Lab 01 实验报告

PB20000296 郑滕飞

代码结构介绍：

代码主文件名为 lab1.m，开头部分如下：

```
function Funcollect = lab1
    Funcollect.plusimage = @plusimage; %plus two images
    Funcollect.timesimage = @timesimage; %times two images
    Funcollect.bright = @bright; %increase or decrease brightness
    Funcollect.gamma = @gamma; %increase or decrease gamma
    Funcollect.equalize = @equalize; %histogram equalization
    Funcollect.match = @match; %histogram maching
    Funcollect.PDF = @PDF; %PDF of an image
    Funcollect.gnoise = @gnoise; %Gauss noise
    Funcollect.pnoise = @pnoise; %Poisson noise
    Funcollect.snoise = @snoise; %salt&pepper noise
    Funcollect.convolution = @convolution; %concolute by given kernel
    Funcollect.medianfilter = @medianfilter; %median filter
end
```

代码中通过主函数 lab1 调用了自己写的各种函数，这些函数的输入一般是图片的名称，而输出的默认名称则是 result_加函数名。如，测试文件夹中有名为 a.jpg 的图片，对其实现均衡化的过程是：

```
func = lab1;
func.equalize('a.jpg');
```

运行结束后，所在的文件夹内即会出现名为 result_equalization.jpg 的图片作为输出，而函数的返回值是输出的图片对应的矩阵。

代码文件夹中的测试图片有 a-f，x-y 八张，前六张为拍摄的照片，后两张为下载的专辑封面(均为合成图片)，以下直接以测试图片的编号作为代称。

逐点操作：

1、图像加法/乘法

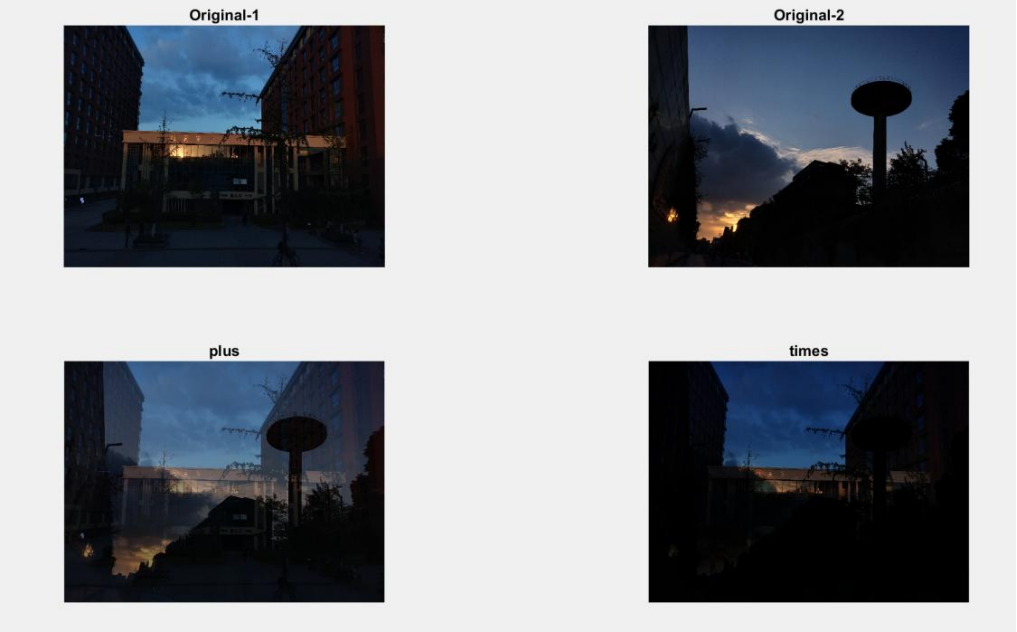
函数名：plusimage/timesimage

输入参数：输入图像 1 名称、输入图像 2 名称、输出图像名称[可选]

直接仿照讲义进行平均/点乘归一化操作即可，乘法的归一化部分为：

```
J1 = double(I1(1:R,1:C,:));
J2 = double(I2(1:R,1:C,:));
J = J1.*J2;
M = max(J(:));
m = min(J(:));
J = uint8(255*(J-m)/(M-m));
```

最终效果(测试图片为图片 b 与图片 f)：

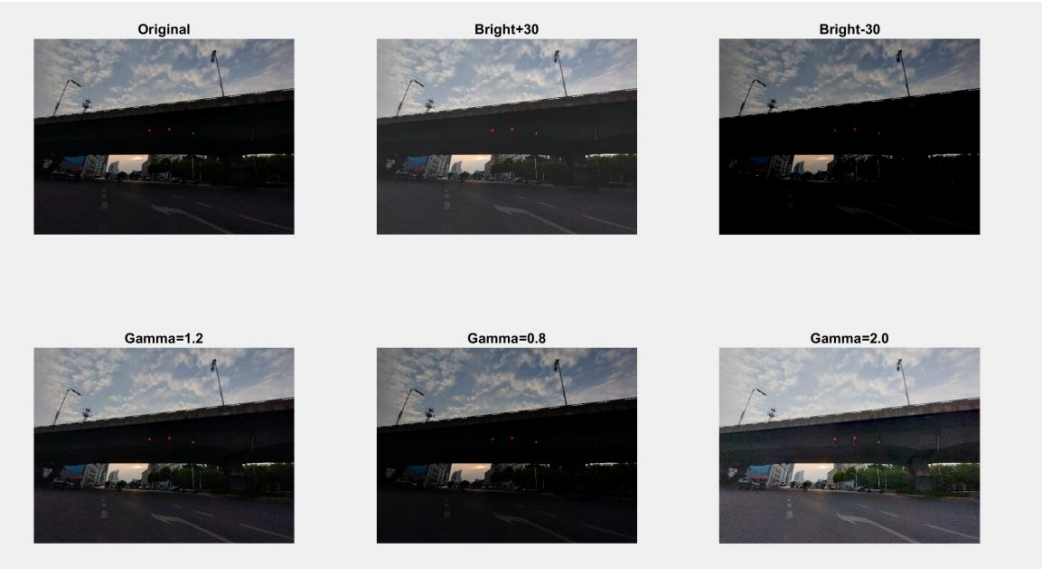


2、亮度/Gamma 值调整

函数名：bright/gamma

输入参数：输入图像名称、调整值、输出图像名称[可选]

按照讲义的提示对每个像素进行操作，亮度调整为同加减(注意上下限)，Gamma 值为输入对应指数，效果如下(测试图片为图片 e)：



整体来看，在调整幅度大时，Gamma 值调整的表现会比直接调整亮度自然很多。

3、直方图与 PDF

函数名：PDF

输入参数：输入图像的矩阵

直方图可直接通过 histogram 统计，为了方便之后操作，需要将 PDF 显式地表现出来，

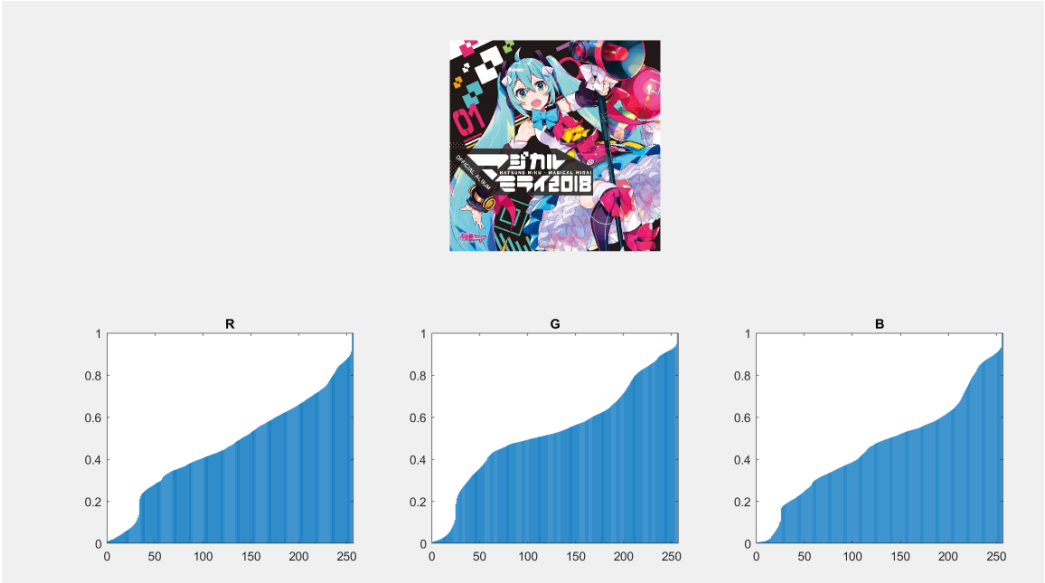
因此需要 PDF 函数统计。

函数的主体如下：

```
function h = PDF(I)
    [R, C, B] = size(I);
    h = zeros(256,1,B);
    for g = 0:255
        h(g+1,1,:) = sum(sum(I==g));
    end
    h = h / (R * C);
    for i = 1:B
        for g = 0:255
            h(256-g,1,i) = sum(h(1:256-g,i));
        end
    end
end
```

先统计像素个数，再除以总数得到比例，最后累加成为 PDF。

以图片 y 为例，三个通道的 PDF 如下[采用辅助脚本 draw_pdf 绘制]：



4、均衡化

函数名：equalize

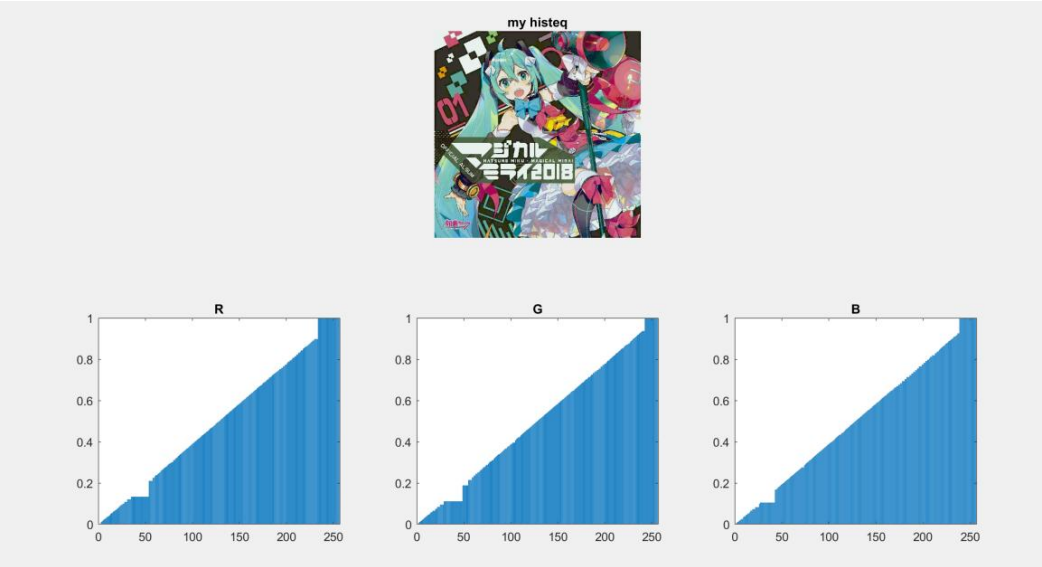
输入参数：输入图像名称、输出图像名称[可选]

利用上课学习的算法，将每个通道中亮度为 0-255 的 256 段拼成尽量接近均匀分布的样子，核心代码如下：

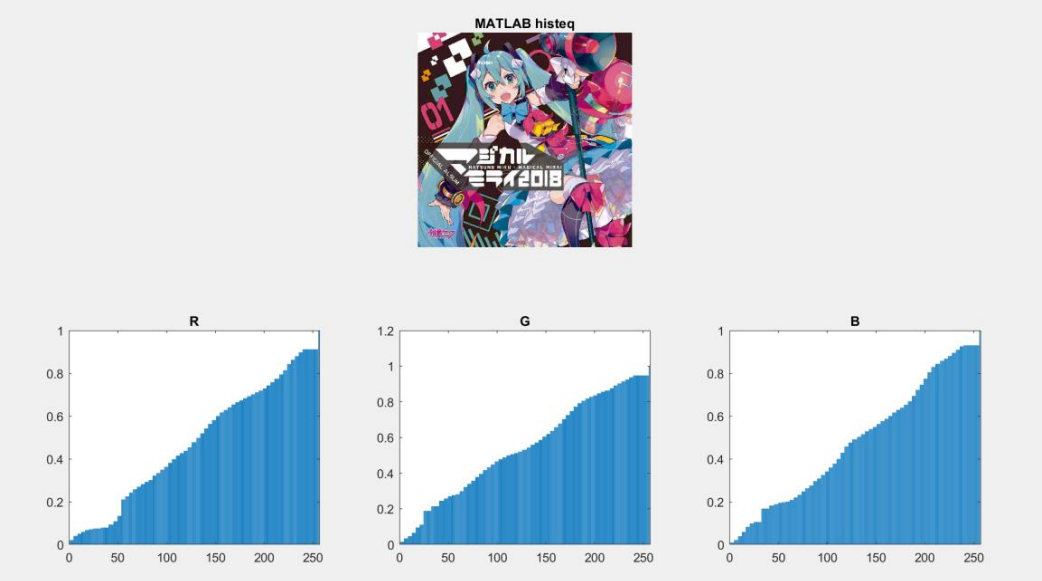
```
h = PDF(I);
h = ceil(h*256) - 1;
J = zeros(R,C,B,'uint8');
b = 1;
for i = 1:numel(I)
    J(i) = h(I(i) + 1,1,b);
```

```
    if (mod(i,R*C) == 0)
        b = b + 1;
    end
end
```

