

Interactive Digital Photomontage

计算摄影学课程项目报告

林昭炜 - 2020.6

3170105728

1 简介

生活中并不是每一张照片都是完美的，我们常常需要去利用多张图片合成一张图片从而消除瑕疵。常见的应用场景有家庭照片中不是所有人都在看镜头，所以需要合成一张大家都在看镜头笑的照片。

Interactive Digital Photomontage¹ (简称论文) 提出了一个框架，通过用户选定的标签对图像进行分割，分割完成之后再梯度域上进行图像的融合，最终获得理想的效果。

在这次项目中，我基于课程提供的代码框架，利用 OpenCV 实现了 GUI，实现了论文中描述的算法，尝试了使用 Laplacian 代替 Edge。然后改进了代码框架里面的源码，让计算时间节省了近 90%。

2 实验环境

编译环境: Visual Studio 2019, C++17

运行环境: Windows 10, 16GB RAM, i7 7700HQ

OpenCV: 4.1.1

Eigen: v3.3.3 Nuget包管理安装

3 算法框架

论文的算法是允许用户在一系列源图像中，指定一些像素点，而这些像素点作为我们初始的标签，然后利用 Graph Cut 对这一系列图像进行标签优化，最终得到一张带有选择那张图片作为合成算法的标签的图片。然后进行梯度域的融合 (Gradient Domain Fusion) 获取最终的图像。

2.1 Graph Cut

首先谈一下如何应用 Graph Cut, Graph Cut 一般分为两个循环，内循环在接收到两个标签的时候，它会选择使得惩罚函数最小的标签，外循环则遍历所有的标签。

而我们的惩罚函数定义如下：

$$C(l) = \sum_p C_d(p, L(p0)) + \sum_{p,q} C_i(p, q, L(p), L(q)). \quad (1)$$

其中 $\sum_p C_d(p, L(p0))$ 是数据项, $\sum_{p,q} C_i(p, q, L(p), L(q))$ 是平滑项。

数据项负责表达了用户指定的一些像素的作为标签的时候，我们应该给其他像素多大的惩罚。而平滑项则表达了像素与像素之间的距离。如 Fig.1 所示， s, t 分别代表两个 Label, 而红色和蓝色的边代表这个像素和标签之间的距离。黄色的边则代表了平滑项。我们的目标是从黄色的边缘中找到一条切割线从而切割图像。

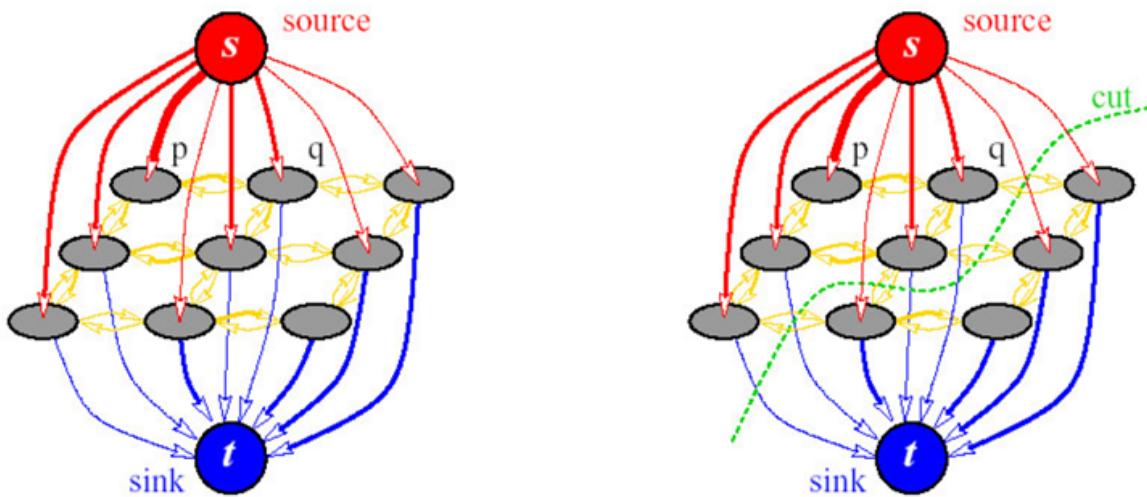


Fig. 1

2.1.1 Max Flow

解决 Graph Cut 的一个基本的思想是 Max Flow 这个图论算法, Max Flow 解决的问题是我们从 Source 到 Sink 之间，我们用不同粗细的水管相连，我们要如何找到最大的水流。Max Flow 基本的算法步骤如下：

```

1 | MaxFlow(graph)
2 |   while true
3 |     path = find a path in graph // bfs/dfs
4 |     if path not found:
5 |       break
6 |     calculate residue along path
7 |   end

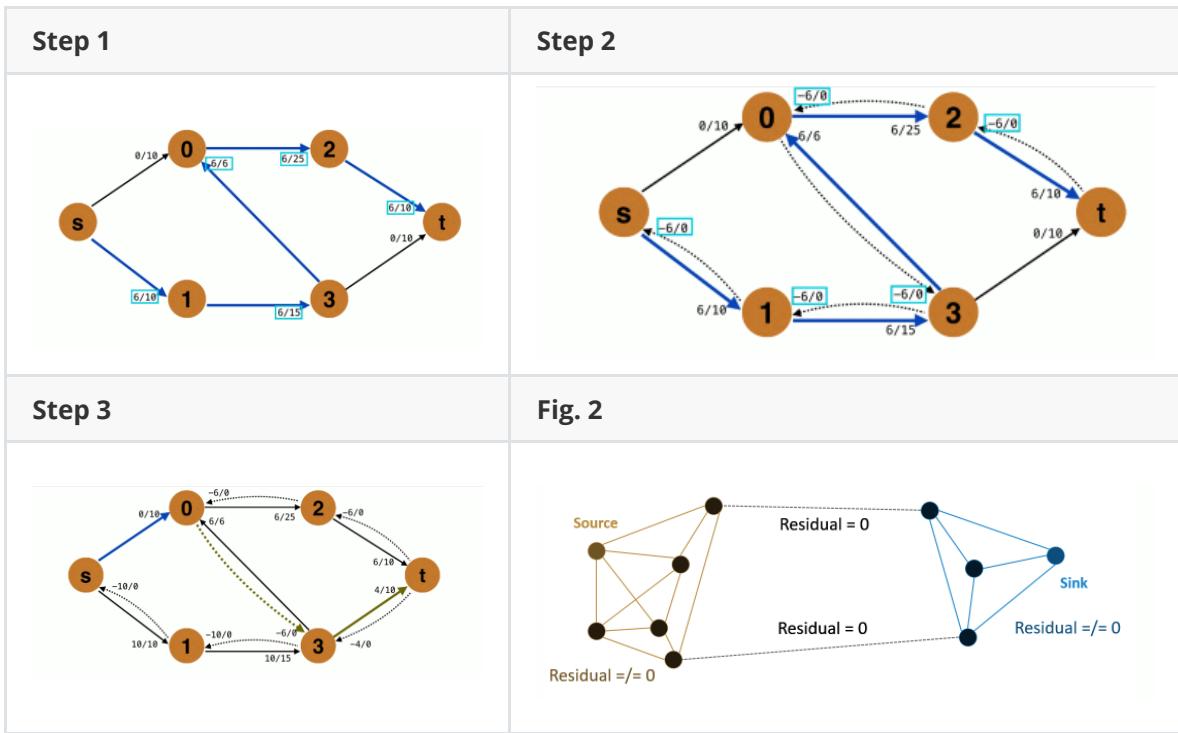
```

