The Chinese University of Hong Kong, Shenzhen

CIE6004 Image Processing and Computer Vision

Homework 1 Report
Poisson Image Editing

Name: Xiang Fei

Student ID: 120090414

Email: xiangfei@link.cuhk.edu.cn

Date: 2022.10

Table of Contents

List of Figures ii

List of Tables iii

1 Introduction 1

2 Design and Method 2

2.1 Poisson Equation . . . 2

2.1.1 What is a poisson equation? . . . 2

2.1.2 Differentiation and Convolution . . . 2

2.2 Poisson Image Editing . . . 3

2.2.1 Construct the poisson equation . . . 3

2.2.2 Solve the poisson equation . . . 4

2.2.3 Determine the guided vector field . . . 6

2.3 My code implementation . . . 6

2.3.1 check_neighbors . . . 6

2.3.2 indices . . . 7

2.3.3 extract_inner_boundary . . . 7

2.3.4 importing_laplace . . . 8

2.3.5 mixing_laplace . . . 8

2.3.6 edge_detector . . . 9

2.3.7 texture_flatten . . . 10

2.3.8 get_coeff_matrix . . . 11

2.3.9 poissonImageEditing . . . 12

3 Execution 15

4 Result 17

4.1 test 1 . . . 17

4.2 test 2 . . . 17

4.3 test 3 18

4.4 test 4 18

4.5 test 5 19

5 My feeling **20**

Bibliography **21**

List of Figures

1 Poisson Image Editing based tools 1

2 The tools that I implemented in this homework 1

3 Guided interpolation notations 3

4 Solve the poisson equation 5

5 check_neighbors 6

6 indices 7

7 extract_inner_boundary 8

8 importing_laplace 8

9 mixing_laplace 9

10 edge_detector 10

11 texture_flatten 11

12 coeff_matrix 12

13 poisson1 13

14 poisson2 13

15 poisson3 14

16 poisson4 14

17 ig_example 15

18 ig_eg_file 15

19 mg_example 16

20 mg_eg_file 16

21 tf_example 16
22 tf_eg_file 16
23 merged_result_ig1 17
24 merged_result_mg1 17
25 merged_result_ig2 17
26 merged_result_mg2 18
27 merged_result_ig3 18
28 merged_result_mg3 18
29 merged_result_ig4 19
30 merged_result_mg4 19
31 merged_result_tf5 19

List of Tables

1 Introduction

In homework 1, we are required to write a program to reproduce Poisson Image Editing based on the given paper. Image editing has now been integrated into people's lives and has great application value. In the given paper, interpolation method based on solving the Poisson equation is used to construct a variety of new tools for seamless editing of image regions. The first set of tools can seamlessly cloning a transparent or opaque area of the source image to the target area, and the second set of tools is based on similar mathematical principles, which allows users to seamlessly modify the appearance of the image in the specified area. These changes can affect textures, lighting, area-specific colors within a certain range, or allow a rectangular area to be repeatedly tiled into a larger rectangular area. The tools are showed as follows.

Poisson Image Editing based tools:

- Seamless Cloning:
  - Importing gradients
  - Averaged gradients
  - Mixing gradients
- Selection Editing:
  - Texture flattening
  - Local illumination changes
  - Local color changes
  - Seamless tiling

Figure 1: Poisson Image Editing based tools

In this homework, we are mainly asked to reproduce the seamless cloning tools. I implement the importing gradients, mixing gradients, and texture flattening, which are all introduced in detail in the given paper.

The tools that I implemented in this homework

1. Importing gradients
2. Mixing gradients
3. Texture flattening

Figure 2: The tools that I implemented in this homework

2 Design and Method

2.1 Poisson Equation

2.1.1 What is a poisson equation?

The Poisson equation is a partial differential equation in mathematics commonly found in electrostatics, mechanical engineering, and theoretical physics. It is named after the French mathematician, geometer and physicist Poisson. Poisson first obtains the Poisson equation in the absence of a gravitational source, $\Delta\phi = 0$ (Laplace equation); when considering the gravitational field, there is $\Delta\phi = f$ (f is the mass distribution of the gravitational field). It is later extended to electric and magnetic fields, as well as thermal field distribution. This equation is usually solved by the Green's function method, and it can also be solved by the method of separation of variables and the method of characteristic line.

2.1.2 Differentiation and Convolution

In the computer world, data is represented in discrete space. For images, the basic computing unit is the pixel. Consider first the differentiation of one-dimensional arrays:

$\bigtriangledown$ represents the first-order differential computation at x (first-order central derivative):

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x-h)}{2h} \tag{1}$$

Δ represents the second-order differential computation at x (second-order central derivative):

$$\frac{d^2 f(x)}{dx^2} = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \tag{2}$$

As $h \to 0$, the result of the above differential formula will gradually approach the true differential value. For an image, the smallest divisible unit h here is a pixel, which means the distance between pixels, which can be regarded as 1. And it is easy to find that the second-order differential formula is the same as using a 1×3 convolution filter $[1, -2, 1]$ to a one-dimensional array (the convolution center is at x).