PDF 以 255 为上限归一化后直接成为了查找表的形式，再根据查找表改变 I 的每个通道像素的亮度即可。对上方的图片 y，均衡化的效果如下：

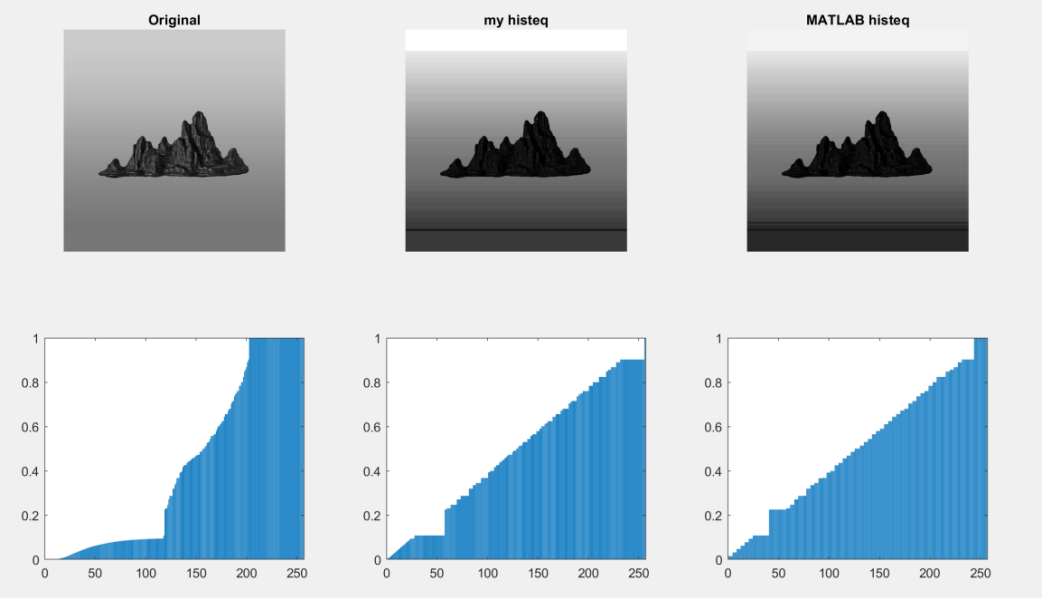


值得注意的是，均衡化后存在一定的变色，这是由于分别将三个通道进行均衡化时，原本总亮度偏低的通道会在均衡中被调高，因此导致变色。MATLAB 自带的均衡化功能 histeq 则解决了这样的问题：



自带的均衡化携带了一部分原有图片的 PDF 特征，在基础之上让其变得更加平缓，从而更好保持原有的颜色特征。

除了这样对彩色图片的均衡化外，对灰度图(提取了图片 x 的红色通道)的两种均衡化也存在一定差别：



在突然上升处和末尾处，自带的 histeq 保留了更多特征。不过总体来说，对灰度图均衡化 的区别并不像彩图那么大。

5、匹配

函数名：match

输入参数：输入图像名称、目标图像名称、输出图像名称[可选]

与之前的直方图均衡化类似，在得到目标图片各个通道的 PDF 后通过调整分段即可使原图的颜色接近目标图。

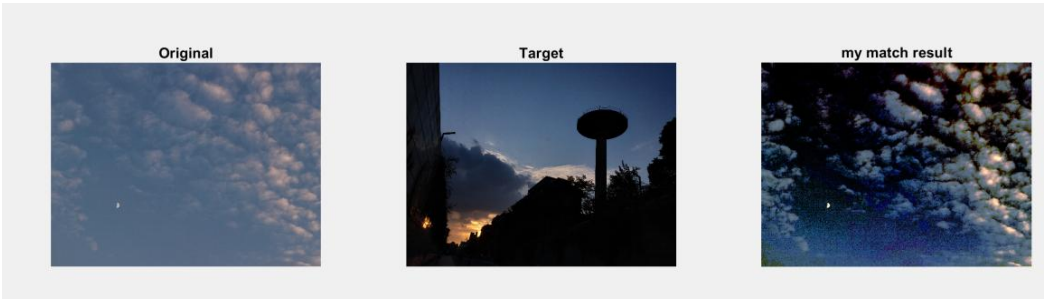
代码如下：

```
function J = match(inputname1, inputname2, outputname)
    I = imread(inputname1, "jpg");
    [R, C, B] = size(I);
    Tar = imread(inputname2, "jpg");
    h = PDF(I);
    g = PDF(Tar);
    LUT = zeros(256,1,B);
    for b = 1:B
        j = 0;
        for i = 0:255
            while (g(j+1,1,b) < h(i+1,1,b)) && (j < 255)
                j = j + 1;
            end
            LUT(i+1,1,b) = j;
        end
    end
    J = zeros(R,C,B,'uint8');
    b = 1;
    for i = 1:numel(I)
```

```
            J(i) = LUT(I(i) + 1,1,b);
            if (mod(i,R*C) == 0)
                b = b + 1;
            end
        end
        if (~exist('outputname','var'))
            outputname = 'result_match.jpg';
        end
        imwrite(J, outputname, 'jpg');
end
```

对 J 的写入过程与直方图均衡化时类似，不过构建查找表的操作(参考自讲义)更加复杂，主要思想是通过调整 i，j 的“前进速度”来达成效果。

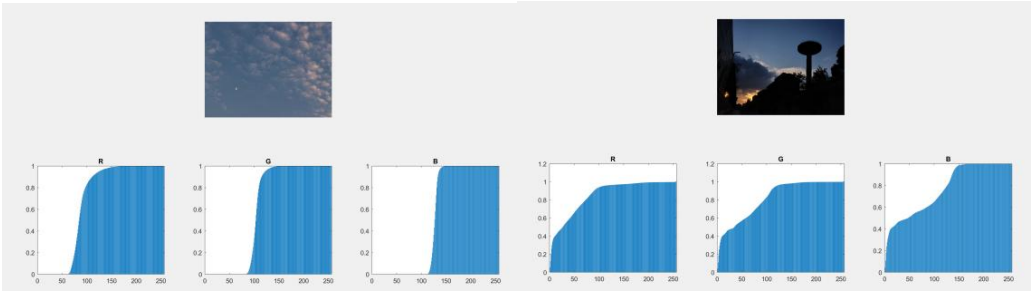
输入图片 a 与图片 f，结果如下：



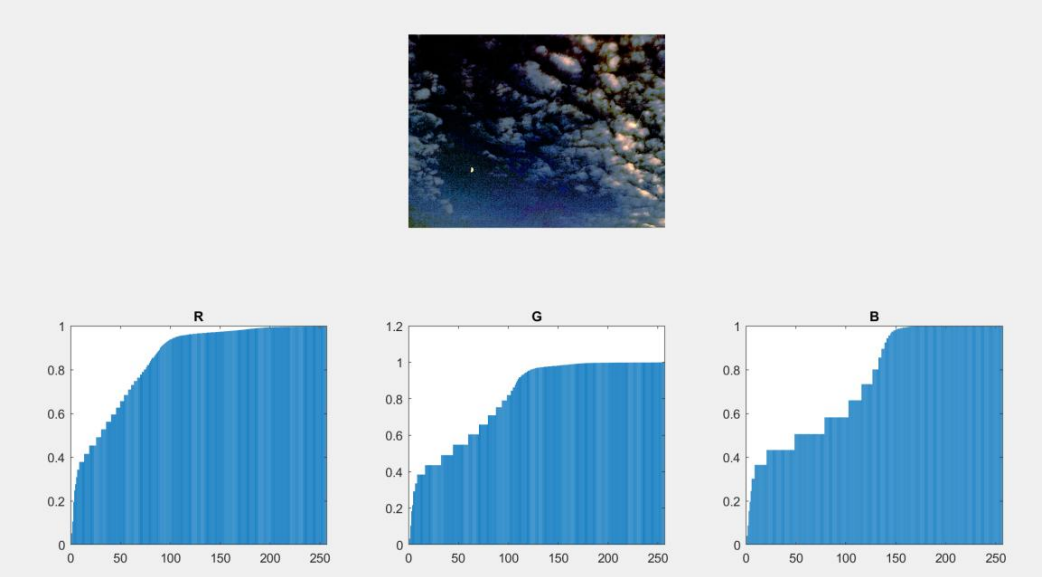
MATLAB 自带的匹配直方图则和上方操作不尽相同。它并不会将三个通道分别匹配，而是在接收目标图片的整体直方图后进行统一处理：

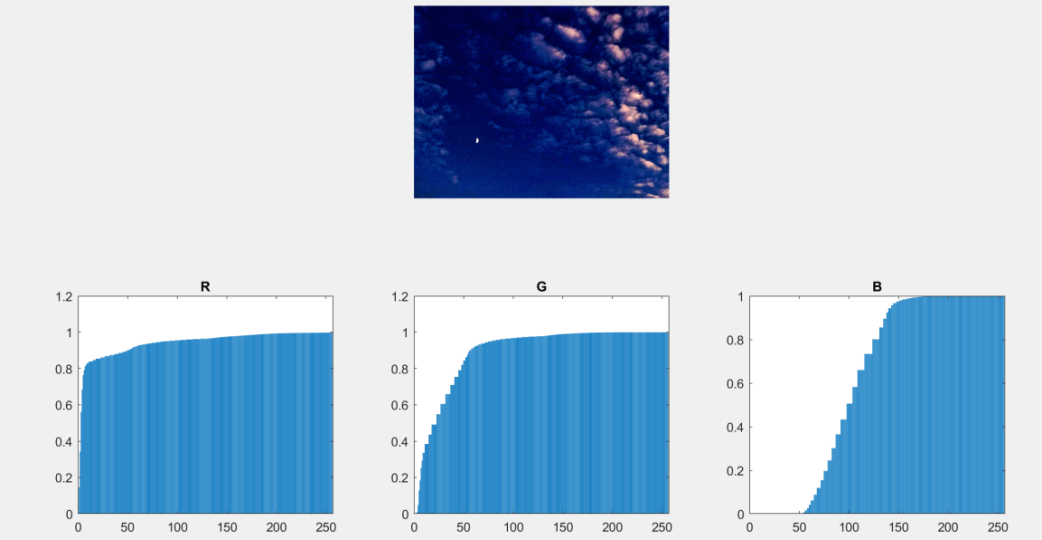
与直方图均衡化时的情况相似，MATLAB 自带的结果保留了更多的颜色信息，整体与原图的相似程度更高。

两张原图的 PDF 如下：



而匹配后，自己实现与自带实现的 PDF 则是下图：





总体来说，自己实现的 PDF 会更接近目标图片，而自带实现后的 PDF 则与原本的 PDF 特征存在一定相似，这个结论与直方图均衡化时是类似的。

卷积与滤波：

1、高斯/泊松/椒盐噪音

函数名：gnoise/pnoise/snoise

输入参数：输入图像名称、噪声幅度、输出图像名称[可选]

高斯噪声与泊松噪声都是在原图片上加上对应分布的随机变量矩阵：

```
Rand = normrnd(0, value, [R,C,B]);
J = uint8(I + uint8(100*Rand));
```

```
Rand = poissrnd(value, [R,C,B]) - value;
J = uint8(I + uint8(20*Rand));
```

其中，value 为代表幅度的输出参数。生成泊松噪声的过程中，由于输入的 value 即为泊松分布的期望，原始的随机阵减去 value 后期望为 0，这样添加噪声后总体的亮度不会

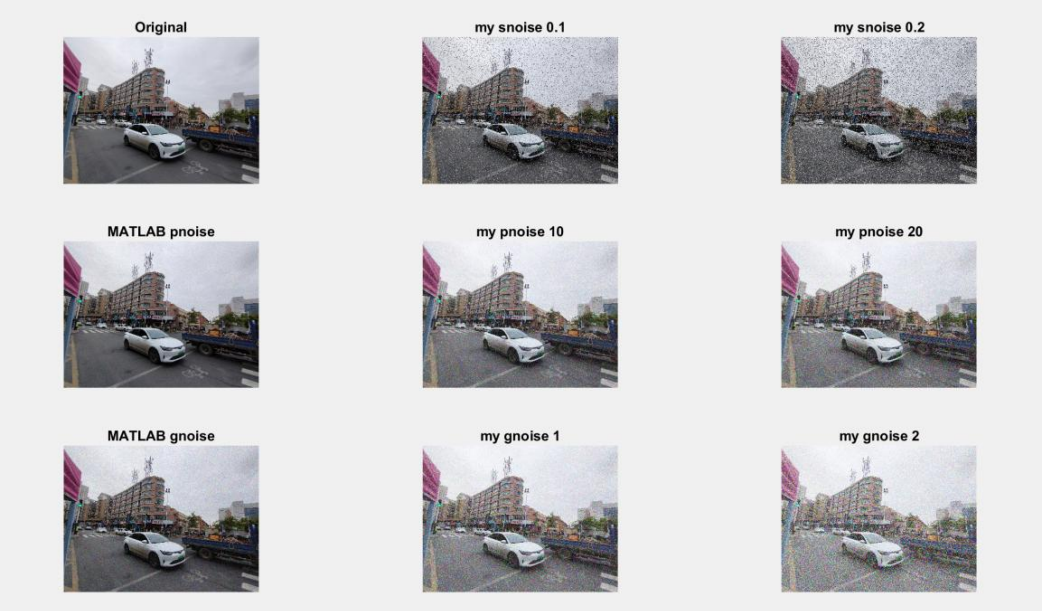
变化。

而椒盐噪声的代码为：

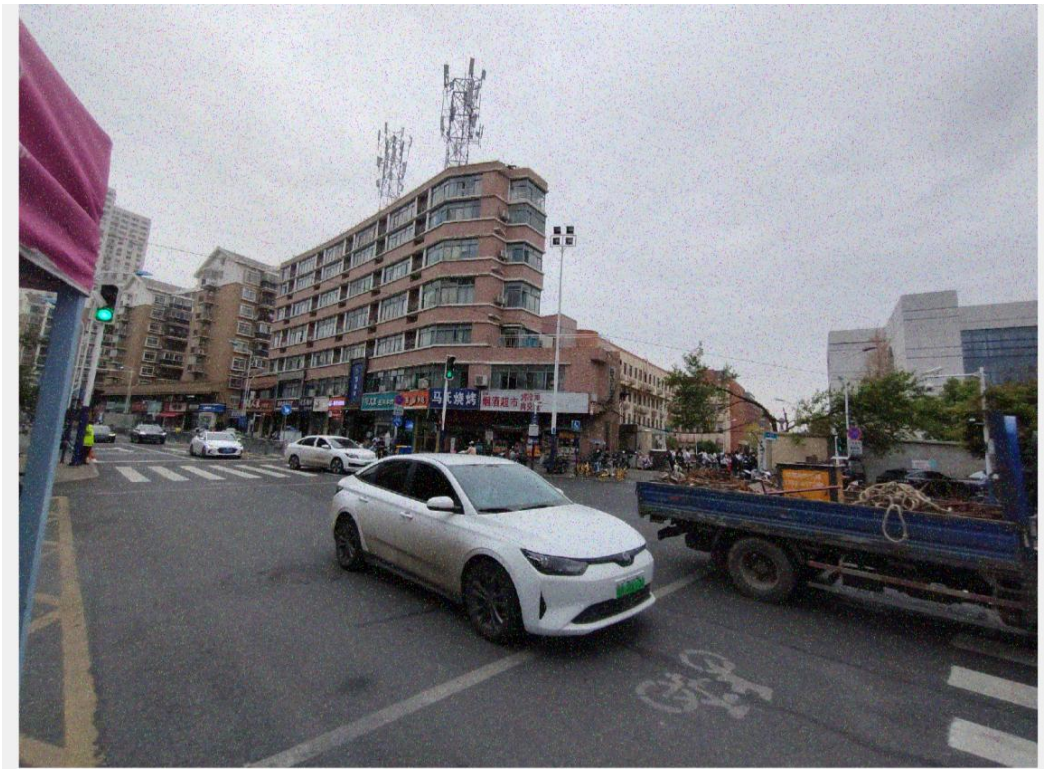
```
Rand = rand([R,C]);
J = zeros(R,C,B,'uint8');
for i = 1:R
    for j = 1:C
        if (Rand(i,j)<value/2)
            for b = 1:B
                J(i,j,b) = 0;
            end
        elseif (Rand(i,j)>1-value/2)
            for b = 1:B
                J(i,j,b) = 255;
            end
        else
            J(i,j,:) = I(i,j,:);
        end
    end
end
```

