中山大學
SUN YAT-SEN UNIVERSITY

《计算机视觉与模式识别》
Bonus 1
Testing Report

学院名称	:	数据科学与计算机学院
学生姓名	:	吴若宇
学号	:	14301030
专业（班级）	:	14级计算机科学与技术
时间	:	2017 年 3 月 27 日

Image Morphing

——人脸变换

1、 测试环境

Ubuntu 15.10

2、 测试数据


3、 测试过程

写了一个runScript.py的python脚本，没有其他的附加参数。跑这个脚本的时候，它会运行Bonus1.cpp程序，把所有生成的中间图片放在transformProcess这个文件夹里面，并在最后把所有的这些图片制作成一个动态的gif（依赖imageio package)。脚本的代码如下：

```python
import os
import subprocess
import imageio

def main():
    sourceImagePath = "../img/1.jpg"
    destImagePath = "../img/2.jpg"
    outImagePath = "../transformProcess/"
    args = ['./Bouns1']
    args.append(sourceImagePath)
    args.append(destImagePath)
    # interval #
    args.append("40")
    subprocess.call(args)

    imageName = []
    for _imageName in os.listdir(outImagePath):
        imageName.append(_imageName)
    imageName.sort(key=lambda x: int(x.strip(".jpg")))

    images = []
    for _imageName in imageName:
        images.append(imageio.imread(outImagePath + _imageName))
    imageio.mimsave("../animated.gif", images, fps=5)


if __name__ == "__main__":
    main()
```