In fact, differentiation operations on discrete data (such as images) can be fully converted to convolution operations.

Then, consider the Laplace operator used to process image (2-dims array for every channel):

$$\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{3}$$

This time, the filter size is 3×3, and the matrix is:

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \tag{4}$$

2.2 Poisson Image Editing

2.2.1 Construct the poisson equation

After the background knowledge is ready, let's turn our attention back to the paper "Poisson Image Editing".

In the image fusion task, when the foreground is placed on the background, two points need to be guaranteed:

- The main content of the foreground itself is as **smooth** as possible compared to the background.
- The border is seamless, that is, the pixel values of the foreground and background at the border point position, and **the border needs to be consistent**.

Focus on two words: smooth content and consistent boundaries. What is smoothing? It can be understood that the foreground and background gradients of the image are the same. What does the same boundary mean? It can be understood that the pixel values are the same on the boundary. Another picture to illustrate:

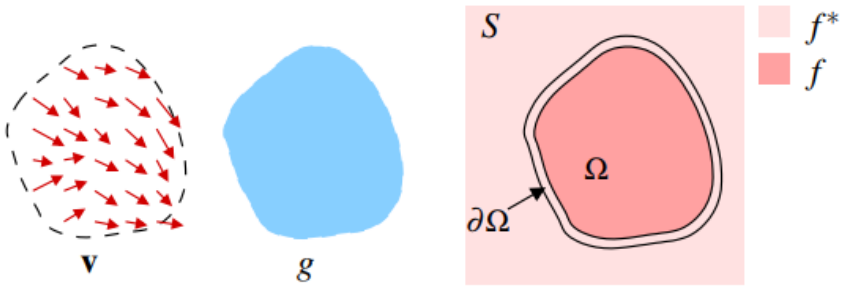

Figure 3: Guided interpolation notations

In the above figure, g represents the foreground image that needs to be synthesized, and $\mathbf{v}$ is the gradient field of g. S is the background image, Ω is the area covered by the foreground in the merged target image, and $\partial\Omega$ is the boundary of Ω. Let the pixel representation function of the merged image inside Ω be f, and the pixel value representation function outside Ω is f^*.

At this time, **smooth** can be represented as:

$$min_f \iint_\Omega |\bigtriangledown f - \mathbf{v}|^2 \tag{5}$$

The border needs to be consistent can be represented as:

$$f|_{\partial\Omega} = f^*|_{\partial\Omega} \tag{6}$$

Then, we let:

$$F = |\bigtriangledown f - \mathbf{v}|^2 = (\bigtriangledown f_x - \mathbf{v}_x)^2 + (\bigtriangledown f_y - \mathbf{v}_y)^2 \tag{7}$$

After substituting into the Euler-Lagrange equation, we have:

$$\frac{\partial F}{\partial f} = \frac{d}{dx}[\frac{\partial F}{\partial(\bigtriangledown f_x - \mathbf{v}_x)^2}] + \frac{d}{dy}[\frac{\partial F}{\partial(\bigtriangledown f_y - \mathbf{v}_y)^2}] \tag{8}$$

Therefore:

$$0 = \frac{d}{dx}[2(\bigtriangledown f_x - \mathbf{v}_x)] + \frac{d}{dy}[2(\bigtriangledown f_y - \mathbf{v}_y)] \tag{9}$$

$$0 = (\frac{\partial^2 f}{\partial x^2} - \frac{\partial \mathbf{v}}{\partial x}) + (\frac{\partial^2 f}{\partial y^2} - \frac{\partial \mathbf{v}}{\partial y}) \tag{10}$$

$$\Delta f = \mathbf{div}\ \mathbf{v} \tag{11}$$

The Poisson equation for this problem under the constraints of the Dirichlet boundary is:

$$\Delta f = \mathbf{div}\ \mathbf{v} \tag{12}$$

$$f|_{\partial\Omega} = f^*|_{\partial\Omega} \tag{13}$$

2.2.2 Solve the poisson equation

In fact, the way to solve the poisson equation is introduced in the given paper in detail, you can see the graph below:

For discrete images the problem can be discretized naturally using the underlying discrete pixel grid. Without loss of generality, we will keep the same notations for the continuous objects and their discrete counterparts: S, Ω now become finite point sets defined on an infinite discrete grid. Note that S can include all the pixels of an image or only a subset of them. For each pixel p in S, let N_p be the set of its 4-connected neighbors which are in S, and let $\langle p,q\rangle$ denote a pixel pair such that $q \in N_p$. The boundary of Ω is now $\partial\Omega = \{p \in S \setminus \Omega : N_p \cap \Omega \neq \emptyset\}$. Let f_p be the value of f at p. The task is to compute the set of intensities $f|_\Omega = \{f_p,\ p \in \Omega\}$.

