Lab 0 实验报告

PB20000296 郑滕飞

简易的 Vector STL：

1、基础 SetSize 操作

其他操作都较为简单，直接对应位置操作(包含越界检查)，或是如插入、删除一样，对应的扩大/缩小数组规模与移动结合即可，因此主要需要完成的是修改大小的操作。代码如下：

```
void DArray::SetSize(int nSize) {
    if (nSize < 0 || nSize == m_nSize) return;
    if (nSize <= m_nSize) {
        m_nSize = nSize;
        return;
    }
    double* now = new double[nSize];
    for (int i = 0; i < m_nSize; i++) now[i] = m_pData[i];
    for (int i = m_nSize; i < nSize; i++) now[i] = INFINITY;
    delete m_pData;
    m_pData = now;
    m_nSize = nSize;
}
```

若不合法或长度未变，不需要任何操作，对于缩减长度，只要修改 m_nSize 为缩减后的长度即可。值得注意的是，缩减后部可能后方存在已分配的空间，不过每次删除都是直接清理了 m_pData 所有空间，因此不会产生内存碎片。

当需要扩容时，由于简易向量不存在预先分配的考虑，需要整体重新分配，并将原来部分复制到新分配的开头处，为防止内存溢出，将新分配的内存重置为 INFINITY。

2、增加效率的改进

为增加效率，在新建立向量时将 max 与 size 设置成一样，而关键的操作在扩容时：

```
void DArray::Reserve(int nSize) {
    if (nSize <= m_nMax) return;
    if (m_nMax == 0) m_nMax = 2;
    while (m_nMax < nSize) m_nMax *= 2;
    double* now = new double[m_nMax];
    if (m_nMax < m_nSize) {
        std::cout << "error";
        return;
    }
    for (int i = 0; i < m_nSize; i++) now[i] = m_pData[i];
    for (int i = m_nSize; i < m_nMax; i++) now[i] = INFINITY;
    delete m_pData;
    m_pData = now;
    m_nSize = nSize;
```

```
}

// set the size of the array
void DArray::SetSize(int nSize) {
    if (nSize < 0 || nSize == m_nSize) return;
    if (nSize <= m_nMax) {
        m_nSize = nSize;
        return;
    }
    Reserve(nSize);
}
```

这里，只要需要的长度没有超过已分配的 max，都可以直接分配，否则进入扩建环节。当还未分配空间时，先将 max 设为 2，否则每次翻倍 max，直到达到需要的大小，并对应把多余的部分置为 Infinity。这样一来，已分配长度不减，在频繁操作时可以增加效率。

3、模板的使用

模板在代码层面没有过多可以赘述的，基本都是查找替换可以解决的问题，将原本的 double 替换成一个模板 DataType 类型。而在书写 main 时，模板类就可以发挥功能，例如：

```
    DArray<int> b;
    b.PushBack(21);
    ...
    b.Print();

    DArray<char> c;
    c.PushBack('a');
    ...
    c.Print();
```

这就可以构造不同类型的动态数组。

4、效果展示

前两个动态数组的显示效果是完全一样的，如下(在数组为空时我设定了打印 empty，inf 是 SetSize 后未分配的结果)：

```
2.1
2.1 3 3.1 3.2
3 3.1 3.2
4.1 3 3.1 3.2
4.1 3 3.1 3.2
4.1 3 3.1 3.2
4.1 3 3.1 3.2
4.1 3 3.1 3.2
21
empty
22 inf inf inf inf
100 97 98 99
```

使用模板的效果如下：

```
2.1
2.1 3 3.1 3.2
3 3.1 3.2
4.1 3 3.1 3.2
4.1 3 3.1 3.2
4.1 3 3.1 3.2
4.1 3 3.1 3.2
4.1 3 3.1 3.2
21
empty
22 0 0 0 0
d a b c
```

值得注意的是，在 double 类型下的 INFINITY 会被 int 转换为 0。

一元多项式：

1、链表实现的基本构造

对于链表实现的一元多项式，最关键的是 AddOneTerm 函数，无论是建立、加减、乘法，都可以看成一项项添加的结果，代码如下：

```
PolynomialList::Term& PolynomialList::AddOneTerm(const Term& term) {
    for (auto it = m_Polynomial.begin(); it != m_Polynomial.end(); it++)
{
        if ((*it).deg > term.deg) {
            continue;
        }
        if ((*it).deg == term.deg) {
            (*it).cof += term.cof;
            return (*it);
        }
        m_Polynomial.insert(it, term);
        return (*it);
    }
    m_Polynomial.push_back(term);
    return m_Polynomial.back();
}
```

由于项是次数从大到小排列的，

2、清理与用法

在单纯相加后，可能会出现系数为 0 的项，而清理的目的是消除为 0 的项。具体过程如下：

```
void PolynomialList::compress() {
    bool flag = false;
    do {
```

```
        flag = false;
        for (auto it = m_Polynomial.begin(); it != m_Polynomial.end();
it++) {
            if ((*it).cof == 0) {
                m_Polynomial.erase(it);
                flag = true;
                break;
            }
        }
    } while (flag);
}
```

循环删除，直到不存在 0 项为止。结合增加项与清理，计算就容易实现了，如乘法：

```
PolynomialList PolynomialList::operator*(const PolynomialList& right)
const {
    PolynomialList p;
    for (Term a : m_Polynomial) {
        for (Term b : right.m_Polynomial) {
            Term now(a.deg + b.deg, a.cof * b.cof);
            p.AddOneTerm(now);
        }
    }
    p.compress();
    return p; // you should return a correct value
}
```

3、映射实现的优化

映射实现下，添加项可以自动由 m_Polynomial[deg] += cof 来实现(若不存在 deg 这项会自动创建 0 项后相加)，因此过程可以简化。而清理的过程类似，得到乘法：

```
PolynomialMap PolynomialMap::operator*(const PolynomialMap& right) const
{
    PolynomialMap p;
    for (auto a : m_Polynomial) {
        for (auto b : right.m_Polynomial) {
            p.m_Polynomial[a.first + b.first] += a.second * b.second;
        }
    }
    p.compress();
    return p; // you should return a correct value
}
```

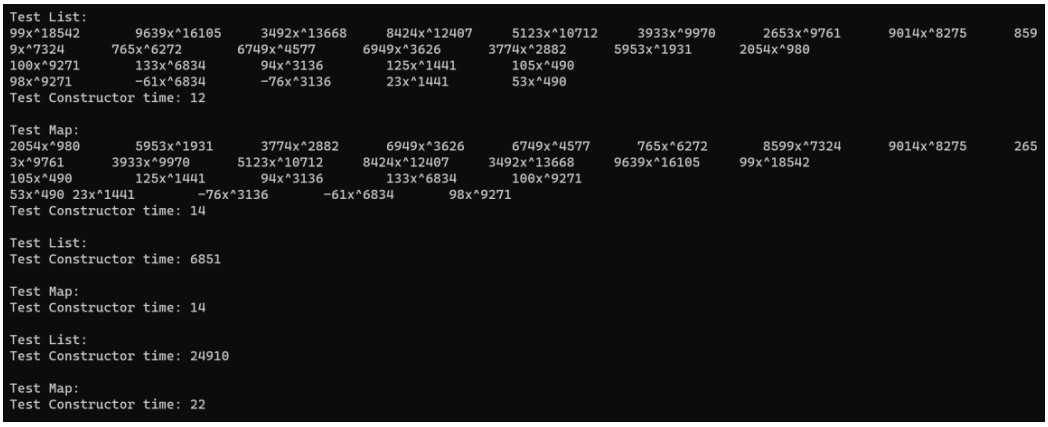
值得注意的是这里的 a、b 都是 pair 模板类，而不是之前手动定义的结构体 Term。

4、效果展示

打印采取了依次打印系数和次数的方式，对链表实现的运算结果为：

```
-3x^4   2x^1
-3x^4   4x^1
-6x^4   6x^1
2x^1
9x^8    -18x^5  8x^2
```

而链表与映射实现的对比为：



可以发现，在项数增加时，映射由于其内部搜索树的构造，可以大幅提升效率。

总结：

总体来说是比较轻松的一次实验[尤其是在已经熟悉 STL 的用法时]，希望未来课程的实验难度不要太恐怖(x

Lab 1 图像颜色变换 实验报告

PB20000296 郑滕飞

摘要

本文对彩色图灰度化与颜色风格迁移两个问题给出了可行的算法。对灰度化问题，文章给出了一个度量优劣的方法，并以此对比了固定比例加权与 PCA 方法的适用情况，并得到了固定比例在一般情况下可以适用的结论；对颜色风格迁移，文章通过直方图匹配算法进行计算，并且加上了光滑化与均衡化参数，使得结果更加符合要求的视觉效果。

代码结构

`draw_pdf.m` 绘制图片三种颜色通道的 PDF

参数：`I`[图片的三维张量]

`lab1.m` 主要功能实现

`show_gray.m` 灰度化效果展示

参数：`name`[文件名]、`if_count`[是否计算灰度化优劣]

`show_match.m` 风格迁移效果展示

参数：`name1`[输入文件名]、`name2`[目标文件名]、`if_draw_pdf`[是否绘制 PDF]

灰度化：

1、问题描述

将三维张量格式存储的图像的 RGB 通道合成为一个灰度通道，希望尽可能多地保持图像信息，同时希望保证一些美观性。

2、分析建模

由于压缩为灰度图需要区分图像，实际上也就是希望直观的颜色差别能反馈到灰度的差别上。出于方便计算，刻画两个颜色的视觉相差直接采取三维向量的距离，而灰度图的差别则是直接以灰度相差的绝对值刻画。由于`(0,0,0)`和`(255,255,255)`的距离应该对应于灰度图中 `0` 与 `255` 的距离，应该将两个距离化归到相同比例后比较，也即：

$$d([R_1,G_1,B_1],[R_2,G_2,B_2],C_1,C_2)=\left|\sqrt{\frac{(R_1-R_2)^2+(G_1-G_2)^2+(B_1-B_2)^2}{3}}-|C_1-C_2|\right|$$

将此作为两个彩色点被化为灰度点后与原来的接近程度(这里$[R_1,G_1,B_1]$化为C_1，$[R_2,G_2,B_2]$化为C_2)。

值得注意的是，这个度量的考虑并没有考虑到明暗，只是以视差比较，这是由于实际情况下灰度化要使相近明暗的不同颜色区分开，很可能无法做到明暗的规律。

对所有点对计算此度量是过于复杂的，也不符合实际的视觉规律，因此，对所有相邻点计算此度量后取平均值，作为最终度量，下面记像素点 `x`，灰度化为 `f(x)`，图像行数 `R` 列数 `C`，则度量为：

$$\frac{1}{R(C-1)+C(R-1)}\sum_{y\ \text{adjoins}\ x}d\big(x,y,f(x),f(y)\big)$$

以此作为图像灰度化好坏的基本刻画。

对灰度化问题，这里对比两种方法：加权平均与 PCA 类方法。加权平均这里考虑直接平

均与辉度平均(RGB 权重为 0.299,0.587,0.114)两种，而这里 PCA 类方法意味着，想要找到一个投影直线，使得原来三维空间中的像素点在直线上区分最大。

不过，直接进行 PCA 后，得到的特征向量(权重)若所有分量都为正或为负，直接进行加权即可(为负可直接都取相反数，需要将所有权重和归为 1 后加权)。但当部分分量为正部分分量为负时(相当于发现的最佳区分投影平面不经过第一卦限)，加权后的结果并不存在一个直接的明暗关系解释。这也就意味着，得到的灰度基本只能作为区分颜色的存在。

这时，有两个可能的处理方式：第一，作这条投影直线在第一卦限的投影，也即化为有一个分量为负(若有两个则取相反数)后直接去掉这个分量，然后进行加权归一化(同样需要所有权重和为 1)。第二，直接将投影到的坐标最小值设为 0，最大值设为 255 后进行线性映射。两种处理方式的对比见结果部分，从语义上来说，第一种方式更能保持原本亮度的语义，而第二种满足 PCA 的原始要求，可以做更大的区分。

3、代码实现

这里主要给出 PCA 部分的代码:

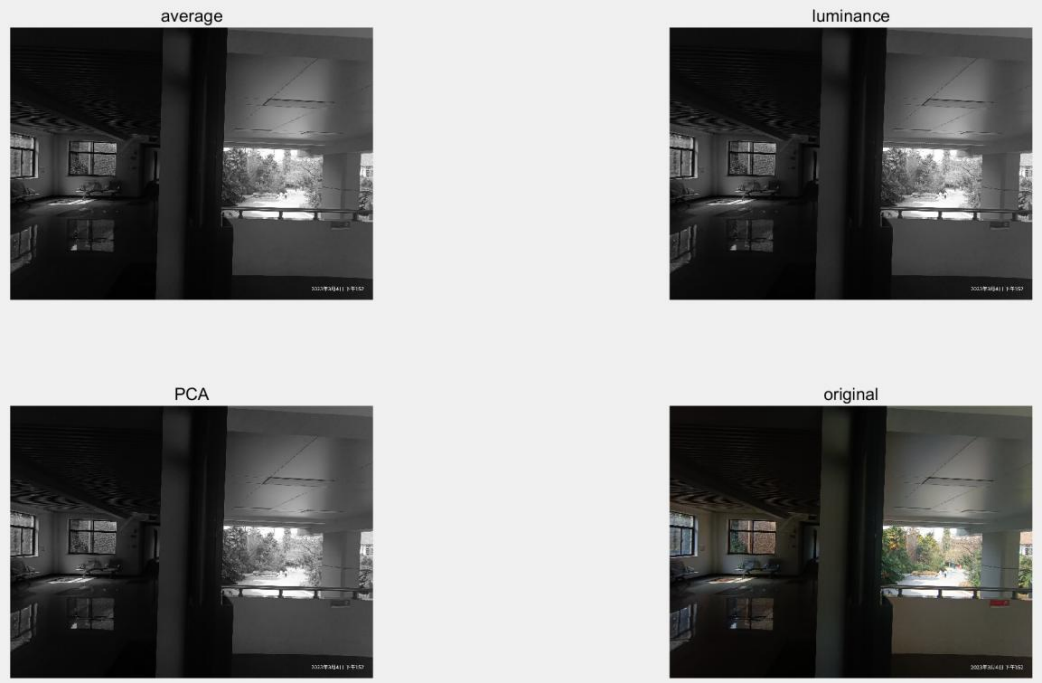
```
function J = gray_pca(I, deal_neg)
    [R, C, B] = size(I);
    T = double(reshape(I, [R*C, B]));
    T_cen = T - mean(T, 1);
    cov = T_cen' * T_cen;
    [V, D] = eig(cov);
    [~, index] = max(diag(D));
    v = V(:, index);
    flag = sum(v < 0);
    if (flag == 0)||(flag == 3)
        v = v / sum(v);
        J = uint8(reshape(T * v, [R,C]));
    else
        if flag == 2
            v = -v;
        end
        if deal_neg == 0
            v = max(v, 0);
            v = v / sum(v);
            J = uint8(reshape(T * v, [R,C]));
        else
            res = T * v;
            res = res - min(res);
            res = res / max(res) * 255;
            J = uint8(reshape(res, [R,C]));
        end
    end
end
```

首先，将图像展平归一化后计算所有像素点对应的协方差矩阵，并得到其最大特征值与所对的特征向量。若特征向量均正或均负，归一化后计算即可，否则，通过 deal_neg 参数

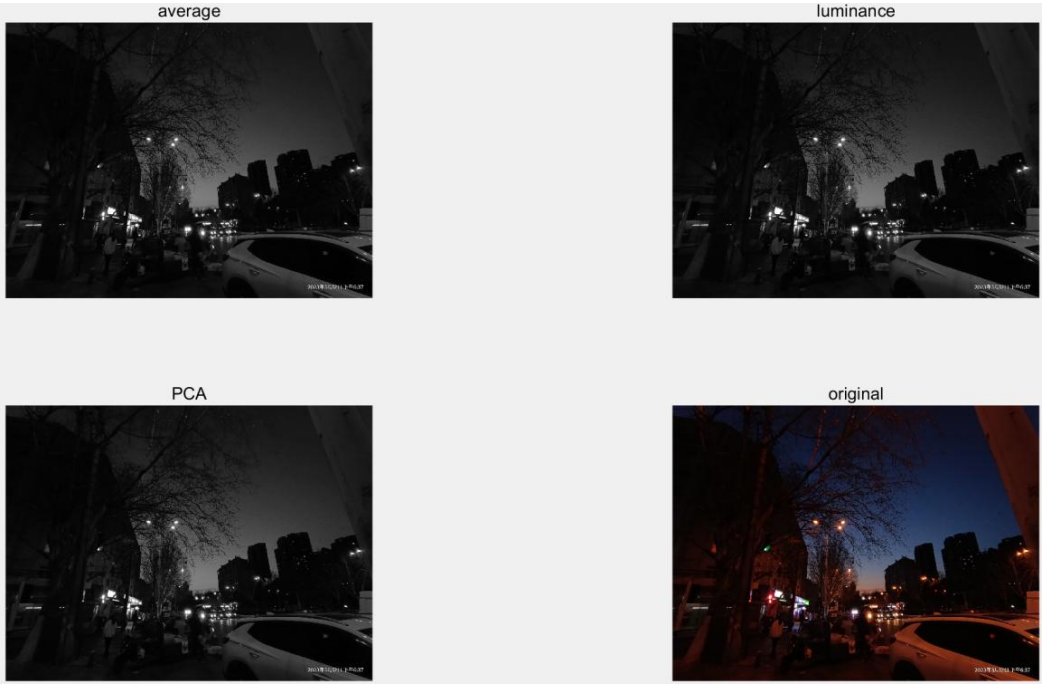
决定直接舍弃一个负分量还是重新映射。

4、结果展示

在实际测试时，对于普通的照片与网络上下载的图片，颜色分布相对均匀，并不会出现需要控制负分量的情况，各方法效果示意如下：

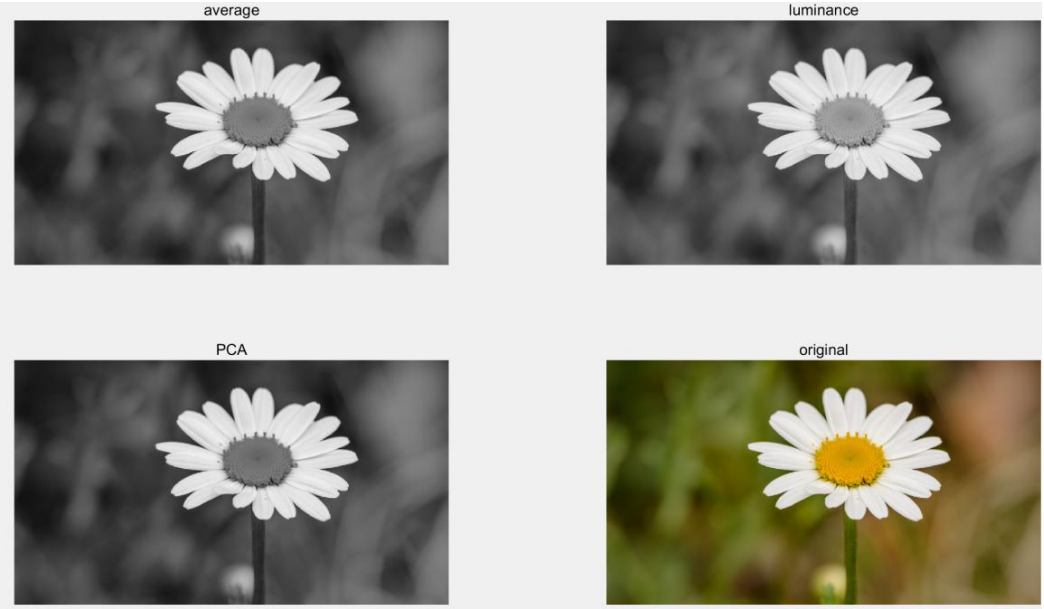


从肉眼来看，三个方法的结果基本没有差别，而事实上计算分数度量依次为：`0.0497`、`0.0573`、`0.0502`，虽然看起来有数值差别，但这个差别基本是可以忽略的(平均差距没有到 1 的亮度差，意味着基本无区别)，从结果来看，直接平均的时间效率是最高的，也即对一般图片直接采用平均是能得到不错的结果的，以下是另一个例子：



