

# 数字图像处理第一次大作业实验报告

2016011359 计65 曾军

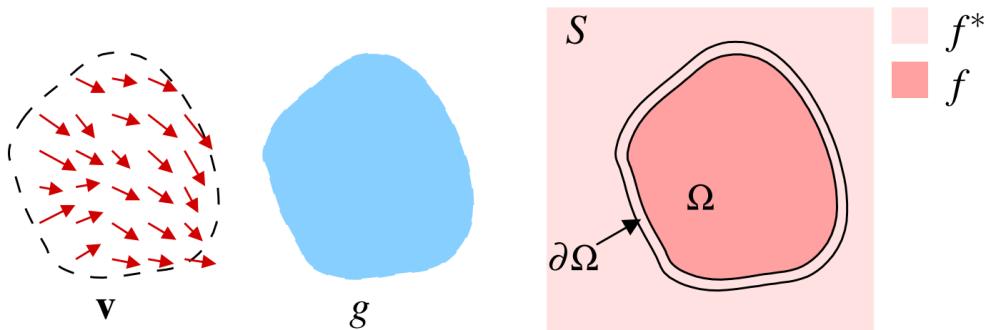
## 实验目的 #

本次实验主要包括两个部分：泊松图像编辑和人脸融合。这两个任务的目标类似，都是要实现两幅图像“自然”的融合：泊松图像编辑主要从图像的梯度和散度入手，抓住了心理学研究上的人类视觉对局部对比度变化的敏感度远高于对整体亮度变化的敏感度的结论，使两幅图像在融合边界处完全一致，在保持融合区域内部梯度的同时，整体的调整亮度/颜色。人脸融合主要是实现两幅图像的融合，但是我们要融合的图像往往是关键点不匹配的，对于关键点不匹配的图像，我们就要使用三角剖分和“triangle warping”来使得两幅图像的关键点匹配，匹配好了之后我们就可以将两幅图像按比例相加得到新的图像。

## 实验内容 #

### Poisson Image Editing

- 实现思路：



泊松图像编辑利用了人类视觉对局部对比度变化的敏感度远高于对整体亮度变化的敏感度的结论，因此我们在融合两幅图像时要保证融合区域内部梯度不变和融合区域的边界等于融合目标图像（即背景图像）的边界。在这个问题中， $S$ 是背景图像， $g$ 是待融合图像， $\Omega$ 是图像要融合的目标区域， $\partial\Omega$ 是要融合的目标区域的边界。这个问题要求： $\min_f \int_{\Omega} |\nabla f - v|^2$ , with  $f|_{\partial\Omega} = f^*|_{\partial\Omega}$ ，其中 $v$ 被称作引导向量场，在本问题中就是待融合图像 $g$ 的梯度场。

这个问题在离散域上就退化为求一个 $n * n$ 的像素矩阵的问题，我们知道矩阵的边界和内部的梯度，想要求出矩阵内部的像素值，这个问题可以转化为线性方程组 $Ax = b$ 的求解问题，已知梯度和边界条件求解像素矩阵，推导发现方程的数目和未知像素的数目契合，这表示这个方程有解，通过解这个方程，我们便可以得到融合区域内部的像素值，方程的具体形式如下（以 $3 * 3$ 矩阵为例）：

$$\begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ 1 & 1 & -4 & 1 & 1 \end{bmatrix} \bullet \begin{bmatrix} R_1 & G_1 & B_1 \\ R_2 & G_2 & B_2 \\ R_3 & G_3 & B_3 \\ R_4 & G_4 & B_4 \\ R_5 & G_5 & B_5 \\ R_6 & G_6 & B_6 \\ R_7 & G_7 & B_7 \\ R_8 & G_8 & B_8 \\ R_9 & G_9 & B_9 \end{bmatrix} = \begin{bmatrix} uR_1 & uG_1 & uB_1 \\ uR_2 & uG_2 & uB_2 \\ uR_3 & uG_3 & uB_3 \\ uR_4 & uG_4 & uB_4 \\ divR_5 & divG_5 & divB_5 \\ uR_6 & uG_6 & uB_6 \\ uR_7 & uG_7 & uB_7 \\ uR_8 & uG_8 & uB_8 \\ uR_9 & uG_9 & uB_9 \end{bmatrix}$$

