COMP 6341
Computer Vision

Assignment 1

Due date: Feb 14th

Zirui Qiu
40050008

Objective:

In this assignment, we implemented some classical computer vision algorithm by scratch. This report will specifically talk about the image filtering, Harris corner detector, and image subsampling with concrete examples.

Part 1(image Filtering):

In this part, we filled out the missing code in function “filiter2d.py” in “utils.py” as shown in figure. To filter each pixel in image, we need to obtain the point values around the pixel we deal with, which shows in code is image[m:m+Hk, n:n+Wk], ‘m’ and ‘n’ are current hight, width value. We use the obtained matrix to multiply the decided filter, and then we can have the results of a single filtered pixel.

```
for m in range(Hi):
    for n in range(Wi):
        ### YOUR CODE HERE (replace ??? with your code)
        image_cube = image[m:m+Hk, n:n+Wk]
        result = np.sum(image_cube * filter)
        out[m, n] = result
        ### END YOUR CODE
```

In next step, we implement the ‘partial_x’ and ‘partial_y’ function in order to find gradient which will be used in further steps. Here, we used Sobel fiter.

M_x

-1	0	1
-2	0	2
-1	0	1

M_y

1	2	1
0	0	0
-1	-2	-1

And in the actual code it should be written as:

For x derivative:

```
# define your derivative filter to be size 3*3
filter = np.array([[-1,0,1],[-2,0,2],[-1,0,1]])
out = filter2d(img, filter)
```

For y derivative:

```
# define your derivative filter to be size 3*3
filter = np.array([[1,2,1],[0,0,0],[-1,-2,-1]])
out = filter2d(img, filter)
```

Part2(Edge):

In part 2, we need to filled out in missing code of edge detector and apply it to a iguana image. The first step is applying Gaussian kernel as a filter to filter image. The purpose of this step is to smooth the image.

```
# Smooth image with Gaussian kernel
img_smotth = filter2d(img,gaussian_kernel())
```

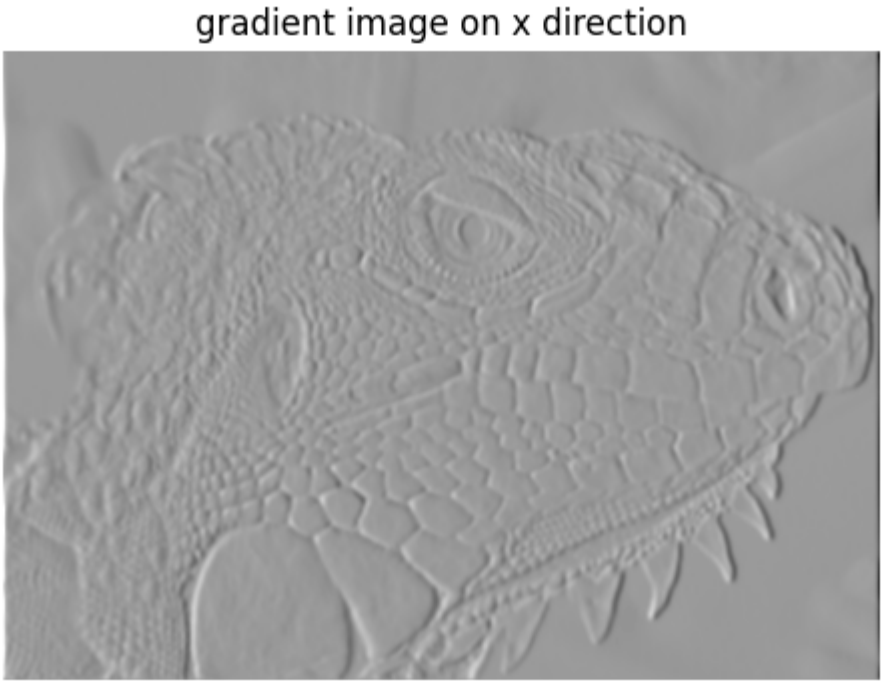
Then, we calculate x and y gradient respectively.

```
# Compute x and y derivate on smoothed image
img_deriv_x = partial_x(img_smotth)
img_deriv_y = partial_y(img_smotth)
```

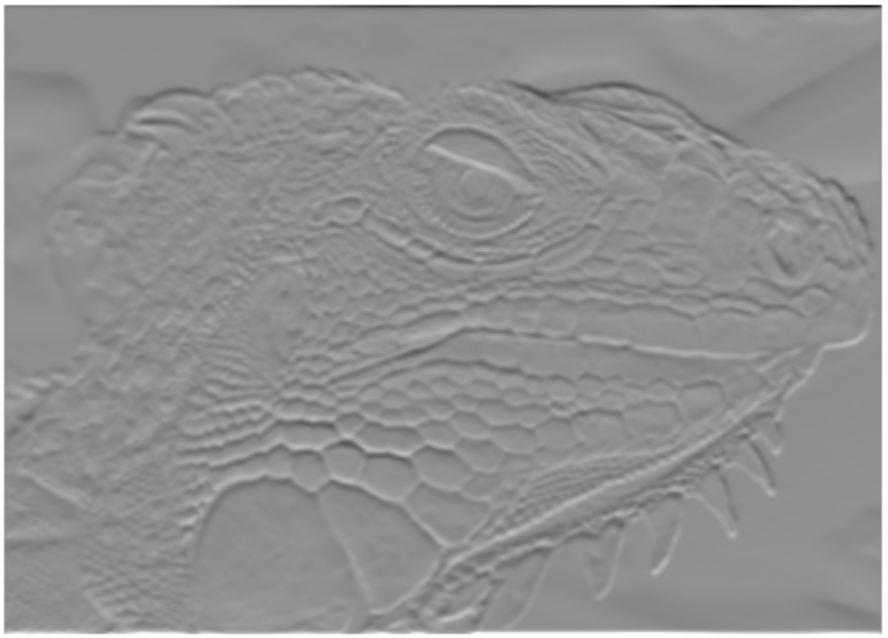
To generate the final result, we need to calculate gradient magnitude. We square root the summation of x derivate square and y derivate square.

```
# Compute gradient magnitude
Hi, Wi = img.shape
img_grad = np.zeros((Hi, Wi))
for m in range(Hi):
    for n in range(Wi):
        img_grad[m, n] = math.sqrt(img_deriv_x[m, n]** 2+ img_deriv_y[m, n]** 2)
```