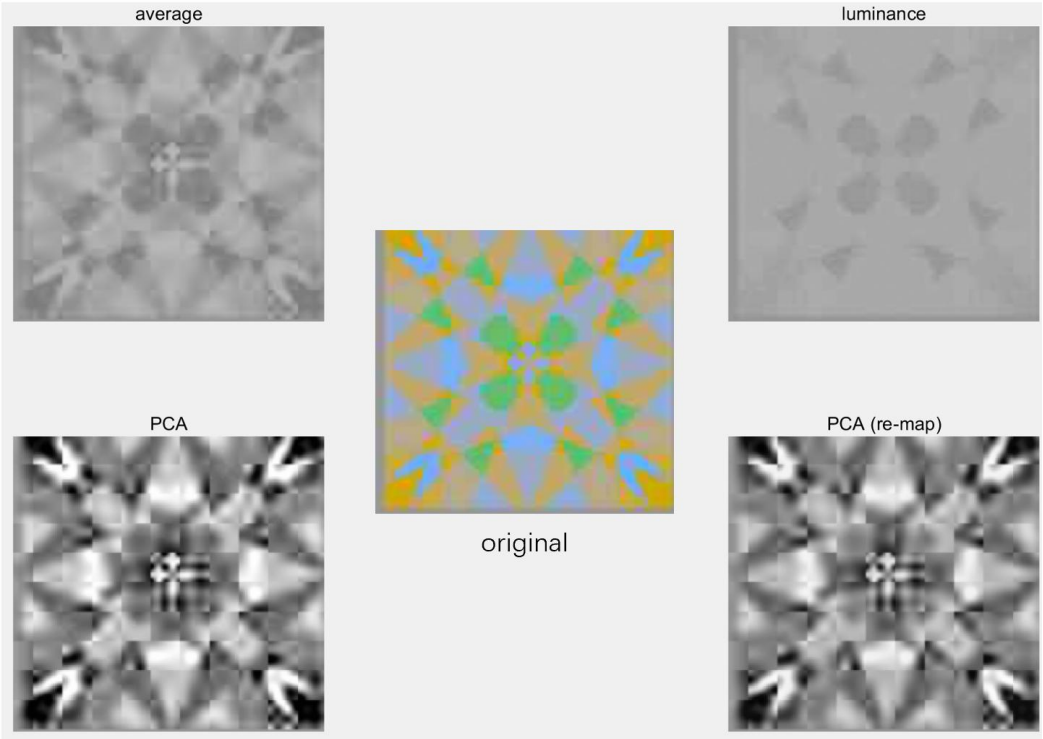
三种方法的误差分别是 0.1211、0.1347、0.1221。

但是，当图像的颜色成分比较单一时，PCA 方法的误差性能就会较好，如：



各方法的误差为 0.1748、0.2316 与 0.1668。

最后，我测试了一个非常极端的样例[样例来自同学]，直接 PCA 方法产生负分量特征向量，效果如下：



这时平均值与辉度平均的误差为 3.1465 与 4.2311，而下方左右的 PCA 方法误差分别为 2.5689、1.9330(其中左侧为舍弃负分量，右侧为重新映射)，可以发现效果并无显著区别，而舍弃负分量后视觉对比反而更强烈。

颜色匹配：

1、问题描述

将目标图片的颜色风格迁移到指定图片，并保证风格自然。

2、建模与实现

为了刻画颜色风格，需要引入 RGB 某个通道的概率密度函数(PDF)，也即 f(x)代表此通道中亮度小于等于 x 的像素所占的比例。建模中，我们认为两个图片的三个通道的 PDF 越接近，颜色风格就越接近。此外，由于我们需要保证原图的颜色信息不变，只能对每个通道进行亮度对亮度的对应映射，于是，采取简单的思路构造查找表：

```
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
```

这里查找表 LUT 的构造思路是，从 i,j 均为 0 开始，若将 i 映射为 j 还未到目标图片小于等于 j 的比例，就将 i 映射为 j，并且 i+1，否则 j+1，重新考虑。

但是，实际操作发现，直接如此匹配会产生两个问题：当两个图片颜色分布有较大差异时，匹配后颜色变化可能太大，导致原本连续的地方产生明显分界。第二，由于将每个通道单独考虑，可能让三个通道不协调，由此，加入了两个参数 if_conv[下称 c]和 if_equal[下称 e]考虑。

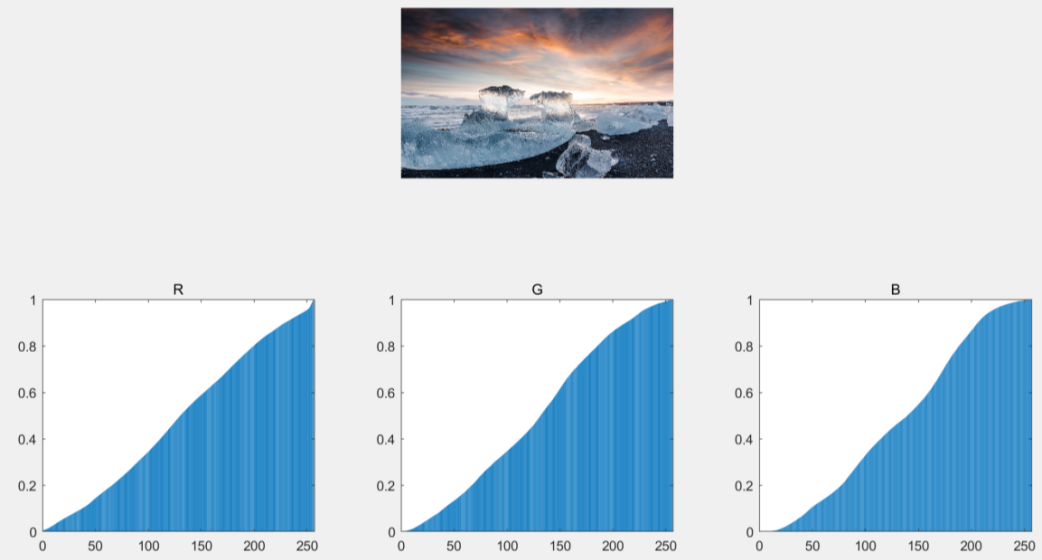
具体代码如下：

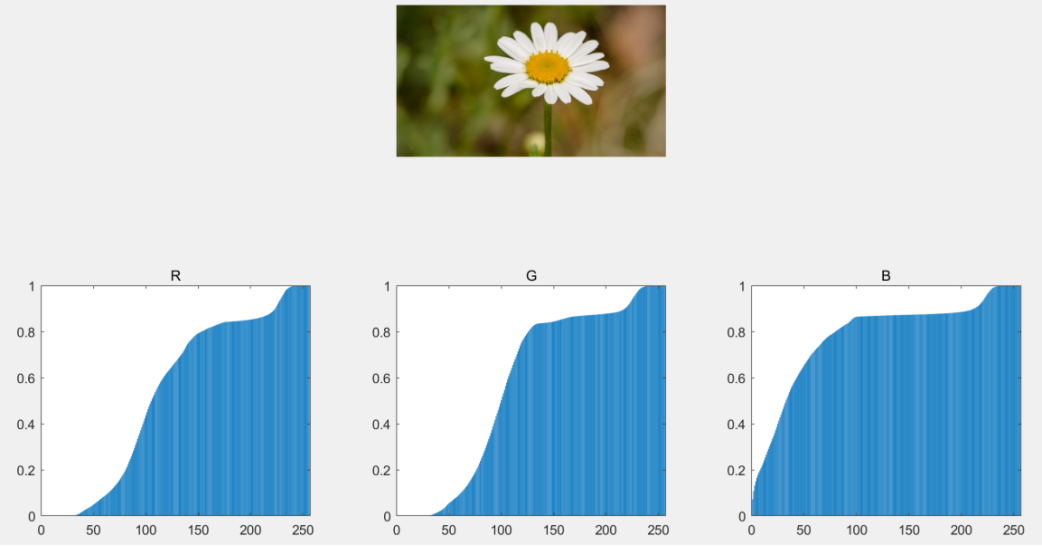
```
if if_conv > 0
    T = zeros(2 * if_conv + 256, 1, B);
    T(if_conv+1:if_conv+256, 1, :) = double(LUT);
    T(if_conv+257:2*if_conv+256, 1, :) = 255;
    for i = 1:256
        LUT(i,1,:) = sum(T(i:i+2*if_conv,1,:), 1) ...
            / double(1 + 2 * if_conv);
    end
end
if if_equal > 0
    T = double(LUT);
    r = if_equal / 10;
    s = 1 - 2 * r;
    LUT(:,1,1) = s * T(:,1,1) + r * T(:,1,2) + r * T(:,1,3);
    LUT(:,1,2) = s * T(:,1,2) + r * T(:,1,3) + r * T(:,1,1);
    LUT(:,1,3) = s * T(:,1,3) + r * T(:,1,1) + r * T(:,1,2);
end
```

c 相当于在查找表同一个通道内进行一定程度的卷积，从而进行光滑化，而 e 则是对三个通道的原本对应颜色作了融合，保存了一部分原有图像的颜色区分。利用这两个参数，可以达到比较好的整体效果。

3、结果展示

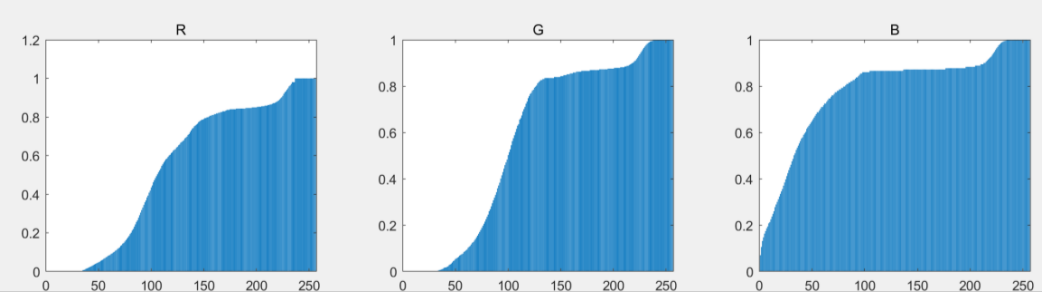
首先举一个例子来说明直方图的匹配：



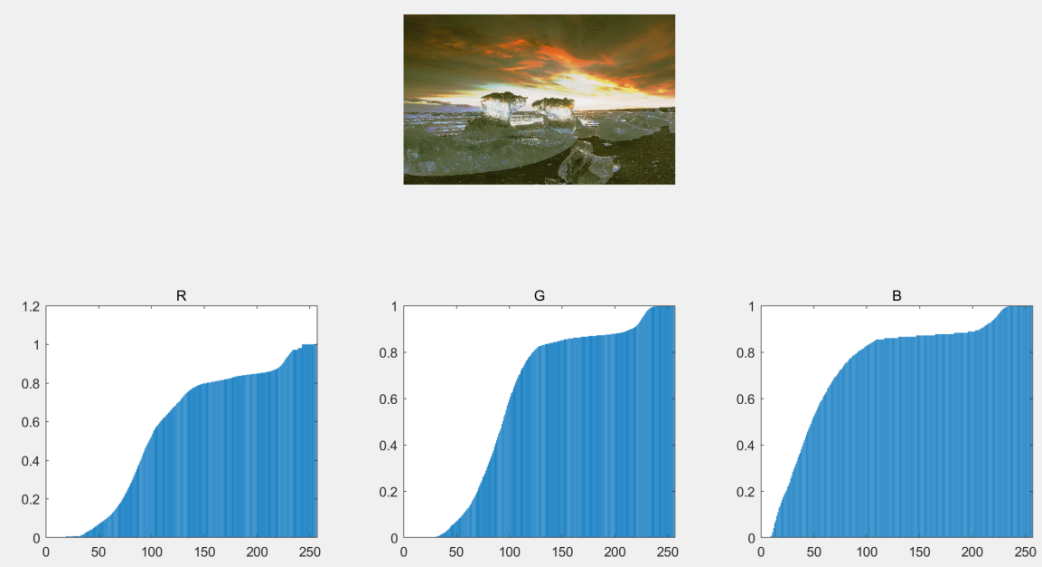


上方第一张为原图，第二张为目标图片，可以发现直方图的分布存在不小差距，而直接进行对应颜色的映射结果如下：

其直方图为：



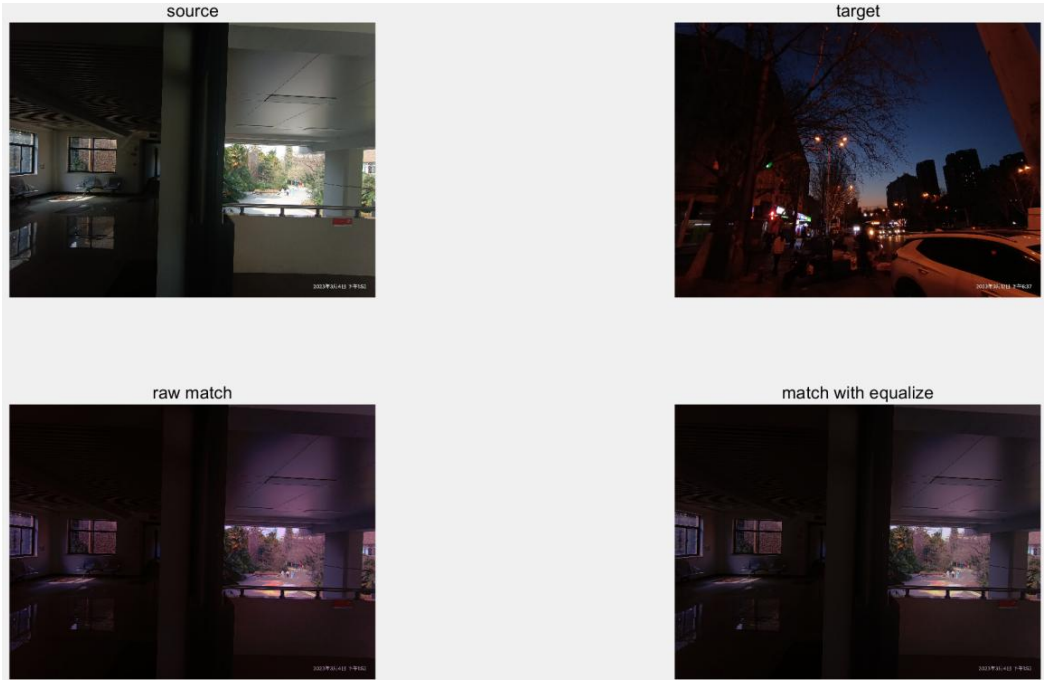
而采用 c=2，e=1（此为之后展示的默认参数)后的效果与直方图为：



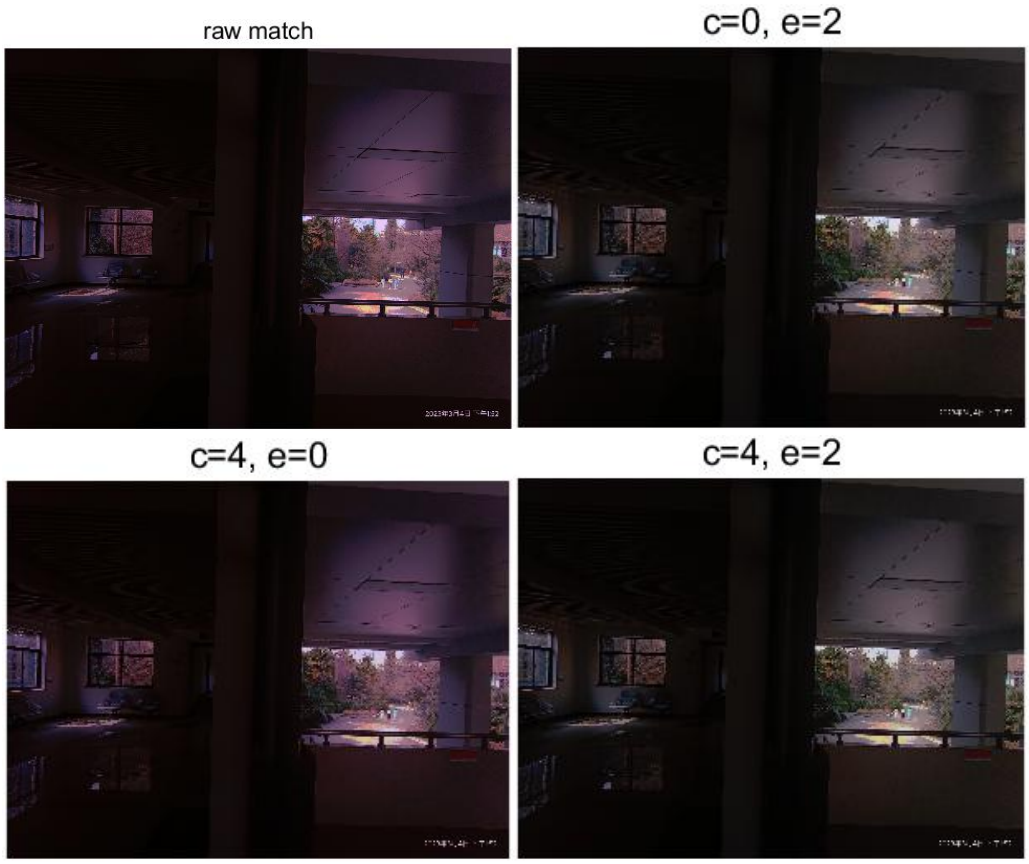
可以发现，由于保留了原有图像的一些颜色信息，卷积后的图像更加自然，以下为对比：

总体来说，e 的值对颜色风格的影响是明显的，而 c 虽然在这个例子中并不是非常明显，

但增大时也有清晰的光滑效果。另一个例子如下：



注意图片右侧中间的部分，原本的阳光在没有光滑化时会显得非常奇怪，而加入光滑化部分后就显得自然了很多，在 c 变化时的光滑程度比 e 变化时更加明显，下面是更细节的比较：