一个具体的例子可以看图片中的 **Step 1 - Step 3**,

Step 1 找到一条边缘, 图中蓝色线段就是一条可行的边缘

Step 2 构建 Residual Graph, 即沿着找到的路径计算 residual

Step 3 重复 Step1, 注意这里找到的路径中包含了虚线(黄色虚线), 代表我们反悔了之前的决定。



2.1.2 Min Cut

Max Flow 和 Min Cut 本质是一样的问题, 我们可以把所有的边缘分为残差为 0 (Residual=0) 的边和残差为 1 的边, 我们认为切断残差为 0 的边就能断开 Source 和 Sink 的连接, 这个可以用反证法证明: 如果切断这些还是有连接, 那么所有的边缘残差都不为0, 那么一定有一条从 Source 到 Sink 的非零路径。

同样的我们证明 Source 和 Sink 最小的 Cut 一定是我们之前的 Cut 办法: 从 Source 到 Sink 的最大流是 Residual = 0 的边之和, 不妨设这个和为 T , 那么我们要切割掉的水管水流之和一定大于等于 T , 如果去切割 Residual 不等于 0 的边, 我们切割的水管大小(权重)大于管内水流, 那么我们切割的总权重一定大于 T .

2.2 数据项

公式 (1) 中的数据项可以有很多种不同的定义方式, 在这里我们列举一下:

指定图像 (Designated Image): 当用户选中一个像素的时候, 我们给他低的惩罚, 其他一律高的惩罚。见 Fig. 3

指定颜色 (Designated Color): 我们知道了用户选中的像素的颜色集合, 我们可以用其他像素和这个集合比较, 我们可以选择距离这个集合某个像素最大的或者距离这个集合某个像素最小的。见 Fig. 4

指定亮度 (Designated Color): 我们知道了用户选中的像素的亮度集合, 我们可以用其他像素的亮度和这个集合比较, 我们可以选择距离这个集合某个像素最大的或者距离这个集合某个像素最小的。

似然概率 (Likelihood): 我们把所有颜色分成比如 20 个类别, 然后我们去统计每一个类别中, 用户选中的颜色在这个类别里的数量和这个类别里所有的像素个数之比, 这个是每个像素的似然概率, 我们可以用来作为数据惩罚项。Fig. 5 演示的是一个类似于颜色直方图的类别柱状图, 淡蓝色代表属于这个类别但是用户没选中的, 深蓝色是属于这个类别并且用户选中了它。

擦除惩罚 (Eraser): 这是一个迭代的目标, 就是当前这个像素和之前合成的图像之间的区别, 这个适用于用户每次增加一个label, 我们基于上一次的计算结果来得到数据项。

除此之外还有很多种方法去定义数据项。

Fig. 3

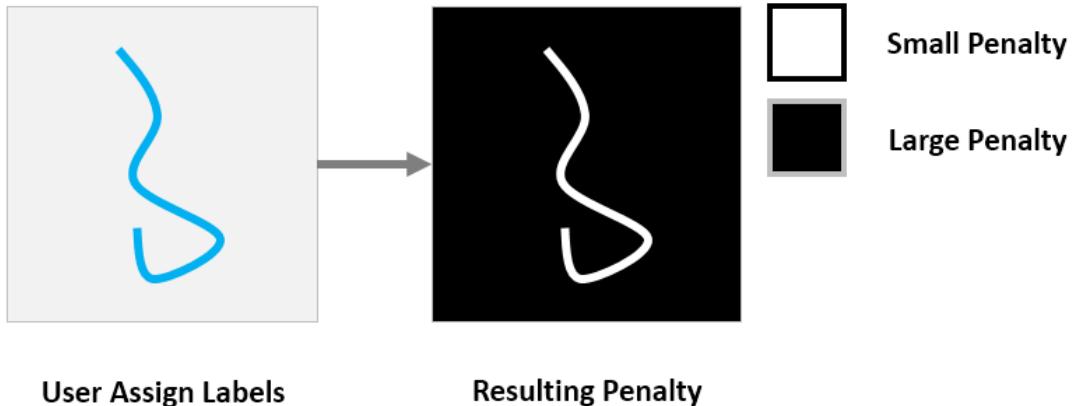


Fig. 4

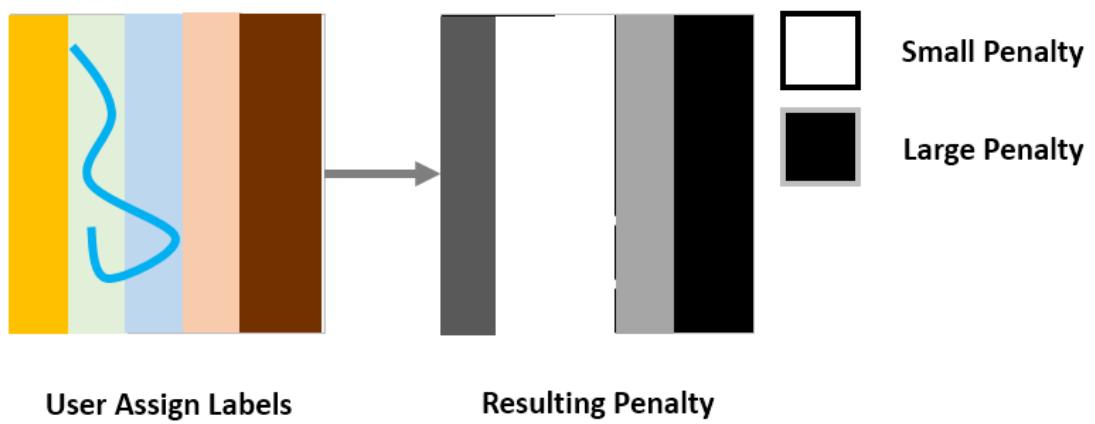
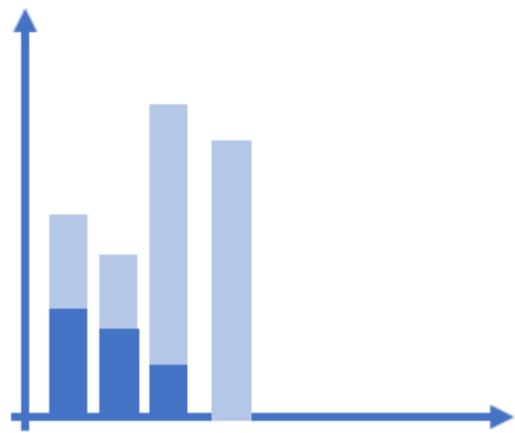