这里的 value 表示污染的像素点的比例，通过生成 01 之间的随机数后判断，将一部分像素点随机变成了白点或黑点。

对图片 d 添加噪声的结果如下：



自带的高斯噪声与泊松噪声(默认参数)比我自己预设的参数要低不少，不过实现方式是基本一致的。而自带的椒盐噪声则是对三个通道分别进行判断操作，可以看出图片中 RGB 通道被提升或拉低后的点：



以下的滤波均针对默认参数下的系统实现噪声，彩色图来源即为上方的图片 d，灰度图为图片 c 的蓝色通道。

2、卷积滤波

函数名：gnoise/pnoise/snoise

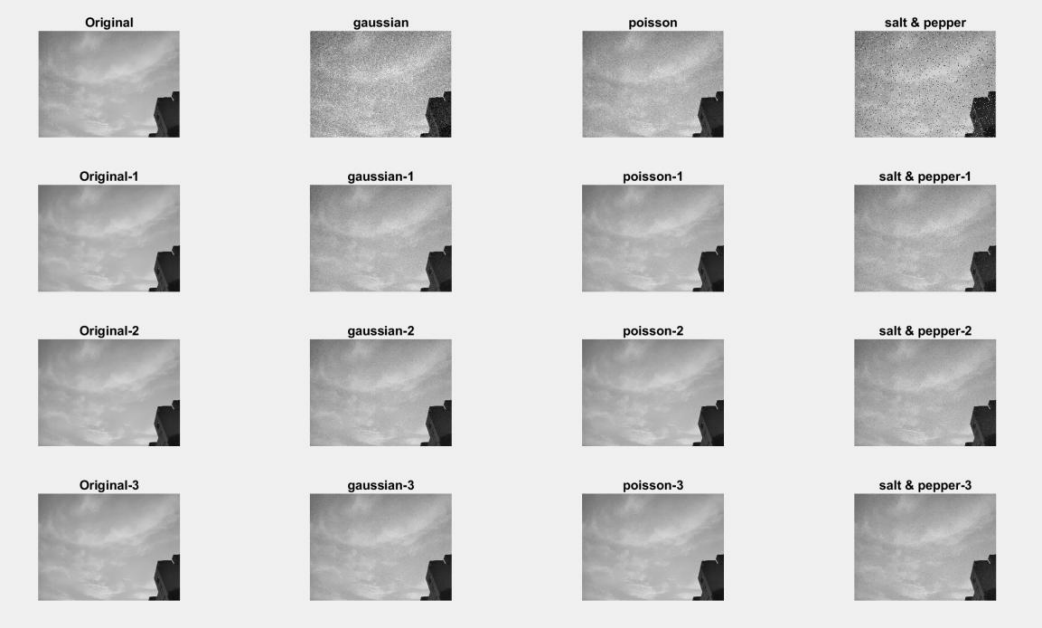
输入参数：输入图像名称、噪声幅度、输出图像名称[可选]

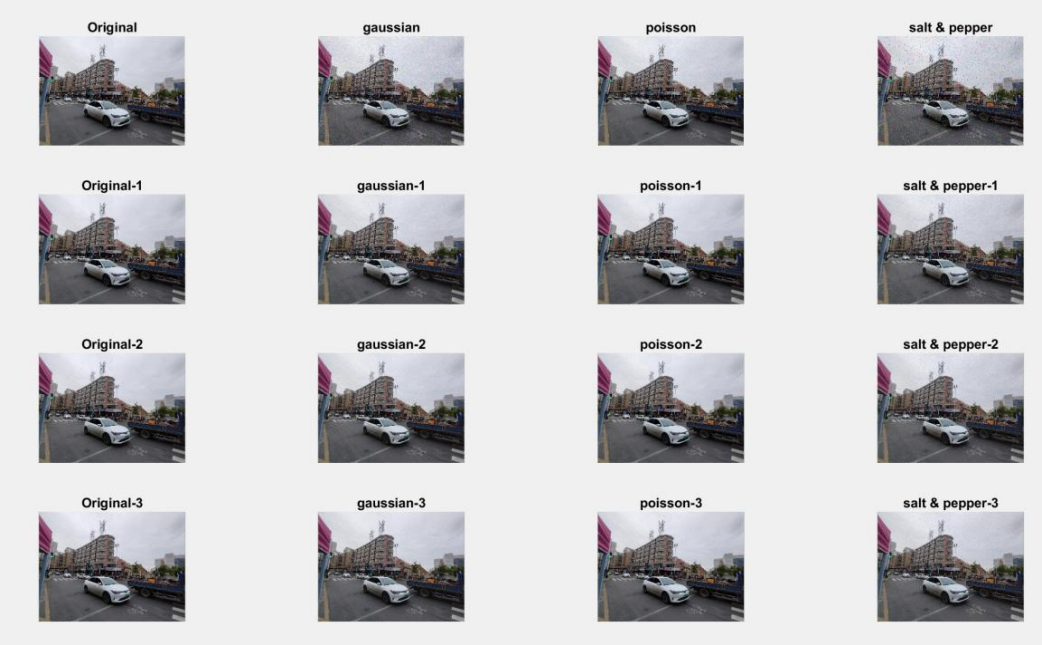
均值滤波与高斯滤波都是卷积滤波，对代码中实现的 3 维方阵来说，均值的系数均为 1/9，而高斯滤波的系数则是[0.05,0.1,0.05;0.1,0.4,0.1;0.05,0.1,0.05]。

具体的卷积代码为：

```
I = imread(inputname, "jpg");
[R, C, B] = size(I);
I0 = zeros(R+2,C+2,B,'uint8');
J = zeros(R,C,B,'uint8');
I0(2:R+1,2:C+1,:) = I;
for b = 1:B
    for c = 1:C
        for r = 1:R
            J(r,c,b) = sum(sum(double(I0(r:r+2,c:c+2,b)).*kernel));
        end
    end
end
```

此处，可以看作最边缘添加了一圈黑色的像素用来卷积，按照给定的 kernel 对周围的八个像素和原像素作一定程度的平均处理。(下方图片的输出方式来自脚本 temp 与 temp2，需要在 a=lab1 后令 I 为输入图像的矩阵，便可得到输出。修改部分代码可进行不同滤波。)

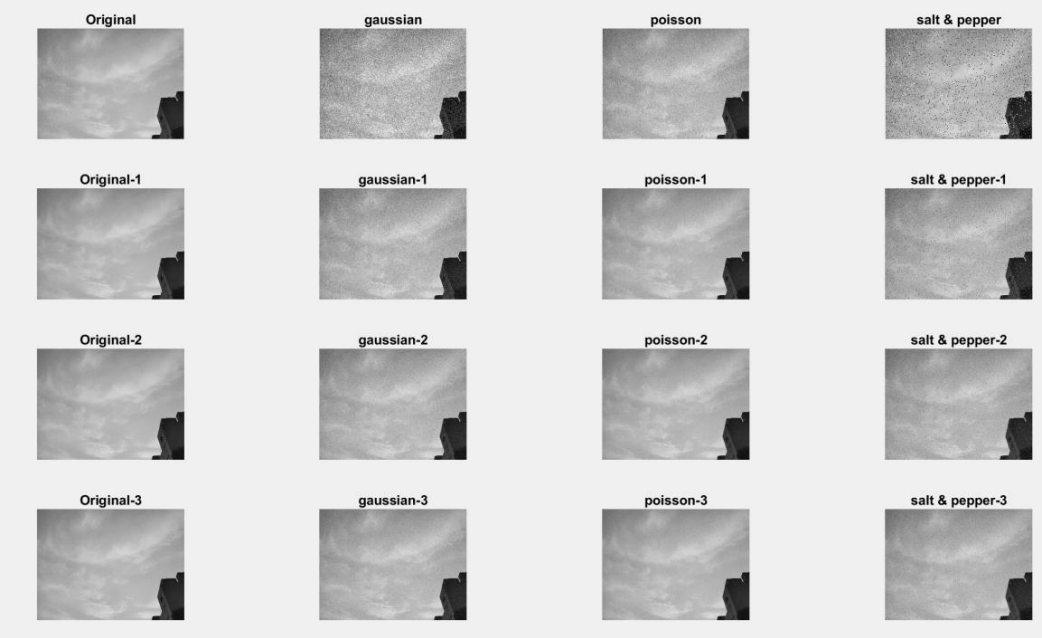


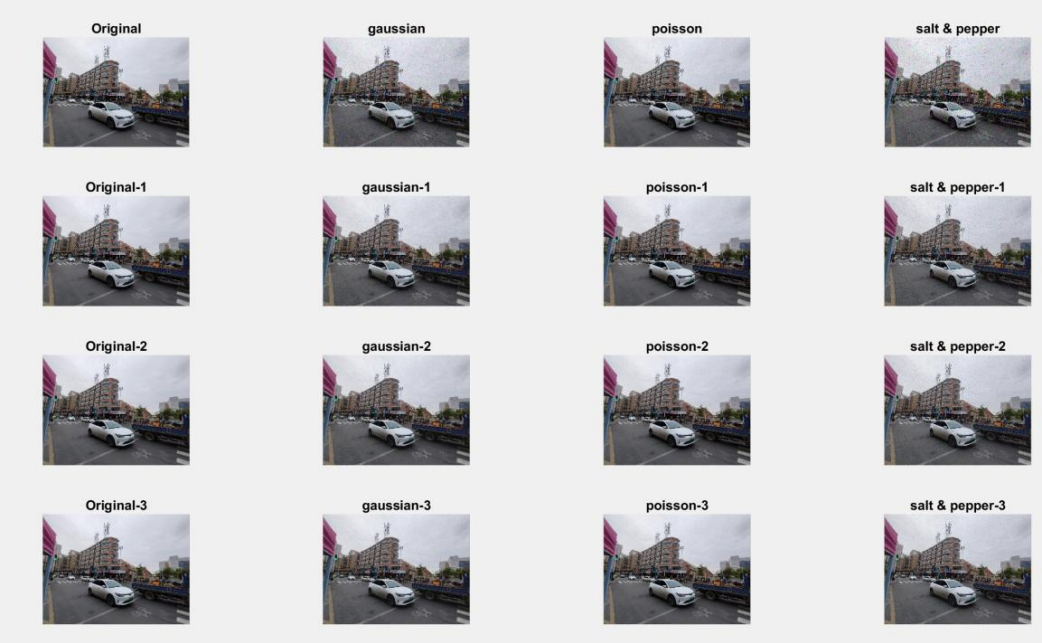


上方是我自己实现的均值滤波的结果(123 代表三次滤波)，可以发现，均值滤波对噪音有一定的去除效果，但图片会明显出现模糊（下方为原图与三次滤波后的文字部分对比）。

与之相对，下方是自己实现的高斯滤波的效果，滤波效率比起均值来说要降低一些，不过对图像信息的保存也比均值滤波强一些。

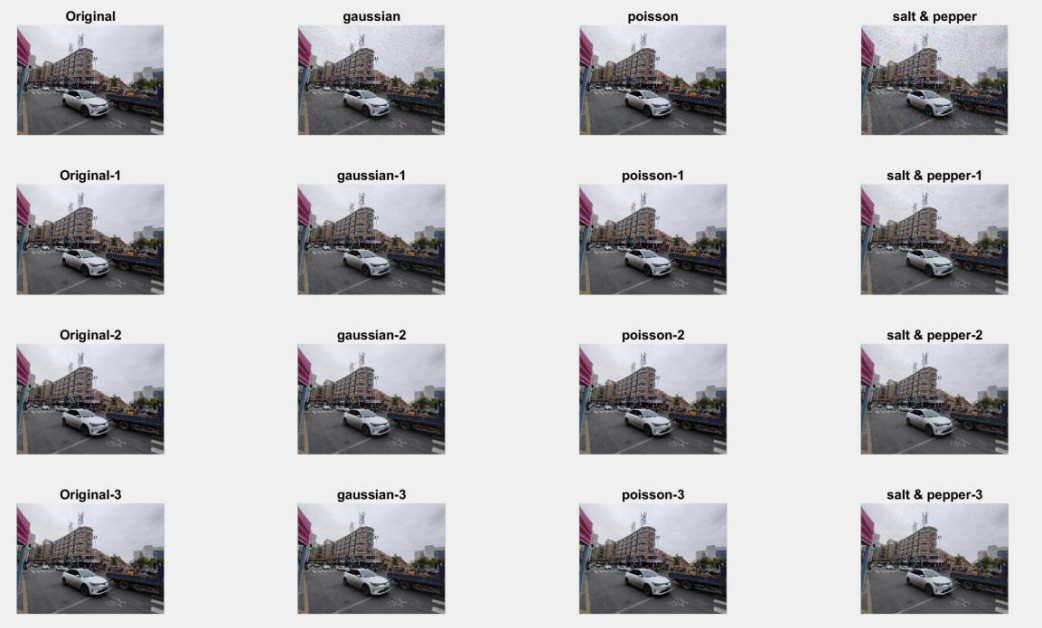
值得一提的是，自己实现的卷积操作没有用傅里叶等方法进行优化，因此非常缓慢，将这十六张图跑出来需要数分钟时间。