讨论与总结：

在有度量的情况下，灰度化另一个可以考虑的方法是度量下的约束最优化问题，不过由于构造方程考虑的边界情况较为麻烦，这里并没有实现，可以作为后续工作。此外，分析亮度的保持等要素可能可以得到更合理的度量，这也是可以完成的后续工作。不过，在限定的时空开销下，PCA 方法已经展现出了较好的性能。

关于画风迁移，更复杂的问题是，考虑颜色分布的情况下(而不是只作为像素)，如何找到一个比 PDF 更好的比较对象(进而可在此比较对象下进行优化)。另一个可能的思路是，直接取部分特征点颜色后利用梯度填充出原图像(这个思路同样也可被灰度图重填色采用)。由于画风是一个较为抽象的问题，想要找到较好的算法不仅需要数学方面的建模，还需要更多艺术的考量。

总体来说，算法的实现不算复杂，但想到问题的构造是更困难的一步。

*参考文献：数学建模 ppt + 上学期数学实验 ppt

Lab 2 矩阵分解 实验报告

PB20000296 郑滕飞

摘要

本文基于矩阵的低秩化逼近构造了图像的压缩存储与低秩恢复算法。对于压缩存储算法，本文讨论了其作为恢复方法的优势和局限性；而构造低秩恢复算法是时对比了各个参数下的差异，并在给定蒙版加以简单的区域光滑化下得到了更好的结果。

代码结构

gen_input.m 生成低秩图像存在扰动的输入

参数：N[输入维度]、mode[生成模式]

low_rank.m 低秩逼近实现

参数：D[输入图像]、epoch[迭代次数]、Om[干扰蒙版]、

lam[W 收缩参数]、al[E 收缩参数]、smooth[是否光滑化干扰区域]

zip.m 图像压缩实现

参数：I[输入图像]、ratio[压缩率]

test*.m 若干测试文件，输入参数示例见注释

图像压缩：

1、问题建模

图像压缩存储是指将图像以尽可能少的空间存储，同时保留尽量多的信息。对于$A_{m\times n}$，若其秩为 r，则可以分解成$A = P_{m\times r}Q_{r\times n}$，存储 P、Q 的压缩比例是$\frac{m+n}{mn}r$。在长宽接近时，这个比例可以近似成$\frac{2r}{n}$。只要压缩后的矩阵$r < \frac{n}{2}$，就可以达到节省空间的效果。

同时，为了保存信息，对矩阵采用奇异值分解的方法，对$A = USV^T$，其中 S 为奇异值由大到小排列构成的对角阵，若 S 只保留前 r 个对角元，则只需要存储 U 的前 r 列与 V 的前 r 行即能还原。

2、代码实现

为了进行简单情况的判断，本文构造了一个数据生成的函数，可以生成不同大小的低秩图像，以窗口噪声、随机噪声与两类噪声叠加三种模式添加噪声，分别效果如下：

在图像压缩与恢复中，都将见到这些噪声的影响。

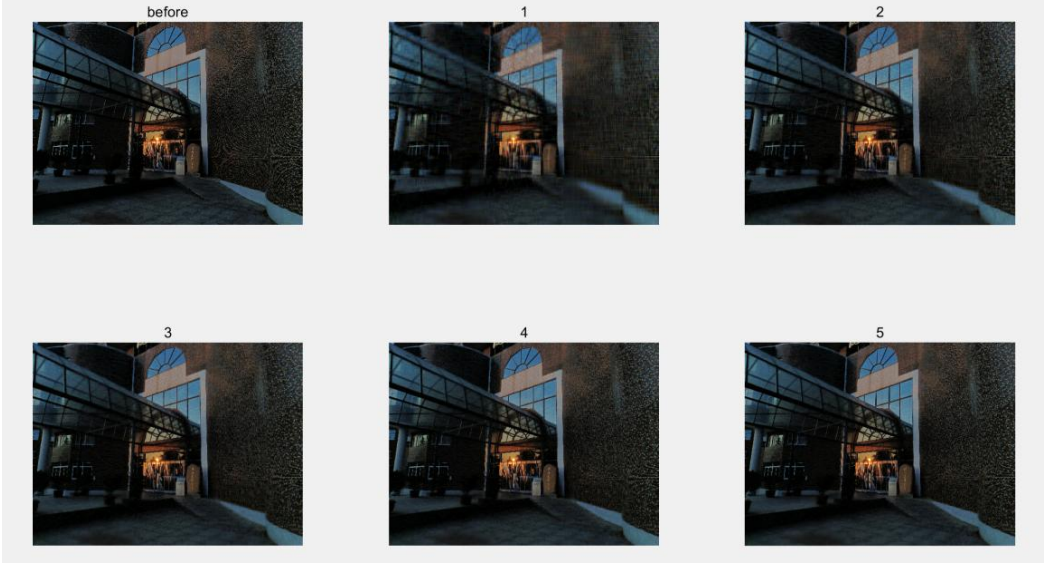
主要代码则计算出对角元保留的比例后进行去除：

```
function J = zip(I, ratio)
    [U, S, V] = svd(double(I));
```

```
    t = min(size(I));
    n = int32(t * ratio);
    for i = n+1:t
        S(i,i) = 0;
    end
    J = uint8(U * S * V');
end
```

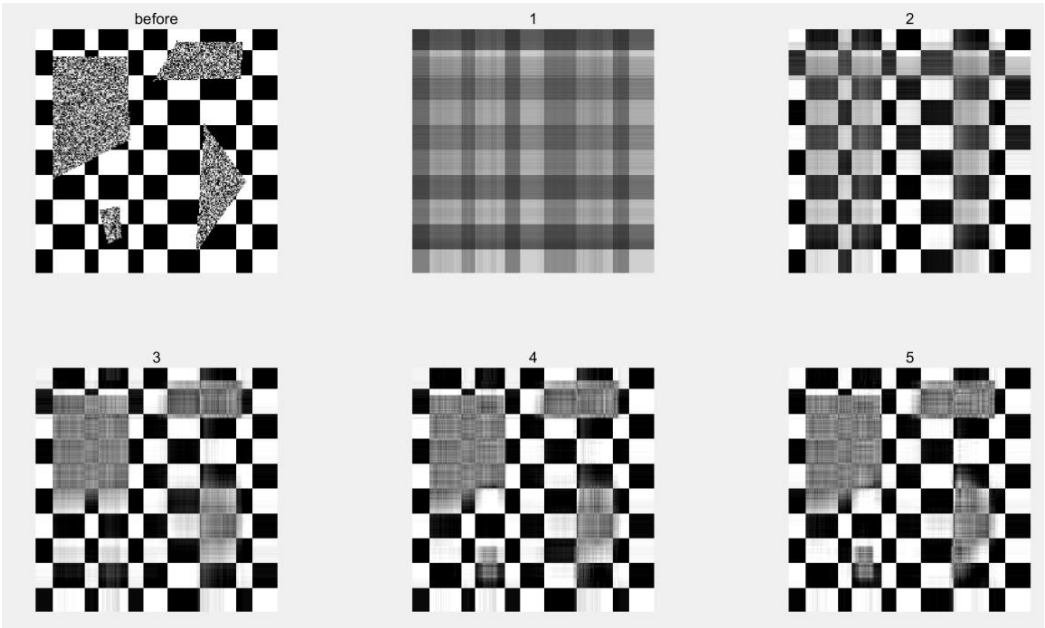
3、结果展示

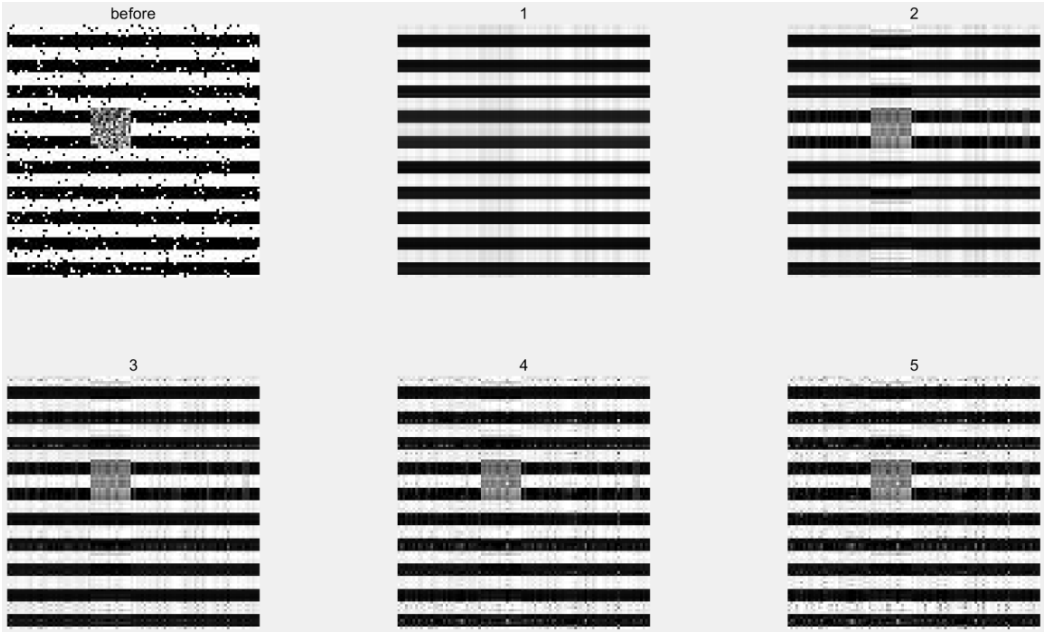
此处展示的 1 到 5 分别是压缩率为 r 的 1 到 5 倍的情况，其中压缩率是指保留对角元个数除以总对角元个数。三通道分别进行压缩。对一般照片效果如下，其中 r=0.01：



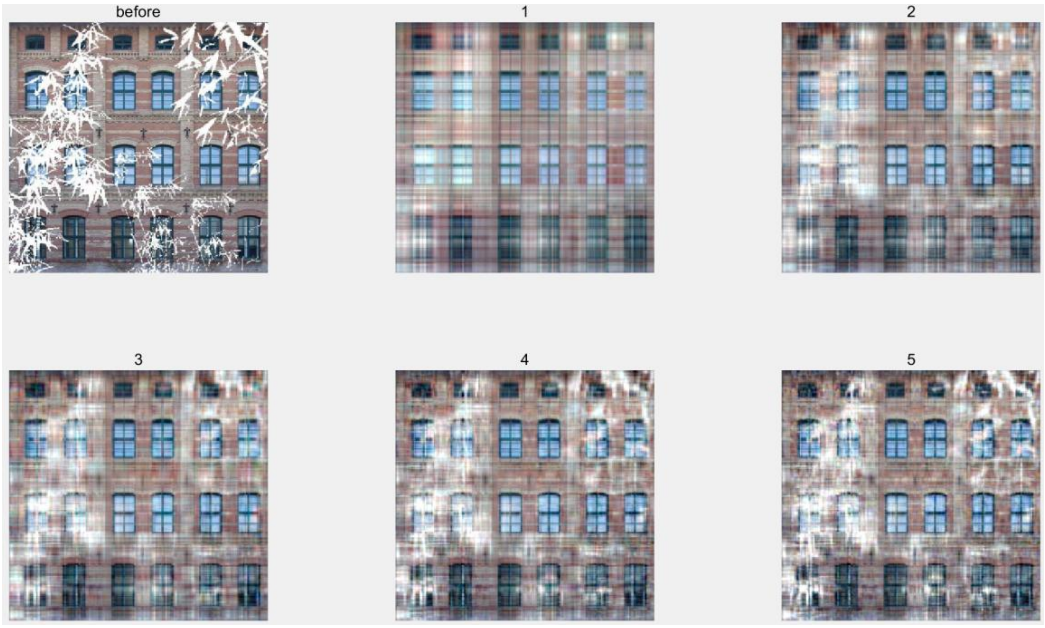
可以看出，从保留百分之三开始，除了字以外已经较难看出差异。

对有噪声图像的效果如下，其中 r 分别为 0.002、0.01：





从结果上看，理论来说通过选取合适的压缩比例，也可以做到一定程度的低秩图像恢复。但事实上，一方面是其只能一次得到简单的低秩近似，一方面无法控制噪声区域，如下图：



在这种较复杂的噪声情况下，只是低秩分解就难以控制了，因此，我们需要更加有效的图像恢复算法。

低秩恢复：

1、算法简述

所参考论文的核心迭代算法分为两步，即：

I 更新误差，在满足$P_{\Omega[i]}(B_1 W B_2^T + E) = P_{\Omega[i]}(D)$的条件下更新：

$$(W^i, E^i) = \text{argmin}_{W,E}||W||_* + \lambda||W||_1 + \alpha||E||_1$$

II 更新误差区域，根据得到的E, W更新$\Omega[i]$至$\Omega[i+1]$。

这其中，B_1, B_2是给定的正交基，W 是核心优化目标，下标星号代表奇异值之和范数，用于控制低秩，λ项用于控制连续性，保证其结果在低秩中是较为规整的，最终通过$B_1 W B_2^T$还原图像。E 代表优化结果的图像和原图的相差。由于很多时候损坏区域Ω不能完全确定，E 中较大的部分可以视为损坏区域，而较小的则可从损坏区域中消除，达到自适应更新的效果。

为了进行 E、W 的求解，文章将硬约束转化为了软约束，并额外引入了乘子与 A，进行序贯更新以达到整体的更新效果。此外，文章后方也提及了提前指定损坏区域时的结果，与针对损坏区域进行光滑性优化(即其更可能是连续的)，这也是本报告讨论的部分。

2、代码实现

核心 E、W 迭代部分的代码直接取自论文的公式，见附上的代码。此处主要讨论参数、初值与损坏区域的更新问题。参数会在之后进行对比，初值设置如下：

```
[R, C] = size(D);
D = double(D);

% initialize
mu = 0.04;
[B1, W, B2] = svd(D);
if (~exist('Om', 'var'))
    Om = ones(R, C);
else
    Om = double(Om);
end
E = zeros(R, C);
Y1 = zeros(R, C);
Y2 = zeros(R, C);
```

这里 Omega 是一个零一矩阵，而在 Omega 上投影事实上是通过逐位相乘来实现的。若有给定蒙版，则选用给定蒙版，否则认为初始保护所有像素，B1 和 B2 通过对原始图像进行 SVD 分解来实现，这样初始的 W 是简单的对角阵，E 全部为 0，方便后续更新。

(如果希望进行化简，事实上这一步就可以利用图像压缩去除 W 后方的对角元，不过直接进入迭代过程也可以。)

而对区域的更新、光滑化，这里都采取了简单手段：

```
    % update Om
    for i = 1:R
        for j = 1:C
            if (abs(E(i,j)) > eps)
                Om(i,j) = 0;
            end
        end
    end
```

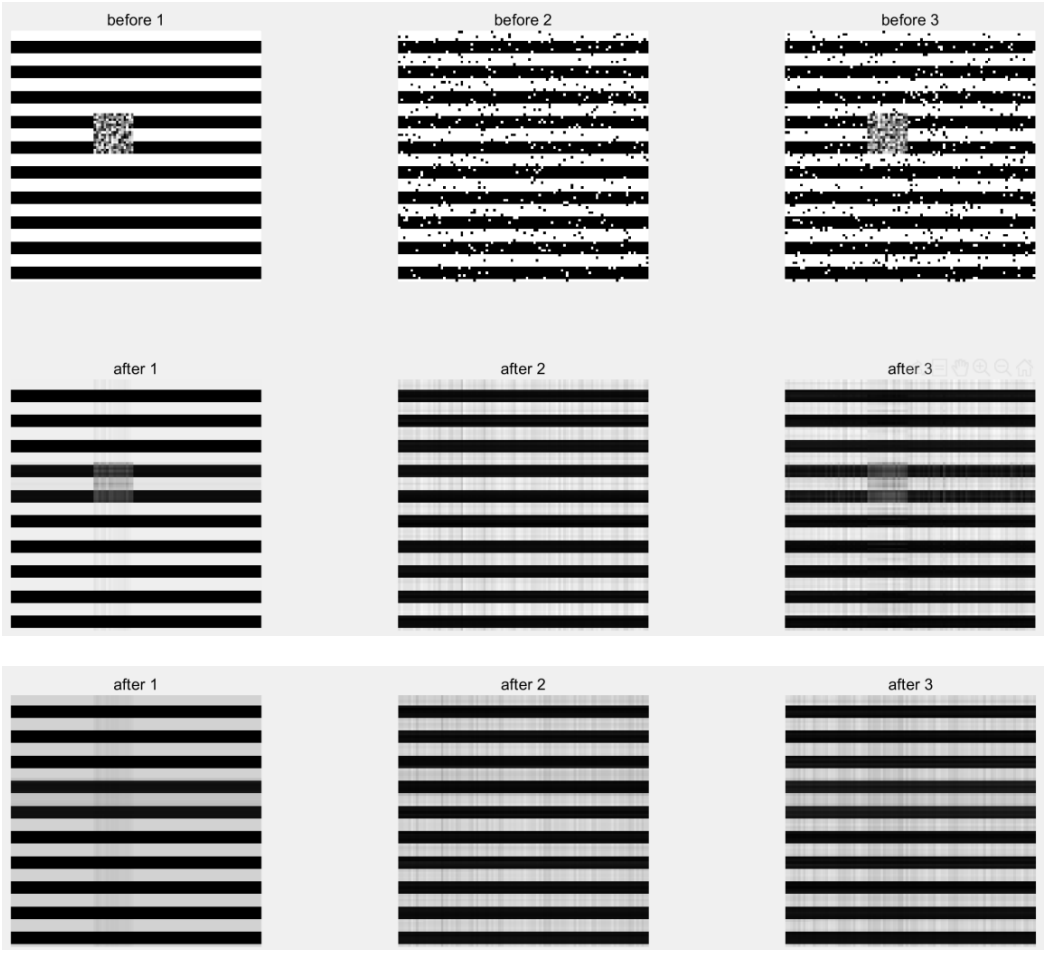
```
% smooth
if (smooth)
    sum = Om(2:R-1,2:C-1) + Om(1:R-2,2:C-1) + Om(3:R,2:C-1) ...
        + Om(2:R-1,1:C-2) + Om(2:R-1,3:C);
    Om(2:R-1,2:C-1) = (sum >= 2);
end
```

