

数字图像处理第一次大作业报告

计 64 翁家翌 2016011446

2018.4

目录

1 测试环境	2
2 Point Processing	2
2.1 实现方式	2
2.2 运行说明	2
2.3 实验结果	2
2.3.1 明度变化	2
2.3.2 对比度变化	2
2.3.3 Gamma 变换	4
2.3.4 直方图均衡化	4
2.3.5 直方图匹配	4
3 Image Fusion	6
3.1 实现方式	6
3.2 运行说明	6
3.3 算法细节	6
3.4 实验结果	7
3.4.1 Test1	7
3.4.2 Test2	7
4 Face Morphing	8
4.1 实现方式	8
4.2 运行说明	9
4.3 算法细节	9
4.4 实验结果	10
4.4.1 人脸关键点 +Delaunay 三角剖分	10
4.4.2 Cruz&Hillary	10
4.4.3 Hilary&Trump	10
4.4.4 Trump&Cruz	10
4.4.5 Face++ API	10

1 测试环境

本套程序在 Ubuntu 14.04 LTS 和 Ubuntu 16.04 LTS 上使用 Python2/Python3 均测试通过。在运行之前，请安装 python 的第三方包 `opencv` 和 `numpy`，使用命令

```
1 pip install -r requirements.txt --user
```

即可安装。

2 Point Processing

2.1 实现方式

该部分仅需要读取图片中的每个点的颜色数据进行相应的处理即可。我实现了明度、对比度、`Gamma` 值、直方图均衡化和直方图匹配的功能。在编程中，我使用了 Python 语言，利用 `opencv` 库用于图片数据的读入，并使用 `numpy` 库进行数组操作。

2.2 运行说明

只要在命令行中输入

```
1 ./point_processing.py <args>
```

即可运行或查看帮助，如图1所示：

```
n+e:~/github/dip2018/1/point_processing ./point_processing.py -h
usage: point_processing.py [-h] [-i INPUT] [-o OUTPUT] [-b GAIN] [-c CONTRAST]
                            [-g GAMMA] [-e] [-m STYLE]

Simple point processing demo

optional arguments:
  -h, --help    show this help message and exit
  -i INPUT      Input image filename
  -o OUTPUT     Output image filename
  -b GAIN       Brightness parameter
  -c CONTRAST   Contrast parameter
  -g GAMMA     Gamma parameter
  -e            Histogram equalization
  -m STYLE      Histogram matching, input style image's filename
```

