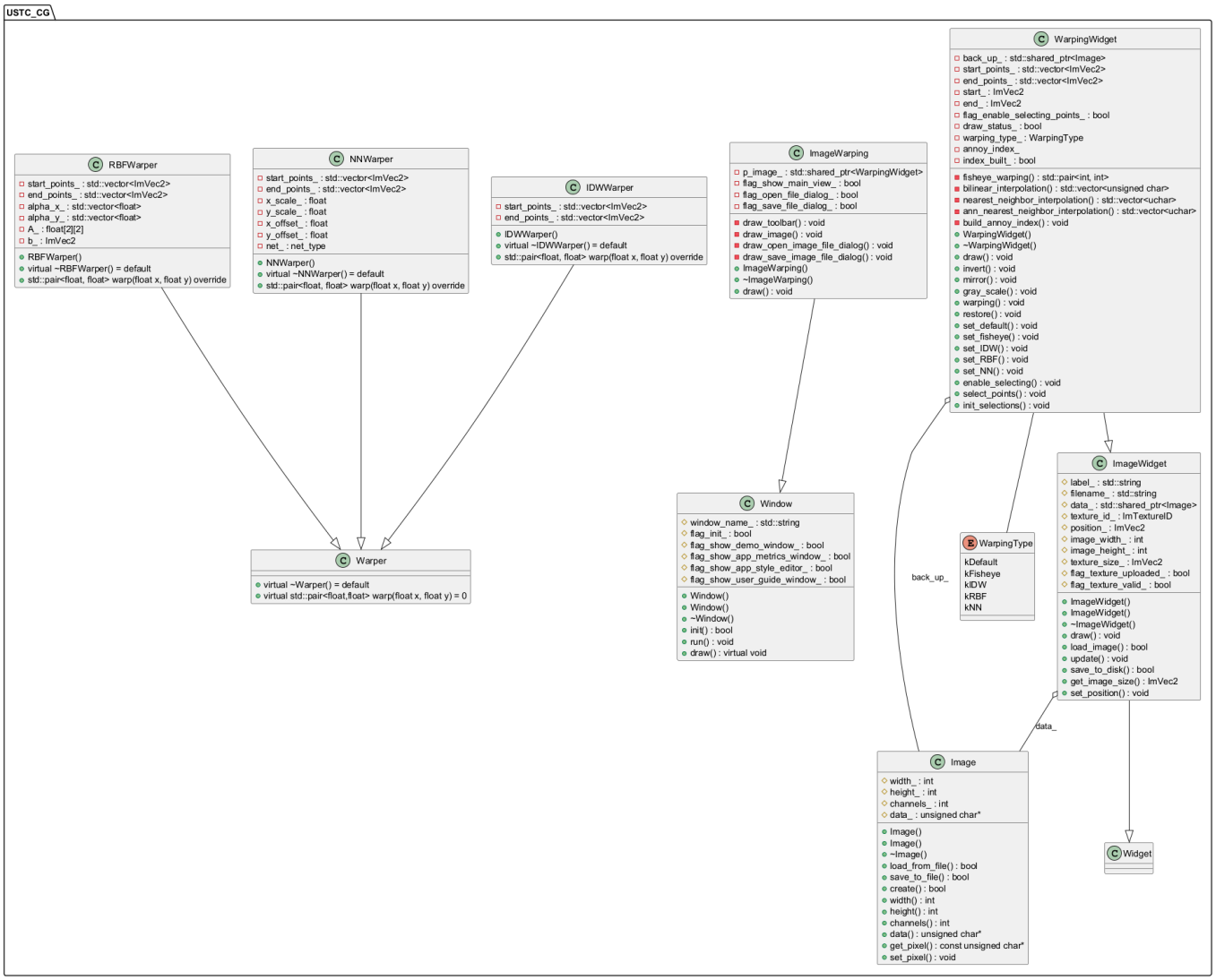
# HW2 Report

# 展示类图




# 期望效果

- 用户通过鼠标点击操作实现图片的warp
- 用户鼠标点击分为两组点，起点和终点
- warp操作后，起点处的像素值被映射到终点处的像素值
- 其他点位得到平滑的处理

# 如何实现?

## 问题描述

给定 $n$ 对控制点 $(\boldsymbol{p}_i, \boldsymbol{q}_i)$，其中 $\boldsymbol{p}_i, \boldsymbol{q}_i \in \mathbb{R}^2$，$i = 1, 2, \cdots, n$，

