

CS172 Computer Vision I:

Homework 1

Zhiyuan Liang
30459281

liangzhy@shanghaitech.edu.cn

Abstract

The homework is to implement the generic interpolation machinery of poisson blending.[1]This technology makes seamless cloning possible.

1. Introduction

The task is to seamlessly clone a part of the source image to an area of the destination image in a generative way. The basic idea of the paper is to solve the optimization problem $\min_f \iint_{\Omega} \|\nabla f - v\|^2$ s.t. $f|_{\partial\Omega} = f^*|_{\partial\Omega}$ whose solution is $\Delta f = \text{div } v$. The constraint keeps the generative image is seamless. The goal function makes the generative image keep similar to the source image and continuous with the boundary of destination.

2. Implementaion

2.1. Usage

The workspace contains /img folder and main.py and poisson.py. In /img folder, there are different folders. Each folder corresponds to an artwork and contains dst.jpg, src.jpg and mask.jpg which are all trimmed and aligned to the same size. The mask.jpg contains only pixels whose values are 0 and 255.

```
python3 main.py -f foldername
```

Then the artwork will be generated in the /img/foldername.

```
python3 main.py -f foldername -m
```

-m flag can be added to use mixing gradient method to calculate divergence.

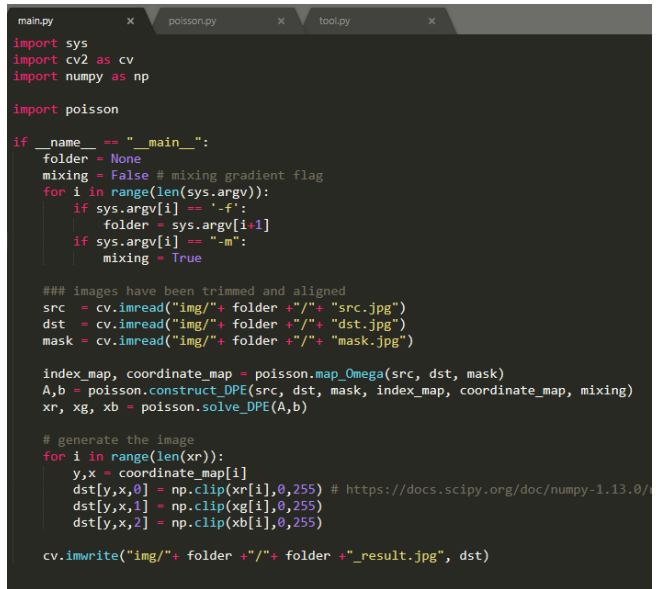
2.2. Code Structure

In main.py, I handle the input argument of program, invoke map_Omega, construct_DPE and solve_DPE defined in poisson.py and finally I generate the image with the results returned by the solve_DPE function.

In poisson.py, I implement how to construct the discrete poisson equation and how to solve the equation.

2.3. Implementaion Details

2.3.1 main.py



```
import sys
import cv2 as cv
import numpy as np

import poisson

if __name__ == "__main__":
    folder = None
    mixing = False # mixing gradient flag
    for i in range(len(sys.argv)):
        if sys.argv[i] == '-f':
            folder = sys.argv[i+1]
        if sys.argv[i] == '-m':
            mixing = True

    ### images have been trimmed and aligned
    src = cv.imread("img/" + folder + "/" + "src.jpg")
    dst = cv.imread("img/" + folder + "/" + "dst.jpg")
    mask = cv.imread("img/" + folder + "/" + "mask.jpg")

    index_map, coordinate_map = poisson.map_Omega(src, dst, mask)
    A, b = poisson.construct_DPE(src, dst, mask, index_map, coordinate_map, mixing)
    xr, xg, xb = poisson.solve_DPE(A, b)

    # generate the image
    for i in range(len(xr)):
        y, x = coordinate_map[i]
        dst[y, x, 0] = np.clip(xr[i], 0, 255) # https://docs.scipy.org/doc/numpy-1.13.0/
        dst[y, x, 1] = np.clip(xg[i], 0, 255)
        dst[y, x, 2] = np.clip(xb[i], 0, 255)

    cv.imwrite("img/" + folder + "/" + folder + "_result.jpg", dst)
```

The main.py is trivial and the code is easy to understand. In main.py, I handle the input argument of program, invoke map_Omega, construct_DPE and solve_DPE defined in poisson.py and finally I generate the image with the results returned by the solve_DPE function.

2.3.2 poisson.py

There are three functions defined in this poisson.py.

The map_Omega function is to map vector x (index of list) with image pixels (coordinate of pixel).

All pixels whose values are 255 in mask define the area Ω to clone the source image in the destination image. The

```

main.py x poisson.py x tool.py x
import numpy as np
from scipy import sparse
from scipy.sparse import linalg
...
input: source, destination and mask image
output: index_map and coordinate_map
...
### map vector x with image pixels
def map_Omega(src,dst,mask):
...
input: source, destination, mask image, index_map, coordinate_map and mixing gradient flag
output: A and b (the discrete poisson equation)
...
### construct Discrete Poisson Equation
def construct_DPE(src, dst, mask, index_map, coordinate_map, mixing=False):
...
input: A and b (the discrete poisson equation)
output: xr, yg and xb (solutions of the equation)
...
### solve Discrete Poisson Equation in rgb channels respectively
def solve_DPE(A,b):

```

```

...
input: source, destination and mask image
output: index_map and coordinate_map
...
### map vector x with image pixels
def map_Omega(src,dst,mask):
    h,w,_ = src.shape

    coordinate_map = []
    index_map = np.zeros([h,w])
    idx = 0
    for i in range(h):
        for j in range(w):
            if mask[i,j,0] == 255:
                coordinate_map.append((i,j))
                index_map[i,j] = idx
                idx += 1
    return index_map, coordinate_map

```

boundary of pixels of 255 and pixels of 0 are the boundary of Ω , which is $\partial\Omega$. All interior pixels in the area Ω will be the unknown vector x , which is the f in the formulation mentioned in introduction. I store the all interior pixels' coordinates in a list and the index of them in a 2-D array with the same size as the image.