图 1: 查看 point_processing 帮助

2.3 实验结果

2.3.1 明度变化

明度变化效果图如图2所示。左图增加了 80 明度，右图减小了 80 明度。

2.3.2 对比度变化

对比度变化如图3所示。对比度斜率小于 1 则表示减小对比度，大于 1 表示增加对比度。



图 2: 明度变换演示效果图



图 3: 对比度变换演示效果图

2.3.3 Gamma 变换

Gamma 变换如图4所示。当 $\gamma > 1$ 的时候图像变得更加明亮，反之图像变得更加暗淡，但是最亮处和最暗处极值处得到了保留。



(a) $\gamma = 0.5$

(b) $\gamma = 1.5$

图 4: Gamma 变换演示效果图

2.3.4 直方图均衡化

对于图片的每一个通道进行直方图均衡化，效果如图5所示。



图 5: 直方图均衡化效果图

2.3.5 直方图匹配

我在搜狗壁纸¹中抠了几张图片下来，演示效果如图6所示。

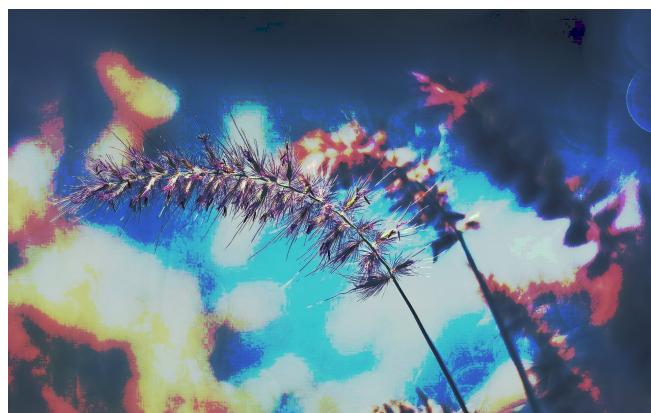
¹<http://bizhi.sogou.com/index.html>



(a) 原图



(b) 目标图



(c) 匹配结果

图 6: 直方图匹配效果图

3 Image Fusion

3.1 实现方式

在该部分中，考虑到实现效率，我采用 C++ 编写代码，其中图片的读入和输出采用 github 上的开源仓库"stb"²实现。

3.2 运行说明

首先编译 C++ 文件：

```
1 g++ image_fusion.cpp -o image_fusion -O2
```

在当前目录下生成可执行文件image_fusion。

使用命令

```
1 ./image_fusion <args>
```

即可运行或查看帮助，如图7所示：

```
n+e:~/github/dip2018/1/image_fusion ./image_fusion
Poisson image editing [un] -- powered by n+e [89-71]
Usage:
no argument      show this message and exit
-s SRC           src filename
-m MASK          mask filename [50-51]
-t TARGET         target filename
-o OUTPUT         output filename (only support .png) logs stat.py
-h HEIGHT        where to put src into target, specify HEIGHT
-w WIDTH         which to put src into target, specify WIDTH
-i ITERATION     how many ITERATION would you prefer, more is better
-b NUMBER        output less than NUMBER iterate result
-p NUMBER        output result every NUMBER iteration
-r ITER          resume iteration from ITERth png file
tensorboard --logdir ./ --port 8013
Example:
./image_fusion -s test1_src.jpg -t test1_target.jpg -m test1_mask.jpg -o test1_result.png -p 100 -b 10 -i 5000 -h 50 -w 100
```

图 7: 查看 image_fusion 帮助

3.3 算法细节

主体算法采用泊松图像融合算法 [4, 5]，梯度计算方式为

$$\nabla(x, y) = 4I(x, y) - I(x-1, y) - I(x, y-1) - I(x+1, y) - I(x, y+1)$$

将原图求完梯度之后，将该梯度匹配到目标图上的某一区域，本质上是一个解线性方程组的问题。形式化地，设有 N 个像素点需要匹配到目标图片中，则需要求解线性方程组

$$A\vec{x} = \vec{b}$$

其中 \vec{x} 代表融合后的图片中像素点的值，矩阵 A 的大小 $\sim N \times N$ ，列向量 \vec{x} 和 \vec{b} 的大小 $\sim N$ ，并且 A 的每一行至多只有 5 个非零元素，并且对角线上的元素均为 4。对

²<https://github.com/nothings/stb>

于第一组测试用例， $N = 2150$ ；对于第二组测试用例， $N = 16854$ 。因此 A 是一个巨大的稀疏矩阵。

考虑到矩阵求逆的复杂度为 $O(N^3)$ 太高，并且某些情况下连 A 都无法直接以矩阵形式存储，因此无法直接从公式

$$\vec{x} = A^{-1}\vec{b}$$

求得 \vec{x} 。此处采用 Jacobi Method 迭代求解出 \vec{x} 的值，详见 [1]。

3.4 实验结果

3.4.1 Test1

使用命令

```
1 time ./image_fusion -s test1_src.jpg -t test1_target.jpg -m  
test1_mask.jpg -o test1_result.png -i 5000 -h 50 -w 100
```

可得到如下结果

```
1 iter 5001 err 0.000000 0.000000 0.000000  
2 real 0m0.221s  
3 user 0m0.212s  
4 sys 0m0.008s
```