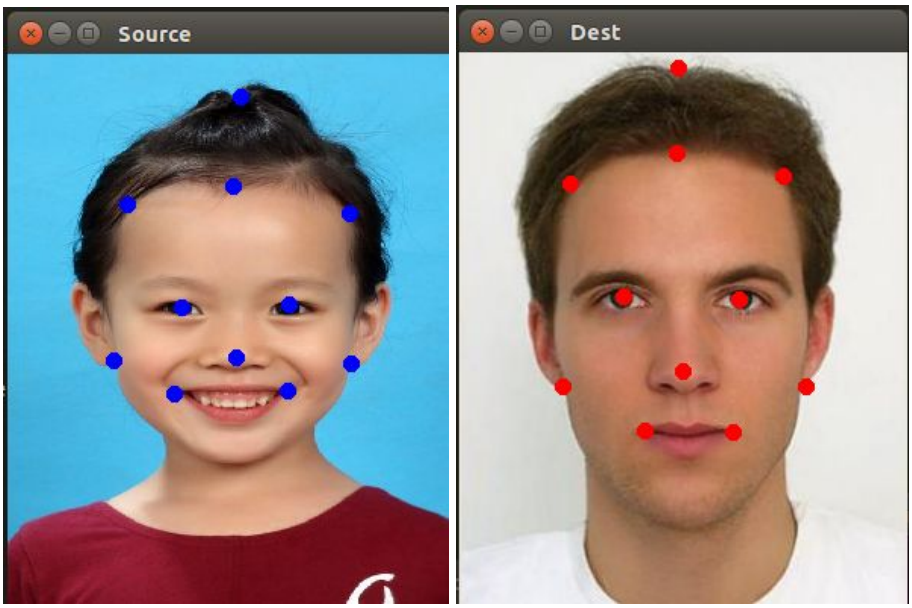
4、 实验原理、代码

（1）控制点获取

这里就直接调用了CImg的CImgDisplay类，分别对两张图打控制点，需要注意的是，两张图的控制点不仅要在特征上需要一一对应，还需要在**顺序上一一对应**，因为程序不大可能自己知道哪些点是相互对应的。图片的左上、左下、右下和右上四个角点是默认的四个角点，这是在imageMorphing的构造函数中设置的。代码如下：

```
void getControlPoint() {
    CImg<double> sourceCopy = source;
    CImg<double> destCopy = dest;

    CImgDisplay sDisp(sourceCopy, "Source");
    CImgDisplay dDisp(destCopy, "Dest");

    while (!sDisp.is_closed()) {
        sDisp.wait();
        if ((sDisp.button() & 1) && sDisp.mouse_x() != -1 && sDisp.mouse_y() != -1) {
            sourceControlPoint.push_back(Point(sDisp.mouse_x(), sDisp.mouse_y(), sourceControlPoint.size()));
            sourceCopy.draw_circle(sourceControlPoint.back().theta, sourceControlPoint.back().rho, 5, sourcePointColor);
            sourceCopy.display(sDisp);
        }
    }

    while (!dDisp.is_closed()) {
        dDisp.wait();
        if ((dDisp.button() & 1) && dDisp.mouse_x() != -1 && dDisp.mouse_y() != -1) {
            destControlPoint.push_back(Point(dDisp.mouse_x(), dDisp.mouse_y(), destControlPoint.size()));
            destCopy.draw_circle(destControlPoint.back().theta, destControlPoint.back().rho, 5, destPointColor);
            destCopy.display(dDisp);
        }
    }

    if (sourceControlPoint.size() != destControlPoint.size()) {
        cerr << "the number of control points are not equal" << endl;
        exit(-1);
    }
}
```

打点时候的效果图如下（*该图只是示例，要得到好效果需要打更多控制点*）：



（2）Delaunay三角形剖分

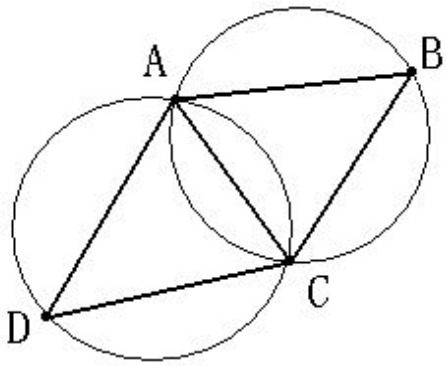
在打好控制点之后要做的就是三角形剖分，我使用的是Delaunay三角形剖分算法。在这里就解释一下这个算法。

【定义】Delaunay边：假设E中的一条边e（两个端点为a,b），e若满足下列条件，则称之为Delaunay边：存在一个圆经过a,b两点，圆内(注意是圆内，圆上最多三点共圆)不含点集V中任何其他的点，这一特性又称空圆特性。

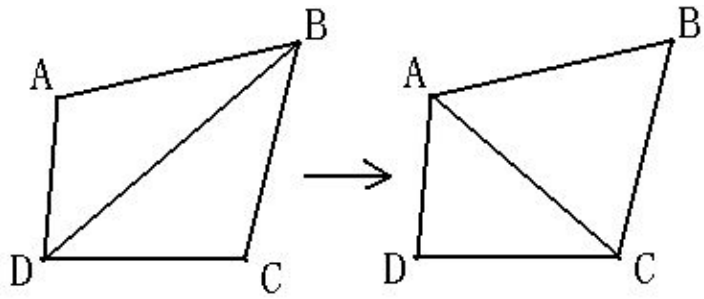
【定义】Delaunay三角剖分：如果点集V的一个三角剖分T只包含Delaunay边，那么该三角剖分称为Delaunay三角剖分。

要满足Delaunay三角剖分的定义，必须符合两个重要的准则：

1、空圆特性：Delaunay三角网是唯一的（任意四点不能共圆），在Delaunay三角形网中任一三角形的外接圆范围内不会有其它点存在。如下图所示：



2、最大化最小角特性：在散点集可能形成的三角剖分中，Delaunay三角剖分所形成的三角形的最小角最大。从这个意义上讲，Delaunay三角网是"最接近于规则化的"的三角网。具体的说是指在两个相邻的三角形构成凸四边形的对角线，在相互交换后，六个内角的最小角不再增大。如下图所示：