通过将 E 较大的区域置为损坏区域来进行更新(这里并没有恢复工作，事实上是在光滑化过程进行的)，而通过将自己置为相邻元素中较多的一类来进行光滑化(由于我们可以容许比实际更大的误差区域，这里放松了光滑的要求，只要周围 5 个有 2 个 1 就置为 1)。

3、基本效果

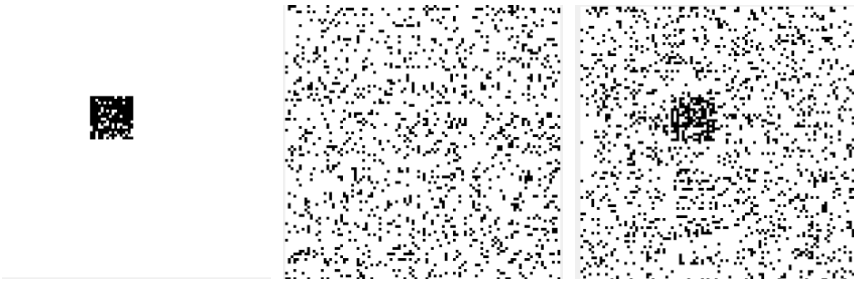
实际测试发现，迭代中 mu 需要以较低量级开始，而增加过程也不宜过快，因此取初始的 mu = 0.004，指数增加速率 1.009。两个 eta 参数仿照论文，均设为 3。决定是否是误差区域的 epsilon 设为 5，这是一个较为宽松的限制，理由仍然是我们可以容许比实际更大的误差区域。于是，之后对比的效果主要是迭代次数、lambda、alpha 的选取与是否光滑化、是否使用蒙版。[以下若无特殊说明，lambda=0.001，alpha=0.85，与论文一致。]

将人工生成的扰动图像(见上一部分)进行处理，原图、5 次迭代与 10 次迭代，无光滑化效果分别如下:

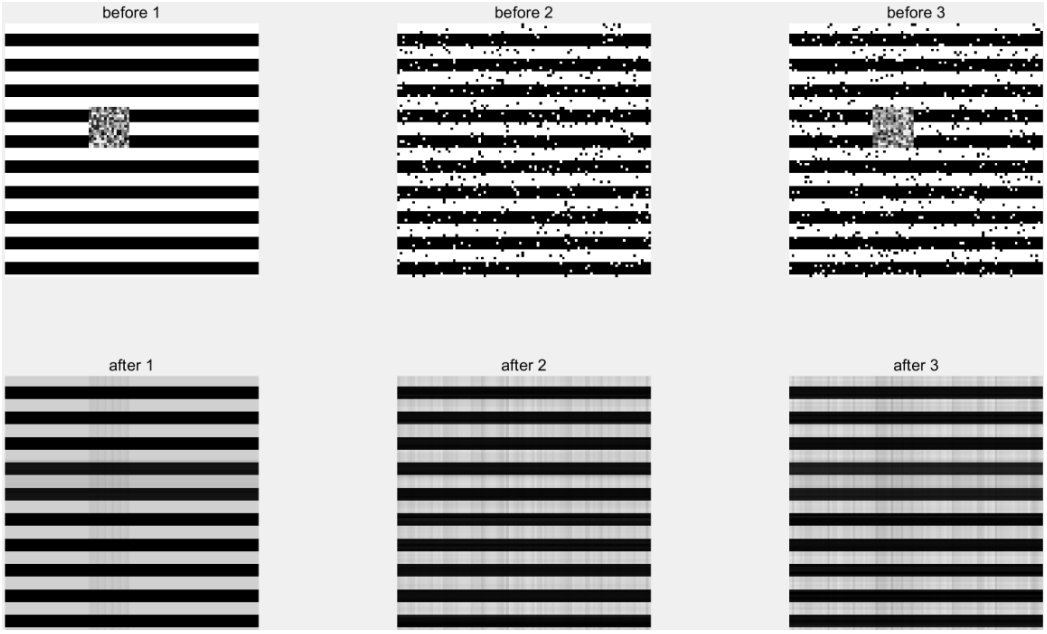


可以看出，对随机分布的误差，极小的迭代已经基本消除了特征，而固定窗口时则在更

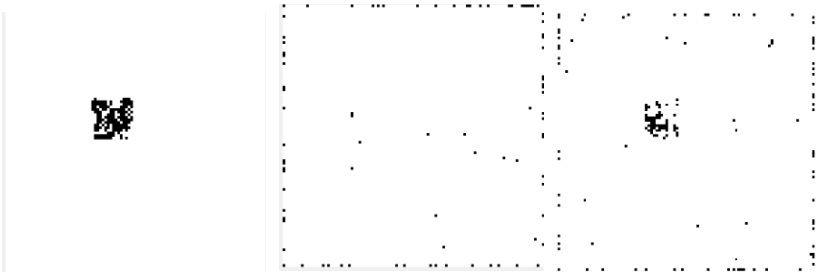
高次数迭代才消除了窗口的特征，但仍然有一定的颜色残留。对三种误差，一次迭代找到的损坏区域如下，可发现，当只有窗口时，误差与窗口非常一致，但添加随机下就有更大的扰动。



在添加区域光滑化时，10 次迭代总体结果如下：



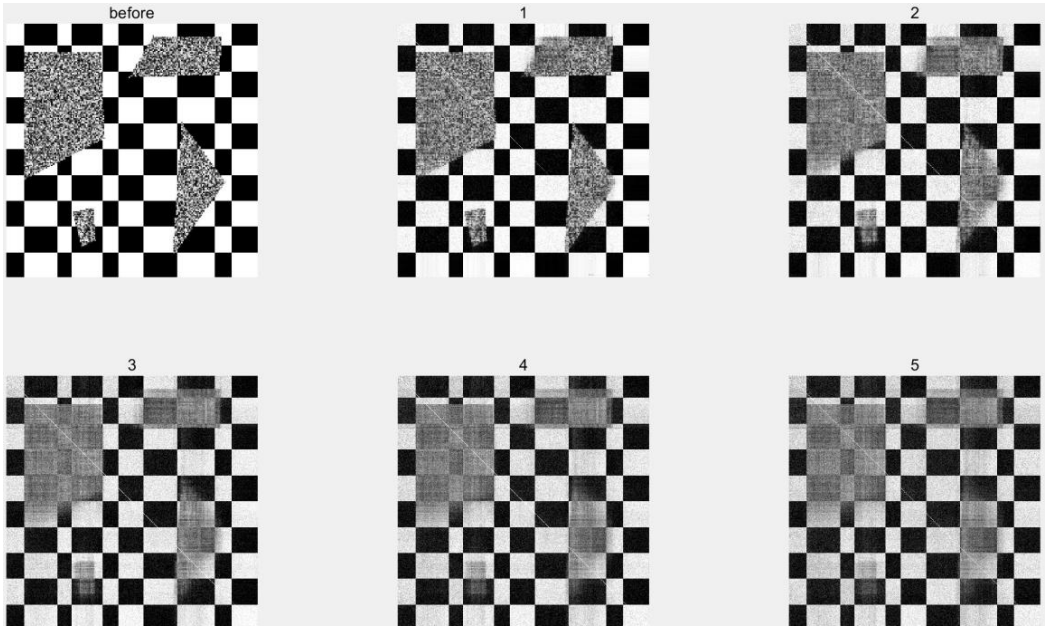
看上去无太大差别，不过一次迭代损坏区域此时为：



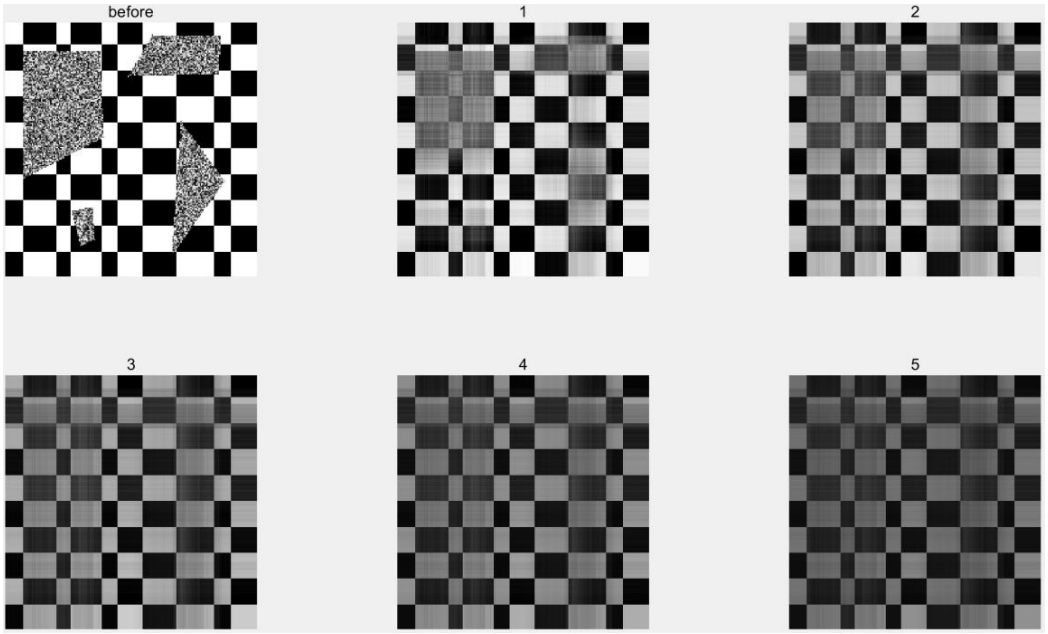
对连续的窗口误差有了更精确的结果。两次迭代时第一个窗口的误差已经识别为了：

非常接近准确。

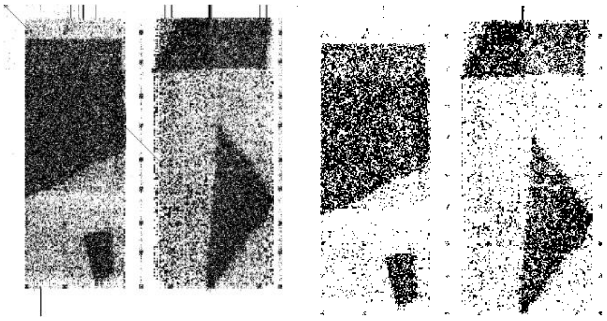
对更复杂的扰动的例子，若仍以之前的参数，迭代效果如下(1 到 5 中的每两个之间迭代了 7 次)：



可发现，这样的扰动下，必须加强连续性参数，否则得到的结果会非常不光滑。取 `lambda` 为 `50`，`alpha` 为 `0.5`，每步之间迭代两次得到：

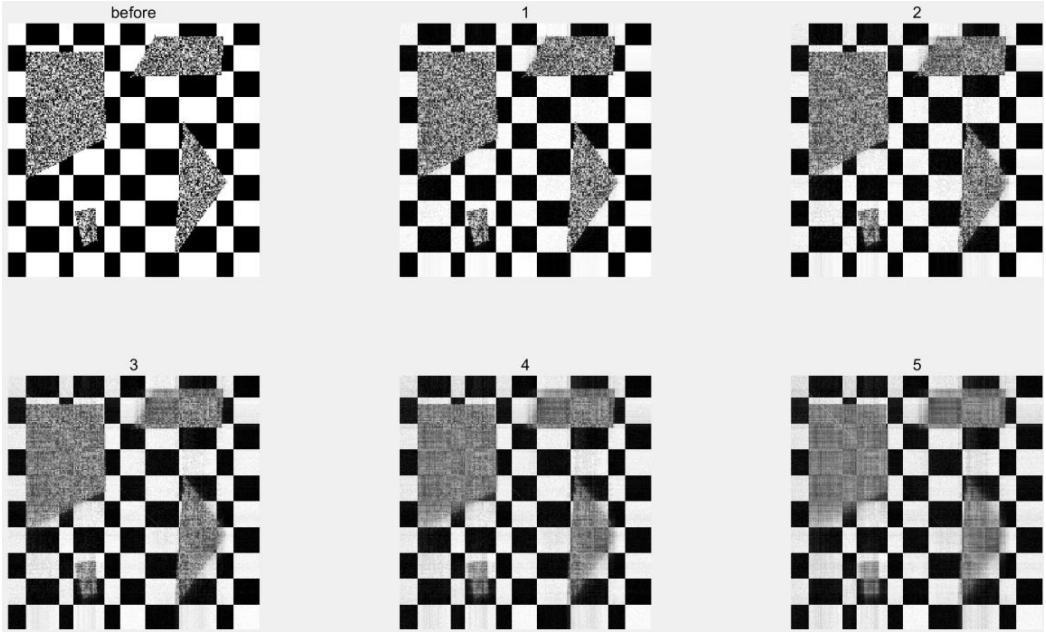


比之前有着总体更好的效果。而损坏区域在添加光滑化前后识别为：



也可见简单的光滑化策略的效果。

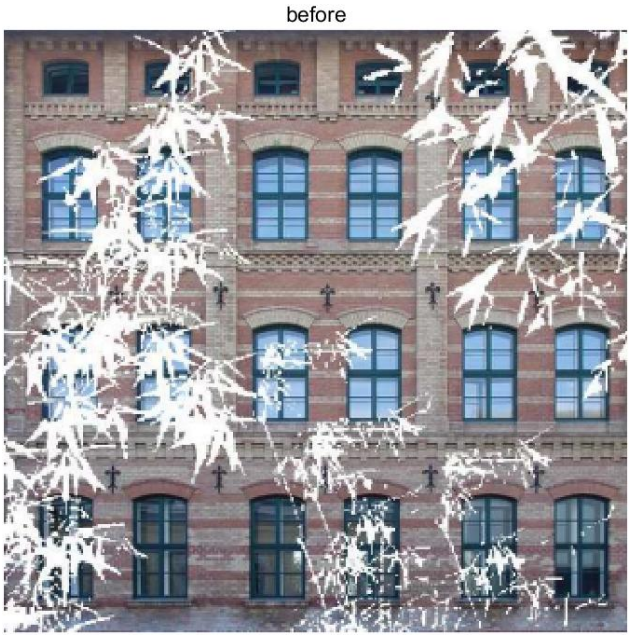
刚才的例子中，lambda 变化的影响非常明显。而若 lambda 仍为 0.001，alpha 也缩小到 0.001，每步迭代 7 次效果如下：



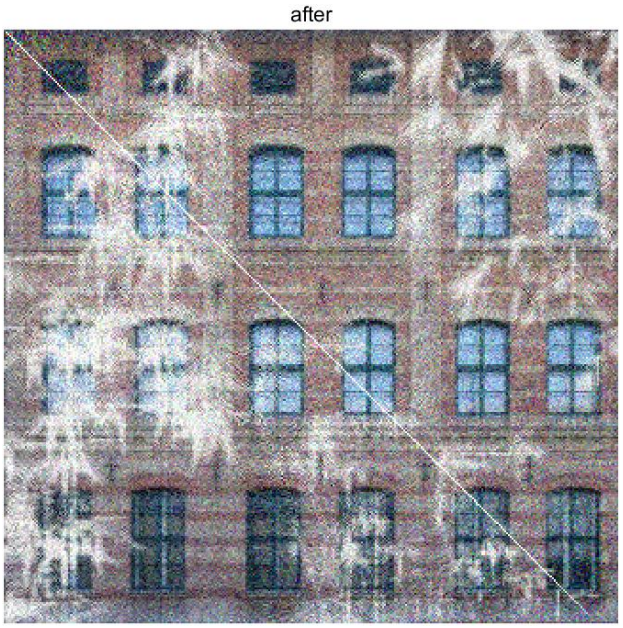
由于 alpha 减小阻拦了 E 的收缩幅度，从而影响了误差区域的更新，迭代的效率是下降的。

