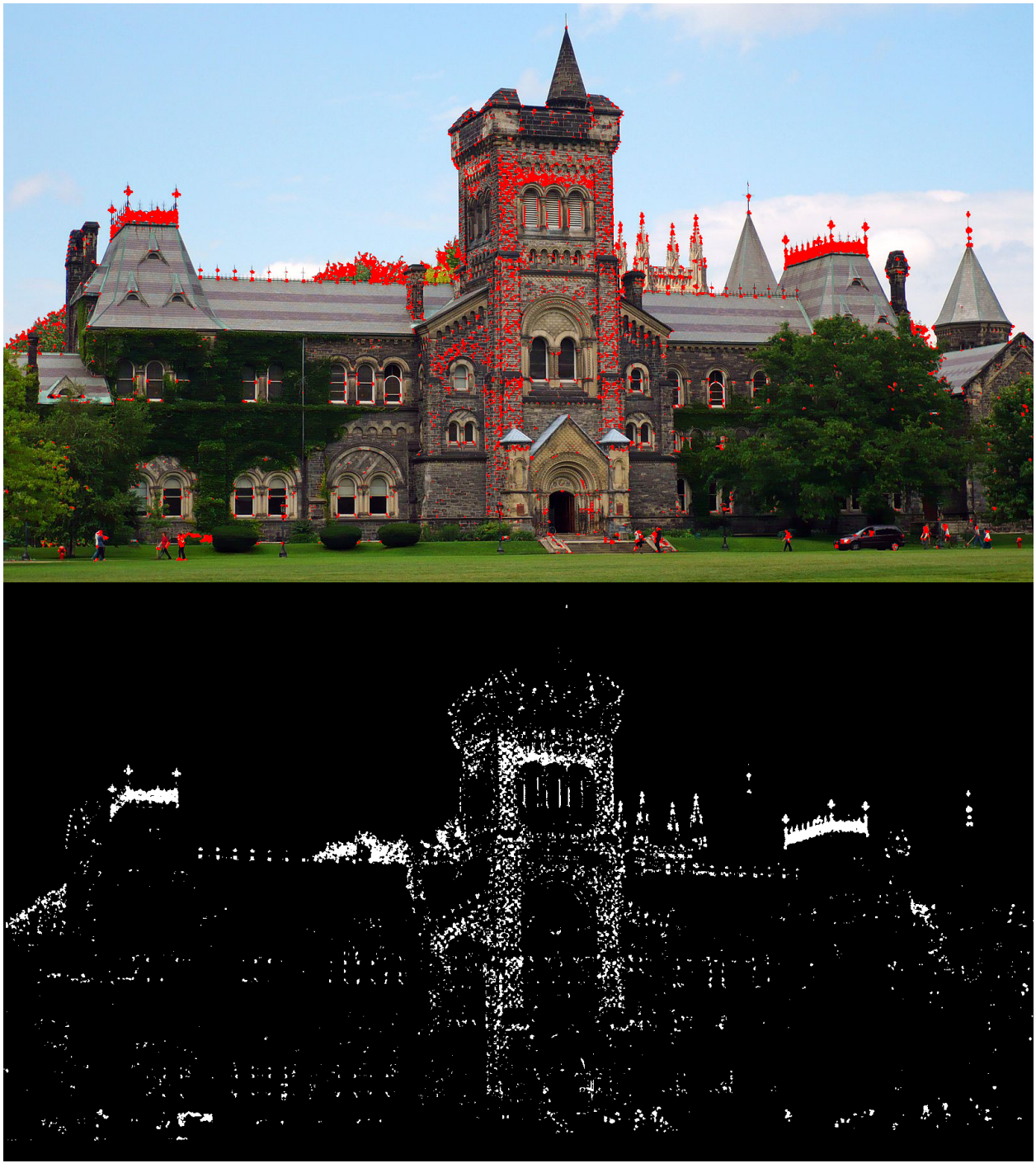
This is the scatterplot for I_2(ex5.jpg).