可以看到 5000 轮之后误差为 0，并且运行速度为 0.2s 左右，十分快。合成效果如图8所示。



图 8: 第一组数据合成效果

动画效果见 [pic/test1.gif](#)。

3.4.2 Test2

使用命令

```
1 time ./image_fusion -s test2_src.png -t test2_target.png -m  
test2_mask.png -o test2_result.png -i 25000 -h 150 -w 150
```

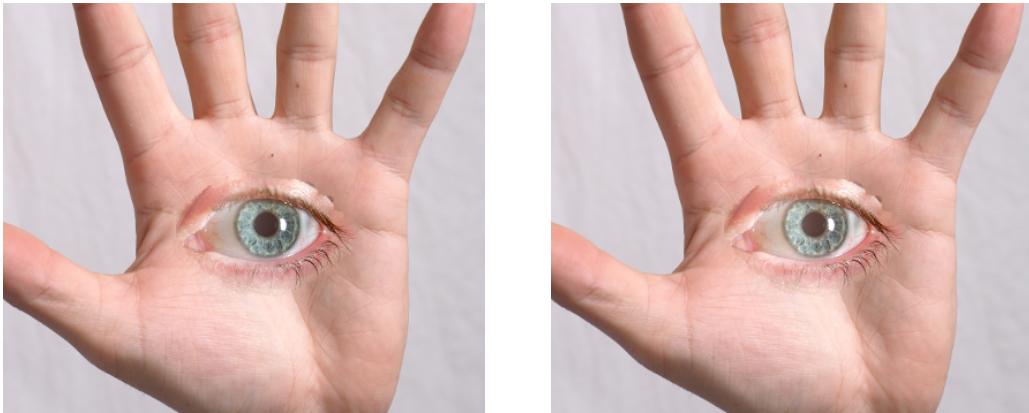
可得到如下结果

```
1 iter 25001 err 0.000000 0.000000 0.000000
2 real    0m5.012s
3 user    0m4.992s
4 sys     0m0.012s
```

可以看到 25000 轮之后误差为 0，并且运行速度为 5s 左右。合成效果如图9(a)所示。

事实上，图像在几千轮迭代的时候效果并不比 25000 轮差，使用 3000 轮迭代的测试情况如下，效果如图9(b)所示。由于我迭代的初始值选取的是目标图片的像素值，因此选用更少的迭代次数的话，合成的图片会更接近原图。

```
1 iter 3001 err 1416.263428 842.223511 960.723694
2 real    0m0.677s
3 user    0m0.660s
4 sys     0m0.004s
```



(a) 25000 轮迭代

(b) 3000 轮迭代

图 9: 第二组数据合成效果

动画效果见 [pic/test2.gif](#)。

4 Face Morphing

4.1 实现方式

在该部分中，考虑到实现效率，我采用 C++ 编写核心代码，Python 编写外部调用接口和 JSON 格式处理，主程序为 `merge_traditional.py`。另外，我还使用 Face++ 的开放接口 Merge Face API (V1)³实现了用深度方法将两张图片进行融合，主程序为 `merge_Face++.py`。

³<https://console.faceplusplus.com.cn/documents/20813963>

4.2 运行说明

使用命令

```
1 ./merge_traditional <args>
```

或者

```
1 ./merge_Face++.py <args>
```

即可运行或查看帮助，如图10所示：

```
n+e:~/github/dip2018/1/face_morphing ./merge_traditional.py -h
usage: merge_traditional.py [-h] [-t TEMPLATE] [-m MERGE] [-o OUTPUT]
                               [-r RATE]

Merge two faces with traditional implements

optional arguments:
  -h, --help            show this help message and exit
  -t TEMPLATE           Template image filename
  -m MERGE              Merge image filename
  -o OUTPUT             Output image filename
  -r RATE               merge rate (0~100), 0: template; 100: merge

real    0m0.677s
user    0m0.668s
sys     0m0.004s
n+e:~/github/dip2018/1/face_morphing/
```