Fig. 5



2.3 平滑项

公式(1)中的平滑项也有很多定义方式。

Math Colors

最简单的是两个颜色的欧式距离，注意这里我们有两个 Label，我们对比的是两张图片对应两点的欧式距离之和，如 Fig. 6 所示，对应的形式化表达是：

$$||S_{L(p)}(p) - S_{L(q)}(p)|| + ||S_{L(p)}(q) - S_{L(q)}(q)|| \quad (2)$$

这里 p, q 分别对应两个像素点， $L(p), L(q)$ 代表两个像素点所在的图片， $S_{L(p)}(q)$ 代表 p 所在的图片对应的 q 位置的像素。

注意当 $L(p) = L(q)$ 的时候，我们认为惩罚是 0。

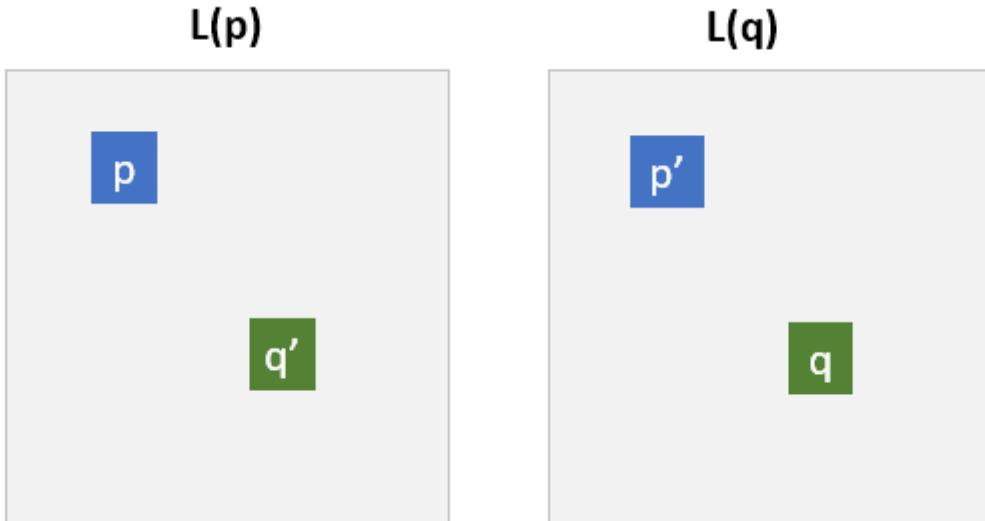


Fig. 6

Match Gradients

这是计算梯度的欧式距离，这和颜色的比较是一样的，除了梯度其实有x和y这两个梯度方向，所以我们不是比较 RGB 这样三维向量的欧式距离，而是六维向量之间的欧式距离。我们可以用 Sobel 算子计算结果。对应的公式如下：

$$||\nabla S_{L(p)}(p) - \nabla S_{L(q)}(p)|| + ||\nabla S_{L(p)}(q) - \nabla S_{L(q)}(q)|| \quad (3)$$

Match Gradients & Colors

梯度和颜色平滑相，其实是把两个上面两个加起来即可，但是这会造成不稳定的实验结果，主要原因是 Gradient 的复制相比 Colors 不是特别稳定，所以有时候会偏向 Gradient 的结果，有的会偏向 Colors 结果。

2.4 Inertia

有一个 Intuition 是，当用户画好几个像素的时候，一般用户会希望这些像素周围的点也是同样的 Label。对此我们可以所以我们可以给靠近用户 Label 的像素更低的惩罚，给较远的高惩罚，保证了用户绘制的周围区域也被打上相同的 Label。如 Fig. 7 所示

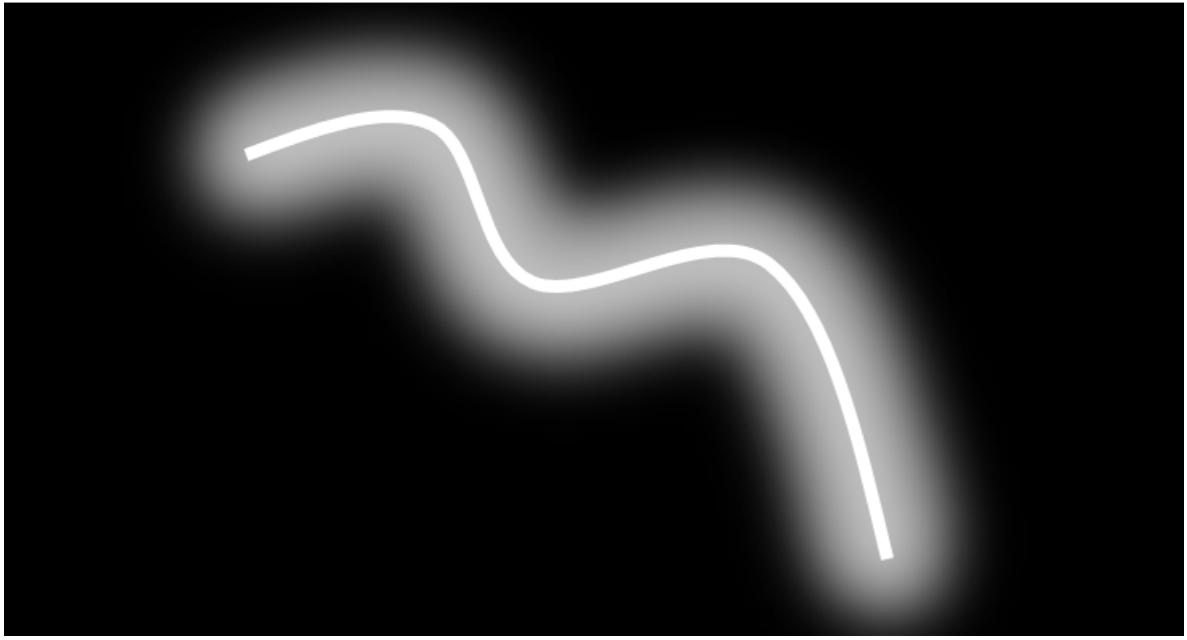


Fig. 7

2.5 Gradient Domain Fusion

Graph Cut往往能呈现一个非常优质的效果，但是如果我们仔细观察图片 Fig. 7 对比 Fig. 9 的分割图，我们还是可以看出由于两张图片的亮度的差异，导致在边缘的地方有些许的合成痕迹，所以我们希望能够消除这个问题。消除问题的核心想法是：我们在边缘上的点，其梯度应该和原来的图像保持一致，所以我们希望能满足一个边界条件，这个边界条件被叫做纽曼边界条件 (Neumann Boundaries)：

$$\begin{aligned} v_{x+1,y} - v_{x,y} &= \nabla I_x(x, y) \\ v_{x,y+1} - v_{x,y} &= \nabla I_y(x, y) \end{aligned} \quad (4)$$