希望得到一个函数 $f : \mathbb{R}^2 \to \mathbb{R}^2$，满足插值条件:

$$f(\boldsymbol{p}_i) = \boldsymbol{q}_i, \quad \text{for } i = 1, 2, \cdots, n.$$

# Method 1: Inverse Distance-weighted Interpolation

```cpp
std::pair<float, float> IDWWarper::warp(float x, float y)
{
    float sum_w = 0.0f;
    float sum_wx = 0.0f;
    float sum_wy = 0.0f;
    constexpr float mu = 2.0f;
    constexpr float epsilon = 1e-9f;

    for (size_t i = 0; i < start_points_.size(); i++)
    {
        // calculate the drift to the control point
        float dx = x - static_cast<float>(start_points_[i].x);
        float dy = y - static_cast<float>(start_points_[i].y);

        float dist_sq = dx * dx + dy * dy;

        // calculate the weight term sigma_i = 1/(dist^u + epsilon)
        float sigma = 1.0f / (powf(dist_sq, mu / 2) + epsilon);

        // f(p) = \sum_{i=1}^n w_i(p)q_i
        // w_i(p) = sigma_i(p) / sum_sigma
        sum_w += sigma;
        sum_wx +=
            sigma * static_cast<float>(end_points_[i].x - start_points_[i].x);
        sum_wy +=
            sigma * static_cast<float>(end_points_[i].y - end_points_[i].y);
    }
    if (sum_w < epsilon)
    {
        return { x, y };
    }
    return { x + sum_wx / sum_w, y + sum_wy / sum_w };
}
```

代码实现假设线性映射$D_i$是恒等变换

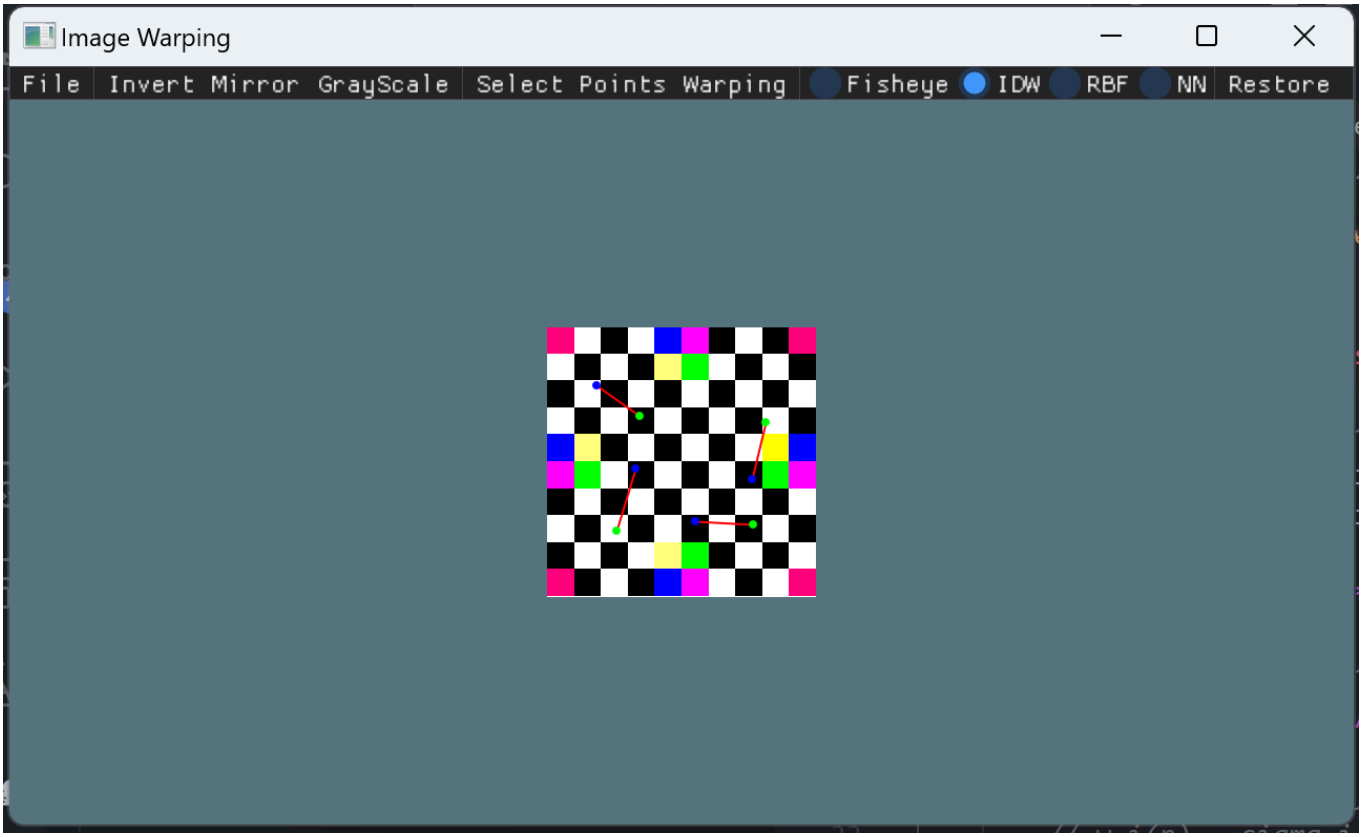
$f(p) = \sum_i w_i(p) f_i(p) = \sum_i w_i(p)(q_i + p - p_i) = \sum_i w_i(p)(q_i - p_i) + p$