For Dirichlet boundary conditions defined on a boundary of arbitrary shape, it is best to discretize the variational problem (3) directly, rather than the Poisson equation (4). The finite difference discretization of (3) yields the following discrete, quadratic optimization problem:

$$\min_{f|_\Omega} \sum_{\langle p,q\rangle \cap \Omega \neq \emptyset} (f_p - f_q - v_{pq})^2, \text{ with } f_p = f_p^*, \text{for all } p \in \partial\Omega, \quad (6)$$

where v_{pq} is the projection of $\mathbf{v}(\frac{p+q}{2})$ on the oriented edge $[p,q]$, i.e., $v_{pq} = \mathbf{v}(\frac{p+q}{2}) \cdot \vec{pq}$. Its solution satisfies the following simultaneous linear equations:

$$\text{for all } p \in \Omega, \quad |N_p| f_p - \sum_{q \in N_p \cap \Omega} f_q = \sum_{q \in N_p \cap \partial\Omega} f_q^* + \sum_{q \in N_p} v_{pq}. \quad (7)$$

When Ω contains pixels on the border of S, which happens for instance when Ω extends to the edge of the pixel grid, these pixels have a truncated neighborhood such that $|N_p| < 4$. Note that for pixels p interior to Ω, that is, $N_p \subset \Omega$, there are no boundary terms in the right hand side of (7), which reads:

$$|N_p| f_p - \sum_{q \in N_p} f_q = \sum_{q \in N_p} v_{pq}. \quad (8)$$

Figure 4: Solve the poisson equation

And the last equation in the above graph can be changed to a linear system of equations:

$$Af = b \quad (14)$$

The right side of the equal sign is the Laplacian filter result for each pixel in the image g, which is easy to understand. Each element of the unknown function f constitutes the column vector to the left of the equal sign. The coefficient matrix A describes the most critical Laplacian filter operator of f.

But here, for boundary points, since their neighbors are not 4, so we can not just use the laplace operator I introduced before. We need to do the neighbor analysis first, so that we can then construct the coefficient matrix.

2.2.3 Determine the guided vector field

Importing gradients:

Just use the gradients of the source image

Mixing gradients:

Use the larger gradients between the source image and the target image

Texture gradients:

Use a binary mask times the gradients of the target image (here in fact, target and source images are the same), A good choice for the mask is an edge detector (will be introduced in detail later)

2.3 My code implementation

In this part, I will introduce my code implementation through illustrate the functions I have written in detail.

2.3.1 check_neighbors

This function is used to check whether a pixel has a neighbor in omega area for one of the four directions. The input is the binary mask, and the neighbor pixel coordinate to be checked. First, we get the height and width. And then, we check whether the neighbor pixel to be checked is masked by the input mask. If not, then it is a neighbor. In fact, when we use it to check pixel (x,y)'s neighbor, we just invoke this function four times and pass (x+1,y), (x-1,y), (x,y+1), (x,y-1) respectively, and the return value is a boolean variable for each direction.

```
def check_neighbors(mask, y, x):

    h, w = mask.shape

    if((y>=0) and (y<=h-1) and (x>=0) and (x<=w-1)):
        if(mask[y][x]==1):
            return True
        else:
            return False
    else:
        return False
```

Figure 5: check_neighbors

2.3.2 indices

This function is used to get whether a pixel has neighbor pixel (four directions) and a list for omega point's index, which is very important for later processing (As I said before, it is used to compute the coefficient matrix and the right hand array b in the linear system). The input of this function is the binary mask. We first get the height and width of the mask. Then, we use some operators in numpy such as nonzero, reshape, concatenate to get a list omega_list to store the valid (not be masked) pixels' coordinates. And then, for each valid pixel, we check its neighbors, and use a list has_neighbor to store the information. The elements in this list is also a list, whose first value is whether the pixel has right neighbor, then is left, bottom, up. Last, we use a list omega_yx to store the index of the valid pixels' coordinate for later processing. The return is these three lists.

```python
def indices(mask):
  #To get whether a pixel has neighbor pixel and a list for omega's index

  ## height and width of mask
  h, w = mask.shape

  ## get the coordinates that the elements are not 0 in mask, which is the omega area.
  omega = np.nonzero(mask)
  x = np.reshape(omega[1],[omega[1].shape[0],1])
  y = np.reshape(omega[0],[omega[0].shape[0],1])
  omega_list = np.concatenate([y,x],1)

  has_neighbor = []
  omega_yx = np.zeros((h,w),dtype=np.int32)
  for index in range(omega_list.shape[0]):

    ## pixel location
    i, j = omega_list[index]

    ## check whether there are neighbors (right, left, down, up)
    has_neighbor.append([check_neighbors(mask,i,j+1),check_neighbors(mask,i,j-1),check_neighbors(mask,i+1,j),check_neighbors(mask,i-1,j)])

    ## store index to list
    omega_yx[i][j] = index

  return omega_list, np.array(has_neighbor,dtype=bool), omega_yx
```