- 具体实现：

- 移动mask和融合图像到指定的位置：

因为我们的mask和融合图像的目标位置并不是一一对应的，这在之后的内部梯度和边界计算中都是需要进行偏移的，要注意不能只将计算出来的结果进行偏移，这会使得边界条件不能满足导致融合失败，要同时偏移图像和mask：

```

def given_offset(src, mask, target, offset=(0, 0)): # If the mask is not
    aligned with the target image, we need to shift it... But how to shift it
    is important
    resize = target.shape
    new_src = np.zeros(resize)
    new_mask = np.zeros(resize)
    [offset_col, offset_r] = offset
    [col, r, depth] = src.shape
    [new_col, new_r, new_depth] = resize
    for c in range(0, col):
        for row in range(0, r):
            if 0 <= c+offset_col < new_col and 0 <= row+offset_r < new_r:
                new_mask[c+offset_col, row+offset_r] = mask[c, row]
                new_src[c+offset_col, row+offset_r] = src[c, row]
    return [new_src, new_mask]

```

- 计算融合图像的内部梯度，使用Laplace算子对融合图像做卷积，求得其梯度场，其中Laplace算子为：

$$L = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} :$$

```

def compute_laplacian(img):
    laplace_kernel = np.array([[0, -1, 0], [-1, 4, -1], [0, -1, 0]])#
    Laplace kernel
    return cv2.filter2D(img, -1, laplace_kernel)

```

- 找到矩阵A和向量b，求解方程Ax = b：

向量 $b$ 的值就是上述已知条件：内部梯度和边界条件，我们可以通过mask计算出内部点和边界点，然后使用计算出来的梯度值和已知的边界条件计算向量 $b$ :

```
#-----get b according to mask-----#
laplacian = compute_laplacian(src)
for loc in range(0, N):
    [col, r] = segment[loc]
    b[loc] = laplacian[col, r]
#-----considering border condition-----#
# Referring to: https://blog.csdn.net/hjimce/article/details/45716603
if border_detection(col, r, mask) == True:
    for i, j in adjacent_pixel(mask, col, r):
        if mask[i, j] == 0:
            b[loc] += target[i, j]
```

矩阵 $A$ 如上图实际上是一个稀疏矩阵，对应了Laplace卷积的过程：矩阵的对角元素为4，mask内的元素为-1，通过上述描述我们可以求解出对应的 $x$ ，也就得到了融合后的图像的像素值：

```
#-----get A according to mask-----#
for loc in range(0, N):
    A[loc, loc] = 4
    [col, r] = segment[loc]
    for point in adjacent_pixel(mask, col, r):
        if point in segment:
            A[loc, segment.index(point)] = -1
x = sparse_alg.spsolve(A, b)
print(x)
return x
```

- 最终效果：





- 说明：

参考博客 <https://blog.csdn.net/hjimce/article/details/45716603>，由于图像的离散性，我们不能直接计算融合图像内部的梯度，这样会使得边界处有一定的误差。根据博客中的方法，我们应该先计算融合图像和背景图像的梯度，然后再对梯度融合之后的图像求散度，这样会减小离散情况带来的误差。但是在实现时一直有散度计算边界上的问题，猜测是实验提供mask锯齿状过于明显导致边界计算出现问题，因为这个问题在自己找的矩形mask上并没有出现，这个问题也困扰了我很久，一直没有解决。

另外本实验参考了github和CSDN博客，所有参考点都在代码中声明。

## Face Morphing

- 实现思路：

人脸融合实际上是一个两幅图像相融合的问题，如果这两个图像足够“规矩”，原则上我们可以直接将对应点的像素值插值即可，但是实际上的图像关键点的位置都是不相同的，这也使得简单的图像融合的效果很差，甚至在很多情况下直接失效。对于直接融合失效的图像，我们考虑对两幅图像进行**德劳内三角剖分**：标出两幅图像中对应的关键点，两幅图像的关键点集进行三角剖分，对剖分后的三角形集合我们可以根据所学的知识对三角形进行一一对应的warping，warping之后两幅图像的关键点就大致一一对应了，这样我们就可以进行插值。