算法解释就到这里。我没有用一般用的Lawson算法或者Bowyer-Watson算法，而是采用了……暴力算法，虽然算法复杂度是有点高，但是做实验的时候感觉还是不错的，因为在控制点不是很多的时候也能得到相对不错的效果。如下图的代码，对于所有点的（三角形）组合我都遍历了一遍，如果这个三角形的外接圆内没有其他控制点，就把这个三角形的三条边都放到数组里面。这个方法虽然粗糙有很多缺陷，但是总的来说还是够用的。

```
vector<Triangle> triangles;

// for all the combination of points
for (int i = 0; i < points.size() - 2; i ++) {
    for (int j = i + 1; j < points.size() - 1; j ++) {
        for (int k = j + 1; k < points.size(); k ++) {
            // if a, b, c are on the same line
            if (isPointsOnOneLine(points[i], points[j], points[k]))
                continue;

            // no other points are in circumscribed circle
            bool delaunayFlag = true;
            for (int h = 0; h < points.size(); h ++) {
                if (h == i || h == j || h == k)
                    continue;
                if (isInCircumscribedCircle(points[h], points[i], points[j], points[k])) {
                    delaunayFlag = false;
                    break;
                }
            }
            if (delaunayFlag) {
                triangles.push_back(Triangle(points[i], points[j], points[k]));
            }
        }
    }
}
return triangles;
```

其中*isPointsOnOneLine*是用来检验三个点是不是在一条直线的，也就是简单的比较斜率，这里代码就不做展示了

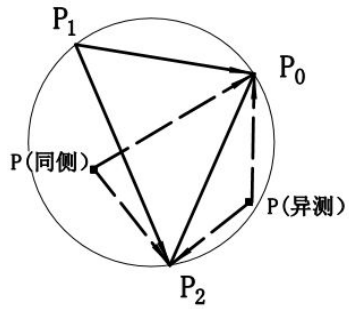
*isInCircumscribedCircle*是用来判断一个点是否在三角形的外接圆内，算法如下：

给定三点$P_0P_1P_2$，假设我们以P_0P_2作为基准边。通过向量积判断P与P_1是否在同侧。

若P与P_1在同侧，那么若$P_0P_1P_2 < P_0PP_2$，则P在圆内，否则在圆外。

若不同侧，那么若$P_0P_1P_2 + P_0PP_2 \geq \pi$，$P$在圆内，否则在圆外。

其中，判断是否同侧就看$Ax_1 + By_1 + c$是否与$Ax_2 + By_2 + c$同号，$Ax + By + c = 0$是基准线的直线方程。角度的话可以利用向量的余弦公式得到。



根据这个算法，我的代码如下：

```
// examine if point m in the circumscribed circle formed by (a, b, c)
// the explanation of algorithm is in the testing report
static bool isInCircumscribedCircle(const Point& m, const Point& a, const Point& b, const Point& c) {
    // if m is on line ab
    if (isPointsOnOneLine(a, b, m))
        return true;

    // calculate the angle
    double acb = angle(a, c, b);
    double amb = angle(a, m, b);

    if (isOnSameSide(a, b, c, m)) {
        if (acb <= amb)
            return true;
        return false;
    }
    else {
        if (acb + amb >= pi())
            return true;
        return false;
    }
}
```

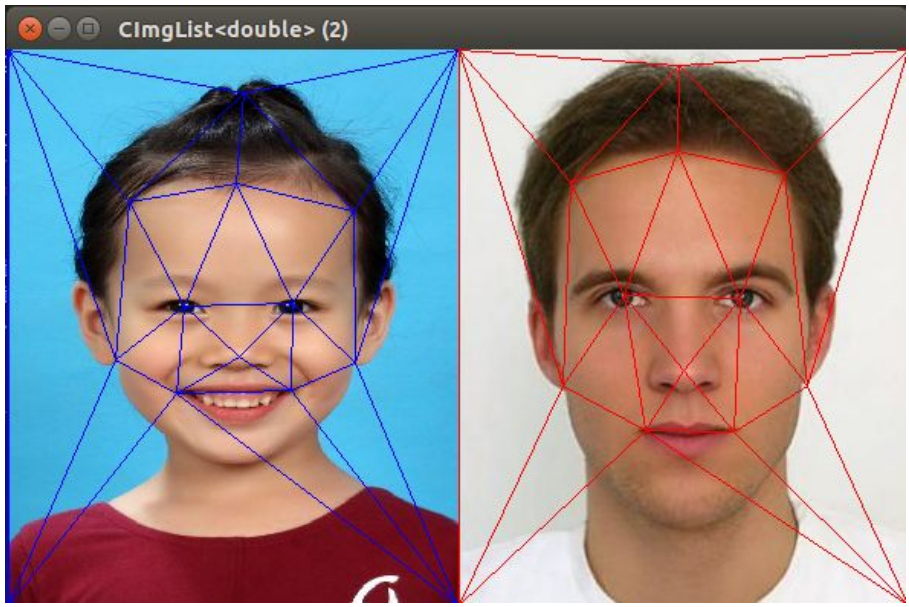
```
// judge if m and p2 are on the same side of line p0p1
static bool isOnSameSide(const Point& p0, const Point& p1, const Point& p2, const Point& m) {
    // line p0p1 is vertical
    if (abs((p0.rho - p1.rho) / double(p0.theta - p1.theta)) < 0.000001) {
        if ((p2.theta - p0.theta) * (m.theta - p0.theta) > 0)
            return true;
        else
            return false;
    }
    else {
        // calculate the efficient of line ab
        double k = (p0.rho - p1.rho) / double(p0.theta - p1.theta);
        double b = p0.rho - k * p0.theta;
        if ((k * p2.theta + b - p2.rho) * (k * m.theta + b - m.rho) < 0)
            return false;
        return true;
    }
}
```

```
// calculate the angle abc
static double angle(const Point& a, const Point& b, const Point& c) {
    int ba[2] = {b.theta - a.theta, b.rho - a.rho};
    int bc[2] = {b.theta - c.theta, b.rho - c.rho};

    return acos((inner_product(begin(ba), end(ba), begin(bc), 0.0)) / (getMagnitude(ba) * getMagnitude(bc)));
}
```

