# CVPR 第五次作业


学　院　名　称：数据科学与计算机学院

学　生　姓　名：谢元新

学　　号：14331311

专 业 （ 班 级 ）：14 软件工程三（5）班

1. 算法：先输入四个对应的点的坐标，然后根据目标图的四个原点的坐标，总共可以列出八个方程，将这八个方程改写成如下的模式：

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -ux & -uy \\ 0 & 0 & 0 & x & y & 1 & -vx & -vy \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ m \\ l \end{bmatrix} \tag{2}$$

则根据两个矩形的八个顶点可以得到下面这个公式：

$$\begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \\ u_4 \\ v_4 \end{bmatrix} = \begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -u_1x_1 & -u_1y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -v_1x_1 & -v_1y_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -u_2x_2 & -u_2y_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -v_2x_2 & -v_2y_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -u_3x_3 & -u_3y_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -v_3x_3 & -v_3y_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -u_4x_4 & -u_4y_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -v_4x_4 & -v_4y_4 \end{bmatrix} \times \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ m \\ l \end{bmatrix} \tag{3}$$

只要计算中间 8*8 矩阵的逆，乘上变换后的 UV 矩阵，就可以求出所有的待定系数。（点的坐标可以用上次作业的 hough 算法得到）（注：这个矩阵是变换后的点到变换前的点的矩阵）

得到变换矩阵后，应用双线性插值的办法得到像素值：

```
cimg_forXY(outputimg, x, y) {
    double px = xs[0] * x + xs[1] * y + xs[2];
    double py = xs[3] * x + xs[4] * y + xs[5];
    double p = xs[6] * x + xs[7] * y + 1;

    double u = px / p;
    double v = py / p;

    int uu, vv;
    if (floor(u) < 0) uu = 0;
    else uu = floor(u);

    if (u + 1 > 299) uu = 298;

    if (floor(v) < 0) vv = 0;
    else vv = floor(v);

    if (v + 1 > 419) vv = 418;

    double t = u-uu, s = v-vv;
    outputimg(x,y,0)=(1-t)*(1-s)*paint(uu,vv,0)+(1-t)*s*paint(uu,vv+1,0)+t*(1-s)*paint(uu+1,v,0)+t*s*paint(uu+1,vv+1,0);
    outputimg(x,y,1)=(1-t)*(1-s)*paint(uu,vv,1)+(1-t)*s*paint(uu,vv+1,1)+t*(1-s)*paint(uu+1,v,1)+t*s*paint(uu+1,vv+1,1);
    outputimg(x,y,2)=(1-t)*(1-s)*paint(uu,vv,2)+(1-t)*s*paint(uu,vv+1,2)+t*(1-s)*paint(uu+1,v,2)+t*s*paint(uu+1,vv+1,2);
```

思考：如何加快运行速度

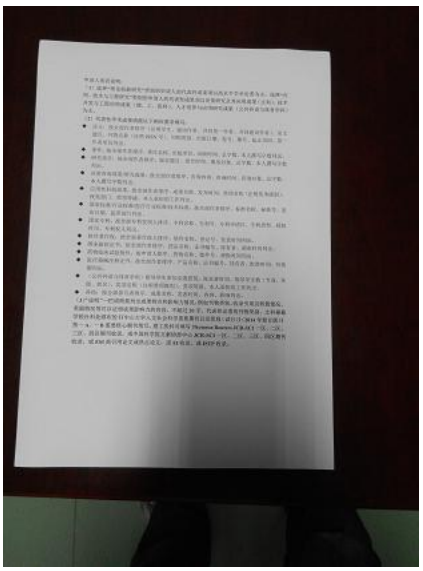
（1） 程序在运行的过程中会大量调用 operator(const x, const y, const z)函数，这会导致极大的开销，如果改用指针操作的话能够加快速度，对于百万像素级别的图片，能加快几十乃至上百倍，效果非常明显

（2） 在计算原图对应的坐标的时候，实际上应用了两重循环，假设 X 是外部循环，第二层循环里里对 X 的变量每次循环都要重复一次，所以可以把 X 坐标的计算提到循环