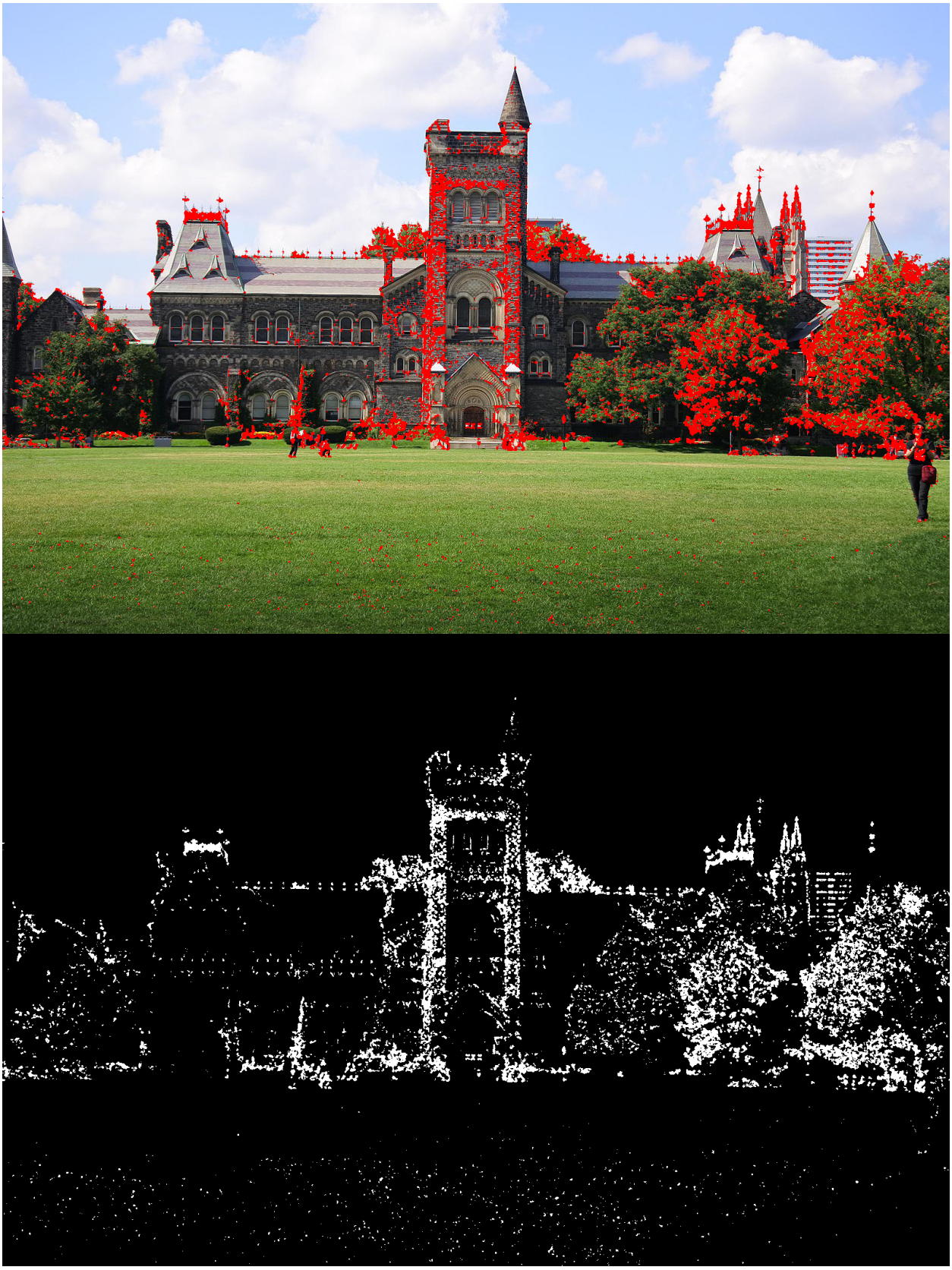

```
83  def scatterplot(src):
84      eigenvalues_1, eigenvalues_2 = eigenvalues(src,3)
85      plt.scatter(eigenvalues_1, eigenvalues_2, alpha=0.6)
86      plt.show()
```

3.

I choose threshold value of 18888 for I_1.



I choose threshold value of 18888 for I_2.



4.

When I Choose a different value for kernel size, I find that when kernel size is bigger, the corners in the picture are bigger. This makes intuition since when we apply larger Gauss filter, we make the energy of the picture more uniform. Therefore, the pixels around around the corner pixel will have value closer to the corner, and it has a wave effect. Therefore, when the Gauss filter is bigger, we will have larger corners just like the following binary images. In this situation, we actually lose the precise of really corners and get a large pixels set like the following pictures. In some cases, we may prefer tom have a large pixels set since we may want to grasp all information about corners and we do not want to miss any of them.