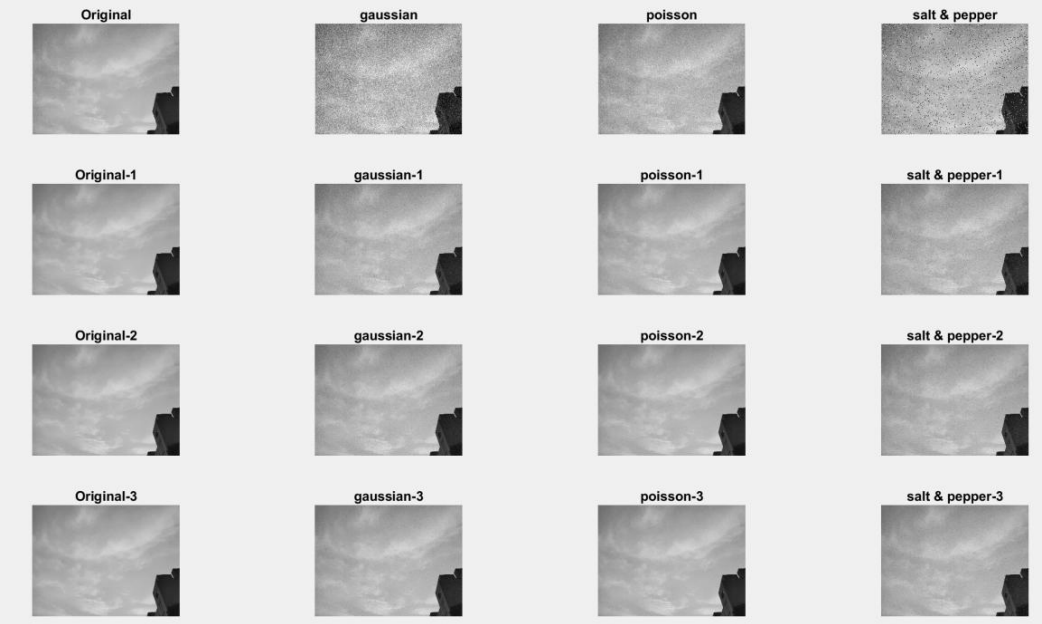




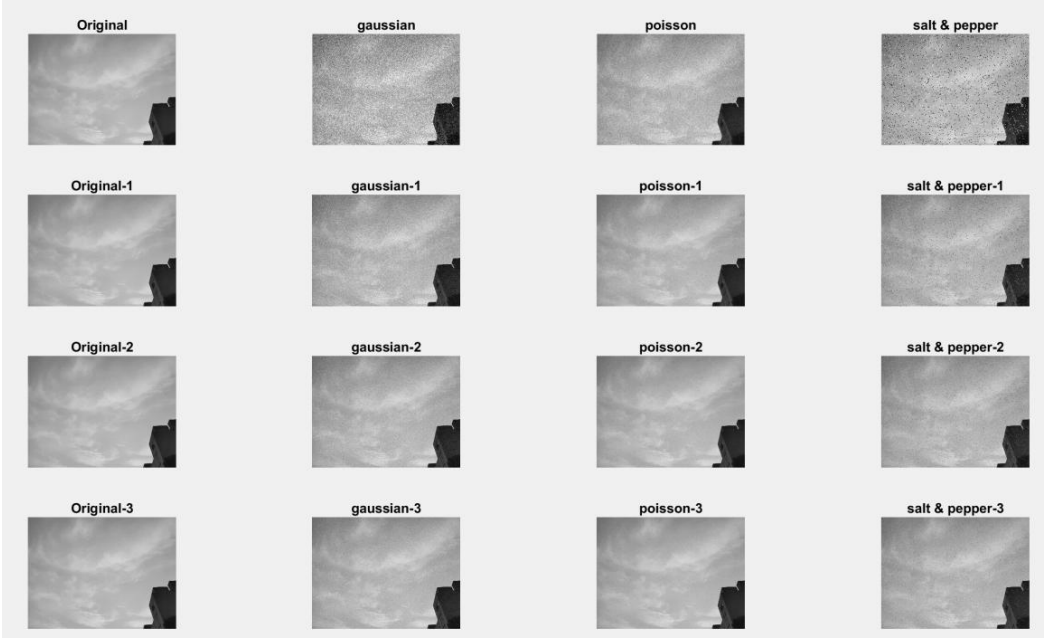
上方三次滤波后的原图可以看出，对清晰度的保存比均值滤波好了很多。

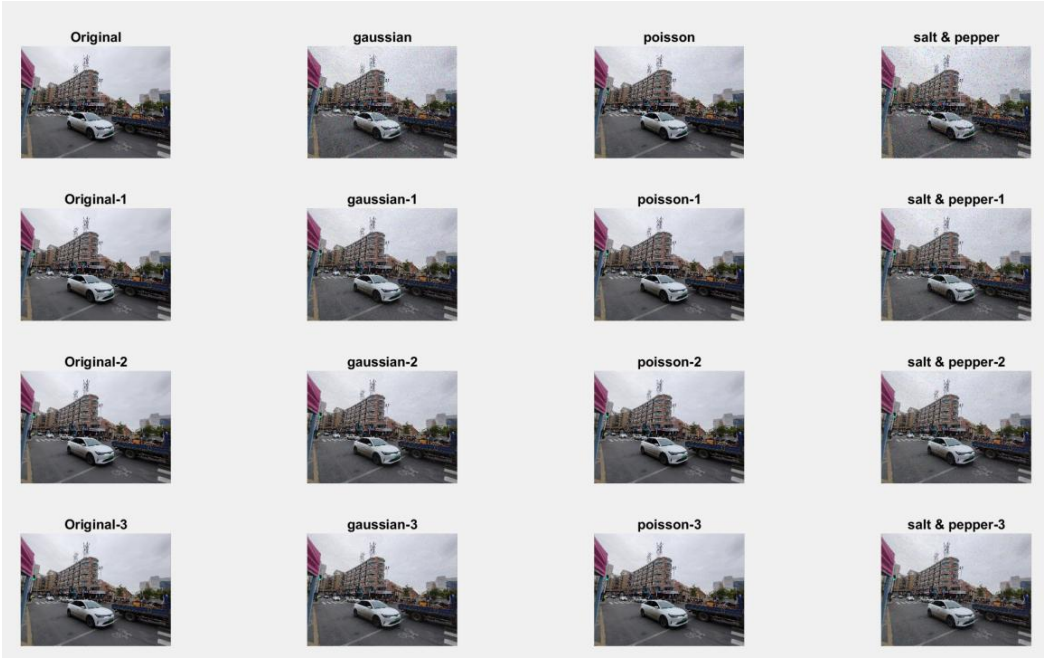
下方展示的是 MATLAB 自带的均值滤波与高斯滤波的结果。总体来说，结果是基本相似的，但由于进行了优化，自带的卷积滤波要比自己写得快很多。


Original
gaussian
poisson
salt & pepper
Original-1
gaussian-1
poisson-1
salt & pepper-1
Original-2
gaussian-2
poisson-2
salt & pepper-2
Original-3
gaussian-3
poisson-3
salt & pepper-3


Original
gaussian
poisson
salt & pepper
Original-1
gaussian-1
poisson-1
salt & pepper-1
Original-2
gaussian-2
poisson-2
salt & pepper-2
Original-3
gaussian-3
poisson-3
salt & pepper-3


烟酒超市


Original
gaussian
poisson
salt & pepper
Original-1
gaussian-1
poisson-1
salt & pepper-1
Original-2
gaussian-2
poisson-2
salt & pepper-2
Original-3
gaussian-3
poisson-3
salt & pepper-3


Original
gaussian
poisson
salt & pepper
Original-1
gaussian-1
poisson-1
salt & pepper-1
Original-2
gaussian-2
poisson-2
salt & pepper-2
Original-3
gaussian-3
poisson-3
salt & pepper-3


烟酒超市
烤冷面
肉夹馍

3、中值滤波

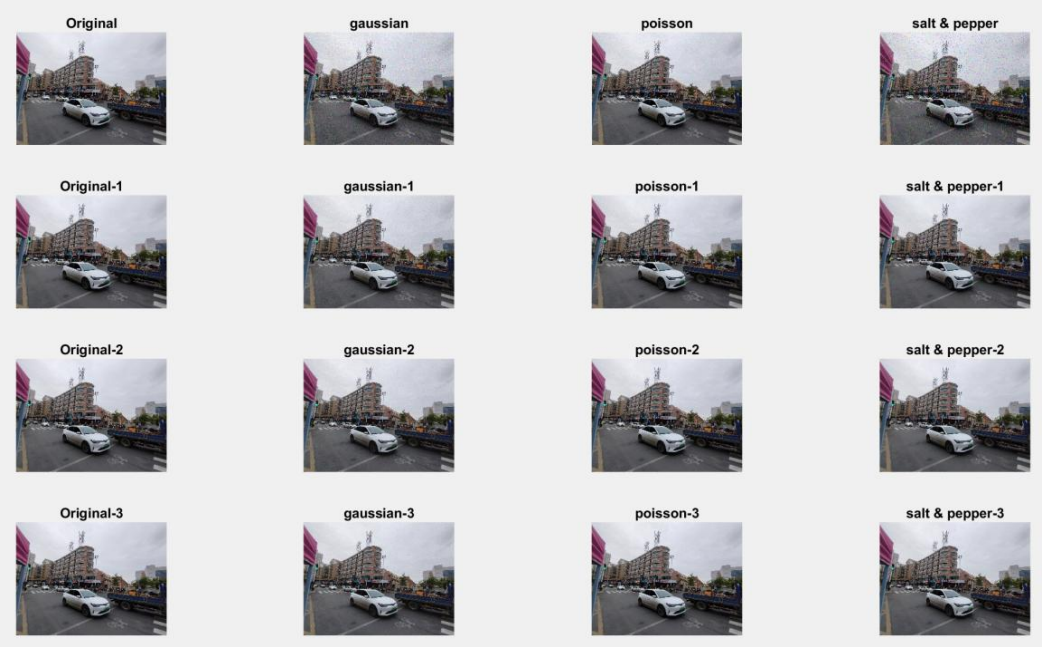
函数名：medianfilter

输入参数：输入图像名称、输出图像名称[可选]

以 3 维方阵对图像进行中值滤波，此处的中值是三个通道分别计算的，而非整体中值。

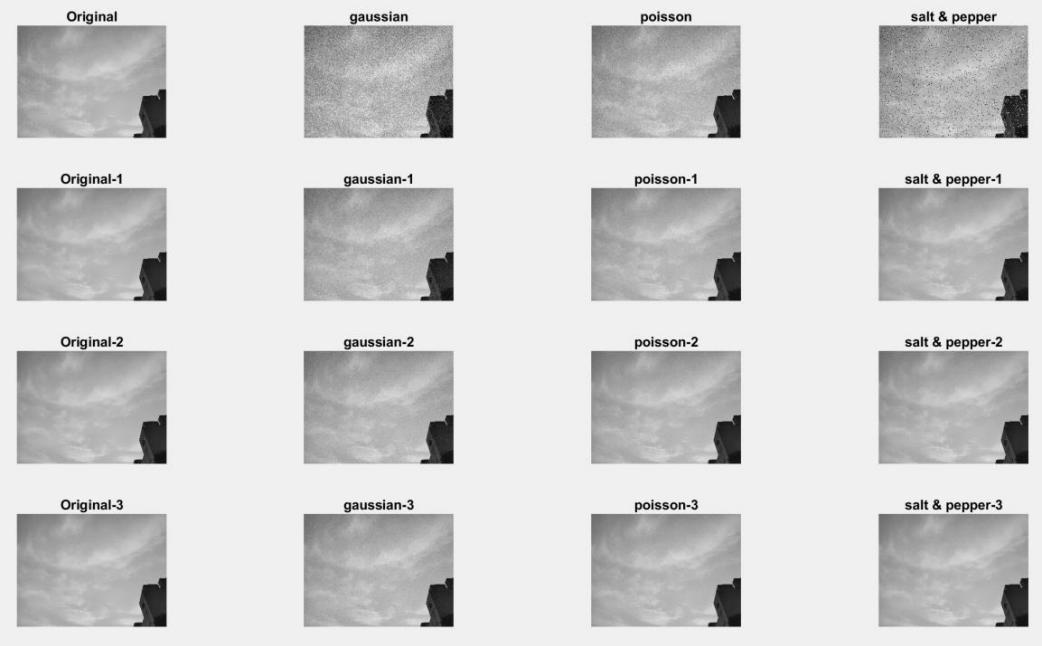
```
I = imread(inputname, "jpg");
[R, C, B] = size(I);
I0 = zeros(R+2,C+2,B,'uint8');
J = zeros(R,C,B,'uint8');
I0(2:R+1,2:C+1,:) = I;
for b = 1:B
    for c = 1:C
        for r = 1:R
            J(r,c,b) = median(I0(r:r+2,c:c+2,b),'all');
        end
    end
end
```