Figure 6: indices

2.3.3 extract_inner_boundary

This function is used to get the mask inner boundary from the original binary mask image using erosion. In fact, erosion is generally used to extract the inner boundary, which consists of boundary points and is part of the region. The extraction of the inner boundary uses the erosion of the image to obtain a shrinkage of the original image, and then performs the XOR operation on the shrinkage result and the target image to realize the extraction of the difference part. The information can be used to determine whether a valid pixel is in boundary.

```python
def extract_inner_boundary(mask):

    ## This function is used to get the mask inner_boundary from the original binary mask image using erosion
    mask = np.array(mask, dtype=np.uint8)
    inner_boundary = mask ^ cv2.erode(mask, np.ones((3, 3), np.uint8))

    return inner_boundary
```

Figure 7: extract_inner_boundary

2.3.4 importing_laplace

In this function, I compute the laplace for all b, g, r channels. For every pixel in omega area, we first get the coordinates of the pixel, and then, we compute the laplace of this pixel for the source image. After that, I consider the dirichlet condition and compute the dirichlet condition value, which is the sum of the neighbor target pixels when the considered pixel in at the boundary, otherwise, this value is 0. And the elements of the variable u in this function stores the sum of the laplace for source image and the dirichlet condition value for each pixel, which can be used as the b in the linear systems I introduced before.

```python
def importing_laplace(source, target, omega, inner_boundary, has_neighbor):

  u = np.zeros((3,omega.shape[0]))
  for k in range(3):
    for index in range(omega.shape[0]):
      ## current location
      i, j = omega[index]
      label = has_neighbor[index]
      source_k = source[:,:,k]
      target_k = target[:,:,k]

      # laplace for source image
      source_grad_value = (label[0]==True)*(source_k[i,j]-source_k[i,j+1])+(label[1]==True)*(source_k[i,j]-source_k[i,j-1])\
          +(label[2]==True)*(source_k[i,j]-source_k[i+1,j])+(label[3]==True)*(source_k[i,j]-source_k[i-1,j])

      ## consider the dirichlet boundary condition
      if (inner_boundary[i][j])==1:
        dirichlet_condition_value = (label[0]==False)*target_k[i,j+1]+(label[1]==False)*target_k[i,j-1]\
          +(label[2]==False)*target_k[i+1,j]+(label[3]==False)*target_k[i-1,j]
      else:
        dirichlet_condition_value = 0

      u[k,index] = source_grad_value + dirichlet_condition_value

  return u
```

Figure 8: importing_laplace

2.3.5 mixing_laplace

In this function, I compute the laplace for all b, g, r channels. For every pixel in omega area, we first get the coordinates of the pixel, and then, we compute the laplace of this pixel for both the source image and target image. After that, I compare the laplace for source image and the laplace for target image and choose the larger one as the guided vector. And I also consider the dirichlet condition and compute the dirichlet condition value, which is the sum of the neighbor target pixels when the considered pixel in at the boundary, otherwise, this value is 0. And the elements of the variable u in this function stores the sum of the laplace

for the guided one and the dirichlet condition value for each pixel, which can be used as the b in the linear systems I introduced before.

```
def mixing_laplace(source, target, omega, inner_boundary, has_neighbor):
  u = np.zeros((3,omega.shape[0]))
  for k in range(3):
    for index in range(omega.shape[0]):
      ## current location
      i, j = omega[index]
      label = has_neighbor[index]
      source_k = source[:,:,k]
      target_k = target[:,:,k]
      ## laplace for source image
      source_grad_r = (label[0]==True) * (source_k[i, j] - source_k[i, j+1])
      source_grad_l = (label[1]==True) * (source_k[i, j] - source_k[i, j-1])
      source_grad_b = (label[2]==True) * (source_k[i, j] - source_k[i+1, j])
      source_grad_u = (label[3]==True) * (source_k[i, j] - source_k[i-1, j])
      ## laplace for target image
      target_grad_r = (label[0]==True) * (target_k[i, j] - target_k[i, j+1])
      target_grad_l = (label[1]==True) * (target_k[i, j] - target_k[i, j-1])
      target_grad_b = (label[2]==True) * (target_k[i, j] - target_k[i+1, j])
      target_grad_u = (label[3]==True) * (target_k[i, j] - target_k[i-1, j])

      source_grad_value = []
      source_grad_value.append(source_grad_r if (abs(source_grad_r)>abs(target_grad_r)) else target_grad_r)
      source_grad_value.append(source_grad_l if (abs(source_grad_l)>abs(target_grad_l)) else target_grad_l)
      source_grad_value.append(source_grad_b if (abs(source_grad_b)>abs(target_grad_b)) else target_grad_b)
      source_grad_value.append(source_grad_u if (abs(source_grad_u)>abs(target_grad_u)) else target_grad_u)
      ## consider the dirichlet boundary condition
      if (inner_boundary[i][j])==1:
        dirichlet_condition_value = (label[0]==False)*target_k[i,j+1]+(label[1]==False)*target_k[i,j-1]\
          +(label[2]==False)*target_k[i+1,j]+(label[3]==False)*target_k[i-1,j]
      else:
        dirichlet_condition_value = 0

      u[k,index] = sum(source_grad_value) + dirichlet_condition_value

  return u
```