Fig. 7

Fig. 8



Fig. 9

要解这个方程，我们可以列一个巨大的矩阵，因为这个矩阵是稀疏的，所以我们可以用共轭梯度法去求解矩阵。在 Lab 3 中我实现了共轭梯度法，这里再次介绍一下原理：

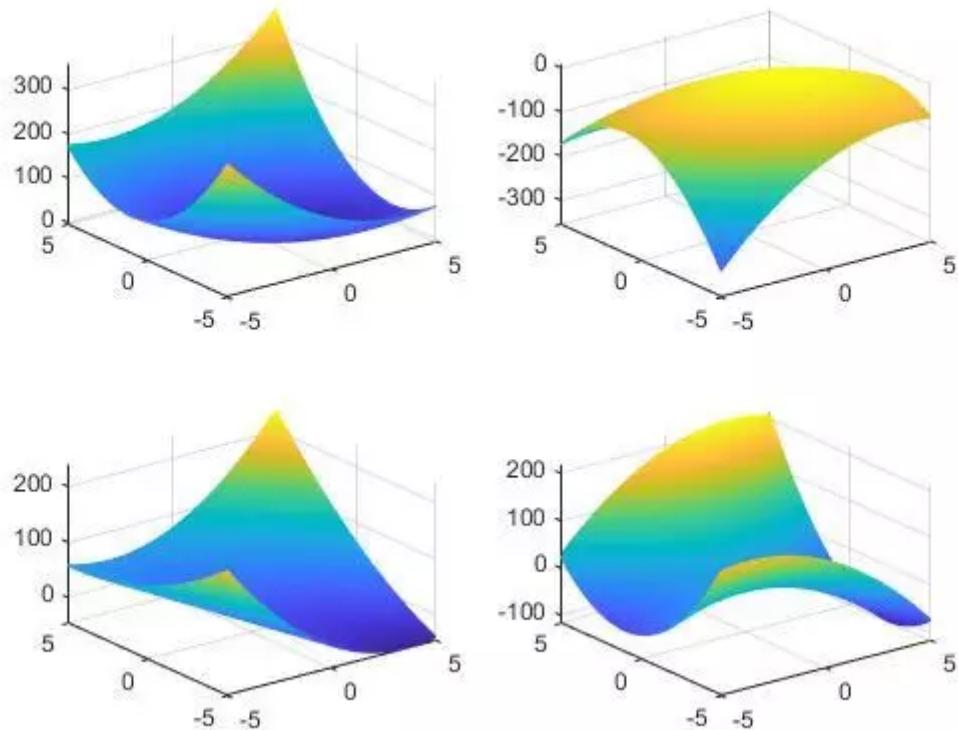
求解 $Ax = b$ 的时候，我们可以转化为对二次型 $f(x) = \frac{1}{2}x^T Ax - b^T x + c$ 导数为0的求解。

即我们有：

$$\frac{df(x)}{dx} = Ax - b. \quad (5)$$

这里只要暴力把矩阵和向量展开相乘即可证明，不在此赘述。

二次型也影响了这个方法的局限性：下图依次时二次型在正定、负定、半正定和不定的情况：



<https://www.jianshu.com/p/3ca4971a1a19>

可以发现当矩阵不定的时候有鞍点，无法通过导数为0求极值。

然后共轭梯度法的关键在于希望每一次迭代每个方向都走到了极致, 也即是说寻找极值的过程中绝不走曾经走过的方向。² 基于这个想法, 我们在每一次前进的时候和前进后误差正交, 这就意味着我们有一个方向不需要再走了。把下一步方向记作 r_t , 目标为 x^* , 那么误差时 $e_t = x^* - x_t$, 那么我们误差与上一步方向正交意味着 $r_{t-1}^T e_t = 0$, 但是我们不知道方向, 所以我们用共轭正交代替 $r_{t-1}^T A e_t = 0$.

首先我们确定步长:

$$\begin{aligned} e_t &= x^* - x_t \implies e_{t+1} = x^* - x_{t+1} = e_t + x_t - x_{t+1}, \quad (6) \\ r_t^T A e_{t+1} &= r_t^T A [e_t + x_t - x_{t+1}] = r_t^T A [e_t - \alpha_t r_t] = 0, \\ &\implies \alpha_t = \frac{r_t^T A e_t}{r_t^T A r_t}, \text{ 而 } \nabla f(x) = Ax - b, \\ \implies \alpha_t &= \frac{r_t^T A (x^* - x_t)}{r_t^T A r_t} = \frac{r_t^T (b - Ax_t)}{r_t^T A r_t} = -\frac{r_t^T \nabla f(x_t)}{r_t^T A r_t}. \end{aligned}$$

而方向的确定希望每次都是共轭正交, 我们可以通过 $-\nabla f(x_t)$ 确定我们的梯度方向, 我们可以很方便的用施密特正交化把之前走过的方向消除从而达到与之前正交:

$$r_t = -\nabla f(x_t) + \sum_{i < t} \frac{r_i^T A \nabla f(x_t)}{r_i^T A r_t} r_i. \quad (7)$$

所以我们更新向量就是: $x_{t+1} = x_t + \alpha_t r_t$.