4、复杂样例

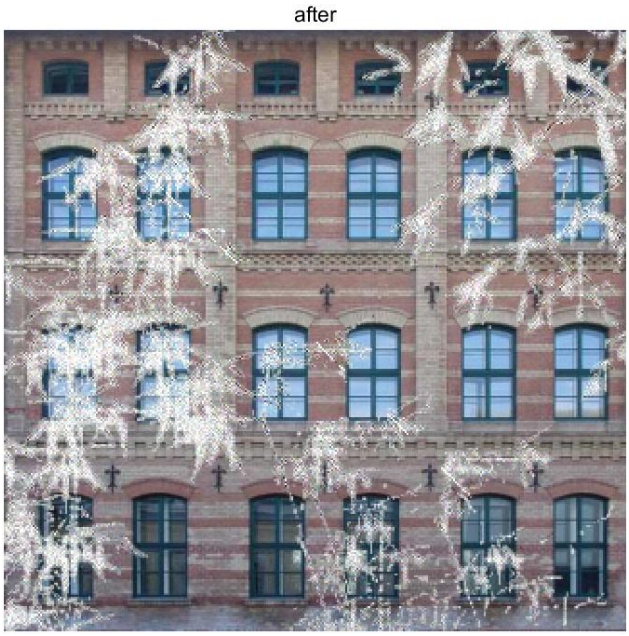
最后，我们通过一个复杂的例子来考察蒙版的影响。图像如下：



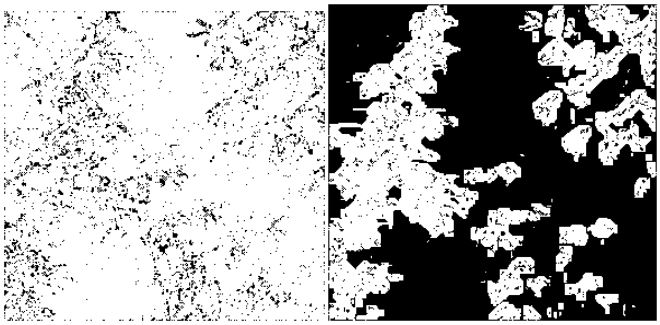
出于测试结果，将 lambda 设置为 0.01，初始的 mu 设置为 0.04，并开启光滑化。首先是无蒙版 10 次迭代的结果(RGB 分别迭代)：



而当将 R 分量大于 200 的部分(由损坏为白色，可如此粗略估计)设为蒙版，十次迭代的结果为：



两者 Omega 的对比如下：



无蒙版时，在损坏区域的改变并不能起到有效效果，这也就是一般情况下基于整体的迭代方法难以使用的原因，由此可发现，蒙版的设置有效地减少了图像在非损坏部分的变化。下方是蒙版下 9 次迭代 12 趟的较好结果，在较少噪声下保存尽量多原始信息：

after

（值得一提的是，图像的中间出现了缝隙，这很可能是由于初始构造了奇异值分解，于是初始矩阵为对角阵，在迭代中对角元部分放大的结果。）

从这里的结果中，亦可见自适应改变区域与光滑化的有效性。

讨论与总结：

复现论文果然还是很痛苦，尤其是其还可能有错、隐藏了一些关键参数的情况下。低秩恢复相关可尝试的算法还有很多，也有着不同改进的优化模型，时间有限，便做到这步为止。

*参考文献：数学建模 ppt 与如下论文：
people.csail.mit.edu/zhangzd/papers/recover_low-rank_texture_final.pdf

Lab 3 数据拟合 实验报告

PB20000296 郑滕飞

摘要

本文基于多项式插值构造了几种平面点列插值的方法，并且进行了对比，在包含光滑程度的不同度量下考察优劣，且结合正则化给出了原始插值的改进办法；此外，本文尝试了利用神经网络进行数据拟合，并与其他方法对比，考察网络结构、参数的效果。

代码结构

`interpol.m` 各种多项式逼近平面点列方法的实现

`test.m` 测试、对比各种方法与正则化的效果。

参数：`X`[输入 x 值列表]、`Y`[输入 y 值列表]

`main.m` 主要测试脚本

`main.ipynb` 神经网络数据拟合的相关结果

平面点列插值：

1、问题建模

平面点列插值的问题为：已知平面上一列点 `X(1:i)`、`Y(1:i)`，求一个依次通过这些点的曲线。为了解决此问题，需要给出最优性的度量，这里考虑两件事：

其一，拟合的精确性，以给出的点到最后拟合成曲线的平均距离进行度量，越小代表越接近。

其二，曲线的光滑性，以拟合后曲线的总长度除以直接顺次连接形成的折线长度进行对比，越小代表越光滑。

值得一提的是，如果只考虑这两点，折线段似乎是最优的结果，但实际上我们对曲线有更高的光滑性要求，譬如基本插值方法中 `X(t),Y(t)`都要求是 `t` 的多项式，这样曲线连接处就具有无穷阶光滑性。

两种误差计算的实现如下：假设 `X`、`Y` 是原插值点，`res_X`、`res_Y` 是结果点列，则：

```
dif_X = res_X(2:lr) - res_X(1:lr-1);
dif_Y = res_Y(2:lr) - res_Y(1:lr-1);
dis = sqrt(dif_X .^ 2 + dif_Y .^ 2);
sub = res_X(2:lr) .* res_Y(1:lr-1) - res_X(1:lr-1) .* res_Y(2:lr);
for i = 1:l
    now = dif_Y * X(i) - dif_X * Y(i) + sub;
    ----
    根据内积确定点是否投影在线段上，从而选择是否改为到端点取值
    [较复杂，详见源代码]
    ----
    err(1) = err(1) + min(abs(now) ./ dis);
end

err(1) = err(1) / l;
ori_dis = sum(sqrt((X(2:l) - X(1:l-1)) .^ 2 + ...
    (Y(2:l) - Y(1:l-1)) .^ 2));
```

```
    dis = sum(dis);
    err(2) = dis / ori_dis;
```

由于结果事实上是拟合出精细的折线段，点到曲线的距离是用点到其中每段折线距离的最小值估算的。计算可知(x,y)到直线$(x_1,y_1),(x_2,y_2)$的距离为(须再结合线段判断)

$$\frac{|(y_2-y_1)x+(x_1-x_2)y+x_2y_1-x_1y_2|}{\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}}$$

而下方正好是这段折线的长度，因此可以进行一定复用提升计算速度。

2、正则化

在实际情况下，拟合精确性是问题更重要的要求，因此假设拟合后的曲线是 C，每个点记为p_i，我们采取的优化手段是对$\sum_i d(p_i,C)+\lambda \mathrm{L}(C)$进行优化，将后一项看作正则化项。

自然，为了得到 X(t),Y(t)，我们需要知道 X、Y 分别与 t 的关系，也就是如何进行参数化。算法中考虑了三种方式：等距、切比雪夫与自适应(为了切比雪夫点使用方便，假设参数区间在[-1,1])。等距方式是假设所给的点处在等距结点，切比雪夫假设其在切比雪夫点，而自适应方法则假设相邻两点之间的距离是参数 t 的距离(也即假设为某种意义的弧长参数)。而无论是哪种，都会变为求解独立的 X-t 与 Y-t 两个插值问题。

当曲线光滑性要求不存在时，只需要考虑精确，这时的插值就是严格插值的结果，算法中采用牛顿插值的办法来计算。但涉及正则化后，情况就更加复杂。

为了保证函数的光滑性，若所给的点为(x_i,t_i)，实际解决的优化问题是

$$\min_{f(x)\in\mathbb{R}_n(x)}\sum_{i=1}^{n}(f(t_i)-x_i)^2+\frac{\lambda}{2}\int_{-1}^{1}\big(f''(x)\big)^2\mathrm{d}\,x$$

接下来对系数进行计算。假设$f(t)=\sum_{j=0}^{n-1}c_j\,t^j$，则右侧展开得到

$$\sum_{i=1}^{n}\left(\sum_{j=0}^{n-1}c_j\,t_i^j-x_i\right)^2+\lambda\int_0^1\left(\sum_{j=0}^{n-1}j(j-1)c_jx^{j-2}\right)^2\mathrm{d}x$$

对c_s求导得到

$$2\sum_{i=1}^{n}t_i^s\left(\sum_{j=0}^{n-1}c_j\,t_i^j-x_i\right)+2\lambda\int_0^1\left(\sum_{j=0}^{n-1}j(j-1)c_j\,x^{j-2}\right)s(s-1)x^{s-2}\mathrm{d}\,x=0$$

$$\sum_{j=0}^{n-1}c_j\sum_{i=1}^{n}t_i^{s+j}-\sum_{i=1}^{n}x_i\,t_i^s+\lambda s(s-1)\sum_{j=0}^{n-1}\frac{j(j-1)}{s+j-3}c_j=0$$

若 j < 2 或 s < 2，后项认为是 0，整理得

$$\sum_{j=0}^{n-1}c_j\left(\sum_{i=1}^{n}t_i^{s+j}+\lambda s(s-1)\frac{j(j-1)}{s+j-3}\right)=\sum_{i=1}^{n}x_i\,t_i^s$$

此式对任何$s=0,\dots,n-1$都成立，从而可以求解 c。可看出，当$\lambda=0$时，要解决的就是简单的插值问题。

实际系数求解过程如下：

```
    Temp = zeros(2*l - 1, l);
    for i = 1:2*l-1
        Temp(i, :) = t .^ (i - 1);
    end
```

```
    A = zeros(l, l);
    b = zeros(l, 1);
    for s = 1:l
        b(s) = sum(X .* Temp(s, :));
    end

    for s = 0:l-1
        for j = 0:l-1
            A(s+1, j+1) = sum(Temp(s+j+1, :));
            if (j > 1 && s > 1)
                A(s+1, j+1) = A(s+1, j+1) + ...
                    lambda * s * (s - 1) * j * (j - 1) / (s + j - 3);
            end
        end
    end

    c = A \ b;
```

这里通过将可能用到的x_i^j提前保存在 Temp 里来提升计算速度。

3、插值实现

前面提到，代码考虑了三种参数化方式：等距、切比雪夫与自适应。此外，还有一个降低光滑性要求的三次样条插值，代码如下：

```
function err = spline_inter(X, Y, N)
    l = length(X) - 1;
    t = -1: 2/l : 1;
    in_t = -1 : 2/N : 1;
    res_X = spline(t, X, in_t);
    res_Y = spline(t, Y, in_t);
    plot(res_X, res_Y, "linewidth", 1.5);
    err = count_err(X, Y, res_X, res_Y);
end
```

由于三次样条的本质是分段二阶光滑，参数化方式并没有太大影响，这里直接采取了等距参数化。同时，也对三次样条进行了两种误差的估算，用于对比。值得注意的是，这里并没有涉及正则化，因为默认三次样条自身已经是要求范围内$||f''||_2$最小的函数。

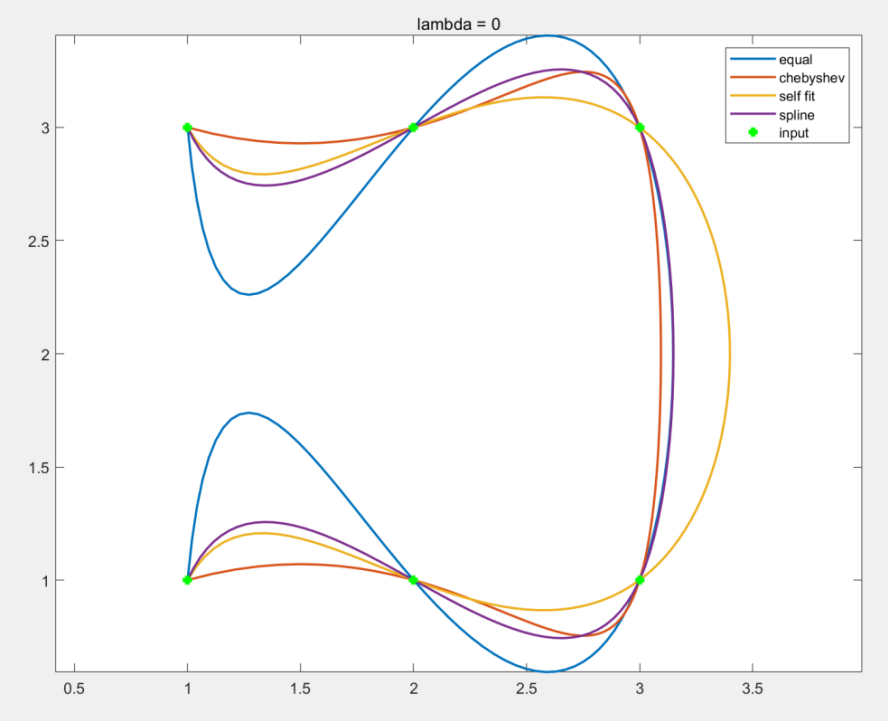
其他的估算均为先选取结点，再调用之前的拟合函数，如自适应选取的方式：

```
    dis = sqrt((X(2:l) - X(1:l-1)) .^ 2 + (Y(2:l) - Y(1:l-1)) .^ 2);
    dis = dis / sum(dis) * 2;
    t = zeros(1, l);
    t(1) = -1;
    for i = 1:l-1
        t(i+1) = t(i) + dis(i);
    end
```

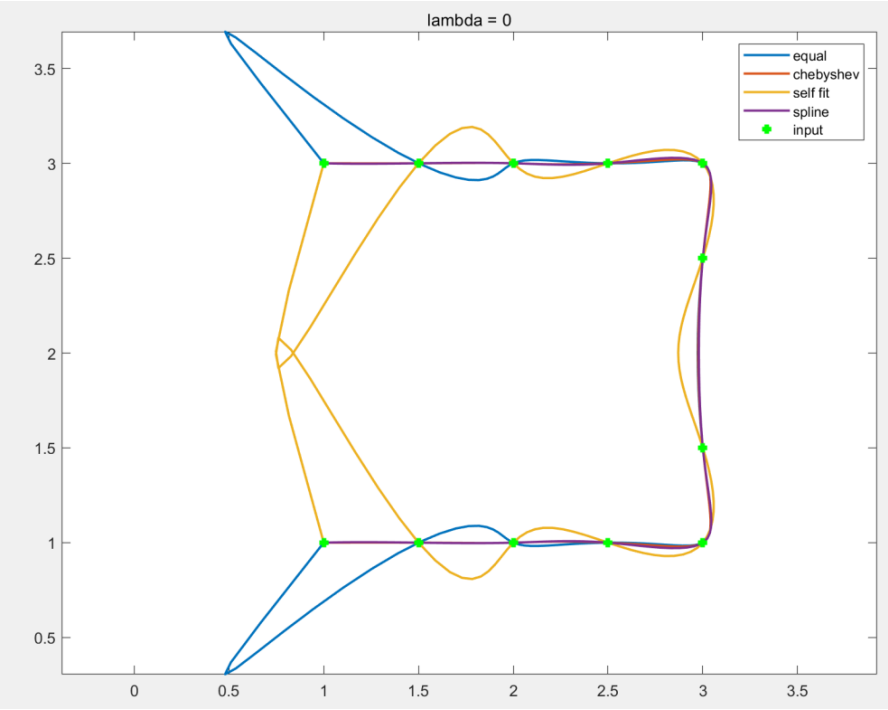
4、基本效果

为了测试各种函数的效果，我构造了四个测试样例。在不添加正则化(即 `lambda = 0`)的情况下，它们的效果分别如下：

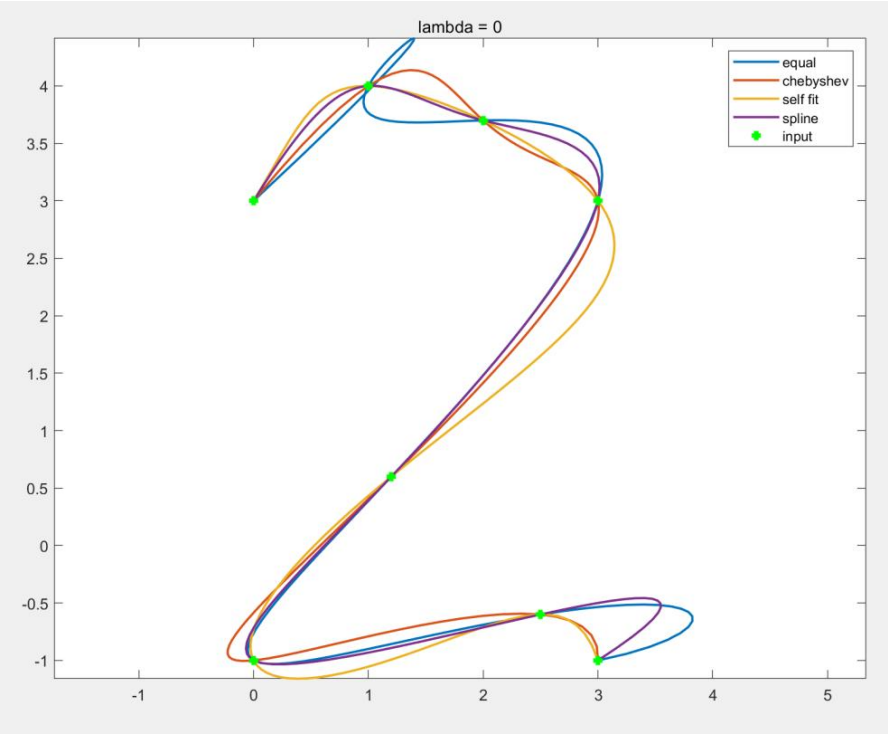
第一个，六点确定的简单方框：



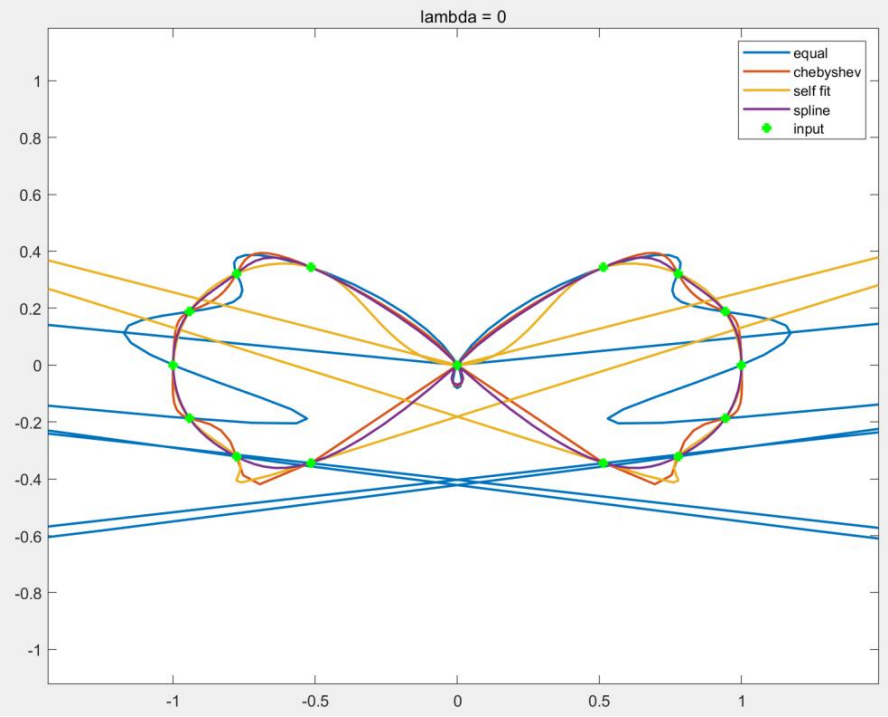
第二个，取点更加精细的方框：



第三个，手写 2 的例子：



第四个，双扭线的部分：



可以直观看出，等距结点的龙格现象极为明显，自适应则次之。与它们相对，切比雪夫结点和三次样条则较为稳定。

事实上，上面四个例子中，误差分别为(误差的第一项代表拟合点到曲线的平均距离，第二项代表长度倍数)：

```
test 1:
lambda = 0:
equal err:     3.198819e-15   1.418106
cheby err:     1.539082e-04   1.064128
self fit err:  5.471751e-15   1.092362
spline err:    0.000000e+00   1.117853

test 2:
lambda = 0:
equal err:     2.788302e-04   1.555441
cheby err:     4.998373e-05   1.006596
self fit err:  2.327637e-14   1.724013
spline err:    2.148143e-04   1.007809

test 3:
lambda = 0:
equal err:     4.609895e-04   1.291796
cheby err:     2.869352e-04   1.058411
self fit err:  4.763231e-04   1.052742
spline err:    5.279877e-04   1.130398

test 4:
lambda = 0:
equal err:     5.392539e-04   92.229898
cheby err:     1.908507e-04   1.097163
self fit err:  5.352625e-04   58.609989
spline err:    2.133998e-04   1.049530
```