Figure 9: mixing_laplace

2.3.6 edge_detector

This function is used to construct the edge detector, which is used to determine the binary mask for texture flattening. We first use cv2.Canny to detect the original edge. And we can use weight parameter to change the thickness of the edge. When we increase weight, the edge will become thicker, which means that more characters of the source image in the later texture flattening task will be preserved.

```python
def edge_detector(gray, weight):
    ## implement the edge detector for texture flattening, use weight to change the thickness of edge.

    ### get edge from filter
    original_edge = cv2.Canny(gray, 100, 200)
    edge = np.zeros((original_edge.shape[0], original_edge.shape[1]), dtype=np.uint8)

    ### make edge bold by using weight source_grad_valueue
    for i in range(edge.shape[0]):
        for j in range(edge.shape[1]):
            if(original_edge[i][j] != 0):
                for m in range(-weight, weight):
                    for n in range(-weight, weight):
                        edge_i = min(max(0,i+m), edge.shape[0]-1)
                        edge_j = min(max(0,j+n), edge.shape[1]-1)
                        edge[edge_i][edge_j] = 1

    return edge
```

Figure 10: edge_detector

2.3.7 texture_flatten

In this function, I compute the laplace for all b, g, r channels. First, I use cv2.cvtColor and pass cv2.COLOR_BGR2GRAY to get the grey image of the source image. And then, use the edge_detector function I introduced before to get the edge mask. After that, for every pixel in omega area, I first get the coordinates of the pixel, and then, we compute the laplace of this pixel for the source image (here, we need to consider both the neighbor label and edge mask label). In addition, I consider the dirichlet condition and compute the dirichlet condition value, which is the sum of the neighbor target pixels when the considered pixel in at the boundary, otherwise, this value is 0. And the elements of the variable u in this function stores the sum of the laplace for source image and the dirichlet condition value for each pixel, which can be used as the b in the linear systems I introduced before.

```python
def texture_flatten(source, target, omega, inner_boundary, has_neighbor):
    u = np.zeros((3,omega.shape[0]))
    gray = cv2.cvtColor(source, cv2.COLOR_BGR2GRAY)
    edge_mask = edge_detector(np.array(gray*255, dtype=np.uint8),10)

    for k in range(3):
        for index in range(omega.shape[0]):
            i, j = omega[index]
            label = has_neighbor[index]
            source_k = source[:,:,k]
            target_k = target[:,:,k]
            ## laplace for source image
            source_grad_value = (label[0]==True)*(source_k[i,j]-source_k[i,j+1])*(edge_mask[i][j])+\
                (label[1]==True)*(source_k[i,j]-source_k[i,j-1])*(edge_mask[i][j-1])+\
            (label[2]==True)*(source_k[i,j]-source_k[i+1,j])*(edge_mask[i][j])+\
                (label[3]==True)*(source_k[i,j]-source_k[i-1,j])*(edge_mask[i-1][j])

    ## consider the dirichlet boundary condition
            if (inner_boundary[i][j])==1:
                dirichlet_condition_value = (label[0]==False)*target_k[i,j+1]+(label[1]==False)*target_k[i,j-1]\
                    +(label[2]==False)*target_k[i+1,j]+(label[3]==False)*target_k[i-1,j]
            else:
                dirichlet_condition_value = 0

            u[k,index] = source_grad_value + dirichlet_condition_value

    return u
```

Figure 11: texture_flatten

2.3.8 get_coeff_matrix

This function is used to construct the coefficient matrix A in the linear system I introduced before. Since only a few of elements of matrix A are nonzeros, so I use sparse matrix to represent A and I use the lil_matrix function is scipy to implement it. And then, for any valid pixel, for the center point, the value in A is 4, and the neighbors are all -1.

```python
def get_coeff_matrix(omega_list, has_neighbor, omega_yx):

    ## get the coefficient matrix A

    ## create empty sparse matrix
    num = omega_list.shape[0]
    A = sp.lil_matrix((num,num), dtype=np.float32)
    dic = {0:[0,1],1:[0,-1],2:[1,0],3:[-1,0]}

    for i in range(num):
        ## fill 4 or -1
        ## center
        A[i, i] = 4
        y, x = omega_list[i]

        for k in range(4):
            if(has_neighbor[i][k]):
                j = omega_yx[y+dic[k][0]][x+dic[k][1]]
                A[i,j] = -1

    return A
```

Figure 12: coeff_matrix

2.3.9 poissonImageEditing

This function can be divided into 4 parts, the used function are introduced very detailed before. The only thing I need to replenish is that I use the linalg.cg in scipy to solve the linear system.