4 实验细节

4.1 Data Term

在实验中我使用了最简单的 Designated Image 的算法, 因为它的效果较好, 所以我没有选择其他方案:

```

1 // 用户指定的像素点
2 if (label != PhotoMontage::undefined)
3 {
4     if (label == 1) {
5         return 0.0;
6     }
7
8     // 这个像素点和用户指定的 label 不一样
9     return large_penalty;
10}
11
12 return large_penalty;

```

4.2 Smooth Term

Smooth Term 中我提供了多种不同的 Smooth Term。

4.2.1 Colors

颜色是直接最简单的计算两次两个像素之间的距离:

```

1 if constexpr (obj == PhotoMontage::kMatchColors ||
2     obj == PhotoMontage::kMatchColorsAndEdges ||
3     obj == PhotoMontage::kMatchColorsAndEdges)
4 {
5     // 两个像素在两张图片上的欧氏距离
6     auto dist = [&Images, &coords, lp, lq](int idx1, int idx2)
7     {
8         auto& a = Images[lp].at<Vec3b>(coords[idx1], coords[idx1 + 1]);
9         auto& b = Images[lq].at<Vec3b>(coords[idx2], coords[idx2 + 1]);
10
11         Vec3d c = a - b;
12         auto res = sqrt(c[0] * c[0] + c[1] * c[1] + c[2] * c[2]);
13         return res;
14     };
15     X = dist(0, 2) + dist(2, 0);
16 }

```

4.2.2 Gradients

在计算梯度的时候我用的是 Sobel 算子, 在 OpenCV 中的 `filter2D` 计算 x 和 y 方向的梯度:

```

1 Mat sobel_x({ 3, 3 }, {
2     -1.f, 0.f, 1.f,
3     -2.f, 0.f, 2.f,
4     -1.f, 0.f, 1.f,
5 });
6
7 Mat sobel_y({ 3, 3 }, {
8     1.f, 2.f, 1.f,
9     0.f, 0.f, 0.f,
10    -1.f, -2.f, -1.f,
11 });
12
13 for (int i = 0; i < n_imgs; i++)
14 {
15     cv::Mat dx, dy;
16     cv::filter2D(Images[i], dx, CV_32FC3, sobel_x);
17     cv::filter2D(Images[i], dy, CV_32FC3, sobel_y);
18     extra_data.ImagesDx[i] = dx;
19     extra_data.ImagesDy[i] = dy;
20 }

```

然后再计算penalty 的时候和颜色类似, 也是计算欧氏距离:

```

1 if constexpr (obj == PhotoMontage::kMatchGradients ||
2     obj == PhotoMontage::kMatchColorAndGradients)
3 {
4     auto& ImDx = ptr_extra_data->ImagesDx;
5     auto& ImDy = ptr_extra_data->ImagesDy;
6
7     auto dist = [&ImDx, &ImDy, &coords, lp, lq](int idx1, int idx2)
8     {
9         auto& ax = ImDx[lp].at<Vec3f>(coords[idx1], coords[idx1 + 1]);
10        auto& ay = ImDy[lp].at<Vec3f>(coords[idx1], coords[idx1 + 1]);
11
12        auto& bx = ImDx[lq].at<Vec3f>(coords[idx2], coords[idx2 + 1]);

```

```

13     auto& by = ImDy[lq].at<Vec3f>(coords[idx2], coords[idx2 + 1]);
14
15
16     Vec3d cx = ax - bx;
17     Vec3d cy = ay - by;
18
19     auto res = sqrt(cx[0] * cx[0] + cx[1] * cx[1] + cx[2] * cx[2] +
20                      cy[0] * cy[0] + cy[1] * cy[1] + cy[2] * cy[2]);
21
22     return res;
23 }
24
25     Y = dist(0, 2) + dist(2, 0);
26 }
```

4.2.3 Edges

这里直接使用了 OpenCV 的 Canny 算法检测边缘:

```

1 for (int i = 0; i < n_imgs; i++)
2 {
3     cv::Mat edge;
4     cv::cvtColor(Images[i], edge, cv::COLOR_BGR2GRAY);
5
6     cv::Canny(edge, edge, 200 / 3, 200);
7     extra_data.ImagesEdge[i] = edge;
8 }
```

然后我们再算法里计算 Edge 的差:

```

1 if constexpr (obj == PhotoMontage::kMatchColorsAndEdges)
2 {
3     auto& ImEdge = ptr_extra_data->ImagesEdge;
4     auto dist = [&ImEdge, &coords](int l)
5     {
6         auto& a = ImEdge[l].at<uchar>(coords[0], coords[1]);
7         auto& b = ImEdge[l].at<uchar>(coords[2], coords[3]);
8
9         return abs(a - b);
10    };
11
12    double Z = dist(lp) + dist(lq) + 1;
13 }
```

4.2.4 计算 Penalty

通过之前的计算，我们就有了所有的 Penalty 所需要的数据，我们要做的是把数据合并，所以平滑项完整的如下:

```

1 template<enum PhotoMontage::Objectives obj>
2 double smoothFnT(int p, int q, int lp, int lq, void* data)
3 {
4     // Interactive Panelty (P)
5 }
```

```

6   // P : X    if match colors
7   // | Y      if match gradients
8   // | X+Y   if match color & gradients
9   // | X/Z   if match color & edges
10
11  // ...
12  // 计算之前所需的数据
13  // ...
14
15  if constexpr (obj == PhotoMontage::kMatchColors)
16  {
17      res = X;
18  }
19
20  if constexpr (obj == PhotoMontage::kMatchGradients)
21  {
22      res = X + Y;
23  }
24
25  if constexpr (obj == PhotoMontage::kMatchColorAndGradients)
26  {
27      res = X + Y;
28  }
29  return res;
30 }
```

4.3 共轭梯度

共轭梯度的伪代码如下:

```

1 r = b - A * x
2 p = r
3 k = 0
4 while true:
5     alpha = dot(r, r) / dot(p, A * p)
6     x = x + alpha * p
7     r1 = r - alpha * A * p
8     if len(r1) < epsilon: break
9     beta = dot(r1, r1) / dot(r, r)
10    p = r1 + beta * p
11    r = r1
12 end while
```

这里常用的对向量的操作是线性和: $\mathbf{r} = \mathbf{a} + k\mathbf{b}$, 我们可以把它抽象成函数, 这里使用了 C++17 的函数, 从而实现并行计算加速:

```

1 template<typename T>
2 inline void vecadd(const std::vector<T>& a, const std::vector<T>& b, T scale_b,
3 std::vector<T>& out) {
4     std::transform(std::execution::par,
5                 a.begin(), a.end(), b.begin(), out.begin(),
6                 [scale_b](auto a, auto b) { return a + scale_b * b; });
7 }
```

还有一个就是两个向量的点积: $\mathbf{r} = \mathbf{a}^T \mathbf{b}$, 同样的抽象成函数:

```

1 | template<typename T>
2 | inline double dotProd(const std::vector<T>& a, const std::vector<T>& b) {
3 |     return std::transform_reduce(std::execution::par,
4 |         a.begin(), a.end(), b.begin(), 0.0, std::plus<>(), std::multiplies<>());
5 |