外。

部分实验结果：

（1） 原图：

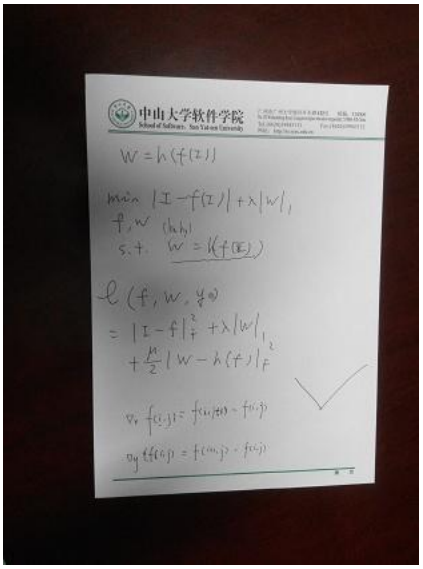


结果图：

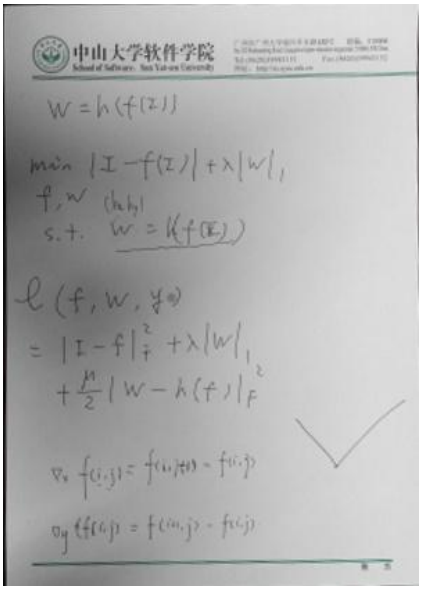
（2） 原图：

结果图：

（3） 原图：


中山大学软件学院
School of Software, Sun Yat-sen University
W = h(f(I))
min |I-f(I)| + λ|W|
f,W
s.t. W = h(f(I))
l(f, W, y)
= |I-f|_F^2 + λ|W|_1
+ μ/2 |W-h(f)|_F

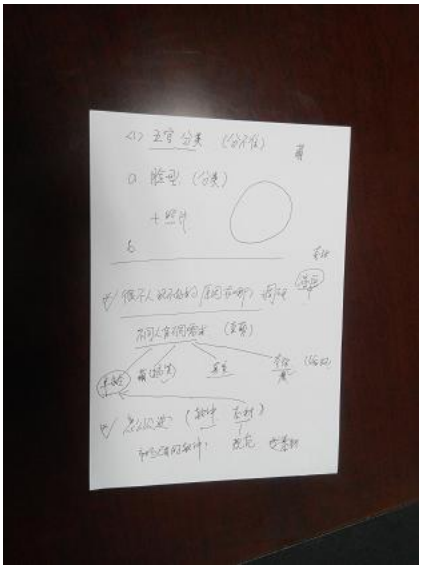
结果图：


中山大学软件学院
School of Software, Sun Yat-sen University
W = h(f(I))
min |I − f(I)| + λ|W|
f, W
s.t. W = h(f(I))
ℓ(f, W, y)
= |I − f|_F^2 + λ|W|_1
+ μ/2 |W − h(f)|_F^2