图 10: 查看 merge_traditional 帮助

4.3 算法细节

此处采用 Mesh-based Morphing[2]，即先将图像划分成若干个三角形区域，然后将相应的三角形区域互相融合得到结果。具体而言，分为如下若干步骤：

1. 获取人脸关键点坐标数据（使用 Face++ Detect API V3⁴，返回 106 点人脸关键点 landmark）；
2. 使用 Delaunay 三角剖分 [3] 划分图像；
3. 对于每个三角形区域，计算仿射变换将其融合；

根据线性代数的相关知识，考虑到仿射变换的本质是矩阵乘法，并且只要确定 Template image 的三个点和 Target image 的三个点就能建立起对应关系，如下：

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} Temp_{1x} & Temp_{2x} & Temp_{3x} \\ Temp_{1y} & Temp_{2y} & Temp_{3y} \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} Tar_{1x} & Tar_{2x} & Tar_{3x} \\ Tar_{1y} & Tar_{2y} & Tar_{3y} \\ 1 & 1 & 1 \end{bmatrix}$$

其中

$$\mathcal{F} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix}$$

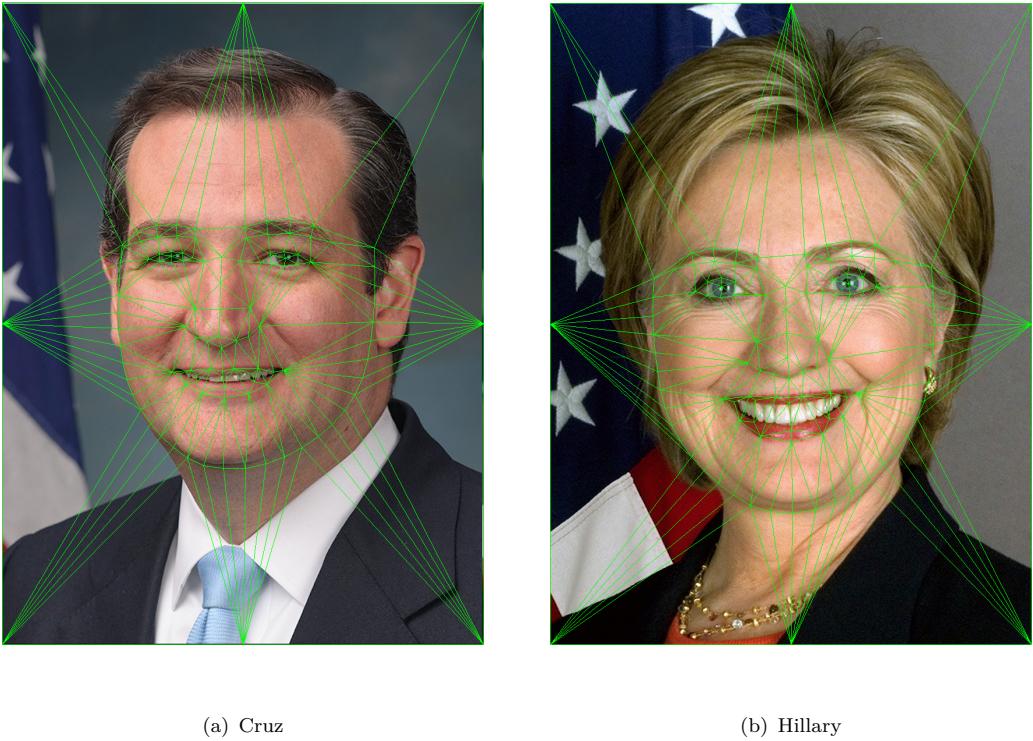
为仿射变换矩阵，直接用高斯消元反解方程或者矩阵求逆就能得到 a, b, c, d, e, f 的值。