The first part: get the coefficient matrix A

```python
def poissonImageEditing(src, mask, tar, method):
    ### create inner_boundary mask
    inner_boundary = extract_inner_boundary(mask) # uint8
    mask = np.array(mask, dtype=np.uint8)
    ### get omega, neighborhoods flag
    omega, has_neighbor, yx_omega = indices(mask)
    print("start the operation\n")
    ### fill A
    print("step1: filling coefficient matrix: A")
    A = get_coeff_matrix(omega, has_neighbor, yx_omega)
```

Figure 13: poisson1

The second part: full fill the b in linear system (here is u) for each color channel

```python
### fill u for each color channel
print("step2: filling gradient matrix: b")
u = np.zeros((3,omega.shape[0]))
u_b = np.zeros(omega.shape[0])
u_g = np.zeros(omega.shape[0])
u_r = np.zeros(omega.shape[0])
## select process type
if(method == "ig"): ## importing laplace
    u = importing_laplace(src, tar, omega, inner_boundary, has_neighbor)
    u_b = u[0,:]
    u_g = u[1,:]
    u_r = u[2,:]
if(method == "mg"): ## mixing laplace
    u =  mixing_laplace(src, tar, omega, inner_boundary, has_neighbor)
    u_b = u[0,:]
    u_g = u[1,:]
    u_r = u[2,:]
if(method == "tf"): ## texture flattening
    u = texture_flatten(src, tar, omega, inner_boundary, has_neighbor)
    u_b = u[0,:]
    u_g = u[1,:]
    u_r = u[2,:]
```

Figure 14: poisson2

The third part: solve the linear system

```
### solve
print("step3: solve Au = b\n")
x_b, info_b = sp.linalg.cg(A, u_b)
x_g, info_g = sp.linalg.cg(A, u_g)
x_r, info_r = sp.linalg.cg(A, u_r)
print("The operation is finishes!\n")
```

Figure 15: poisson3

The fourth part: replace the omega area with the computed pixel

```
### create output by using x
seamless_cloning = tar.copy()
cloning = tar.copy()

for index in range(omega.shape[0]):

    i, j = omega[index]

    ## normal
    seamless_cloning[i][j][0] = np.clip(x_b[index], 0.0, 1.0)
    seamless_cloning[i][j][1] = np.clip(x_g[index], 0.0, 1.0)
    seamless_cloning[i][j][2] = np.clip(x_r[index], 0.0, 1.0)

    ## cloning
    cloning[i][j][0] = src[i][j][0]
    cloning[i][j][1] = src[i][j][1]
    cloning[i][j][2] = src[i][j][2]


return (np.array(seamless_cloning*255, dtype=np.uint8),
        np.array(cloning*255, dtype=np.uint8))
```

Figure 16: poisson4

3 Execution

poisson.py and main.py should be placed in the same folder.

```
python main.py argvs[1] argvs[2] argvs[3]
```

argvs[1] : path to source image

argvs[2] : path to mask image

argvs[3] : path to target image (for texture flattening, it is the same as the source image)

argvs[4] : method (for importing gradients, it is 'ig'; for mixing gradients, it is 'mg'; for texture flattening, it is 'tf')

And there will generate a folder result, which consists the simple cloning image, the result image, and the merged image.

For example:

importing case:

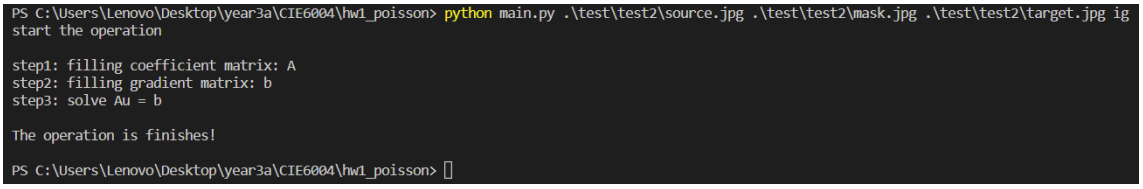

Figure 17: ig_example

The generated folder and files:

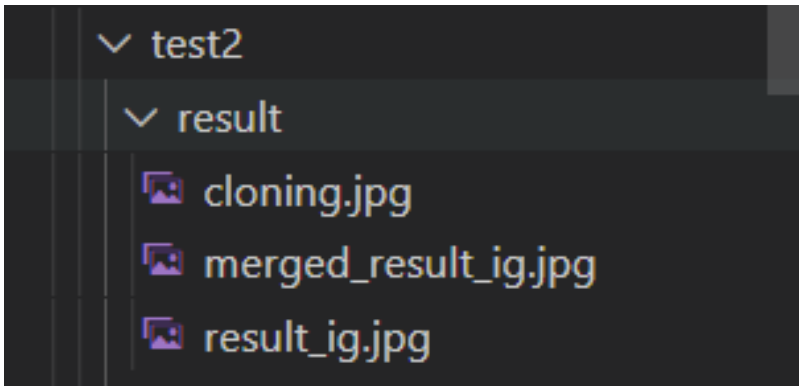

Figure 18: ig_eg_file

mixing case:

```
PS C:\Users\Lenovo\Desktop\year3a\CIE6004\hw1_poisson> python main.py .\test\test2\source.jpg .\test\test2\mask.jpg .\test\test2\target.jpg mg
start the operation

step1: filling coefficient matrix: A
step2: filling gradient matrix: b
step3: solve Au = b

The operation is finishes!

PS C:\Users\Lenovo\Desktop\year3a\CIE6004\hw1_poisson>
```

Figure 19: mg_example

The generated folder and files:

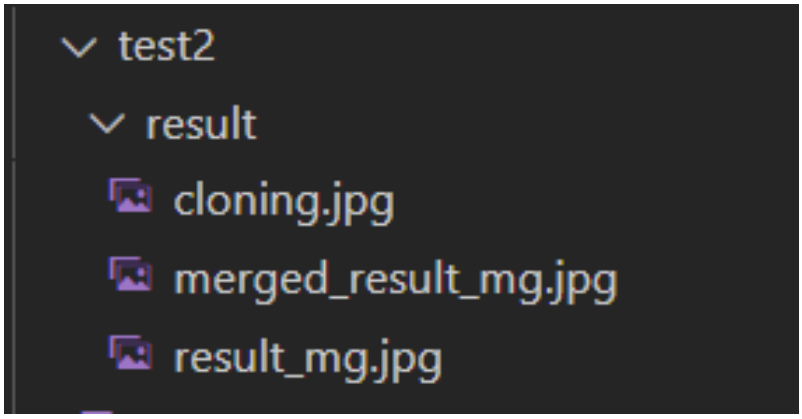

Figure 20: mg_eg_file

texture flatten case:

```
PS C:\Users\Lenovo\Desktop\year3a\CIE6004\hw1_poisson> python main.py .\test\test5\source.jpg .\test\test5\mask.png .\test\test5\source.jpg tf
start the operation

step1: filling coefficient matrix: A
step2: filling gradient matrix: b
step3: solve Au = b

The operation is finishes!

PS C:\Users\Lenovo\Desktop\year3a\CIE6004\hw1_poisson>
```

Figure 21: tf_example

The generated folder and files:

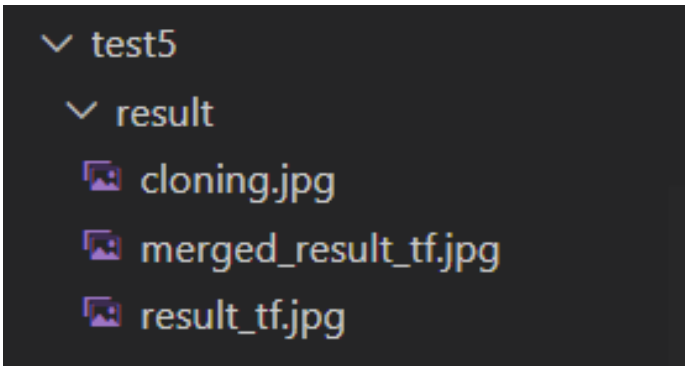

Figure 22: tf_eg_file

4 Result

4.1 test 1

importing gradients case:

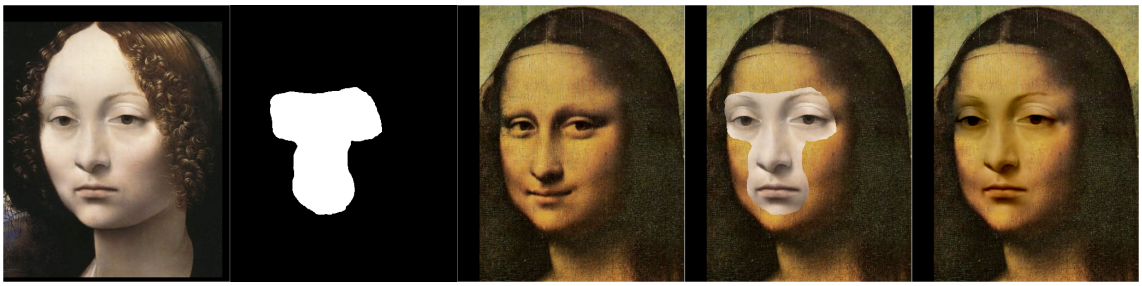

Figure 23: merged_result_ig1

The first is the source image, the second is the mask, the third is the target image, the fourth is the cloning result, the fifth is the importing gradients result.