```

4.3.1 关于 Eigen 的思考

我发现自己的实现和 Eigen 的结果有一定的差距，于是我研究了一下 Eigen 实现的源码，Eigen 引入的一个 Preconditioner，Eigen 默认的 Preconditioner 是把矩阵的对角线与残差相乘得到下一个方向。我模仿了 Eigen 有写了一个共轭梯度法的算法，奇妙的是，我的结果和 Eigen 还是不一样，所以我一步步调试，最终发现了问题所在：我们的点积算法有误差，大约是 $10^{-8} \sim 10^{-10}$ 左右的量级，但是这个会随着迭代的次数放大。我反复尝试不同的点积算法，都和 Eigen 不一样，至今我未能找到原因。我怀疑的原因是 Eigen 使用了 AVX 指令加速计算，但是我的编译器未作优化，使用的是普通的浮点乘法运算，因为两个指令的底层 CPU 实现不同，最终造成误差。

但是这个不影响最终结果，所以我没有采用 Eigen 或者我自己仿制的 Eigen 算法，而是使用了 Lab3 中我写的代码。

4.3.2 共轭梯度法的实现

在实现中我修改了 Lab3 中的代码，允许了用户迭代的初始值，所以最终算法如下：

```

1 | std::vector<double> conjugateGradient(const std::vector<double>& b, double epsilon =
2 | 1e-16, int max_iteration = 1000, const std::vector<double>& initialize =
3 | std::vector<double>()) {
4 |     std::vector<double> x(b.size(), 0);
5 |     std::vector<double> r(b.size());
6 |     std::vector<double> r1(b.size());
7 |
8 |     if (initialize.size()) {
9 |         x = initialize;
10 |     }
11 |
12 |     // r(0) = b - Ax
13 |     applyToVector(x, r);
14 |     vecsub(b, r, r);
15 |
16 |     std::vector<double> p = r; // p(0) = r(0)
17 |     int k = 0;
18 |
19 |     std::vector<double> Ap(b.size());
20 |     int cnt = 0;
21 |     double error = 0;
22 |     while (cnt < max_iteration)
23 |     {
24 |         double rlen = veclen2(r);
25 |         applyToVector(p, Ap); // Ap = A * p(k)
26 |         double alpha = rlen / dotProd(p, Ap); // alpha = (r' * r) / (p' * Ap)
27 |         vecadd(x, p, alpha, x); // x(k+1) = x(k) + alpha * p(k)
28 |         vecadd(r, Ap, -alpha, r1); // r(k+1) = r(k) - alpha * A * p(k)
29 |         auto r1len = veclen2(r1); // r(k+1) sufficiently small:
30 |         error = r1len;
31 |         if (sqrt(r1len) < epsilon) break; // break;

```

```

31     double beta = r1len / rlen; // beta = len2(r(k+1)) / len2(r(k))
32     vecadd(r1, p, beta, p);    // p(k+1) = r(k+1) + beta * p(k)
33     using std::swap;
34     swap(r1, r);             //
35     ++cnt;
36 }
37
38 return x;
39 }
```

4.4 梯度域图像融合

在做实验之前我对如何解想方程 (4) 这样的没有什么概念，经过实验我了解到了解法，我们可以构造这样的矩阵，以一张 2×2 的图片为例，我们把图像 $I(x, y)$ 展平成向量：

$$\begin{bmatrix} -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} I(0, 0) \\ I(0, 1) \\ I(1, 0) \\ I(1, 1) \end{bmatrix} = \begin{bmatrix} I_x(0, 0) \\ I_y(0, 0) \end{bmatrix} \quad (8)$$

所以我们的代码第一步是构造这样的稀疏矩阵：

```

1 // 构建求解泊松方程的稀疏矩阵
2 for (int y = 0; y < height - 1; y++)
3 {
4     int idx = y * (width - 1) * 4;
5     for (int x = 0; x < width - 1; x++)
6     {
7         int col_xy = width * y + x;
8
9         // 第 2k 行，在矩阵里表示 v(x+1,y)-v(x,y)
10        int row_xy = 2 * col_xy;
11        int col_x1y = col_xy + 1;
12        NonZeroTerms[idx++] = Eigen::Triplet<double>(row_xy, col_xy, -1); // -v(x,
y)
13        NonZeroTerms[idx++] = Eigen::Triplet<double>(row_xy, col_x1y, 1); // v(x+1,
y)
14        Vec3f grads_x = color_gradient_x.at<Vec3f>(y, x);
15        b(row_xy) = grads_x[channel_idx];
16
17        // 第 2k + 1 行，在矩阵里表示 v(x,y+1)-v(x,y)
18        int row_xy1 = row_xy + 1;
19        int col_xy1 = col_xy + width;
20        NonZeroTerms[idx++] = Eigen::Triplet<double>(row_xy1, col_xy, -1); // -v(x,
y)
21        NonZeroTerms[idx++] = Eigen::Triplet<double>(row_xy1, col_xy1, 1); // v(x,
y+1)
22        Vec3f grads_y = color_gradient_y.at<Vec3f>(y, x);
23        b(row_xy1) = grads_y[channel_idx];
24    }
25 }
```

然后我们把矩阵改成方阵：

```

1 Eigen::SparseMatrix<double>, Eigen::RowMajor> ATA(width * height, width * height);
2 ATA = A.transpose() * A;
3 Eigen::VectorXd ATb = A.transpose() * b;

```

最后调用我们的矩阵API求解:

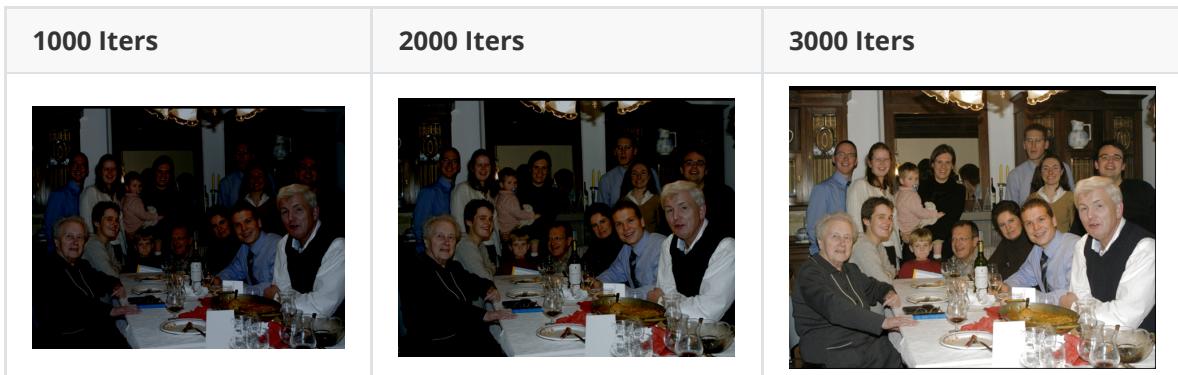
```

1 auto MyMatrix = ConvertFromEigen(ATA);
2 std::vector<double> myATb;
3 myATb.insert(myATb.begin(), ATb.data(), ATb.data() + ATb.rows());
4 auto mysolution = MyMatrix.conjugateGradient(myATb, 1e-10, iterations_, init);