The tricky part is to construct the poisson equation $Ax = b$. The construction contains two parts: one is to design matrix A which comes from laplacian operator and calculate b which comes from the guiding gradient of the source image. The diagonal of the matrix A are all 4 and each row is composed of four -1 in the columns which is related to the up,down,left and right pixels of the current pixel(which is an element of the vector x) and a lot of zero. This is the divergence of the pixel.

The b corresponds to each pixel in the the area Ω . Each element is the divergence of the pixel in that location in the source image. For those pixels in the boundary $\partial\Omega$, the corresponding element in b is the pixel values plus the di-

```

...
input: source, destination, mask image, index_map, coordinate_map and mixing gradient flag
output: A and b (the discrete poisson equation)
...
### construct Discrete Poisson Equation
def construct_DPE(src, dst, mask, index_map, coordinate_map, mixing=False):
    h,w,c = src.shape
    A = linalg.sparse_linalg
    b = np.zeros([h,w])
    for i in range(h):
        A[i,i] = 4
        y,x = coordinate_map[i]
        ...
        if not mixing:
            # if y is very important, force it to be float
            b[i, :] = 1. * src[y, x, :] - src[y - 1, x, :] - src[y + 1, x, :] - src[y, x - 1, :] - src[y, x + 1, :] # calculate divergence
        else:
            # mix mixing gradient
            if abs(np.mean(src[y, x, :]) - np.mean(src[y - 1, x, :])) > abs(np.mean(dst[y, x, :]) - np.mean(dst[y - 1, x, :])):
                b[i, :] = 1. * src[y, x, :] - src[y - 1, x, :] # y is very important, force it to be float
            else:
                b[i, :] = 1. * dst[y, x, :] - dst[y - 1, x, :] # x is very important, force it to be float
            if abs(np.mean(src[y, x, :]) - np.mean(src[y + 1, x, :])) > abs(np.mean(dst[y, x, :]) - np.mean(dst[y + 1, x, :])):
                b[i, :] = 1. * src[y, x, :] - src[y + 1, x, :] # y is very important, force it to be float
            else:
                b[i, :] = 1. * dst[y, x, :] - dst[y + 1, x, :] # x is very important, force it to be float
            if abs(np.mean(src[y, x, :]) - np.mean(src[y, x - 1, :])) > abs(np.mean(dst[y, x, :]) - np.mean(dst[y, x - 1, :])):
                b[i, :] = 1. * src[y, x, :] - src[y, x - 1, :] # x is very important, force it to be float
            else:
                b[i, :] = 1. * dst[y, x, :] - dst[y, x - 1, :] # y is very important, force it to be float
            if abs(np.mean(src[y, x, :]) - np.mean(src[y, x + 1, :])) > abs(np.mean(dst[y, x, :]) - np.mean(dst[y, x + 1, :])):
                b[i, :] = 1. * src[y, x, :] - src[y, x + 1, :] # x is very important, force it to be float
            else:
                b[i, :] = 1. * dst[y, x, :] - dst[y, x + 1, :] # y is very important, force it to be float
        ...
    return A,b

```

```

...
input: A and b (the discrete poisson equation)
output: xr, yg and xb (solutions of the equation)
...
### solve Discrete Poisson Equation in rgb channels respectively
def solve_DPE(A,b):
    ### Too slow !!!
    # xr,_ = np.linalg.lstsq(A,b[:,0], rcond=None) # https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lstsq.html
    # yg,_ = np.linalg.lstsq(A,b[:,1], rcond=None)
    # xb,_ = np.linalg.lstsq(A,b[:,2], rcond=None)

    # A is a sparse matrix with a few 4 and -1 and tons of 0. Then
    A = sparse.lil_matrix(A, dtype=int) # https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.lil_matrix.html
    # Conjugate Gradient Iteration
    xr,_ = linalg.cg(A, b[:, 0]) # https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.cg.html
    yg,_ = linalg.cg(A, b[:, 1])
    xb,_ = linalg.cg(A, b[:, 2])
    return xr,xg,xb

```

vergence, which implies the constraint that $f|_{\partial\Omega} = f^*|_{\partial\Omega}$ whose solution is $\Delta f = \text{div } v$.

The mixing gradient is also implemented. The difference is to select the gradient with higher absolute value of source image and destination(background) image as guiding gradient.

To solve the poisson equation $Ax = b$ which is a typical linear regression problem, I just use the scipy lib to help me to solve the equation. Since A is a very sparse matrix, I use the sparse module in scipy to solve this, it is much faster than normal least square function in numpy.

3. Result

All results can be found in /img folder. Here, I choose some interesting cases.



(a) src



(c) basic method



(b) result



(d) mixing gradient

References

- [1] Patrick Pérez, Michel Gangnet, and Andrew Blake. Poisson image editing. In *ACM SIGGRAPH 2003 Papers*, pages 313–318. 2003.