mixing gradients case:

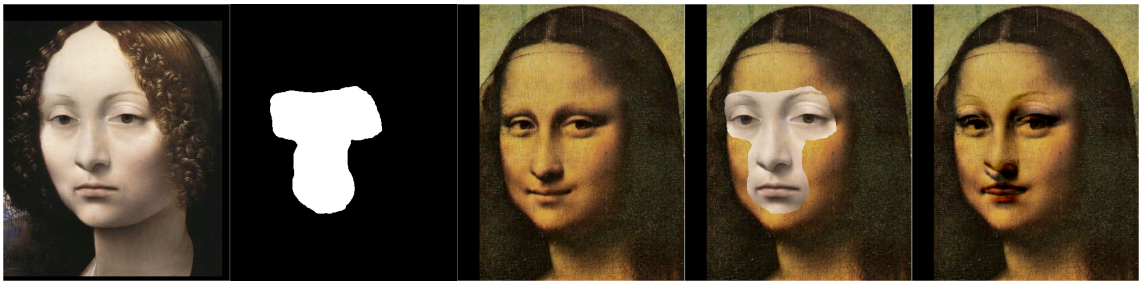

Figure 24: merged_result_mg1

The first is the source image, the second is the mask, the third is the target image, the fourth is the cloning result, the fifth is the mixing gradients result.

4.2 test 2

importing gradients case:

Figure 25: merged_result_ig2

The first is the source image, the second is the mask, the third is the target image, the fourth is the cloning result, the fifth is the importing gradients result.

mixing gradients case:

Figure 26: merged_result_mg2

The first is the source image, the second is the mask, the third is the target image, the fourth is the cloning result, the fifth is the mixing gradients result.

4.3 test 3

importing gradients case:

Figure 27: merged_result_ig3

The first is the source image, the second is the mask, the third is the target image, the fourth is the cloning result, the fifth is the importing gradients result.

mixing gradients case:

Figure 28: merged_result_mg3

The first is the source image, the second is the mask, the third is the target image, the fourth is the cloning result, the fifth is the mixing gradients result.

4.4 test 4

importing gradients case:

Figure 29: merged_result_ig4

The first is the source image, the second is the mask, the third is the target image, the fourth is the cloning result, the fifth is the importing gradients result.

mixing gradients case:

Figure 30: merged_result_mg4

The first is the source image, the second is the mask, the third is the target image, the fourth is the cloning result, the fifth is the mixing gradients result.

4.5 test 5

texture flattening case:

Figure 31: merged_result_tf5

The first is the source image, the second is the mask, the third is the target image (the same as source image), the fourth is the cloning result (the same as source image), the fifth is the texture flattening result.

5 My feeling

As an undergraduate, this was my first exposure to the field of image editing, and I thought Poisson image editing was really a very elegant approach. I feel that I have gained a lot from the whole process this time.

First of all, after my study, I think that the essence of image editing here is to modify the gradient of the image, which I think is very important for my understanding. And I also read the given thesis very seriously, I think this is a very good exercise for me, and it has cultivated my ability to study independently through the thesis. This time I also implemented Poisson image editing from a very low level. I think it is very helpful for me to clearly understand the problem and my ability to code.

However, I also encountered many difficulties in this study. First of all, reading a paper is not an easy process. It is difficult for me to understand this problem very well by relying on the full English paper, so I also found a lot of relevant information on the Internet, which is very helpful for me to understand this problem. Secondly, I didn't understand the method of solving the Poisson equation in the paper very well at first, but by looking up some github source code and some online blogs, I understood that the laplace operator here is not a traditional form, but It is to consider that the four neighborhoods of the boundary point do not necessarily have pixels. I also encountered a lot of difficulties in the implementation of the code, because I am not familiar with the commonly used interfaces in the field of computer vision such as cv2, which makes me need to check the information frequently. And for the implementation of texture flattening, the paper's description of the edge detection mask is very detailed. For the implementation of this piece, I also need to learn and understand the relevant knowledge.

All in all, I learned a lot through this homework!!!

Bibliography

Pérez, Patrick, Michel Gangnet and Andrew Blake (2003). 'Poisson Image Editing'. In: *ACM SIGGRAPH 2003 Papers*. SIGGRAPH '03. San Diego, California: Association for Computing Machinery, pp. 313–318. ISBN: 1581137095. DOI: 10.1145/1201775.882269. URL: https://doi.org/10.1145/1201775.882269.