- 算法知识：

- 德劳内三角剖分：

**德劳内三角剖分**其实并不是一种算法，它只是给出了一个“好的”三角网格的定义，它的优秀特性是**空圆特性和最大化最小角特性**，这两个特性避免了狭长三角形的产生，也使得德劳内三角剖分应用广泛。**空圆特性**其实就是对于两个共边的三角形，任意一个三角形的外接圆中都不能包含有另一个三角形的顶点，这种形式的剖分产生的最小角最大。

关于德劳内三角剖分的求解算法也一直是学术界研究的重点，求解的算法也大致分为三种：暴力枚举，增量法以及分治法，其中暴力枚举的时间复杂度最高为 $O(n^2)$ ，后两种算法的复杂度都为 $O(n \log n)$ 。但是就目前工业上的使用而言，分治法的表现是最好的。本实验中我使用了增量法对标出的特征点集进行三角剖分，使用的增量算法是经典的**Bowyer逐点插入法**，这个算法是Paul Bourke在他的[论文](#)中提出的，也得到了工业界的广泛认可和使用，算法伪代码如下：

```

subroutine triangulate
input : vertex list
output : triangle list
    initialize the triangle list
    determine the supertriangle
    add supertriangle vertices to the end of the vertex list
    add the supertriangle to the triangle list
    for each sample point in the vertex list
        initialize the edge buffer
        for each triangle currently in the triangle list
            calculate the triangle circumcircle center and radius
            if the point lies in the triangle circumcircle then
                add the three triangle edges to the edge buffer
                remove the triangle from the triangle list
            endif
        endfor
        delete all doubly specified edges from the edge buffer
        this leaves the edges of the enclosing polygon only
        add to the triangle list all triangles formed between the point
        and the edges of the enclosing polygon
    endfor
    remove any triangles from the triangle list that use the supertriangle vertices
    remove the supertriangle vertices from the vertex list
end

```

算法具体要维护的是一个待筛选的剖分三角形集合和每个特征点对应的边集合，通过检查每个顶点的和三角形几何中的三角形的空圆特性来判断当前的部分是否为德劳内部分，若不是，则将这个点和三角形的三条边各形成一个新的三角形，达到更细致的剖分，通过分析不难知道增量法的复杂度为 $O(n \log n)$ 。

- 获得一个三角形内的所有点

在获得剖分之后，我们需要对三角形内的每一个点做仿射变换，因此我们需要知道三角形内所有点的信息，然而这在基于只知道三个点的基础上是十分复杂的。借鉴[博客](#)，我们可以将整个三角形转换到三角形的质心坐标系中，我们知道若三角形顶点坐标为 $A, B, C$ ，则在三角形内某一点 $P = a * A + b * B + c * C$ , 其中 $a + b + c = 1$ ，我们可以遍历 $a, b, c$ 得到三角形内所有的点：

```

def get_all_pts(triangle):
    [pt1, pt2, pt3] = triangle.pts
    pts_list = []
    radius = max([distance(pt1, pt2), distance(pt1, pt3), distance(pt2, pt3)])
    a = 1.0 / radius
    for p in np.arange(0., 1., a):
        for q in np.arange(0., 1-p, a):
            p_x = int(p*pt1[0] + q*pt2[0] + (1 - p - q)*pt3[0])
            p_y = int(p*pt1[1] + q*pt2[1] + (1 - p - q)*pt3[1])
            pts_list.append((p_x, p_y))
    return list(set(pts_list))

```

- 具体实现：

- 辅助类：

[Bowyer逐点插入法](#)中需要维护三角形集合和边集合，因此我定义了`Graph`, `Triangle`, `Edge`三个类来分别表示整个图，特定的三角形和特定的边：