中值滤波实现的效果如下：

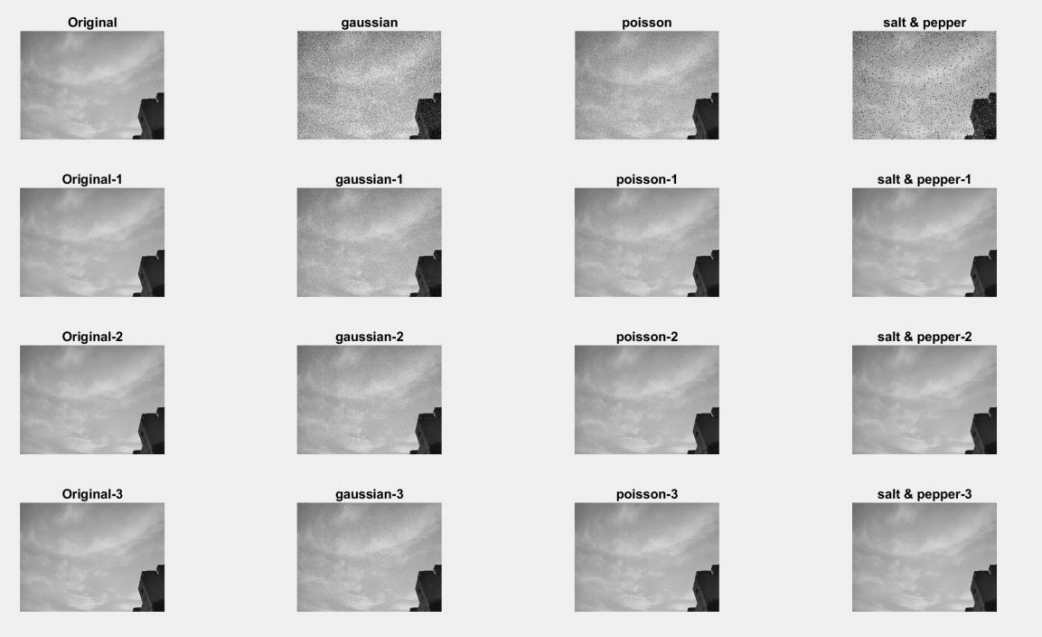


滤波三次后，文字部分仍然有着较高的清晰度，对噪音的去除效果也较为明显。对灰度

图表现如下:



MATLAB 自带的中值滤波 `medfilt2` 只能对单通道进行处理，不过还是比自己实现的代码快很多。对灰度图的结果如下:



总结:

第一次实验的核心感受是，初用 MATLAB 存在各种麻烦和不适应，不过真的做出结果看到图像的变化时还是很开心的(虽然在跑自己的程序时因为太慢一度以为卡死了)。

熟悉矩阵的操作之后，代码写起来就快了很多，希望 Lab2 不会再这么痛苦了。

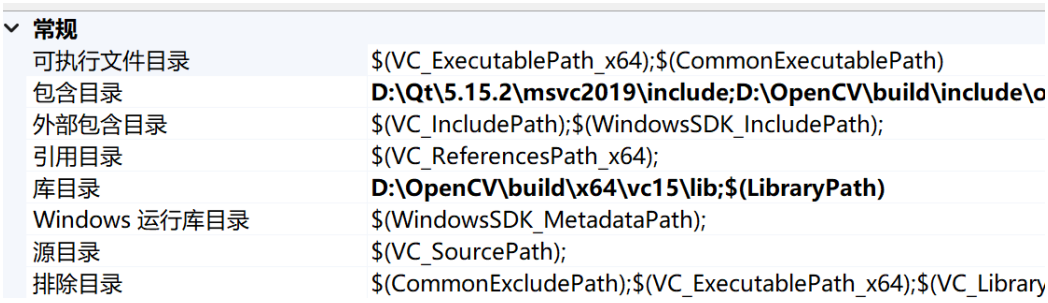
Lab 02 实验报告

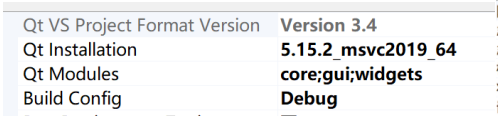
PB20000296 郑滕飞

泊松融合：

1、环境配置

为了所给框架能够运行，需要配置 Qt、OpenCV、Eigen 三个库并将解决方案中的包含目录、库目录、Qt 版本、链接器对应修改，如下图所示。





2、代码实现

在框架中，总共需要补充的有构造 A、构造 b，求解三个过程。事实上，构造的方式是与一维完全类似的，将任何两个相邻的相差与复制进来的图片的相差进行比较，再用总和进行最小二乘的构造。

```
for (int i = 0; i < index_mapto_coor.size(); i++)
{
    x_i = index_mapto_coor[i].x;
    y_i = index_mapto_coor[i].y;
    coefs.push_back(Eigen::Triplet<double>(i, i, 4.));
    //上
    if (cv::Point(x_i, y_i - 1).inside(feasible_rect))
    {
        id = coor_mapto_index[CoorInRect(x_i, y_i - 1, feasible_rect)];
        if (id >= 0)
            coefs.push_back(Eigen::Triplet<double>(i, id, -1.));
    }
    //下
    if (cv::Point(x_i, y_i + 1).inside(feasible_rect))
    {
        id = coor_mapto_index[CoorInRect(x_i, y_i + 1, feasible_rect)];
        if (id >= 0)
            coefs.push_back(Eigen::Triplet<double>(i, id, -1.));
    }
    //左
    if (cv::Point(x_i - 1, y_i).inside(feasible_rect))
    {
        id = coor_mapto_index[CoorInRect(x_i - 1, y_i, feasible_rect)];
        if (id >= 0)
            coefs.push_back(Eigen::Triplet<double>(i, id, -1.));
    }
    //右
    if (cv::Point(x_i + 1, y_i).inside(feasible_rect))
    {
        id = coor_mapto_index[CoorInRect(x_i + 1, y_i, feasible_rect)];
        if (id >= 0)
            coefs.push_back(Eigen::Triplet<double>(i, id, -1.));
    }
}
```

构造 A 的过程中，需要检查每个内点的上下左右是否为内点(由于最后的乘数法求导是针对每个内点)，如果有，就将这对相邻点所造成的系数变化填入矩阵 A。值得注意的是，此处矩阵对角元是 4，但事实上，在四边都有相邻的情况下，直接求导的对角元应是 8，这意味着构造时已经对矩阵整体除以了 2，这也是为什么 push_back 中(i, id)位置被填入了-1，而不是与讲义一维例子一样的-2。

coor_mapto_index 与 index_mapto_coor 两个函数是实现转化的核心。通过给每个内点一个编号并给边界编号-1，二维分布中所有的内点都被展平为一个行向量，而将任何两个内点配对可以成为矩阵的元素。展平二维的分布即是 coor_mapto_index，而将一维的行向量还原到图形中的位置则是通过 intex_mapto_coor。这个做法有两个好处：首先，遍历只需要针对行向量整体进行，而不需要在每排考虑边界条件；其次，只要通过编号，这个方法可以对任何的高维操作(一维自然也可以)，不会受到维度的限制。

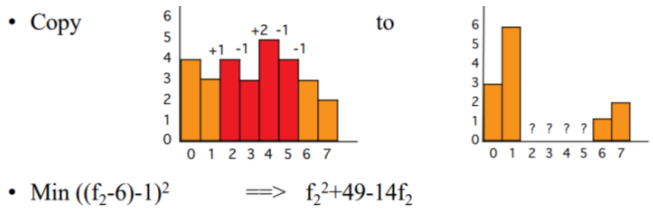
接着是构造 b 的过程：

```
b.resize(inner_points_size, targetimg.channels());
b = laplace_map;

cv::Point ptemp;
// ToDo: 请生成b，提示可能用到的函数CoorInRect(),AddrMatAt()
for (int i = 0; i < inner_points_size; i++)      //所有点
{
    cv::Point pnow = id_to_coor[i];
    for (int k = 0; k < 4; k++)                  //四个方向
    {
        ptemp = pnow + direction[k];
        if (ptemp.inside(frect))
        {
            int id = coor_to_id[CoorInRect(ptemp, frect)];
            if (id == -1) {
                for (int chn = 0; chn < targetimg.channels(); chn++)
                    b(i, chn) += *AddrMatAt(targetimg, ptemp + shift_pos, chn);
            }
        }
        else {
            for (int chn = 0; chn < targetimg.channels(); chn++)
                b(i, chn) += *AddrMatAt(targetimg, ptemp + shift_pos, chn);
        }
    }
}
```

值得注意的是，b 实质上是由两部分组成，一部分是被复制的图片中内点自身之间的差别(也就是离散的梯度)，而这部分已经由 b = laplace_map 所保证了，我们需要完成的是第二部分，也就是边界所引起的变化。

这部分的变化，需要找到处在边界的点(ptemp 存储)，然后将这个点对应的颜色值加到原来的 b 上。此处之所以是简单的直接相加，是由于类似 ppt 一维情况中的$(f_2-6)^2$里的$-12f_2$系数，在求导完移到右侧后除以 2 的结果，恰好是直接相加(剩下的$-2f_2$是来自离散梯度，不需要处理)。



• Min $((f_2-6)-1)^2$ ==> $f_2^2+49-14f_2$

下一步则是确认边界。代码中，先用一个矩形 frect 框定了内点的范围，又把在其中的边界点的“线性坐标”设置为了-1。因此，我开始的思路也是与类似构造 A 时，利用 frect 与 id == -1 判定边界。但是，这样操作后，融合图片中总会有黑色的部分，就像那边的边界被识别成了纯黑像素点一样。

在找同学讨论后发现，此处存在一个隐蔽的问题：frect 框定的是内点的范围，而边界完全可能落在 frect 的外侧，这种情况下，我的算法就会落下这部分边界。容易发现，如果某个内点的相邻点落在外侧，它一定是边界点，因此这里添加一个 else 即可解决问题。

最后是求解方程：

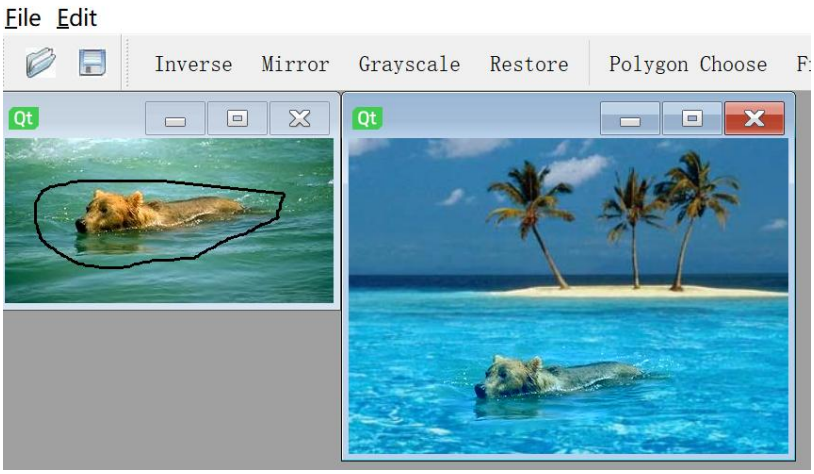
```
X = solver.compute(coef_mat_a).solve(vec_b);
```

```
for (int chn = 0; chn < target.channels(); chn++)
{
    if (X(i, chn) > 255.)
        colorv = 255;
    else if (X(i, chn) < 0.)
        colorv = 0;
    else
        colorv = X(i, chn); //ToDo: 此处应该结合X怎么赋值？很简单，仔细读这个循环和判断是为了什么？
    (*AddrMatAt(result, index_mapto_coor[i] + pos, chn)) = colorv;
}
```

这两步都较为简单。由于类中已经定义好了 solver，此处查询 Eigen 库 solver 的使用方式后发现，可以直接求解(并且，从这里可以学习到 AddrMatAt 的使用方式：它代表了原图中像素点的获得，由此可以应用在计算 b 的过程中)。

在之后，通过改变类中 solver 的定义，可对比不同求解方法。

3、结果展示



此为对示例图片的效果。

代码中自带了计时，但由于 Qt 主窗口运行时无法看到 stdout，此处经过与某热心同学的交流，将输出重定向到了一个文件：

```
int main(int argc, char* argv[])
{
    (void)freopen("when.log", "w", stdout);
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

框架自带的 SimplicialLLT 直接平方根法的速度如下：

```
Time consumed : 501ms
Time consumed : 272ms
Time consumed : 269ms
Time consumed : 270ms
Time consumed : 278ms
Time consumed : 271ms
Time consumed : 269ms
Time consumed : 270ms
Time consumed : 272ms
Time consumed : 267ms
Time consumed : 271ms
Time consumed : 278ms
Time consumed : 272ms
Time consumed : 269ms
Time consumed : 278ms
Time consumed : 268ms
```

除去首次加载，基本在 270ms 左右。

对类似的选区采用 LDLT 法，速度并未产生明显差别(部分差别可能是选区形状细节所致)：

```
Time consumed : 477ms
Time consumed : 267ms
Time consumed : 263ms
Time consumed : 261ms
Time consumed : 261ms
Time consumed : 263ms
Time consumed : 262ms
Time consumed : 262ms
Time consumed : 262ms
Time consumed : 261ms
Time consumed : 275ms
Time consumed : 263ms
Time consumed : 262ms
Time consumed : 262ms
Time consumed : 259ms
Time consumed : 265ms
Time consumed : 258ms
Time consumed : 268ms
```

再尝试稀疏矩阵的 LU 分解(提供的 PoissonFusion.h 中没有 include 对应的头文件，加上 Eigen/SparseLU 即可，而 PoissonFusion.cpp 也需要把一行分解为两行才能适配 LU 分解)：

```
solver.compute(coef_mat_a);
X = solver.solve(vec_b);
```

```
Time consumed : 6683ms
Time consumed : 3501ms
Time consumed : 3496ms
```

可以发现，LU 分解由于未运用正定对称特性，速度比起 LLT 与 LDLT 慢很多。最后尝试 QR 分解，此时需要选取重排方式，我采用了列主元选取：

```
Eigen::SparseQR<Eigen::SparseMatrix<double>, Eigen::COLAMDOrdering<int>> solver;
```

效果是，在选取之前测试的区域时，程序直接卡死，等待 20 秒以上也没有出结果。只有选取很小的区域才能计算出具体结果。

这说明，本题的情况中，稀疏矩阵进行 QR 分解的效率是很低的，因此，本题情况下速度排序应为 LDLT=LLT>LU>QR。

自选图片的泊松融合效果如下：

零范数光滑：

1、基本原理

首先考虑一维情况：

一维情况的基本要求是，在限定 f 的变化次数(相关离散情况下梯度的零范数)时，寻找与目标 g 的最小二乘结果最优的 f。

由于限定变化次数寻找最优的计算过程复杂，将其转化成最优化问题，给变化次数乘一个惩罚项，加入最小二乘结果中，寻找最终的最优结果。这样，惩罚项的系数λ越大，最后结果中的变化次数就会越小。

在二维情况下，这样的变化次数用与相邻像素产生差别的次数来区分，仍然通过乘法项转化为最优化问题，接下来即需要对这个问题进行求解。

在现实操作中，采用的求解方式是傅里叶迭代得到近似数值解(由于像素是 uint8，过于精确的解也不存在意义)，此即提供的算法的主要部分。

2、算法实现

代码输入 lambda 与 kappa 两个参数，并且提供了默认值 0.02 与 2.0。

在准备部分，代码利用傅里叶变换将(1,-1)的行/列向量扩散到了整个图像的大小。