可以看出，对不带正则化的插值，在绘制点没有取到拟合点时(即误差不是 e-14/e-15 这样的机器精度量级)，误差基本都在 e-4 量级，相差不大。但是，长度的倍数差别可以很大。

大部分情况下，切比雪夫插值是表现最好的，长度倍数从没有到 1.1，而等距结点则表现最差，在结点数增多时表现出了明显的误差，甚至在最后一个例子中和自适应结点一起给出了完全失真的拟合。自适应结点的好坏取决于结点的分布，一般来说结果并不好，不过也在一些特定例子中有更好的效果。

值得一提的是样条曲线。样条曲线虽然在好的例子中不如切比雪夫点，但发生复杂的情况，例如曲线自交(最后一个例子)时，由于它只关心局部的连接，可以做到更稳定。仔细观察可以发现，直观上它也的确比切比雪夫插值更光滑。

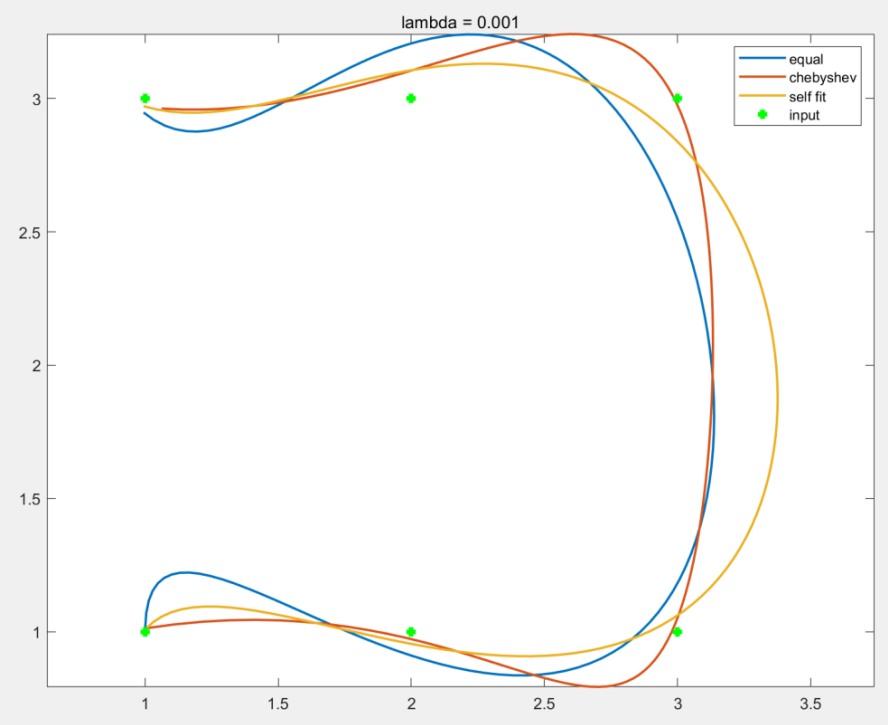
5、正则化的影响

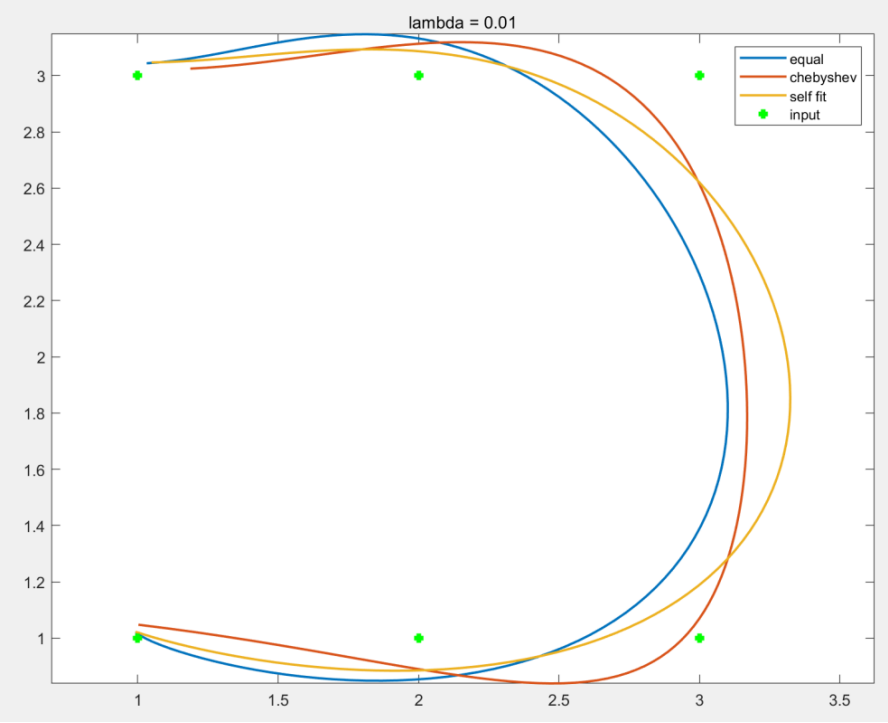
对前三种插值引入正则化后，可以看到明显的效果。以第一个测试为例，引入后误差为：

```
lambda = 0.001:
equal err:     3.197669e-01   1.028084
cheby err:     1.710122e-01   1.032636
self fit err:  1.875879e-01   1.027270
```

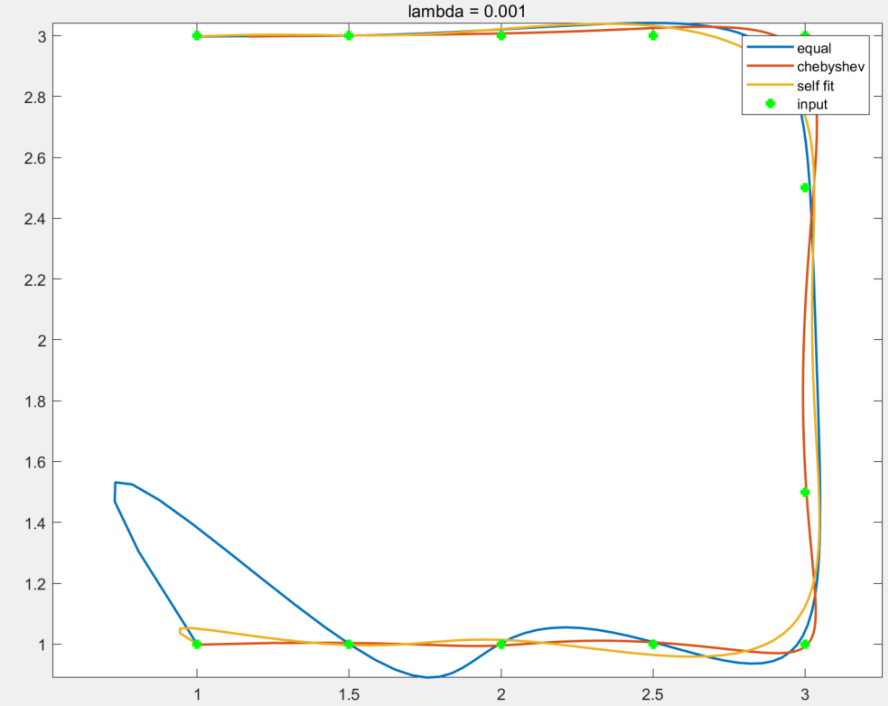
```
lambda = 0.01:
equal err:     3.613341e-01   0.915455
cheby err:     6.365365e-01   0.951235
self fit err:  3.301384e-01   0.964097
```

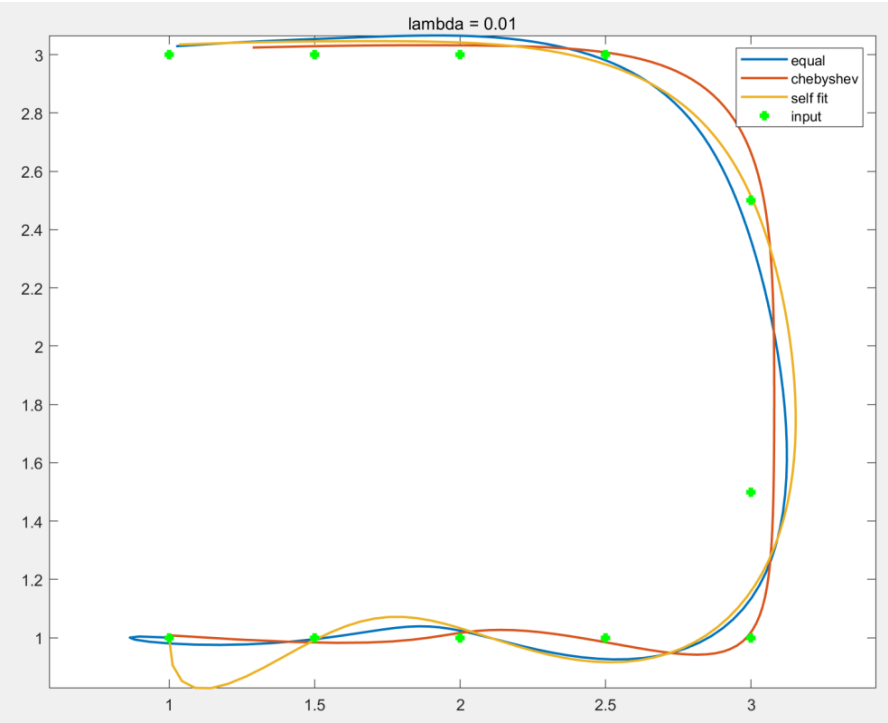
可以明显看到，拟合误差增大了，而光滑程度显著上升，甚至可以低于 1。实际图像如下：





对第二个测试效果为：



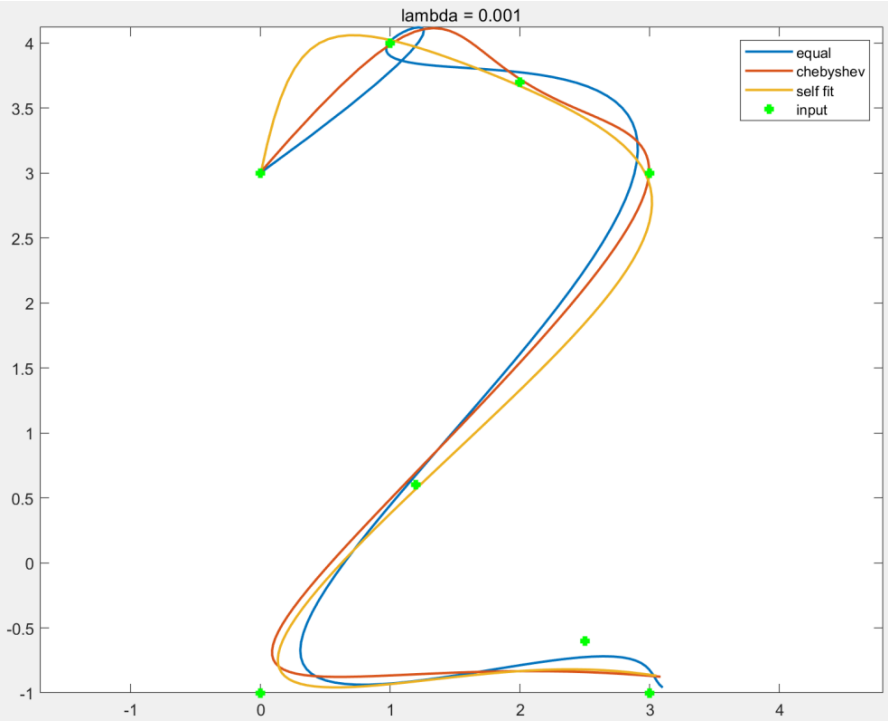


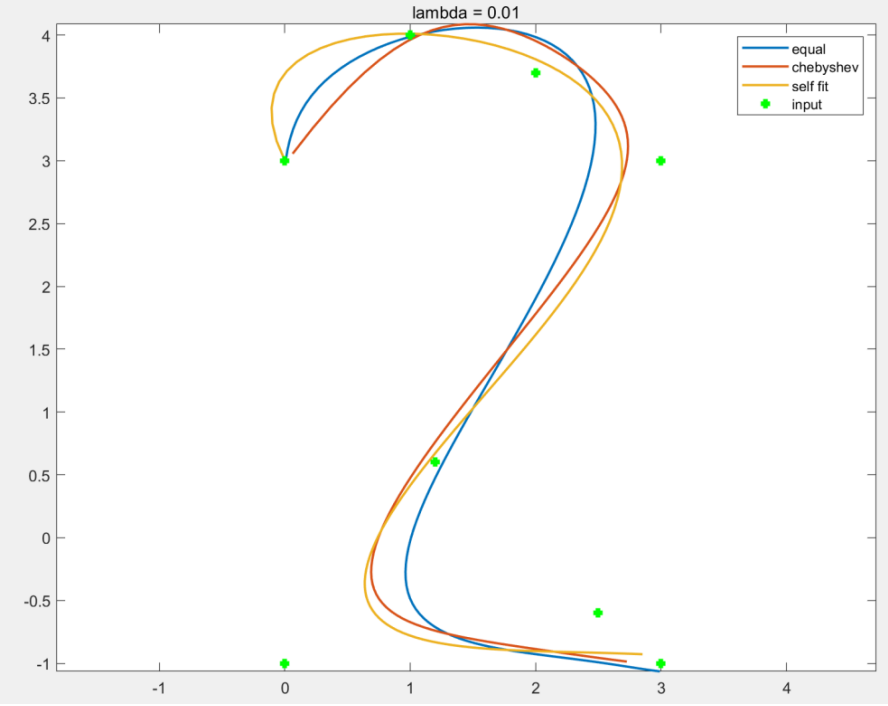
可以直观感受到过程中龙格现象产生的峰被抹平。具体误差为:

```
lambda = 0.001:
equal err:     2.317918e-02   1.170572
cheby err:     1.649986e-01   0.970953
self fit err:  3.687493e-01   0.991702
lambda = 0.01:
```

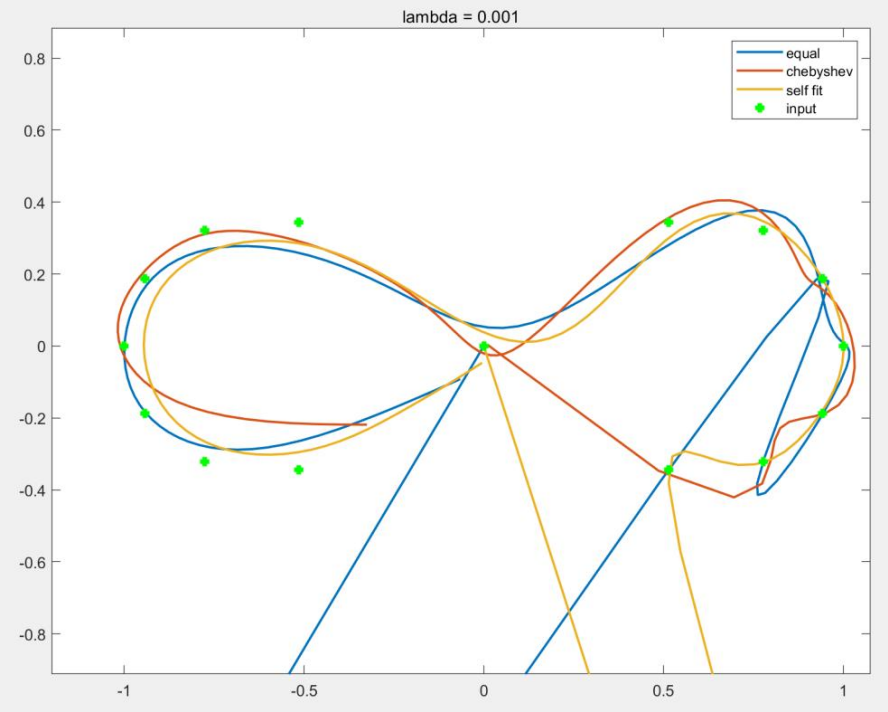
```
equal err:     2.790926e-01   0.988591
cheby err:     4.268067e-01   0.925512
self fit err:  1.435386e-01   0.974336
```