图：`Graph`类起到了整合数据的作用，并且实现了接口 `del_repeated_element(edge_list)` 来满足算法中`delete all doubly specified edges from the edge buffer`的要求：

```

class Graph:
    def __init__(self, vertex_list=[], tri_list=[]):

```

```

    self.vertex_list = vertex_list
    self.tri_list = tri_list

    def add_element(self, ops, element):
        if ops == 'vertex':
            self.vertex_list.append(element)
        elif ops == 'triangle':
            self.tri_list.append(element)

    def del_repeated_element(edge_list):
        N = len(edge_list)
        filter = np.zeros(N)
        deduplicated_list = []
        for i in range(0, N):
            if filter[i] == 0:
                for j in range(i + 1, N):
                    if edge_list[i].equals(edge_list[j]):
                        filter[i] = 1
                        filter[j] = 1
        for i in range(0, N):
            if filter[i] == 0:
                deduplicated_list.append(edge_list[i])
        return deduplicated_list

```

三角形：*Triangle*类维护了三角形的三个顶点的列表，创建时传入三个顶点即可，并且在创建时就会计算这个三角形的外接圆圆形和半径，求解的方法为简单的数学方程求解。若输入的三点共线表明三角形不存在，则停止继续创建：

```

class Triangle:
    def __init__(self, pts=[]):
        self pts = pts
        [pt1, pt2, pt3] = pts
        [x1, y1] = pt1
        [x2, y2] = pt2
        [x3, y3] = pt3
        a = x1 - x2
        b = y1 - y2
        c = x1 - x3
        d = y1 - y3
        e = (pow(x1, 2) - pow(x2, 2) - pow(y2, 2) + pow(y1, 2)) / 2
        f = (pow(x1, 2) - pow(x3, 2) - pow(y3, 2) + pow(y1, 2)) / 2
        self.x0 = (b*f - d*e) / (b*c - a*d)
        self.y0 = (c*e - a*f) / (b*c - a*d)
        self.r = self.dist((x1, y1), (self.x0, self.y0))

    def in_circle(self, pts):
        distance = self.dist((self.x0, self.y0), pts)
        if distance < self.r:
            return True
        return False

    def dist(self, pt1, pt2):

```

```

        [x1, y1] = pt1
        [x2, y2] = pt2
        return math.sqrt(pow(abs(x1-x2), 2) + pow(abs(y1-y2), 2))

    def overlap(self, pts):
        for inner_pt in self pts:
            for outer_pt in pts:
                if inner_pt == outer_pt or inner_pt == (outer_pt[1],
outer_pt[0]):
                    return True
        return False

```

边：起初考虑用四元组来表示一条边，但是这样对节点的先后顺序也敏感，这会给删除边造成很大的影响，因此定义了Edge类来解决这个问题：

```

class Edge:
    def __init__(self, pt1, pt2):
        self.pt1 = pt1
        self.pt2 = pt2
        self.pts = [pt1, pt2]

    def equals(self, edge):
        if edge.pt1 == self.pt1 and edge.pt2 == self.pt2:
            return True
        elif edge.pt2 == self.pt1 and edge.pt1 == self.pt2:
            return True
        return False

```

- 三角剖分算法：

在定义了辅助类和实现了相应的方法之后，实现伪代码变得比较简单，主要就是循环便利点集合和三角形集合，检查空圆性并作出调整即可：

```

def subroutine_triangulate_strategy(pts, height, width):
    hypo_pts = [(-height, -width), (-height, 3*width), (3*height, -width)]
    hypo_triangle = graph.Triangle(hypo_pts)
    tri_list = []
    vertex_list = pts
    img_graph = graph.Graph(vertex_list, tri_list)
    # use area-adding to solve the delaunay triangulation
    tri_list.append(hypo_triangle)
    for pt in vertex_list:
        edge_list = []
        N = len(tri_list)
        for loc in range(N-1, -1, -1):
            if tri_list[loc].in_circle(pt) == True:
                [pt1, pt2, pt3] = tri_list[loc].pts
                edge_list.append(graph.Edge(pt1, pt2))
                edge_list.append(graph.Edge(pt2, pt3))
                edge_list.append(graph.Edge(pt1, pt3))
                del(tri_list[loc])