```
OTF = psf2otf(PSF) computes the fast Fourier transform (FFT) of the point-spread function (PSF) array and creates the optical
transfer function array, OTF, that is not influenced by the PSF off-centering.
```

(此处的扩散可以简单理解为，在对应位置放两个光源后，整个图像大小的场所收获的光强，之后利用 Denormin 进行操作时，事实上是在用 2 维方阵作为卷积的核)。

在之后的迭代中，实际上是交替对 h-v 与 S 进行最小化。h-v 子问题中，辅助变量 h、v 代表两方向的梯度，而 t 则代表其中足够小的部分。将这些部分化为 0 后，进入 S 子问题，根据新的梯度得到整体的核，再以此构造出新的 S。

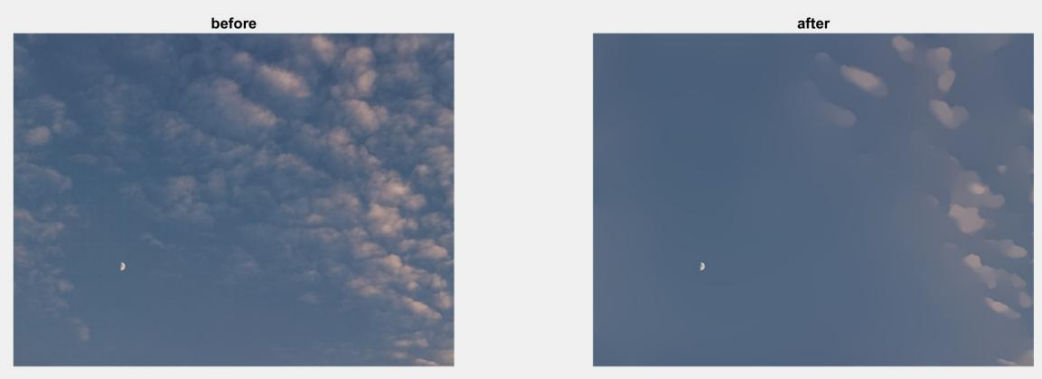
通过足够多地迭代，可以达到结果渐进最优。kappa 代表迭代次数的控制，而 betamax 与 lambda 则控制了最优的情况。

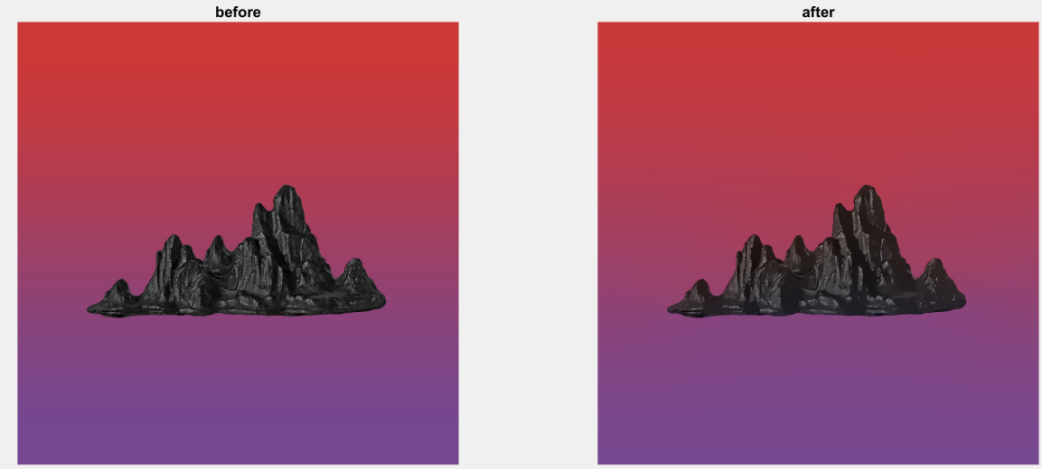
参考：

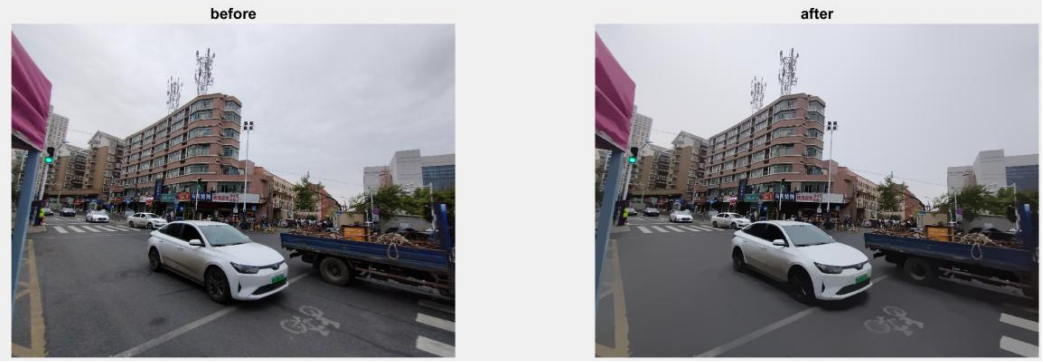
https://blog.51cto.com/u_15458423/4936928

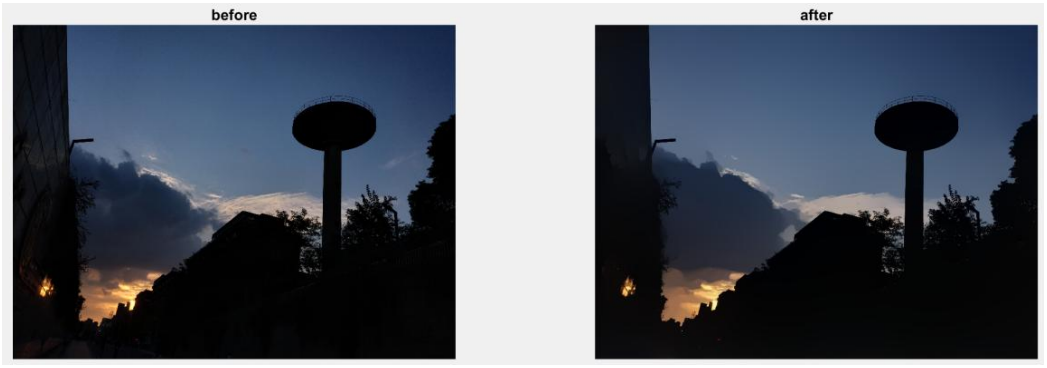
https://blog.csdn.net/qq_38347905/article/details/77646353

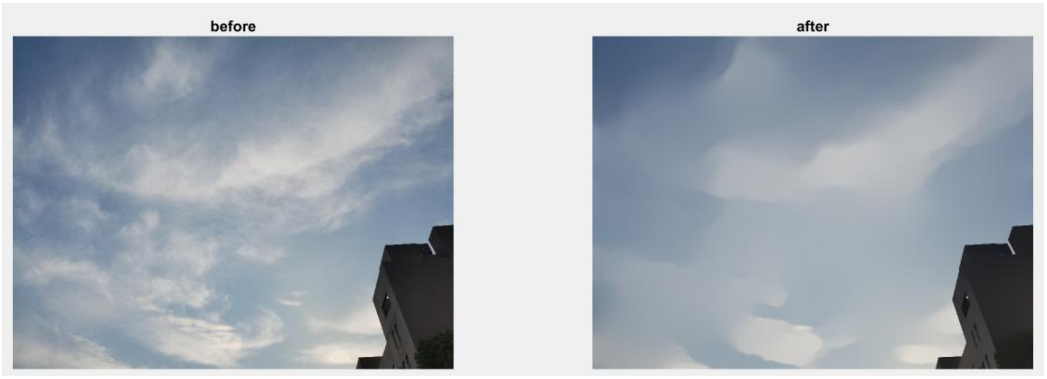
3、不同图片测试(此处均使用默认参数)







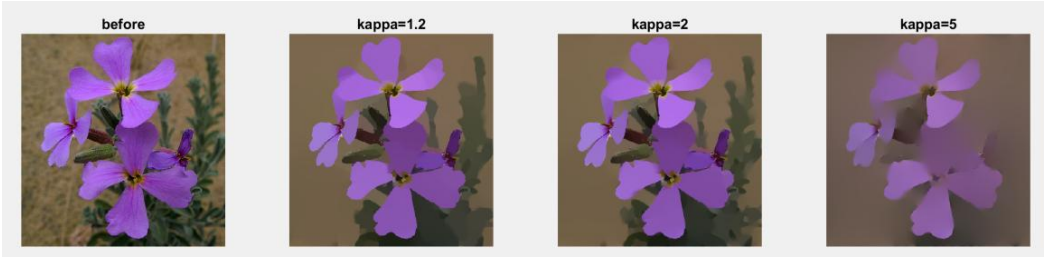




4、修改参数(此处均使用默认图片)

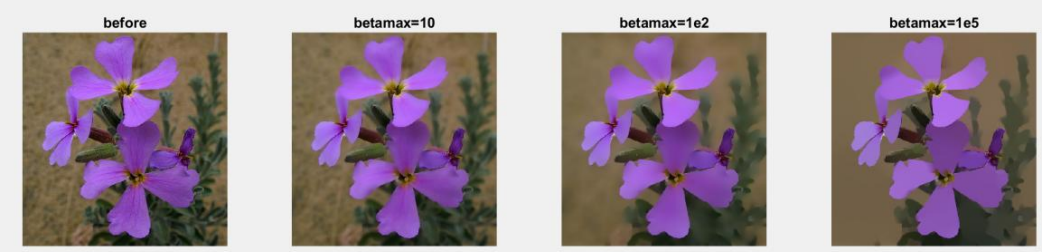
lambda 控制了限定的范围，当 lambda 越大时，lambda/beta 的值即越大，也就会有更多的梯度被控制成 0，这意味着，最终结果看起来会更加“光滑”:

在 lambda 取定 0.05 时，kappa 的减小会增加迭代的次数，让结果变得更加精细:



由此可知，当 lambda 更大时，为了让结果更加精确，需要搭配更大的 kappa。

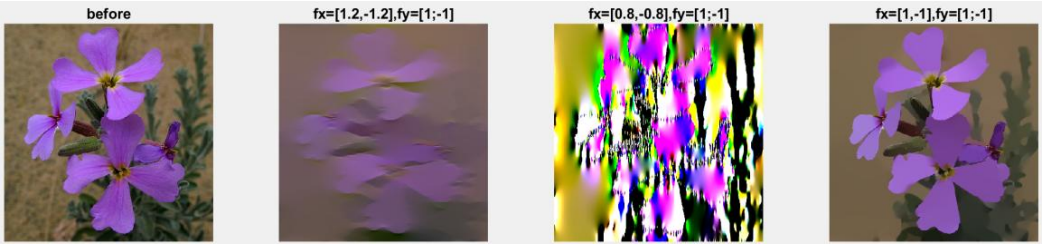
betamax 控制了迭代的上界。增加 betamax 与减小 kappa 会有着类似的效果。不过，由于 lambda/beta 项的存在，在 betamax 很大时效果并不明显，而减小时会有明显差别:



总体来说，`betamax` 越大，图像迭代得越充分。不过，这个充分性与改变 `kappa` 时是“相反”的：改变 `kappa` 时的不充分是由于接近纯色，改变 `betamax` 时的不充分是由于接近原图。

改变初始的核 `fx` 与 `fy` 则更为有趣：

两者同步变化时，当模长越大，图像越接近纯色，而越小时反差则会越明显。而一旦两者不再同步，由于 `xy` 两方向分别处理，结果中能明显看出区分：



总结：

本次实验最难任务：配环境。

本次实验第二难任务：看懂 `L0Smoothing` 代码(尤其是没有接触过 `fft` 在图像处理里的应用时)。

本次实验第三难任务：看懂代码框架。

(于是就拖到了 DDL 最后一天，悲)

Lab 03 实验报告

PB20000296 郑滕飞

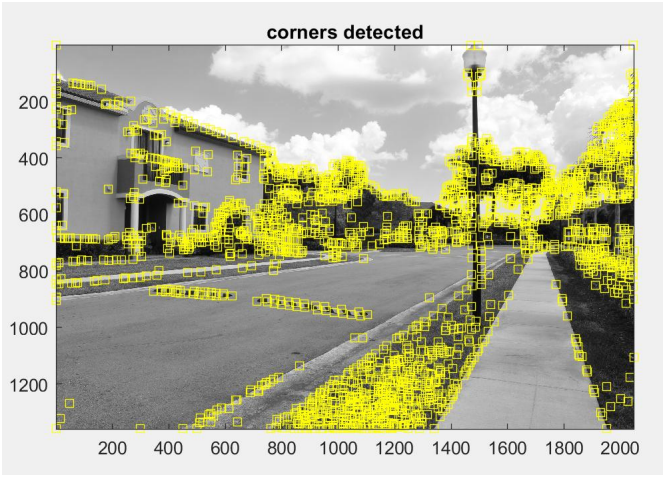
图像拼接：

1、特征点检测

特征点检测方法已经包含在了 MATLAB 框架中，不需要自己完成。其中，第一个函数 harris 用来检测图像的拐角，例如自己用于合成的图书馆图片的检测效果如下：

（由于西区图书馆的墙面有下图所示的花纹，检测到的拐角数量极多）

而示例图片（来自 C++框架文件夹）之一效果如下：



检测拐角的实现方法类似之前所作的边缘检测，在光滑化后确定拐角的位置。

下一个函数是 find_sift，也就是用 sift 方法检测特征点。harris 结束后，取出的是所有拐角的位置，而代表它们具体特征的值则是由每个拐角对应的一个 128 维向量给出。

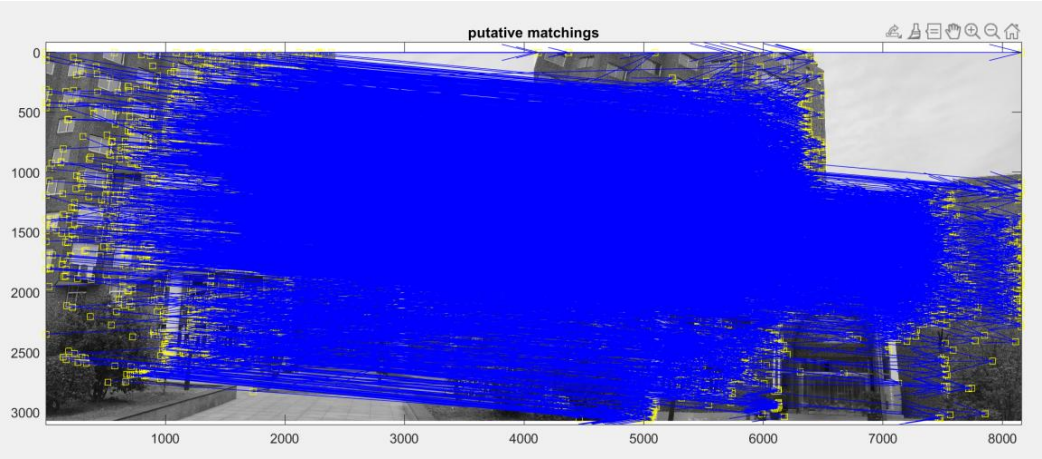
2、推定选择

在讲义的过程中，需要进行特征点的匹配，利用 RANSAC 方法确定最终的匹配结果。由此，对任意两张图片，需要先用 distM 得到它们特征点之间的匹配程度，再进一步完成对应：