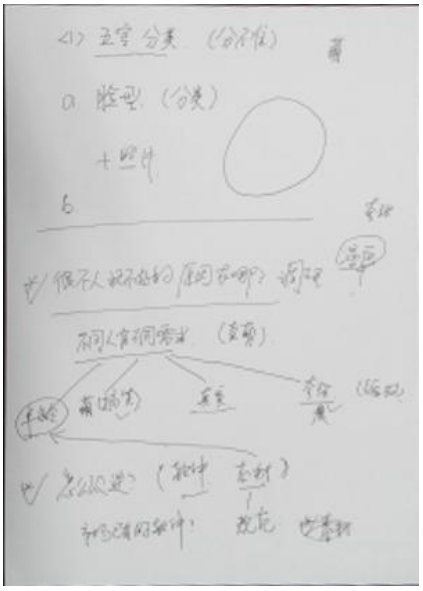
（4） 原图：

结果图：

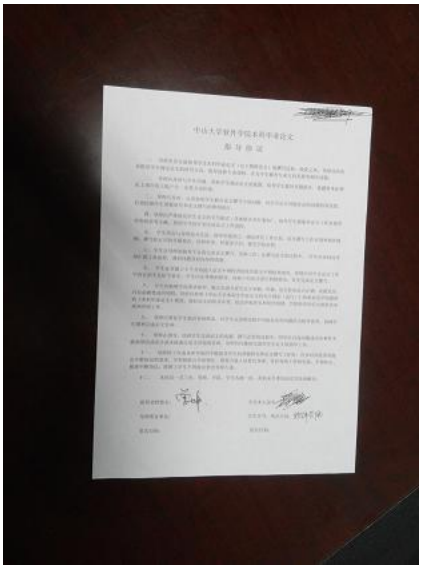
（5） 原图：



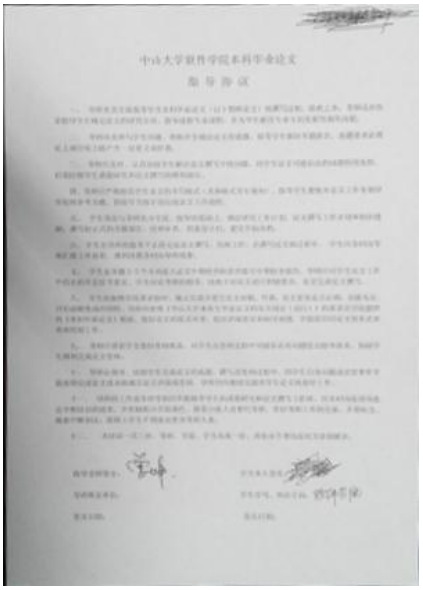
结果图：



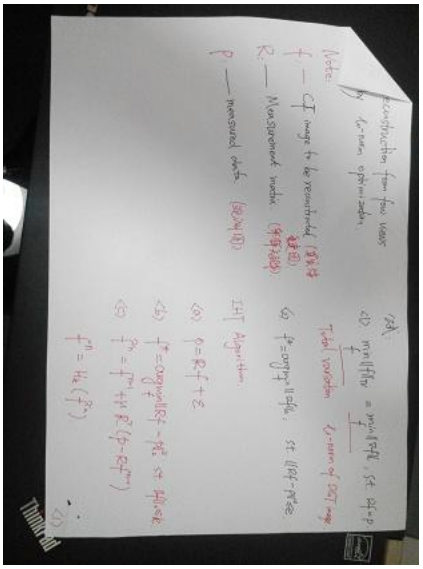
（6） 原图：



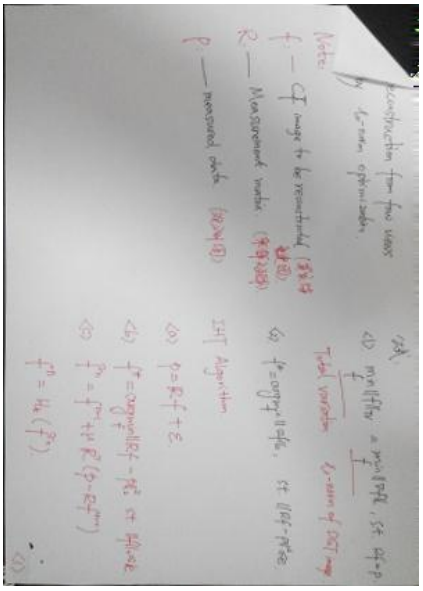
结果图：



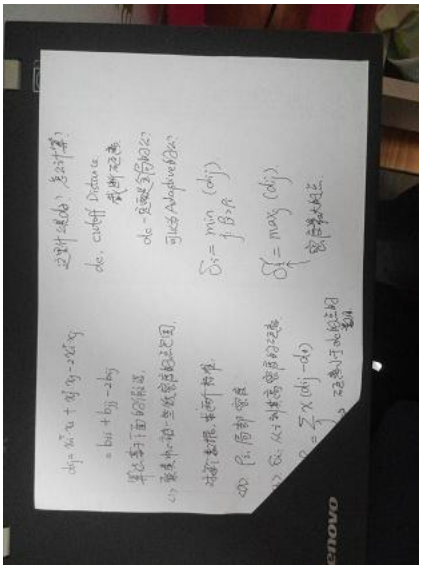

（7） 原图



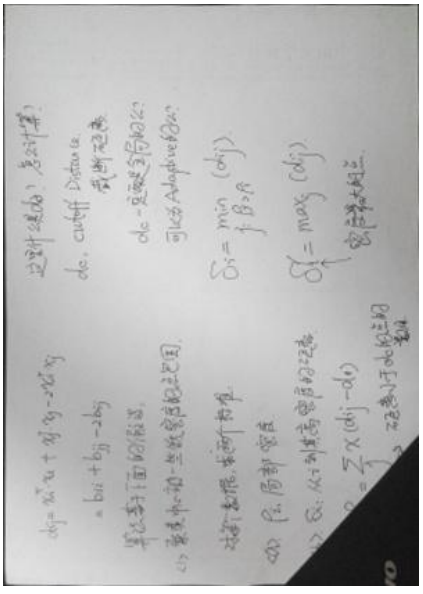
结果图：



（8） 原图：



结果图：



附加题：

算法步骤：

1. 由于时间有限，脸部的特征点的标记是我直接手动标记的，并存在一个文件（coordinate.txt）里，所以第一步是读取两张图片的特征点。

```
ifstream fin("coordinate.txt");
while (getline(fin, line))
{
    vector<string> result;
    split(line, " " , result);
    x1 = atoi(result[0].c_str());
    y1 = atoi(result[1].c_str());
    x2 = atoi(result[2].c_str());
    y2 = atoi(result[3].c_str());

    PT p1, p2;
    p1.x = x1;
    p1.y = y1;
    p2.x = x2;
    p2.y = y2;
    points1.push_back(p1);
    points2.push_back(p2);
}

cout << points1.size() << endl;
cout << "reading finish\n";
```