```

```
edge_list = graph.Graph.del_repeated_element(edge_list)
```

```

for edge in edge_list:
    tri_pts = [pt, edge.pt1, edge.pt2]
    a = pt[0] - edge.pt1[0]
    b = pt[1] - edge.pt1[1]
    c = pt[0] - edge.pt2[0]
    d = pt[1] - edge.pt2[1]
    if b*c != a*d:
        tri_list.append(graph.Triangle(tri_pts))
N = len(tri_list)
for loc in range(N-1, -1, -1):
    if tri_list[loc].overlap(hypo_pts) == True:
        del(tri_list[loc])
return tri_list

```

- 求解流程：

首先对两幅图像的特征点做一个插值得到一个中间态图像，对中间态图像做三角剖分得到一个三角形集合，我们知道中间态图像的三角网格可以通过仿射变换得到两幅图像的中间

网格，放射变换的矩阵可以表示为：
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$
，我们知道三个点的信息，对应的我们需要求解六个未知数，因此方程是有解的，求解的具体过程如下：

```

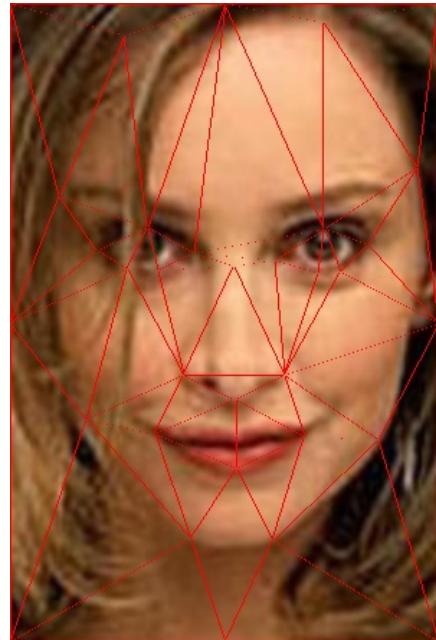
def solve_transform(src, target):
    [(x1, y1), (x2, y2), (x3, y3)] = src
    [(_x1, _y1), (_x2, _y2), (_x3, _y3)] = target
    b = np.array([_x1, _y1, _x2, _y2, _x3, _y3])
    A = np.array([[x1, y1, 1, 0, 0, 0],
                  [0, 0, 0, x1, y1, 1],
                  [x2, y2, 1, 0, 0, 0],
                  [0, 0, 0, x2, y2, 1],
                  [x3, y3, 1, 0, 0, 0],
                  [0, 0, 0, x3, y3, 1]])
    x = solve(A, b)
    return np.array([[x[0], x[1], x[2]], [x[3], x[4], x[5]], [0, 0, 1]])

```

求解到了仿射变换之后我们可以对当前三角网格中的所有点做仿射变换得到他与两幅图像之间的关系，相应的像素值也可以求得，我们对两个像素值做对应比例的插值即可，这样处理完中间态图像的三角网格之后我们就可以得到一个新的图像，人脸融合也就完成了。

- 最终效果：

三角剖分效果：（由于使用的连线算法比较初级，因此效果可能不好）：



人脸融合效果：



发现第二幅的效果稍微有一点差，原因是第二幅图使用了开源的人脸关键点识别库，标出的关键点中只有面部的关键点，而脸型和衣着的关键点都没有标注，若是对其进行手动标注，效果应该会和第一幅差不多。

- **说明：**

分析了德劳内三角剖分的三类算法，发现分治法占优的主要层面是在数据结构上，分治法一般使用DAG或双向边等数据结构，使得边和三角网格查找的复杂度更低，若增量法引入这些数据结构，表现应该也是不俗的。

## 实验总结

本次实验实现了泊松图像编辑和人脸融合算法，难度主要体现在数学知识和算法实现层面，比如我就因为对离散图像的梯度和散度的意义了解的不是很清楚，遇到了很多问题也浪费了很多时间，通过这个实验我收获了很多，感觉自己对数字图像处理的问题也更熟悉了。