```
% 4. Putative matching
% hori
for i = 1:m
    tmp = distM(i,:);
    [first_min,ind1] = min(tmp);
    [second_min,~] = min(tmp(tmp~=min(tmp)));
    if(first_min<second_min*0.8) %invalid matching
        distM(i,:) = 10000;
        distM(i,ind1) = first_min;
    else
        distM(i,:) = 10000;
    end
end

% verti
for i = 1:n
    tmp = distM(:,i);
    [first_min,ind1] = min(tmp);
    [second_min,~] = min(tmp(tmp~=min(tmp)));
    if(first_min<second_min*0.8) %invalid matching
        distM(:,i) = 10000;
        distM(ind1,i) = first_min;
    else
        distM(:,i) = 10000;
    end
end
```

将两组特征点之间的距离(由于矩阵的每列都是 128 个元素，这里的距离就代表不同对特征点的匹配程度)保存成矩阵后，代码进行了行列处理，找到了行列显著比其他距离短的(也就是匹配程度显著高的)特征点对。这里代码的补充只需要仿照水平情况，注意交换行列顺序即可。匹配完成后，通过 cord1，cord2 可知每对特征点之间的坐标对应。

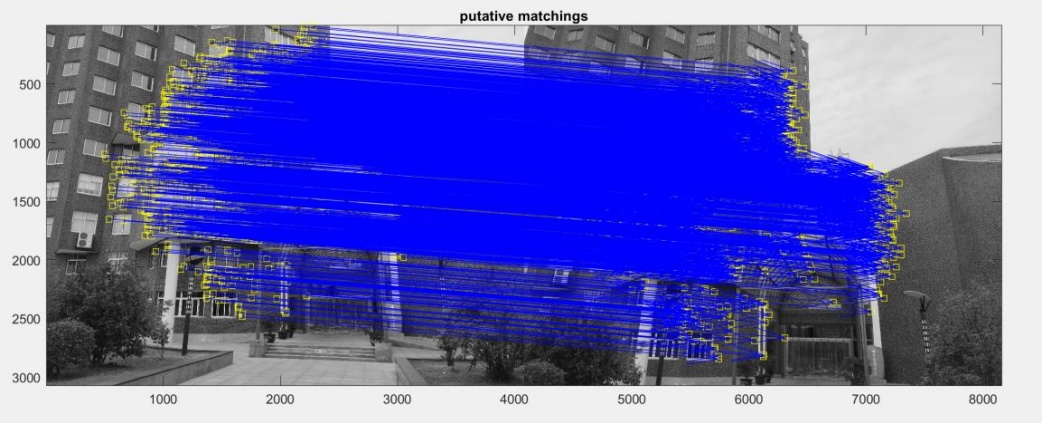


(此为读入的两张图书馆图片的匹配结果)

3、RANSAC

此部分亦不需要自行实现。

通过讲义介绍的 RANSAC 方法，可以将上方特征点中不那么“特征”的部分舍去，得到最终的匹配结果，如读入的两张图书馆图片最终匹配结果如下：



值得注意的是，此时 get_transform 里的返回值 T 已经代表了投影需要的矩阵 T，存在了 MATT{i,j}.T 中。

4、中心图片与投影矩阵

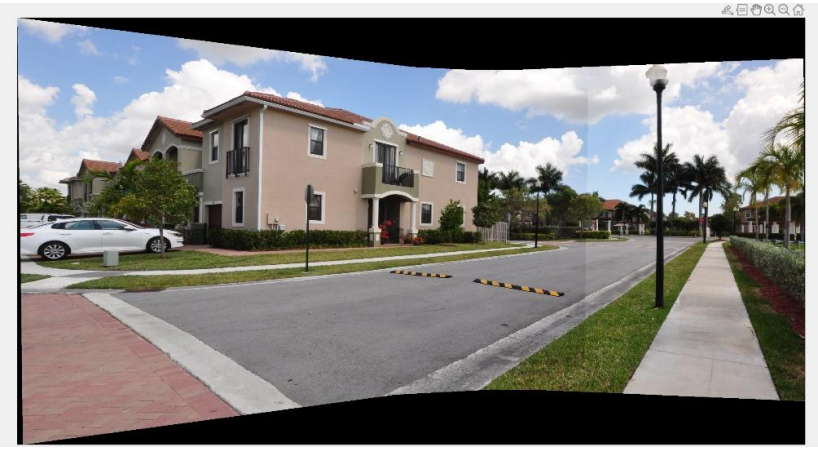
这部分的开始，先将无法充分匹配的图片从图片序列中删去了，再计算出剩下图片的相关情况，选取相关最多的图片作为“中心图片”(也就是主视角)，其他图片往主视角投影。需要自己完成的部分代码如下：

```
function [ T_out ] = get_T_for_all(Ts, row, rlt, Matt)
%%TODO: compute transform matrix (with respect to the central image we just found) for every image
[num, ~] = size(rlt);
for i = 1:num
    if (i ~= row)
        Ts{i} = Matt{i,row}.T;
    end
end
T_out = Ts;
end
```

事实上，由于选定了中心图片，这里并不需要考虑关系树的具体情况，只要将其他图片投影到中心图片的矩阵取出来即可。而对于中心图片自己，进入函数前已将对应的部分置为恒等变换，直接赋值 T_out 即可。

5、图片融合

在完成上面的步骤后，事实上已经可以看到结果，例如对示例图片的结果是：



可以看到非常明显的融合痕迹与不对齐的部分。为了解决这一问题，需要按权重融合。值得一提的是，由于拍摄的每个图片尺寸一致，事实上其权重矩阵是一样的，而在 warping 后会出现不同的位置分配(事实上，一开始给每张图片都配了权重矩阵导致计算很慢，后来意识到这件事，成功加速了运算)：

```
[m,n,b]=size(im{i});
weight = zeros(m,n,b);
for j = 1:m
    for k = 1:n
        weight(j,k,:) = min([j-1,k-1,m-j,n-k]);
    end
end
weight_all = double(imwarp(weight,affine2d(eye(3)),'OutputView', panoramaView));

for i=1:length(im)
    if(i==cent_num)
        continue;
    end
    result{i} = imwarp(im{i},projective2d(Ts{i}),'OutputView', panoramaView);

    weight_now = double(imwarp(weight,projective2d(Ts{i}),'OutputView', panoramaView));

    overlap = (panorama > 0.0) & (result{i} > 0.0);
%     error = sum((panorama(overlap)-result{i}(overlap)).^2)/length(overlap(:))
%     if(error>20)
%         continue;
%     end

    result_avg = uint8((double(panorama).*weight_all ...
        + double(result{i}).*weight_now) ...
        ./(weight_all+weight_now));
    weight_all = weight_all + weight_now;

    panorama = panorama + result{i};
    panorama(overlap) = result_avg(overlap);
end
```

首先，设定按原来像素比例的权重矩阵，具体的权重为到边界距离的最小值。接着，用 weight_all 表示已经叠加完成的权重部分，起初为与中心图片一同 warping 的结果。

在之后的计算中，一张张图片在重叠的 overlap 部分进行加权平均，weight_all 也每次增加。值得注意的是，若某部分在主视角、图片 1、图片 2 中均涉及，加权的结果将会是(w 为 weight，r 为 result，c 为 cent_num，乘除均表示逐点)

$$\frac{\frac{r_c w_c + r_1 w_1}{w_c + w_1}(w_c + w_1) + r_2 w_2}{(w_c + w_1) + w_2} = \frac{\mathrm{r_c w_c} + \mathrm{r_1 w_1} + r_2 w_2}{w_c + w_1 + w_2}$$

正好为三者加权的结果，因此 weight_all 每次直接+=weight_now 是合理的。

6、效果展示

共拍摄了七张关于西区图书馆的照片：

将此七张图融合后的最终效果为:

总结:

有框架的情况下写代码，最关键的是读懂框架的大致思路与步骤，就这次实验来说，还是比预想中顺利一些。值得一提的是，当涉及的处理量较大时，优化的重要性就会凸显。例如，经过调整的 `blending` 比最开始写的时候快了几倍不止，让整体时间得以减小了十多秒。

Lab 04 实验报告

PB20000296 郑滕飞

代码实现：

1、estimate F

内参矩阵直接按照讲义定义即可：

```
% ToDo:坐标变换中的 K=diag(f,f,1)
frames.K = diag([frames.focal_length, frames.focal_length, 1]);
```

对于 esitamteF 函数里的内容，仿照 estimateE，由于 matches 的四行分别存储的是前图与后图里的坐标，将其拆分开即可完成匹配：

```
% ToDo：只需看懂ransacfitfundmatrix()函数，在这里直接调用 并且一行代码即可完成，其实最下面已经给你注释了，所以你甚至可以直接写出来
[F, inliers] = ransacfitfundmatrix(pair.matches(1:2,:), pair.matches(3:4,:),t);
```

由于这里并没有获取真正的内参矩阵，得不到真实的K_r与K_l，直接用之前得到的 K 代替。此外，由于 K 已经是对角阵，也没有必要进行转置，直接乘法即可：

```
pair.E = frames.K*pair.F*frames.K;
```

2、Rt from E

```
% ToDo：triangulation 相当于我们将第一个矩阵[R1|t1]=[I|0],则P{1}=K[I|0] (matlab的话 一行代码可能就行)
P{1} = frames.K*[eye(3) zeros(3,1)];

goodCnt = zeros(1,4);
for i=1:4 % ToDo:计算四种可能，你需要读懂vgg_X_from_xP_nonlin()函数（在相应的.m文件下），PS:如果你够仔细和机智，你会发现有个地
    clear X;
    % first:先得到的是相对1的K[R2|t2]
    P{2} = frames.K*Rt(:,:,i);
    % second:应用vgg_X_from_xP_nonlin()计算
    for j=1:size(pair.matches,2)
        X(:,j) = vgg_X_from_xP_nonlin(reshape(pair.matches(1:4,j),2,2),P,repmat([frames.imsize(2);frames.imsize(1)],1,2));
    end
    X = X(1:3,:) ./ X([4 4 4],:);
    % third:选择（这步我已经给出了，不需要你们写，你们可以根据这句话思考前面second和下一步怎么写）
    dprd = Rt(3,1:3,i) * ((X(:,:) - repmat(Rt(1:3,4,i),1,size(X,2))));%最后选择的规则即为满足(X-C)*R(3,:)是否大于0
    % forth:计算了满足z>0以及(X-C)*R(3,:)>0的个数
    goodCnt(i) = sum(X(3,:) > 0 & dprd > 0);

end
```

这个函数中，其他的部分按注释提示操作即可，较为简单。第四步的 z 坐标事实上即为 X 的第三行，因此可以直接如此书写。而第二步需要利用 vgg_X_from_xP_nonlin，由于 SFMedu 的 151-155 行有完全类似的操作，且作用均为计算匹配的 X 结果，此处将那时匹配的 densematch 改为这时匹配的 matches(稀疏匹配)即可。

3、dense reconstruction

```
% ToDo:仿照给的1完成2，利用非线性拟合得到空间坐标，试着思考为什么这样计算？？？
X = X(1:3,:) ./ X([4 4 4],:);
x1= P{1} * [X; ones(1,size(X,2))];
x2= P{2} * [X; ones(1,size(X,2))];
x1 = x1(1:2,:) ./ x1([3 3],:);
x2 = x2(1:2,:) ./ x2([3 3],:);
Graph{frame}.denseX = X;
% ToDo:给出稠密重建的error，所以你只需要读懂denseMatch()函数，直接在这里补充好sum()函数里
Graph{frame}.denseRepError = sum((Graph{frame}.denseMatch(1:4,:)-[x1;x2]).^2,1);