2. 由于老师说可以直接调用网上的三角剖分函数，但是我没有找到很优秀的 C++写的 delaunay 代码，犹豫了很久，选择了 matlab 自带的库函数，我另外用两幅图的特征点跑了一下，将结果存在 delaunay_result.txt 里，并用 BuildDelaunayEx 直接读取（文件里的整数代表特征点的编号，如 1，2，3 代表一号点，二号点和三号点组成一个三角形）

```
vector<TRIANGLE> t1;
CMyDelaunay d;
d.BuildDelaunayEx(points1, t1);

cout << t1.size() << endl;

cout << "finish delaunay\n";
```

3. 接下来对于中间的每一帧图片，首先计算出中间帧的特征点的位置（采用加权平均）：

```
vector<PT> m;

for (int k = 0; k < t1.size(); ++k)
{
    PT pm;
    pm.x = i / 12.0 * (points2[t1[k].firstPid-1].x - points1[t1[k].firstPid-1].x) + points1[t1[k].firstPid-1].x;
    pm.y = i / 12.0 * (points2[t1[k].firstPid-1].y - points1[t1[k].firstPid-1].y) + points1[t1[k].firstPid-1].y;
    m.push_back(pm);

    pm.x = i / 12.0 * (points2[t1[k].secendPid-1].x - points1[t1[k].secendPid-1].x) + points1[t1[k].secendPid-1].x;
    pm.y = i / 12.0 * (points2[t1[k].secendPid-1].y - points1[t1[k].secendPid-1].y) + points1[t1[k].secendPid-1].y;
    m.push_back(pm);

    pm.x = i / 12.0 * (points2[t1[k].thirdPid-1].x - points1[t1[k].thirdPid-1].x) + points1[t1[k].thirdPid-1].x;
    pm.y = i / 12.0 * (points2[t1[k].thirdPid-1].y - points1[t1[k].thirdPid-1].y) + points1[t1[k].thirdPid-1].y;
    m.push_back(pm);
}
```

4. 对于中间帧的每一个像素点，对于每一个剖分出来的三角形进行判断，看是否在这个三角形中：若在，则运用 warping 的方法进行变换。像素值也采用加权平均。（给定三角形的三个顶点与另外一点判断该点是否在三角形中的算法采用了向量叉乘的方法，运算效率特别快，比用等面积法、角度和的方法**不知道高到哪里去了**）

```
for (int u = 0; u < 490; ++u)
{
    for (int v = 0; v < 700; ++v)
    {
        PT p;
        p.x = u,p.y=v;
        for (int k = 0; k < t1.size(); ++k)
        {
            if (inTriangle(m[k*3], m[k*3+1], m[k*3+2], p))
```

```
PT p1, p2;
p1.x = u * tran1[0][0] + v * tran1[0][1] + tran1[0][2];
p1.y = u * tran1[1][0] + v * tran1[1][1] + tran1[1][2];
p2.x = u * tran2[0][0] + v * tran2[0][1] + tran2[0][2];
p2.y = u * tran2[1][0] + v * tran2[1][1] + tran2[1][2];

int px1 = normalizeX(p1.x), py1 = normalizeY(p1.y);
int px2 = normalizeX(p2.x), py2 = normalizeY(p2.y);

medG(u, v, 0) = (1 - i / 12.0) * origin(px1, py1, 0) + i / 12.0 * target(px2, py2, 0);
medG(u, v, 1) = (1 - i / 12.0) * origin(px1, py1, 1) + i / 12.0 * target(px2, py2, 1);
medG(u, v, 2) = (1 - i / 12.0) * origin(px1, py1, 2) + i / 12.0 * target(px2, py2, 2);
break;
```

这里有一个我觉得很坑的点，一开始我只是对脸部标记了特征点，背景的变换效果不是很好，后来我把照片的四个角点也标记为特征点，这很有效。

思考：这份代码还有很多可以提高的地方，但时间有限我没有时间去完成了，只好在这里提出思路，主要分为效果和运行速度：
效果：我人工标记的特征点效果还不错，但是不如用人脸识别的算法，而且我在这里只对脸部进行了标记，忘了对人物的衣着进行特征标记，导致图像下部的变换效果不如脸部。

运行速度：代码中每一个像素点都要对所在的三角形求一次变换的矩阵，然而其实一个三角形里的点是公用一个矩阵的，所以可以考虑将所有的点先按三角形分类，再对每个三角形进行一次计算就好。