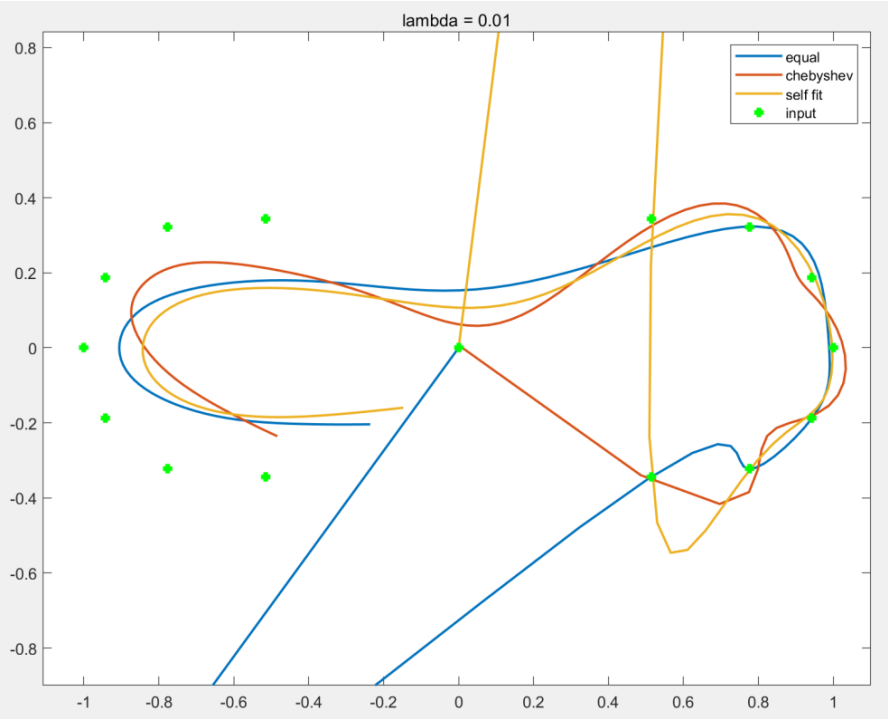
对测试三的效果如下：





在 lambda 过大时，过于光滑很容易导致丢失信息，例如左下角的点几乎被忽略了。这里，误差并没有表现出与之前明显差异的性态，因此不呈现。最后，是测试四的结果：





可以发现，在这种曲线本身复杂的情况下，正则化对信息的丢失是非常明显的，哪怕是切比雪夫结点都产生了很严重的失真。事实上，误差为：

```
lambda = 0.001:
equal err:     6.528153e-02   3.362915
cheby err:     3.470706e-02   0.943017
self fit err:  2.315200e-02   10.175092
```

```
lambda = 0.01:
equal err:     1.426170e-01   1.334299
cheby err:     2.668858e-01   0.817224
self fit err:  8.761602e-02   12.163108
```

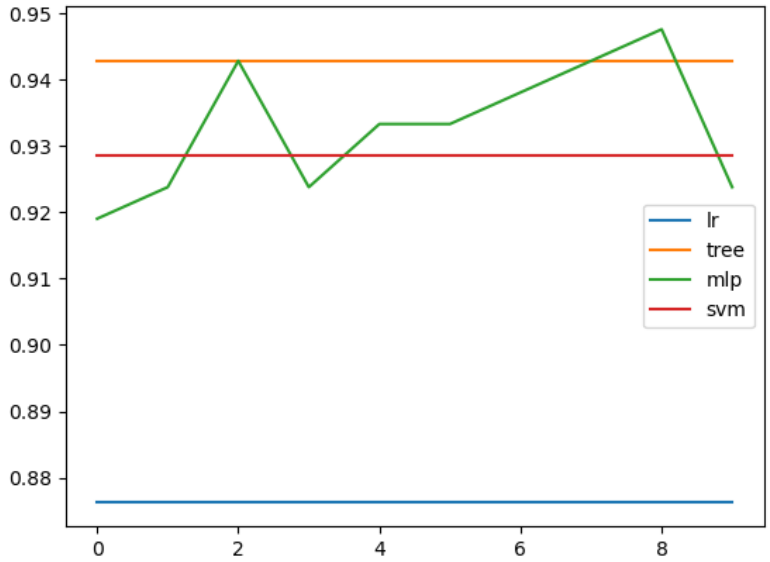
值得一提的是，第二类误差减小的等距选取并没有恢复到更接近原图形的样子，而自适应结点由于曲线的复杂性，第二类误差甚至增大了。这启示着，曲线复杂时，更考虑局部的样条插值才是最能还原实际情况的。

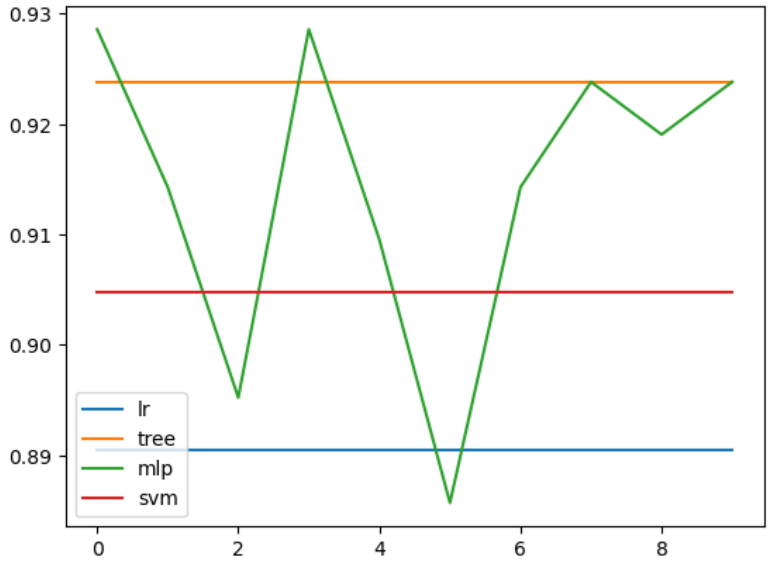
神经网络多分类：

[这次实验主要做的是第一个任务，第二个任务只是简单对昆虫分类写了点调库代码]

1、机器学习模型对比

对于分类模型，有多个机器学习算法可以选择，这里简单将合适参数的回归(LR)、决策树(tree)与 svm 进行对比，在两个例子中分别得到(折线为多次测试结果)：



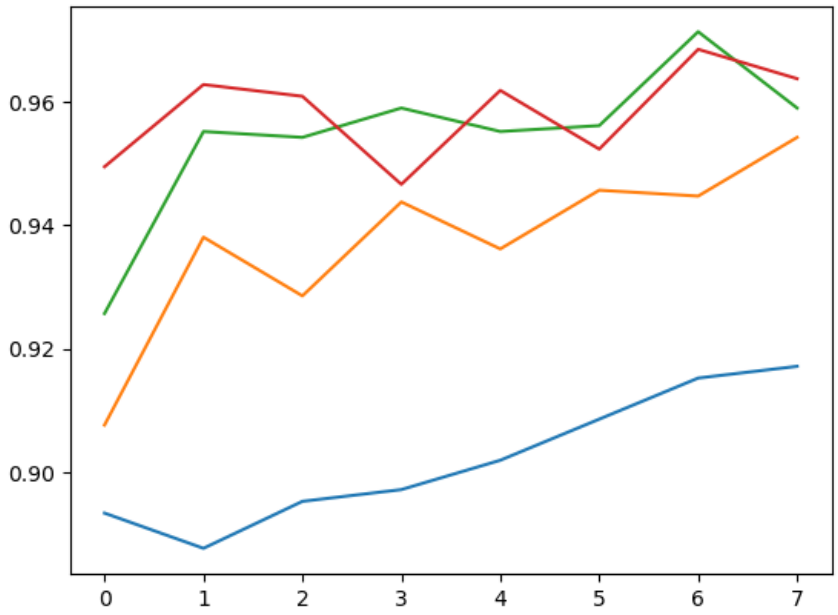


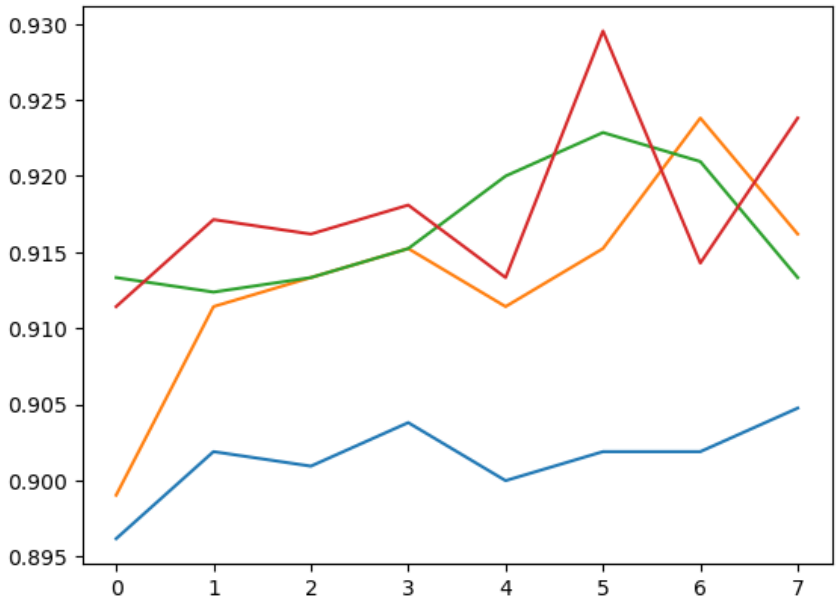
可以看出，与其他网络的稳定相比，神经网络随机初值下会产生更多的扰动，而总体性能则是稳定在 SVM 与决策树中间(这里采取的网络结构是全连接，两隐藏层，各 30 个神经元，迭代至收敛。之后会看到更多参数的对比)。由于本实验的数据集是几乎能精确分割的二维点，决策树效果最好是符合预期的。不过，接下来的调参过程会对比决策树的精确程度数据，给出更好的效果。

此外，在增添了噪声后神经网络结果的稳定程度显著降低，这也意味着其局部最优的多可能性。在这样的简单例子中，事实上是决策树更易于得到稳定良好的结果。然而，神经网络的万能近似特性提供了更多的可能：

2、参数对比

在标准的 Relu 激活函数下，绘制两个例子的参数对比图：

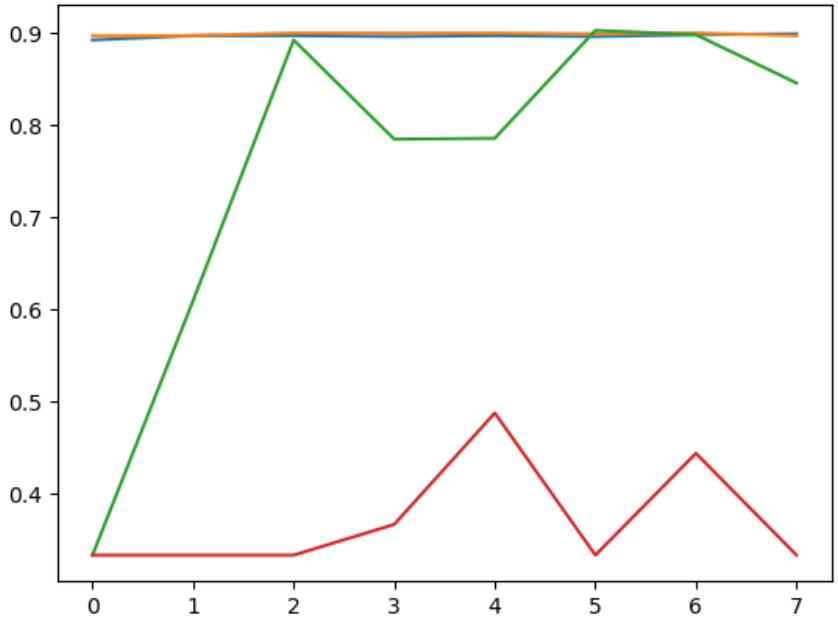




这里，蓝、黄、绿红分别代表 1、2、3、4 层，而每层的神经元个数则是横坐标+1 的七倍，纵坐标是训练五次的平均准确率。

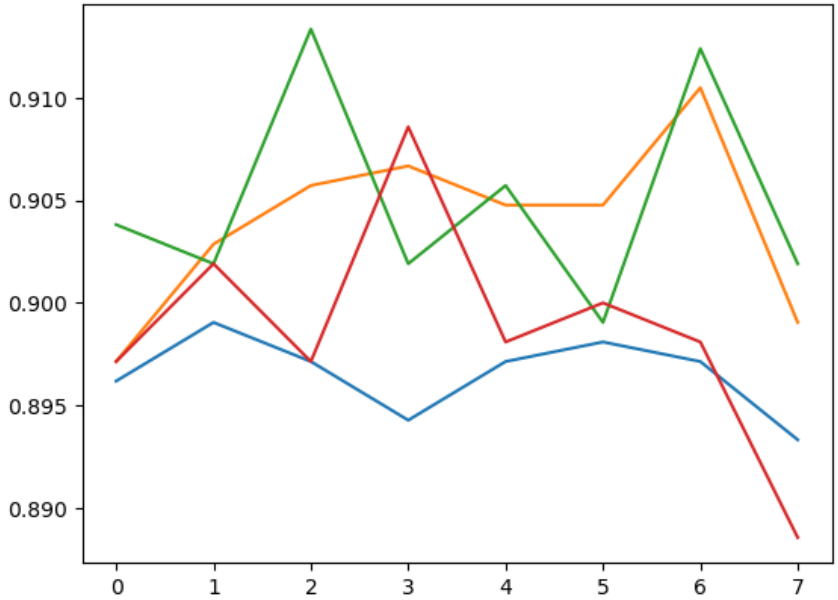
对于不存在扰动的第一个例子，可以鲜明看到结果随着层数的上升而变好，而宽度总体影响并不大，只要不是过小都可以造成提升。于是，只要层数够多、够宽，可以达到稳定 95%以上的准确率，超过了决策树的结果。

然而，第二个例子中，只要层数达到两层，提升就不再明显，也无法达到决策树的结果。这是由于扰动的存在，层数过高、过深容易引起过拟合，极端情况是第二个例子中 logisitic 函数作为激活函数时：



层数提升到三层后，结果非常不稳定，四层甚至带来了极差的结果。

第二个例子中 tanh 作为激活函数时效果如下：



同样，当深度、宽度升高时，结果出现了变差的趋势。

从这样的对比也能看出，logistic 函数由于 0-1 附近的很高斜率，对过拟合非常敏感，而 tanh 的指数也带来了较为敏感的结果。一般来说， Relu 是更加稳定的。

讨论与总结：

在机器学习与深度学习的实践中，为了减少误差的影响、避免过拟合，各种的正则化手段是必要的。而将一个问题在给定的标准下视为优化问题，并且将更多规范化约束作为正则化项进行求解，也是最优化过程中常用的手段。

这次，对第一个任务，我尝试通过自己的想法进行目标误差估算与正则化的求解(的确难算)，取得了不错的效果，也验证了切比雪夫参数化+正则化与样条插值对不同的问题的适用性，收获不少。

*参考文献：数学建模 ppt、机器学习/深度学习与数值分析相关课程内容

Lab 4 微分方程模型 实验报告

PB20000296 郑滕飞

摘要

本文基于图像梯度操作实现了图像选区的泊松融合，并尝试了正则化、颜色匹配等改进方式，对比了其效果。

代码结构

main.m 主函数，输入图像名称实现直接拼接与各种泊松融合对比
pasting.m 单通道直接拼接的实现
poisson_editing.m 泊松融合的实现
match.m 图像颜色风格匹配(取自 Lab 1)

泊松融合：

1、模型建立

为了保证选区正确融合到给定图片中，我们采用梯度操作的方式进行建模，也即考虑如下的优化问题：假设原图为 f，目标图片为 g，区域记为 D，则目标为优化：

$$\min_h \iint_D |\nabla h - \nabla f|^2, \text{s.t.}\,(h-g)|_{\partial \mathrm{D}} = 0$$

下面对这个式子进行简单的解释。由于融合时只能控制选区内部，为保证边界连续性，需要满足新的区域边界处值与目标图相同，这也就是限制条件的含义。而为了保证融合后能自然看出原图的特征，需要保留原图的梯度特征，这也就是优化目标的含义：梯度与原图梯度尽量接近。

在实际计算时，由于图片是离散的，分别对每个通道操作，梯度变为差分，也即要优化(这里定义的 D 为开，与其边界不交，N_p表示 p 的邻域，二维图像中为上下左右)：

$$\min_h \sum_{p\in D, q\in N_p} \left(h_p - h_q - f_p + f_q\right)^2, \text{s.t.}\,(h-g)|_{\partial \mathrm{D}} = 0$$

对 D 中的每个h_p进行求导后，等式两边同除以二，得到最小值须满足

$$\forall p \in D, 4h_p - \sum_{q\in N_p \cap D} h_q = \sum_{q\in N_p} \left(f_p - f_q\right) + \sum_{q\in N_p \cap \partial D} g_q$$

值得注意的是，由于边界的h_p并非变量，会以g_q出现在右侧的常量处。

在 MATLAB 中，这样的系数矩阵构造如下：

```
A = 4 * eye(s);
b = zeros(s, 1);
for i = 1:s
    x = coor(1, i);
    y = coor(2, i);
    b(i) = 4 * I(x, y) - I(x - 1, y) - I(x + 1, y) - I(x, y - 1) ...
        - I(x, y + 1);
    if (region(x - 1, y) > 0)
        A(i, region(x - 1, y)) = -1;
    else
        b(i) = b(i) + J(x - 1 + bias_x, y + bias_y);
```

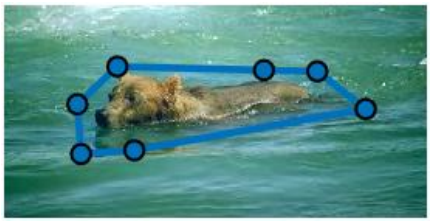
```
        end
        if (region(x + 1, y) > 0)
            A(i, region(x + 1, y)) = -1;
        else
            b(i) = b(i) + J(x + 1 + bias_x, y + bias_y);
        end
        if (region(x, y - 1) > 0)
            A(i, region(x, y - 1)) = -1;
        else
            b(i) = b(i) + J(x + bias_x, y - 1 + bias_y);
        end
        if (region(x, y + 1) > 0)
            A(i, region(x, y + 1)) = -1;
        else
            b(i) = b(i) + J(x + bias_x, y + 1 + bias_y);
        end
    end
```

其中，coor 是点的排序到选区内部点的映射，而 region 中不在选区位置的点是 0，选区中的每个点存储点的排序。例如，若1 × 5数组的选区是(1,2)、(1,3)、(1,4)，则 region 中存储[0, 1, 2, 3, 0]，coor 中存储[[1, 1, 1], [2, 3, 4]]。bias_x 与 bias_y 代表原始图片坐标到目标图片坐标的偏移。