Rt1 = mergedGraph.Mot(:, :, frame);
Rt2 = mergedGraph.Mot(:, :, frame+1);
% ToDo：计算得到相机中心，分别用上面的Rt1和Rt2；
C1 = -Rt1(1:3,1:3)'*Rt1(:,4);
C2 = -Rt2(1:3,1:3)'*Rt2(:,4);
% ToDo：仿照给的view_dirs_1，补充2，并读懂这里是如何计算两个向量的！！！！！
view_dirs_1 = bsxfun(@minus, X, C1);
view_dirs_2 = bsxfun(@minus, X, C2);
view_dirs_1 = bsxfun(@times, view_dirs_1, 1 ./ sqrt(sum(view_dirs_1 .* view_dirs_1)));
view_dirs_2 = bsxfun(@times, view_dirs_2, 1 ./ sqrt(sum(view_dirs_2 .* view_dirs_2)));
% ToDo：两个向量求出相机的拍摄角度cos值
Graph{frame}.cos_angles = dot(view_dirs_1,view_dirs_2);
```

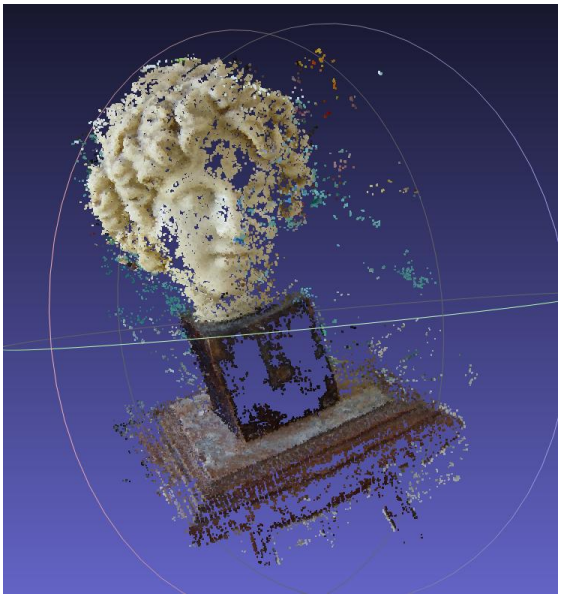
这一部分中，大部分都是仿照已经给出的 1 完成 2，需要自己涉及的是计算误差、计算相机中心与夹角。计算误差部分，观察可发现 densematch 中给出了匹配的坐标值，这里即是计算匹配到的与 x1，x2 中得到的之间的误差。由于后文直接以 0.05 作为限制，这里可以采取不同的方案计算，在后文展示中会对比结果。

相机中心的位置，由于考虑相对坐标，由$Rc + t = 0$可知$c = -R^T t$，从而可直接计算。

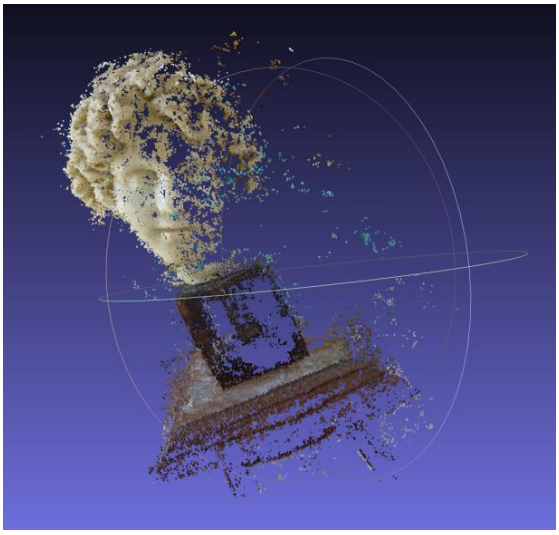
而计算夹角的 cos 值，由于在上一步已经把 X 减去中心的向量归一化，这里直接点乘即可得到夹角。

结果展示：

采用默认的直接计算 E 方法结果如下：

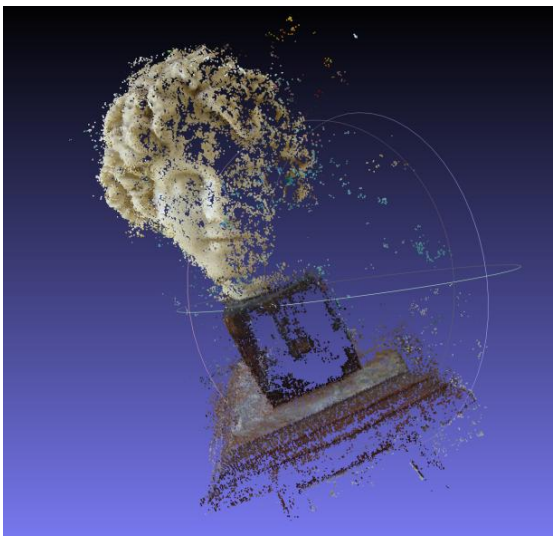


而采用先算 F 后算 E 的结果如下：



可以发现，虽然下方显示 vertices 的个数先算 F 的方法要远大于直接算 E 的方法，但就结果而言，两者不存在太明显的差别。

由于面部的重建似乎缺失了一块，我试着将误差部分除以 `1.5`，以获取更多点，结果如下：



根据顶点数，这时的结果已经和将误差直接设为 `0` 的结果相同，也即所有的点都被选中，说明这已经是匹配的上限，除非在之前降低匹配的精度。

总结：

主要难点在于读框架，`densematch` 那边的误差计算是通过加断点观察那时候哪些变量很接近但还存在差距来找到的误差表达式，而 `vgg` 函数在没看到后方之前也确实不知道怎么调用。

[所幸这些东西在代码的其他部分有提示，所以还是能大概知道应该写啥的，可喜可贺`.jpg`]

Lab 05 实验报告

PB20000296 郑滕飞

算法原理：

1、整体思路

论文中采取的方式是利用 2D 的相机变换进行视频稳像。不过，比起考虑全局的相机移动或是类似光流方法逐像素考虑，论文选择将图片分成一定大小的网格，并对每个网格进行考虑，再综合结果。并且，论文的相机轨迹是只依赖相邻帧进行计算的，避免了全程追踪特征的移动时受到动态模糊等情况的影响。

于是，此相机变换方法进行视频稳像分为四步：

a. 根据视频前后两帧的特征对应得到两帧间每个网格对应的相机变换。

b. 将所有帧之间的相机变换综合，得到一束相机轨迹。

c. 对这束相机轨迹进行光滑化。

d. 综合光滑的结果，得到稳像后的帧。

2、求解两帧间的变换

[论文中给出的代码 demo 即为此部分]

按照论文的方式，这个变换是针对每帧的每个网格求解的。但是，将图片划分为网格后，对应网格未必有良好的特征对应，而求解出每个网格的变换也并不代表整体变换后形状和物理。因此，优化时实际需要有两项：控制变换前后相似度的数据项与控制变换变形程度的正则化项。

根据论文中的方法，通过求解顶点来得到变换。假设对第 t 帧第 i 个网格，其四个顶点在变换后组成的矩阵为$\hat{\mathbf{V}}$，变换前为V，网格中每个匹配特征位置为p与$\hat{p}$，由于p可以写成四个顶点以某个权重求和(此处权重的写法不止一种，由于方形网格，事实上应为对行和列分别取权重组合，即若原来顶点为(a,b)、(a,c)、(d,b)、(d,c)，则将(x,y)先写为(ta+sd,pb+qc)，其中 t+s=p+q=1，再由此得到权重分别为 tp、tq、sp、sq)，我们希望变换后$\hat{p}$相对顶点的权重位置不变，于是数据项如下，其中的w_p类似上方所述。

$$E_d(\hat{V}) = \sum_p ||\hat{V}_p w_p - \hat{p}||^2.$$

另一方面，保持形状不变的正则化项如下：

$$E_s(\hat{V}) = \sum_{\hat{v}} \|\hat{v} - \hat{v}_1 - sR_{90}(\hat{v}_0 - \hat{v}_1)\|^2.$$

其中 s 是原本两边长度的比值，R_{90}则是代表旋转 90 度的矩阵。这项的几何意义是，若变换后相邻三顶点仍为直角三角形，且边长比例与之前一致，则代表形状保持较好，差异越大则误差越大。加入这项后，网格在变换后的形状能更加接近原本的长方形。

将两项乘以系数后相加，作为需要最小化的能量。由于这是一个二次优化问题，有很多求解方法，而在变量只有 8 个的情况下，容易直接得到精确解。根据精确解$\hat{\mathbf{V}}$，单应变换矩阵可从$\hat{V}_i = F_i(t)V_i$解出。

关于求解变换，论文所附代码 homography_4pts.m 中给出了过程。这里的$F_i(t)$并不是一个简单相乘的 2 阶方阵，而是一个平面射影变换矩阵(即 3 阶方阵商掉比例等价类，代码中的做法是让 H(3,3)归一化)，其商去等价类后的 8 个自由度与两边的各 8 个分量是一致的，因此可以解出唯一结果。

(具体的点变换过程是，假设原本的点写为列向量是v，记$F_i(t)\begin{pmatrix}v\\1\end{pmatrix}=\begin{pmatrix}x_1\\x_2\\x_3\end{pmatrix}$，则变换后的$\tilde{v}=\begin{pmatrix}x_1/x_3\\x_2/x_3\end{pmatrix}$，注意到同乘比例不会影响$F_i(t)$的计算结果，因此可以进行归一化)

3、优化与得到相机轨迹

为了保证相机轨迹的稳定性，求解过程还有三个优化的部分：

首先，选取特征点时可以通过 RANSAC 方法舍弃匹配得并不良好的点，论文的实际操作中，在全局和局部各做了一次，保证更好的匹配效果。

其次，为方便估算，可以先考虑全局的相机移动构成射影变换，先全局作一次变换再考虑每个网格的变换[pre-warping]。

最后，总能量中E_d与E_s的比例系数不需要事先指定，可以在过程中通过自适应方法得到。具体做法是，类似E_d可以计算出$F_i(t)$得到的匹配误差，而通过$F_i(t)$和与之相邻的网格中的$F_j(t)$的差值可以得到光滑性误差。如果匹配误差较大，就让E_d项占更高的比例，否则让E_s项占更高的比例。

在优化后，由于$F_i(t)$是每步的变换，关于 t 作连乘即可得到每次相机的轨迹：

$$C_i(t)=C_i(t-1)F_i(t-1),\Rightarrow C_i(t)=F_i(0)F_i(1)\cdots F_i(t-1)$$

(这里论文中每次右乘与点变换时的左乘是相反的，当时推测是由于相机移动方向与相机里的点的坐标移动是相反，具体见后方代码部分。)

4、路径束光滑化与综合结果

由于采用了网格分割，在每一步中都会产生网格内的项与网格间关联的项。对路径束的光滑化来说，网格内的项是某个 i 对应的路径C_i与优化后的P_i产生的

$$\mathcal{O}(\{P(t)\})=\sum_t\left(\|P(t)-C(t)\|^2+\lambda_t\sum_{r\in\Omega_t}\omega_{t,r}(\mathbf{C})\cdot\|P(t)-P(r)\|^2\right)$$

左侧的项代表与原路径的相似程度，右侧则类似之前实验中零范数光滑所产生的光滑化项，对每处梯度进行最小化(Ω_t表示邻域)。由于这里存在大量变量，直接求解法不再适用，于是需要进行迭代求解，论文中提供了一个基于 Jacobi 迭代的方法。(此外，这里的权重系数可以调整，达到精细化控制，左侧权重越大则得到的路径越接近原路径。)

而网格间的项，类似上一部分的光滑性误差，考察的是相邻路径之间的相似程度，即

$$\sum_t\sum_{j\in N(i)}\|P_i(t)-P_j(t)\|^2$$

其中，$N(i)$即代表邻域。类似上方对单条路径迭代求解的方式，这里也可以利用 Jacobi 迭代法进行计算。

有了优化前的路径$C_i(t)$与优化后的路径$P_i(t)$，最终对每帧每个网格进行变换是利用公式$B_i(t)=C_i^{-1}(t)P_i(t)$。这相当于找到光滑化路径后原位置所应该移到的位置，以匹配光滑化的路径。

根据论文的结果，由于可以处理小幅度的摇摆，这样的光滑化可以附带解决快门逐行扫描带来的果冻效应。

代码实现：

第一个模块为通过 KLT

第二个模块，求解两帧之间的变换，直接引用论文中的代码，原理已在上方详细说明。而对于得到路径，过程如下：

```
W = tracks.videoWidth;
H = tracks.videoHeight;
qW = W / MeshSize;
qH = H / MeshSize;
for frameIndex = 2:nFrames
    fprintf('%5d', frameIndex);
    if mod(frameIndex, 20) == 0
        fprintf('\n');
    end
    %TODO：按照论文中的方法，得到path的值，需要用到NewWarping函数和TrackLib.m中的getF函数

    [f1, f2] = tracks.getF(frameIndex - 1);
    homos = NewWarping(f1, f2, H, W, qH, qW, lambda);
    for row = 1:MeshSize
        for col = 1:MeshSize
            path(frameIndex, row, col, :, :) = reshape(homos(row, col, :, :), [3,3])...
                    * reshape(path(frameIndex-1, row, col, :, :), [3,3]);
        end
    end

end
```

这里的 f1，f2 为匹配的特征点，homos 即得到的是两帧之间所有的$F_i(t)$，而 path 在优化前即为$C_i(t)$，从起初的单位阵开始，每次左乘新得到的矩阵即能得到结果。

(此处按照论文中方法写了右乘与左乘对比，可发现右乘的结果显著非常扭曲，应该是错误的，而左乘得到的结果是正确的，合理怀疑论文此处是笔误，或框架中实现的方式和论文不完全一样。)

第三个模块，按照论文的方式进行迭代优化，之后第四个模块计算并输出图片，已由助教实现，过程同样与论文相同，在上方已经解释。

结果展示：

具体展示见 testcase 文件夹中的视频。其中默认的参数为

`lambda = 2; Meshsize = 8; Smoothness = 1; Rigidity = 2;`

视频名字即为改变的参数与改变后的值。这里对效果进行简单的总结：

根据论文提供代码的注释，lambda 增大后路径会变得更加“僵硬”，而减小时会更加柔和，类似之前实验零范数光滑的效果。在实际对比时，能发现其减小时路径更加接近原路径，增大后则更接近接近直线。

Meshsize 决定划分的细致程度，在对比效果时，划分数量的增大会导致运算速度显著减慢。就效果来说，更大的 Meshsize 会让路径更加精细，但也更容易引起变形。这时，配合更大的 Rigidity 可以减少形变。

Smoothness 参数的效果非常明显，影响光滑化的程度，也直接影响稳像的结果(在测试样例中，光滑程度调至 3 的效果非常好)。同样，当光滑化程度调高的时候，可能需要减少形变。

除了已有的例子之外，我还自己拍摄了一个模拟交通工具上高频抖动的例子，在这个例子的处理中，可以发现由于相机移动速度过快，在整体的稳像过程中，会有一些不稳定的帧。文件夹中的样例将 Smoothness 调成了 3，效果较好。