到这里，我们就完成了三角形划分。在这个程序中，我只对source（开始的图）做了三角形划分。dest（终结的图）的三角形划分是依据source的三角形划分来做的（source图和dest图的控制点对应关系在控制点的数据结构中被保存了起来），就是下面简单的几行代码：

```
void syncTriangle() {
    for (int i = 0; i < sourceTriangle.size(); i ++) {
        destTriangle.push_back(Triangle(destControlPoint[sourceTriangle[i].points[0].weight],
                                        destControlPoint[sourceTriangle[i].points[1].weight],
                                        destControlPoint[sourceTriangle[i].points[2].weight]));
    }
}
```

把三角形划分的结果展示出来，可以得到下图（*该图只是示例，只是为了显示清晰，要得到好效果需要打更多控制点*）：



（c）Morphing

其实到了这里才是这次实验的核心部分。

①首先，我们设定了一个interval，表示过渡过程中有多少张中间图，可以依据是第几张中间图来确定变换的ratio，在根据这个ratio对两张图片的控制点进行下式的线性组合，就可以得到中间图的三角形：

$$(u,v) = (x_s,y_s) \times ratio + (x_d,y_d) \times (1 - ratio)$$

代码如下：

```
vector<Triangle> getMiddleTriangle(double ratio) {
    vector<Triangle> middleTriangle;
    for (int i = 0; i < sourceTriangle.size(); i ++) {
        int ax = sourceTriangle[i].points[0].theta * ratio + destTriangle[i].points[0].theta * (1 - ratio);
        int ay = sourceTriangle[i].points[0].rho * ratio + destTriangle[i].points[0].rho * (1 - ratio);
        int bx = sourceTriangle[i].points[1].theta * ratio + destTriangle[i].points[1].theta * (1 - ratio);
        int by = sourceTriangle[i].points[1].rho * ratio + destTriangle[i].points[1].rho * (1 - ratio);
        int cx = sourceTriangle[i].points[2].theta * ratio + destTriangle[i].points[2].theta * (1 - ratio);
        int cy = sourceTriangle[i].points[2].rho * ratio + destTriangle[i].points[2].rho * (1 - ratio);

        middleTriangle.push_back(Triangle(Point(ax, ay, -1),
                                          Point(bx, by, -1),
                                          Point(cx, cy, -1)));
    }
    return middleTriangle;
}
```