这里的四个 if、else 即表示分别考察四个方向是否到达边界，到若到达则在常数项添加系数，否则在系数矩阵添加。这样构造矩阵后，进一步对 A 进行稀疏化(若直接用稀疏矩阵构造，插入过程复杂度高，这里选择先用稠密矩阵再稀疏求解)，并得到系数 c = A \ b。计算完成后，利用 coor 将选区形状恢复，并平移到目标图片上的对应位置即可。

2、基本效果

主函数中，确定图片后，可利用 MATLAB 自带的 draw_polygon 方法在图上进行交互选区：

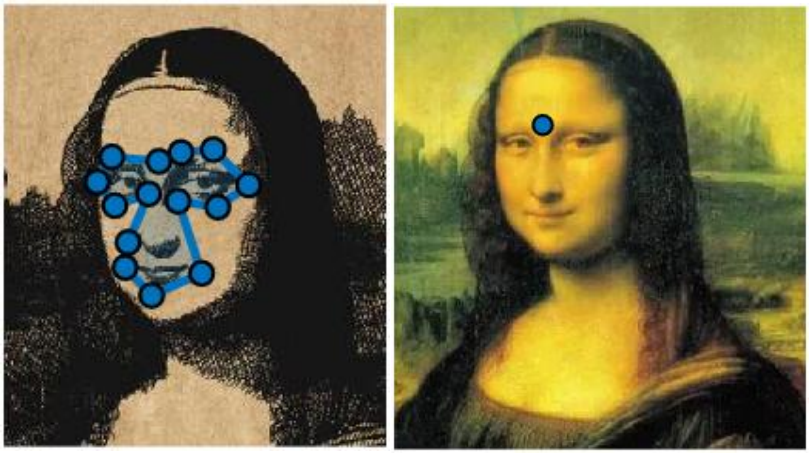


接着利用 draw_point 工具选择第二个图像中的插入点(这个点对应的是多边形第一个顶点平移到的位置，用它确定 x、y 的偏移)：

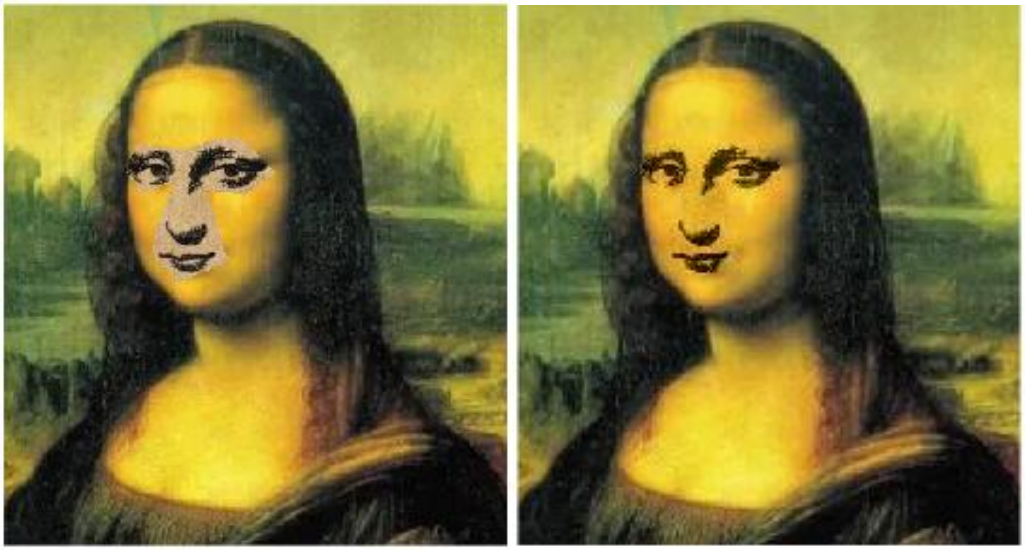
即可输出直接拼接与泊松融合的效果对比：

可以发现，经过泊松融合后，原图融入目标图片要自然很多，虽然细看水波还是有一定不和谐，但总体基本无法分辨。

以下为另一个例子，选区与移动位置为：



最终效果



若进行反向的“换脸”，可以得到：

3、正则化

从第一个、第三个例子中，虽然进行了融合，但仍然有很明显的不和谐感。第一个例子是由于，在主色调只有蓝色的大海中，原来黄色、绿色的区分被一并变得偏蓝了，导致熊看起来颜色很浅。第三个例子类似，由于两个图像本身的**颜色风格**不同，单纯记录梯度很可能仍然不和谐，亦忽略了原图除边界的信息。为此，我引入了正则化项

$$\lambda \sum_{p \in D} (h_p - g_p)^2 + \mu \sum_{p \in D} (h_p - f_p)^2$$

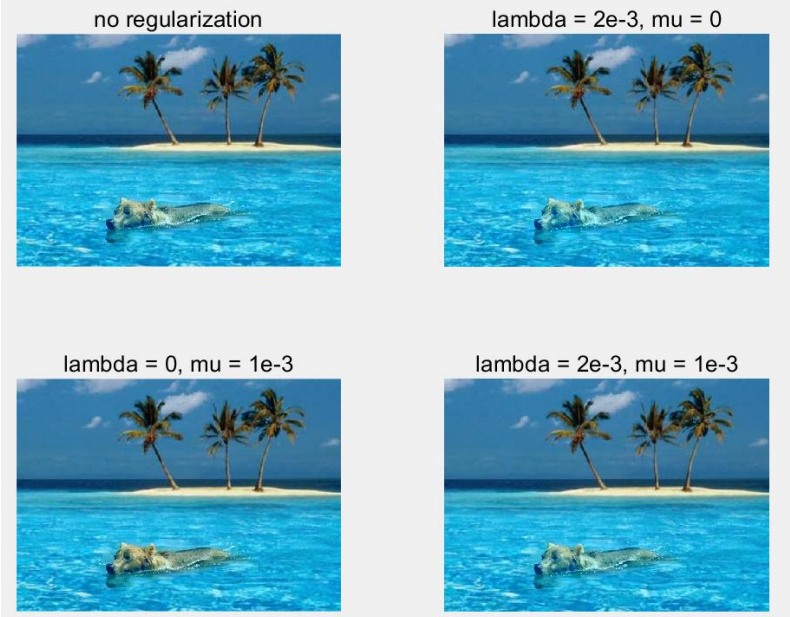
由式子可直接看出，第一项代表与目标图片的相似度，第二项则代表与原图的相似度。在求导后，它们会反应在 A 的对角线与 b 的梯度部分中，也即构造矩阵时 A 初始应为：

```
A = (4 + lam + mu) * eye(s);
```

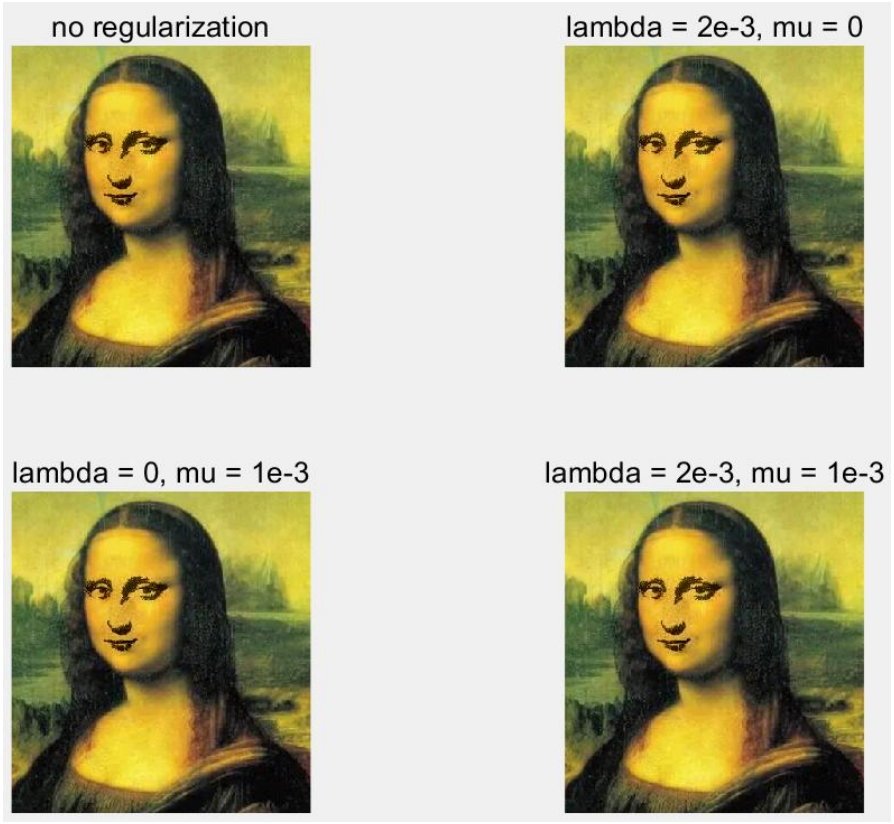
而 b(i)则应初始化为：

```
b(i) = (4 + mu) * I(x, y) - I(x - 1, y) - I(x + 1, y) - I(x, y - 1) ...
          - I(x, y + 1) + lam * J(x + bias_x, y + bias_y);
```

根据测试，正则化系数一般取为 1e-3 量级较为合适。对第一个例子，效果为：



可以发现，`lambda` 增大时熊的颜色更浅，更接近背景，而 `mu` 增大时前景颜色则更为凸显。后两个例子效果分别为：



对颜色本来差别不大、原始效果就很不错的第二个例子，效果并不明显，而对第三个例子，即使增大了 lambda，依然能看到颜色的不自然。因此，我们需要寻求更有效的措施。

4、颜色匹配

之前提到，这样的不自然主要是图像的颜色风格不同导致的。具体来说，由于两张图三个通道边界处与下降情况的不一致，导致了合并后原本的颜色依然凸显，没有完全融入。诚然，考虑实际三维点的距离可以解决这个问题，但这又会导致计算量的大幅增加，最终，我选择利用之前完成的颜色风格匹配代码，具体操作为：

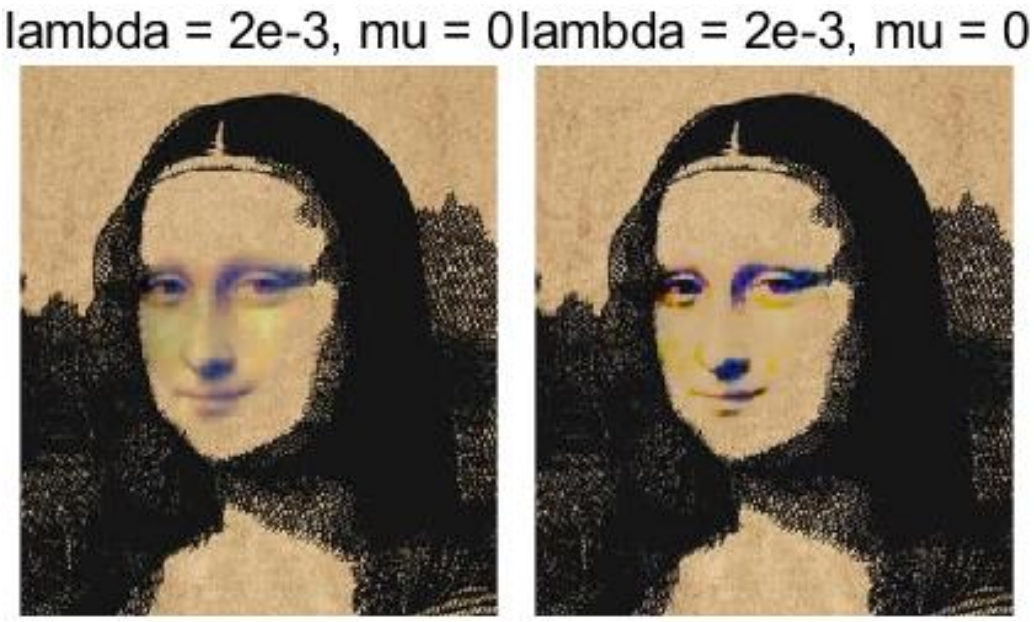
```
T = match(J, I, 0, 0);
for i = 1:3
    K(:,:,i) = poisson_editing(T(:,:,i), I(:,:,i), ...
        region, bias_x, bias_y, 0, 0);
end
K = match(K, J, 0, 0);
imshow(K);
```

也就是说，先将目标图片的颜色风格匹配到原图，利用匹配后的图片进行泊松融合后，再匹配回目标图片。这里另一件值得一提的事是，之所以不直接将选区的颜色风格匹配到目标图片，是因为我们需要保证在原图的环境下进行泊松融合，以使选区更加自然。

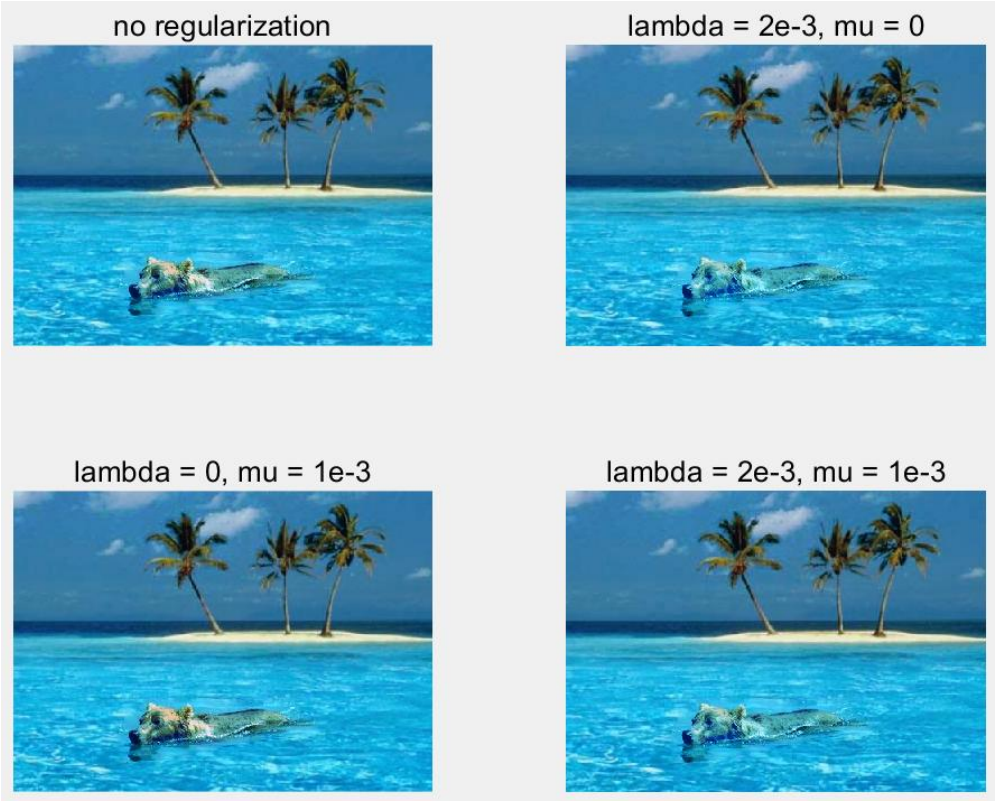
下面直接展示第三个例子的效果(添加匹配过程后仍然可以选取正则化参数)：

可以发现，之前一直无法处理的第三幅图，在颜色匹配后融入就自然了很多。虽然一些黄色的像素依然显示了原图的信息，但在调高 lambda 后也可以清除，这里给出直接的对比，

左侧为直接融合，右侧为匹配后融合，右侧基本保留了最重要的五官特征，而面部基本完全融入了目标图片：

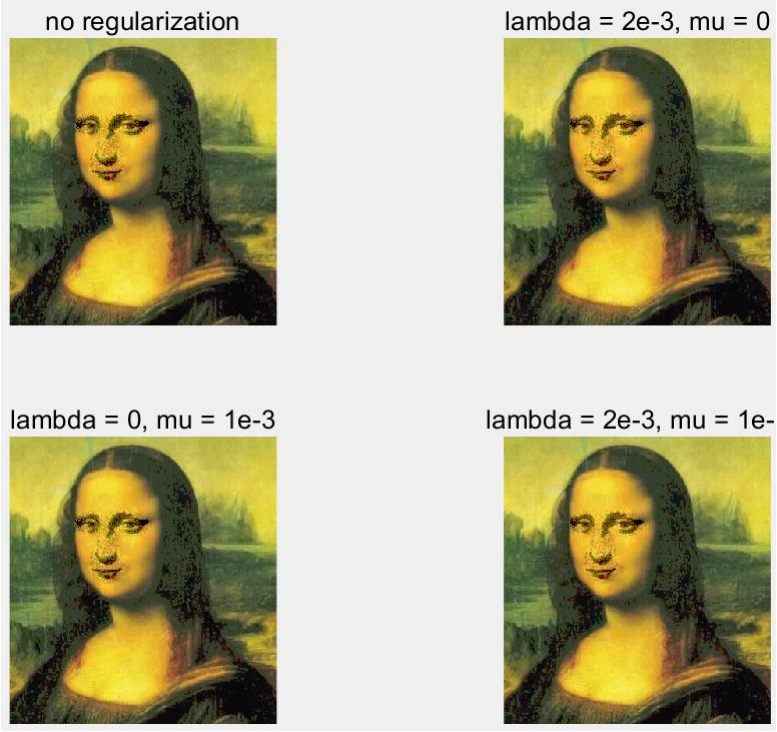


对第一个例子，匹配后融合的效果如下：



在这个例子中，原图的确更加明显了，为了保留熊的特征，将 `mu` 调至 `1e-3`，与最初直接融合相比，已经基本不存在失真了。

不过，颜色风格迁移的处理未必一定会有好的效果，例如对第二个例子：



由于原图有很多细碎的纹理质地，在颜色迁移中变形，反而不如直接进行泊松融合。

讨论与总结：

在数学实验学过一遍原理之后，这次实验的难度并不高，不过探索优化方法、以至利用之前的实验进行优化的思路还是颇有意思的。其实关于优化结果还有更多的想法，例如添加通道均衡的正则化项等，不过考虑到完成时间与时间复杂度，还是以单通道直接进行较为便利。

*参考文献：数学建模 ppt、数学实验相关课程内容