```

4.4.1 梯度域图像融合的优化

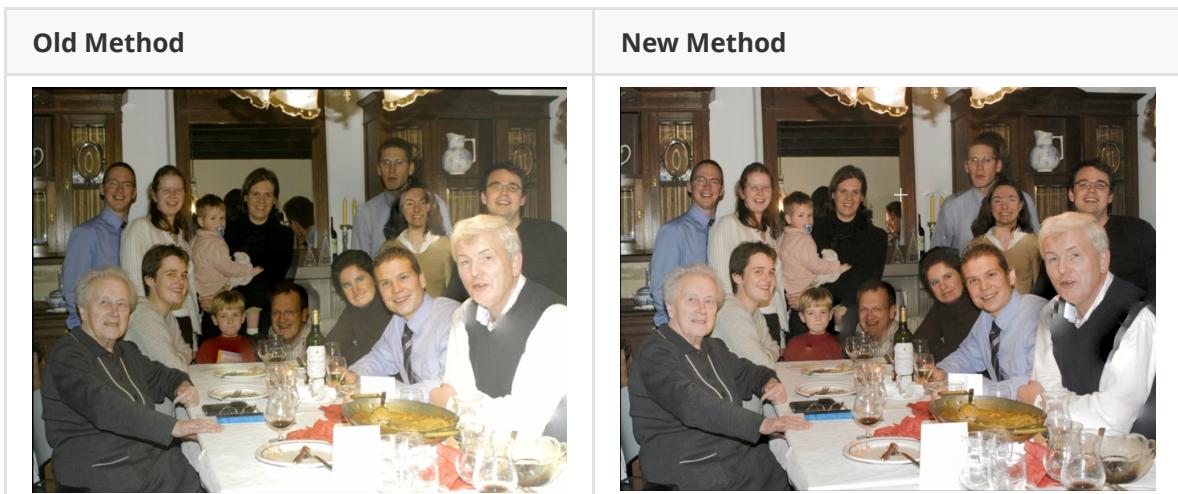
在实验中我发现梯度域图像融合需要很久，甚至要等待长达 5 分钟才有结果，我们可以观察一下不同迭代次数的图像:



我们可以看到问题的根源是图像从 0 开始迭代，这必然会速度慢。其实我们有一个很好的初值，就是 Graph Cut 的结果，这样做好处有很多:

- 迭代次数大大减少，只要 50 次就可以达到之前 3000 次的效果
- 能收敛，不会出现画面整体的问题

第二个好处可能优点抽象，我们可以观察下面两个图片:



左边是旧方法，右边是新方法，两者都是 Graph Cuts 的时候失败了，所以会导致梯度域图像融合也会失败，但是用了新的方法之后图像右下角不会莫名其妙出现光，新方法整体画面的亮度比例是至少正确的。

5 实验成果展示

5.1 GUI 详解



GUI 分为三个部分 左侧按钮区，中间绘画和展示区，右边相册区。

中间的展示区可以用来绘制显示计算结果，或者用户也可以在上面用画笔绘制。

右侧相册区用鼠标滚轮允许用户选择一张源图像。算法会扫描目录下所有的源图像展现在相册区。当用户在中间绘制完成，切换到别的源图像时候，相册会显示用户绘制的标签预览。

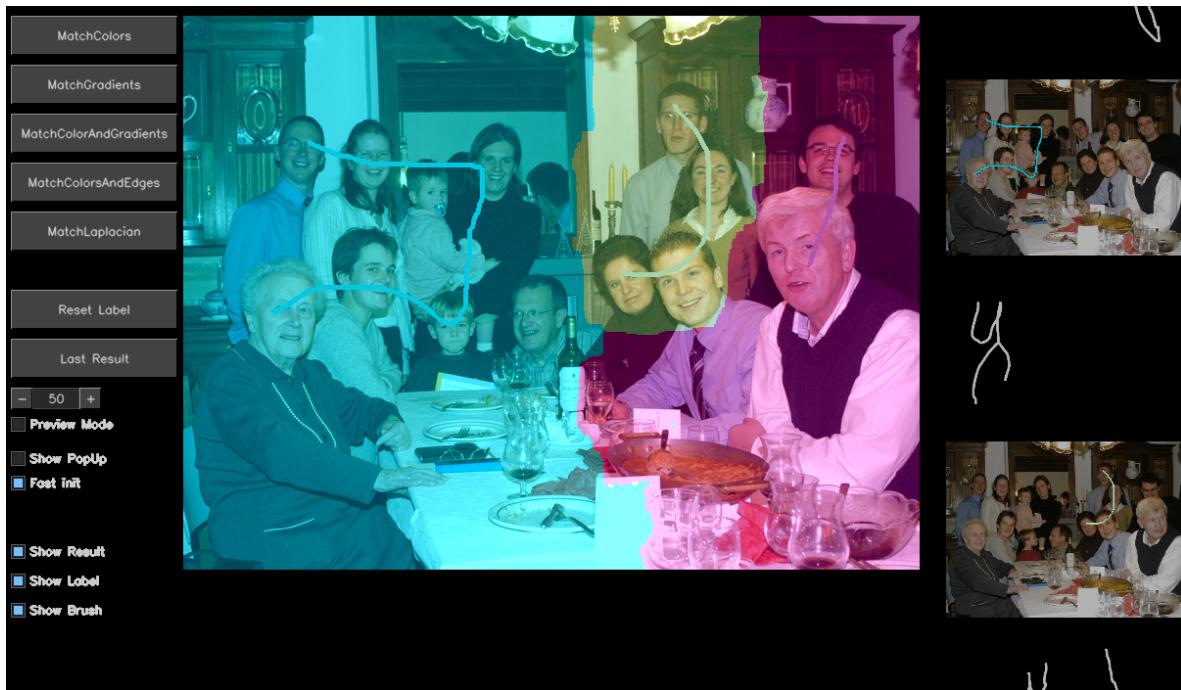
首先是左侧按钮区，从 Match Color 到 Match Gradients, Match Color & Gradient 还有 Match Colors And Edge 是不同的优化目标，点击就会开始 PhotoMonatge 算法。Reset Label 会清空 Label, Last Result 会显示上次的计算结果。