②在得到某一个ratio对应的中间图的剖分三角形之后，我们需要求每个三角形到source图和到dest图的变换矩阵，这样我们才能做双线性插值。变换的方式如下：

$$u = A_1 x + A_2 y + A_3$$

$$v = A_4 x + A_5 y + A_6$$

总共有三组点，相当于六个方程，解出六个参数。我们可以像Ex5一样把这个方程组写成 $Ax = b$ 的形式，然后通过 $x = A^{-1} b$ 解出六个参数。由于这里和Ex5的透视变换几乎一样，所以就不加赘述。代码如下：

```
vector<double> getTransformMatrix(Triangle begin, Triangle end) {
    // Ax = b
    CImg<double> A(6, 6, 1, 1);
    CImg<double> b(1, 6, 1, 1);

    // give values to A
    for (int row = 0; row < 3; row ++) {
        A(0, row) = begin.points[row].theta;
        A(1, row) = begin.points[row].rho;
        A(2, row) = 1;
        A(3, row) = 0;
        A(4, row) = 0;
        A(5, row) = 0;
    }
    for (int row = 3; row < 6; row ++) {
        A(0, row) = 0;
        A(1, row) = 0;
        A(2, row) = 0;
        A(3, row) = begin.points[row - 3].theta;
        A(4, row) = begin.points[row - 3].rho;
        A(5, row) = 1;
    }

    // give value to b
    for (int row = 0; row < 3; row ++) {
        b(0, row) = end.points[row].theta;
    }
    for (int row = 3; row < 6; row ++) {
        b(0, row) = end.points[row - 3].rho;
    }

    CImg<double> tranformMatrixCImg = A.get_invert() * b;

    vector<double> tranformMatrix;
    for (int i = 0; i < 6; i ++) {
        tranformMatrix.push_back(tranformMatrixCImg(0, i));
    }

    return tranformMatrix;
}
```

③得到中间图的各个三角形到source图和dest图的各个三角形的变换关系之后，我们就可以进行双线性插值了，对于中间图的每一个像素点，先看它在中间图的哪个三角形内，这个由下面的算法解决：

在平面上给定一点$p(x_{0_{y_0}})$，判断是在三角形ABC中，可以用面积法来判断。

若$S(ABC) = S(PBC) + S(PAB) + S(PAC)$，则该点在三角形内部。

若$S(ABC) < S(PBC) + S(PAB) + S(PAC)$，则该点在三角形外部。

三角形的面积可以用海伦公式求得。

代码如下：

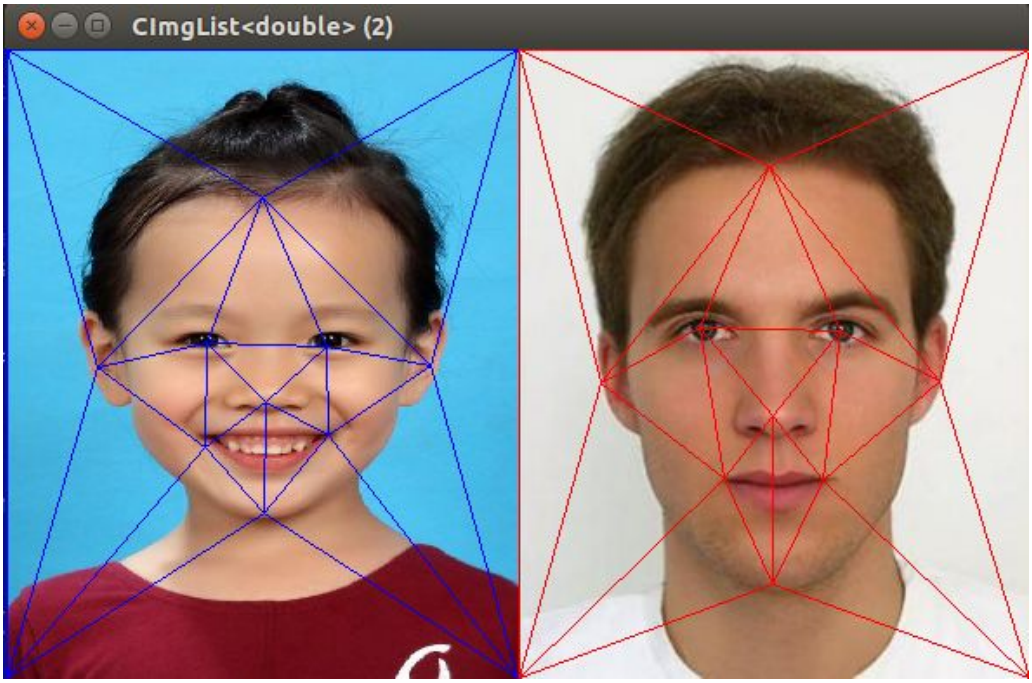
```
double getArea(Point a, Point b, Point c) {
    double ab = sqrt(pow((a.theta - b.theta), 2) + pow((a.rho - b.rho), 2));
    double bc = sqrt(pow((b.theta - c.theta), 2) + pow((b.rho - c.rho), 2));
    double ac = sqrt(pow((a.theta - c.theta), 2) + pow((a.rho - c.rho), 2));
    double s = (ab + bc + ac) / 2;
    return sqrt(s * (s - ab) * (s - bc) * (s - ac));
}

bool isPointInTriangle(Point x, Triangle tri) {
    double sum = getArea(x, tri.points[0], tri.points[1]) + getArea(x, tri.points[0], tri.points[2]) + getArea(x, tri.points[1], tri.points[2]);
    double triArea = getArea(tri.points[0], tri.points[1], tri.points[2]);
    if (abs(sum - triArea) <= 0.00001)
        return true;
    return false;
}
```