The visualized results show in the figures below.



gradient image on y direction



gradient magnitude

Part 3(Corner):

In this part, we mainly implemented Harris corner detector and apply it on a building image. There are 3 major steps.

1. Computing the “corner response” map on each pixel

   The response computed on each pixel is based on the principle of Harris corner algorithm.

$$R = \det(M) - \alpha \, \mathrm{trace}(M)^2$$

   R is the response of the pixel. M is the second moment matrix of partial derivatives Ix and Iy at each pixel.

$$M = \begin{bmatrix} \sum_{x,y} w(x,y) I_x^2 & \sum_{x,y} w(x,y) I_x I_y \\ \sum_{x,y} w(x,y) I_x I_y & \sum_{x,y} w(x,y) I_y^2 \end{bmatrix}$$

   We use the response formula above to filled out the code in function ‘harris_corner’. I_x and I_y are partial derivates.

```
response = None

### YOUR CODE HERE
I_x = partial_x(img)
I_y = partial_y(img)
Ixx = filter2d(I_x**2, gaussian_kernel())
Ixy = filter2d(I_y*I_x, gaussian_kernel())
Iyy = filter2d(I_y**2, gaussian_kernel())
# determinant
detA = Ixx * Iyy - Ixy ** 2
# trace
traceA = Ixx + Iyy
#response
response = detA - k * traceA ** 2
### END YOUR CODE

return response
```

The visualized result of this step:

corner response map on each pixel

2. Thresholding on the response: In this step, we set up a threshold in order to find strong responses.

```
for rows, response in enumerate(harris_response):
    for cols, r in enumerate(response):
        if r > 0.01* harris_response.max():
            # this is a corner
            corners[rows, cols] = 255
```

The visualized result of this step:

thresholding on the response

3. The last step is to apply Non-Maximum Suppression(NMS). NMS will find out the points with the strongest response in a number of candidate points.

```
# Perform non-max suppression by finding peak local maximum
coordinates = peak_local_max(corners, min_distance=18)
```

The visualized result of this step:

Peak local max

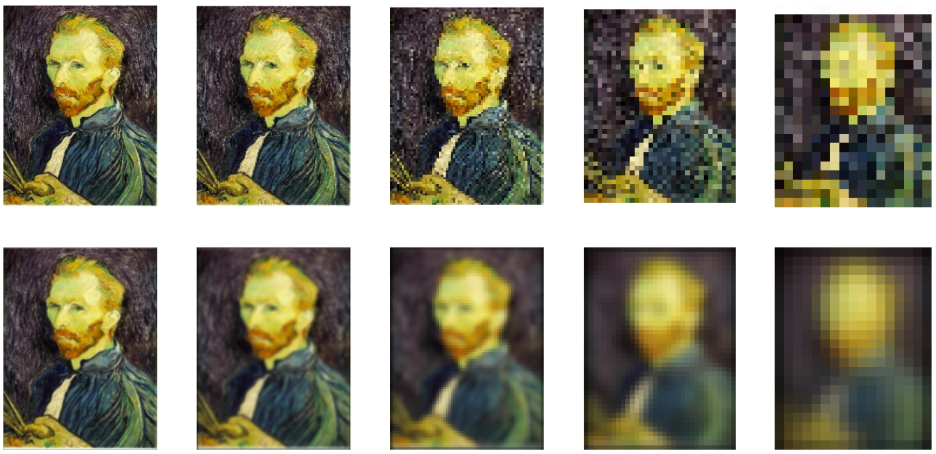
Part4(Image Subsampling):

The last part of this assignment is to subsample the image. In this step, we mainly compared the difference between subsampling with and without antialiasing. The subsampling without antialiasing is already written in the code. The only thing we need to do to realize antialiasing is applying Gasussion kernal to smooth each pixel in subsampling step.

```
# subsampling without aliasing, visualize results on 2nd row
#### YOUR CODE HERE
im_subsample_anti = im.copy()
for i in range(N_levels):
    #subsample image
    im_subsample_anti= im_subsample_anti[::2, ::2,:]
    plt.subplot(2, N_levels, i+6)
    for j in range(im_subsample_anti.shape[2]):
        im_subsample_anti[:,:,j] = filter2d(im_subsample_anti[:,:,j],gaussian_kernel())

    plt.imshow(im_subsample_anti)
    plt.axis('off')
```

The result is shown below. First row is the results of the naïve downsampling, and the second is the results with antialising.



Conclusion:

According to this assignment, we are becoming more familiar with classical computer vision algorithms. On one hand, we are consolidating our knowledge of finding edges using partial derivatives. On the other hand, we are gaining an understanding of the principle of the Harris corner detector and how non-maximum suppression is applied in corner detection. Furthermore, we have learned how a Gaussian kernel is applied in image filtering and have a clearer understanding of why the Gaussian kernel is widely used in image processing.