下面是一个迭代次数选择器。

Preview Mode 选项打开之后会进入快速计算模式。主要是减小分辨率，减少迭代次数快速获得结果，但画质会比较糟糕。

Show PopUp 会用弹窗的形式显示中间的一些过程。注意运行完按 Esc。

Fast Init 是使用了前文提到的梯度域图像融合的优化算法。如果打开了建议 50 次迭代即可，如果关闭选项建议 3000 次迭代。



如果我们运行了算法之后，会增加几个按钮：

Show Result 显示合成图像

Show Label 显示Graph Cuts 的标签

Show Brush 显示用户的画笔

这时候继续点击 Gallery 可以添加标签。

5.2 运行结果

5.2.1 简单家庭照

源图1



源图 2



源图3



源图4



合成结果:





笔画 + 分割



笔画 + 结果

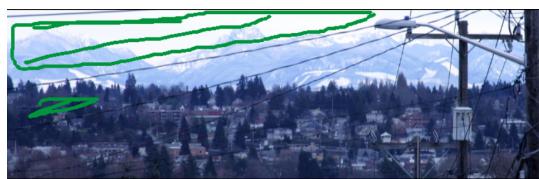


5.2.2 富士山去电线

源图1



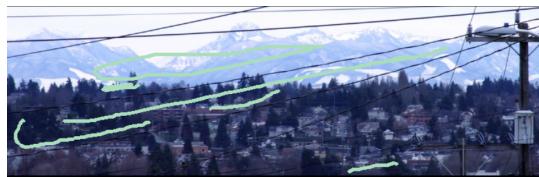
源图2



源图3



源图4



源图5



源图6



源图7



源图8

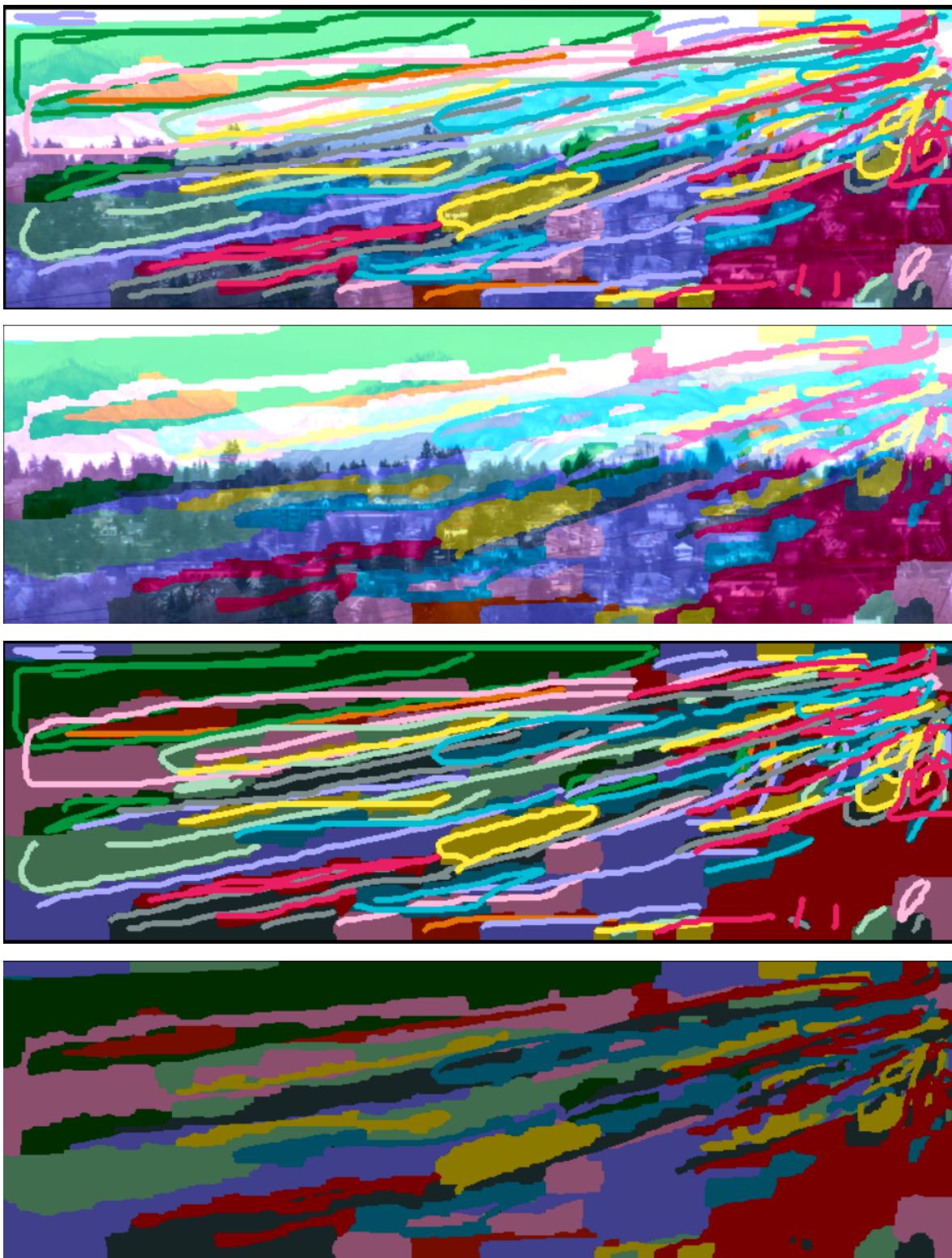


源图9



结果:





5.3 运行速度

我对比了增加了优化和没有优化的速度。括号内是使用的迭代次数。我们可以发现优化后可以节省近 90% 的时间。

Match	Times (Iterations) Old Method	Times (Iterations) New Method
Match Colors	44738ms (3000)	3143ms (50)
Match Gradients	55201ms (3000)	15111ms (50)
Match Colors & Edges	45186ms (3000)	3349ms (50)

6 编译运行

编译说明

由于实验中依赖多个 C++17 特性，包括了：`std::filesystem` 用于扫描目录，
`std::reduce_transform` 用于多线程的向量操作和计算，`if constexpr()` 编译期优化分支结构。

实验中依赖 Eigen 进行转置等运算，通过 VS 的 Nuget 作为 Package 安装。

实验中依赖 OpenCV 4.1.1

运行说明

GUI 请参照 [GUI 详解](#)

1. Aseem Agarwala, Mira Dontcheva, Maneesh Agrawala, Steven Drucker, Alex Colburn, Brian Curless, David Salesin, and Michael Cohen. 2004. Interactive digital photomontage. In ACM SIGGRAPH 2004 Papers (**SIGGRAPH '04**). Association for Computing Machinery, New York, NY, USA, 294-302. DOI:<https://doi.org/10.1145/1186562.1015718> ↗

2. <https://zhuanlan.zhihu.com/p/64227658> ↗