知道一个点在中间图的哪个三角形内之后，我们就通过求得的变换关系转换到source和dest图，进行双线性插值。双线性插值在Ex5已经阐释过了，在这里就不再赘述。值得注意的是，由于像素点是离散化的，变换后的点可能会越界，所以需要进行一些简单的处理。这一部分代码如下：

```
cimg_forXY(target, x, y) {
    bool sourceFindFlag = false;
    bool destFindFlag = false;

    for (int j = 0; j < middleTriangle.size(); j ++) {
        if (isPointInTriangle(Point(x, y, -1), middleTriangle[j])) {
            int uSource = middle2source[j][0] * x + middle2source[j][1] * y + middle2source[j][2];
            int vSource = middle2source[j][3] * x + middle2source[j][4] * y + middle2source[j][5];

            int uDest = middle2dest[j][0] * x + middle2dest[j][1] * y + middle2dest[j][2];
            int vDest = middle2dest[j][3] * x + middle2dest[j][4] * y + middle2dest[j][5];

            if (uSource >= 0 && uSource < source.width() && vSource >= 0 && vSource < source.height()) {
                cimg_forC(source, channel) {
                    target(x, y, 0, channel) += source.linear_atXY(uSource, vSource, 0, channel) * ratio;
                };
                sourceFindFlag = true;
            }

            if (uDest >= 0 && uDest < dest.width() && vDest >= 0 && vDest < dest.height()) {
                cimg_forC(dest, channel) {
                    target(x, y, 0, channel) += dest.linear_atXY(uDest, vDest, 0, channel) * (1 - ratio);
                };
                destFindFlag = true;
            }
            break;
        }
    }
    // might lead to bug because of size issue
    if (!sourceFindFlag) {
        cimg_forC(source, channel) {
            target(x, y, 0, channel) += source.linear_atXY(x, y, 0, channel) * ratio;
        };
    }
    if (!destFindFlag) {
        cimg_forC(source, channel) {
            target(x, y, 0, channel) += dest.linear_atXY(x, y, 0, channel) * (1 - ratio);
        };
    }
};
```

④不断重复①~③，就能得到很多*.jpg，把它们串起来做成一个gif，就能得到一个face morphing的动态演示图了

5、 实验结果



左图为source，右图为dest。上图为三角形划分之后的结果。我的实验interval设为40，所以一共有40张图，比较大，就不放上来了。所以我把这40张图生成的gif附在作业中，命名为animated.gif。

6、 总结及提高

从gif来看，效果还是不错的，达到了这次实验的总体目的，但是在几个地方还是有很大的提高空间：

①在三角形划分的实现中，首先是算法不是很严谨，可能会出现不是很好的三角形划分。其次是复杂度比较高，达到了 $O(n^4)$。以后应该采用一些比较成熟的算法来解决这个问题。

②代码的模块化做的不好。有一些属于点的、线的或是三角形的函数都没有封装。像这一次完全可以写一个三角形的类，把一些对于三角形的操作放在里面，可是一开始没有考虑到这些因素。这样以后要是再用到三角形的一些操作的时候会比较麻烦。

但是从总体上来说对这次实验还是比较满意的，做出了一些有趣的东西。