⁴<https://console.faceplusplus.com.cn/documents/4888373>

4.4 实验结果

4.4.1 人脸关键点 +Delaunay 三角剖分

三角剖分之后的可视化效果如图11所示。我没有按照 Mallick[2] 那样把耳朵、头型、衣肩、领口这些点给标出来，并且标出这些点对人脸融合的意义也不是很大。



(a) Cruz

(b) Hillary

图 11: Cruz&Hillary 的人脸三角剖分之后可视化效果

4.4.2 Cruz&Hillary

将 Cruz 和 Hillary 的人脸融合之后的效果如图12所示。合并同等规模大小的图像在本机运行的时间约为 0.6s。

4.4.3 Hilary&Trump

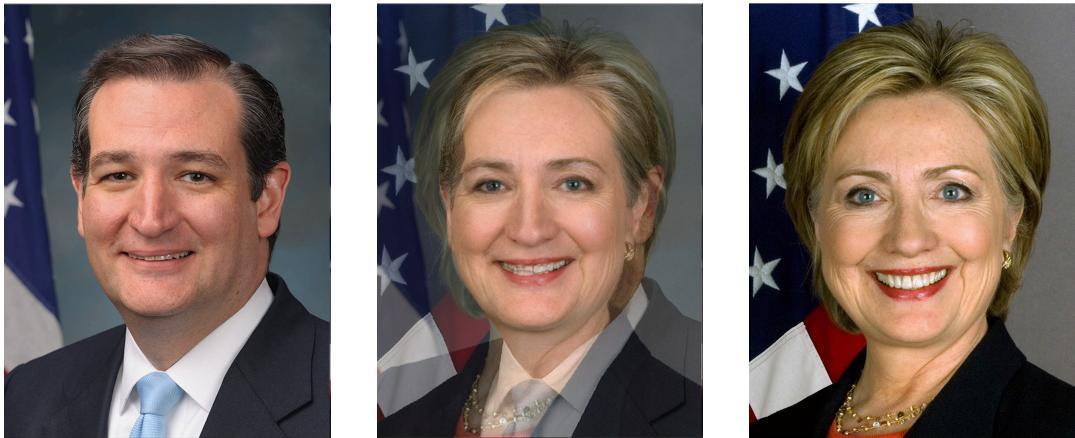
将 Hillary 和 Trump 的人脸融合之后的效果如图13所示。

4.4.4 Trump&Cruz

将 Trump 和 Cruz 的人脸融合之后的效果如图15所示。

4.4.5 Face++ API

使用 Face++ API 的结果如图??所示。生成的图片没有 ghosting，质量不错。运行时间约为两秒多。



(a) Cruz

(b) $0.5C+0.5H$

(c) Hillary

图 12: Cruz+Hillary



(a) Hillary

(b) $0.5H+0.5T$

(c) Trump

图 13: Hilary+Trump



(a) Trump

(b) 0.5T+0.5C

(c) Cruz

图 14: Hilary+Trump



(d) 0.5H+0.5C

(e) 0.5T+0.5H

(f) 0.5C+0.5T

图 15: Face++ API result

参考文献

- [1] Jacobi method. https://en.wikipedia.org/wiki/Jacobi_method. Accessed: 2018-04-12.
- [2] Satya Mallick. Face morph using opencv -- c++ / python. <https://www.learnopencv.com/face-morph-using-opencv-cpp-python/>. Accessed: 2018-04-17.
- [3] n+e. Delaunay 三角剖分. <http://trinkle.is-programmer.com/2015/7/1/delaunay-triangulation.100287.html>. Post datetime: 2015-03-01.
- [4] Patrick Pérez, Michel Gangnet, and Andrew Blake. Poisson image editing. *ACM Transactions on graphics (TOG)*, 22(3):313–318, 2003.
- [5] Christopher J. Tralie. Poisson image editing. <http://www.ctralie.com/Teaching/PoissonImageEditing/>. Accessed: 2018-04-12.