在实践中，我们发现这与简单的认为$D_i = 0$所实现的效果差异很大。

即修改成如下代码：

```
34          sum_w += sigma;
35          sum_wx += sigma * static_cast<float>(end_points_[i].x);
36          sum_wy += sigma * static_cast<float>(end_points_[i].y);
37      }
38      if (sum_w < epsilon)
39      {
40          return { x, y };
41      }
42      return { sum_wx / sum_w, sum_wy / sum_w };
```

# 结果对比




$D_i = 0$ 的效果

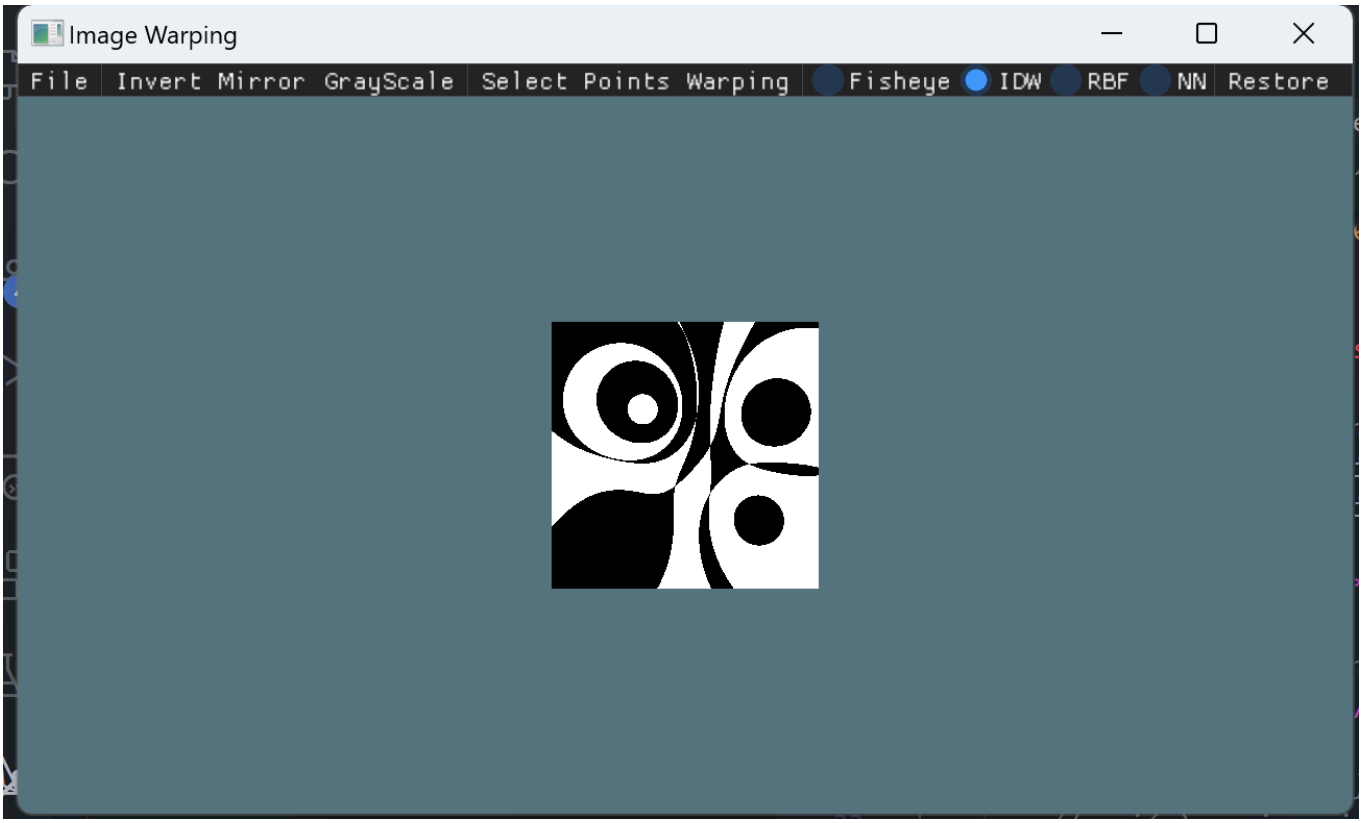


Image Warping
File
Invert
Mirror
GrayScale
Select Points
Warping
Fisheye
IDW
RBF
NN
Restore

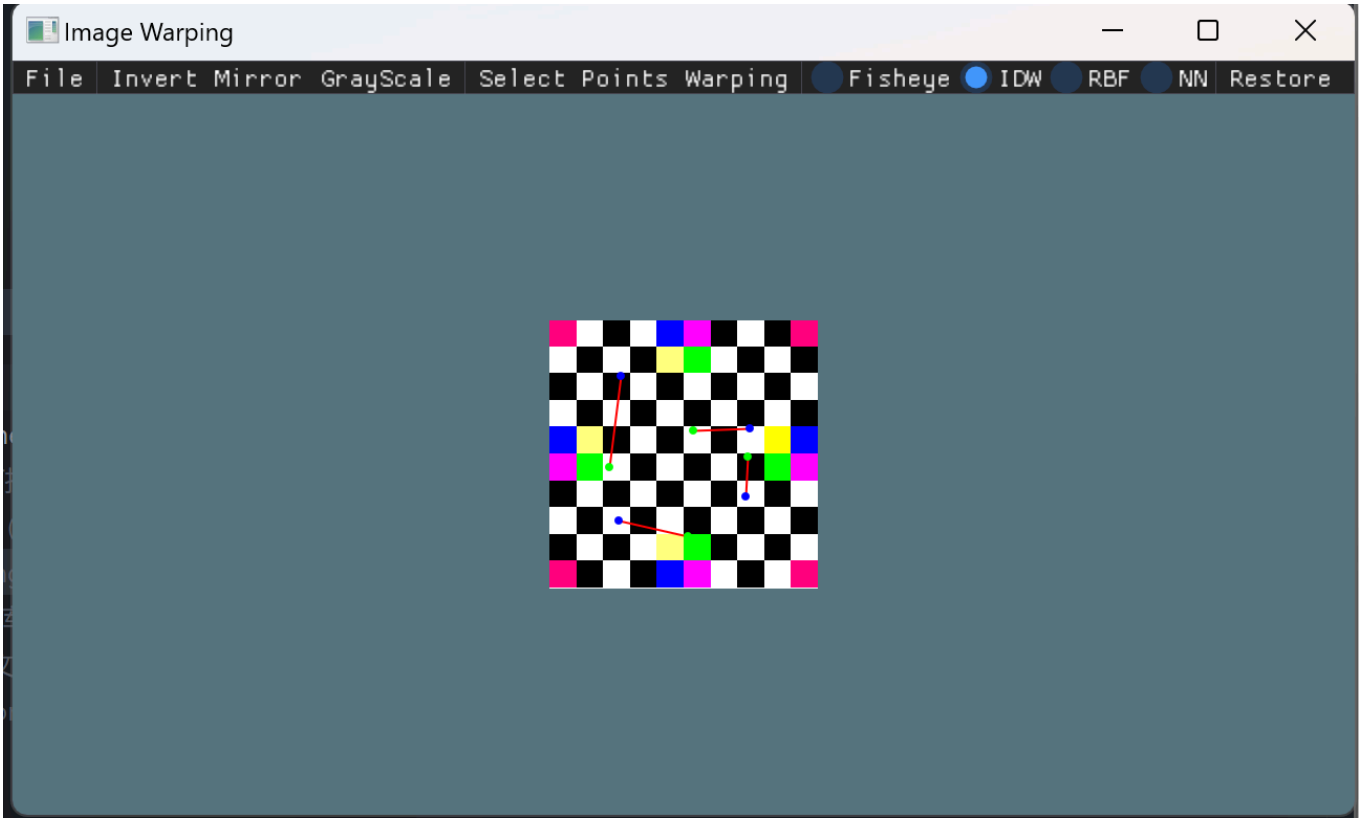
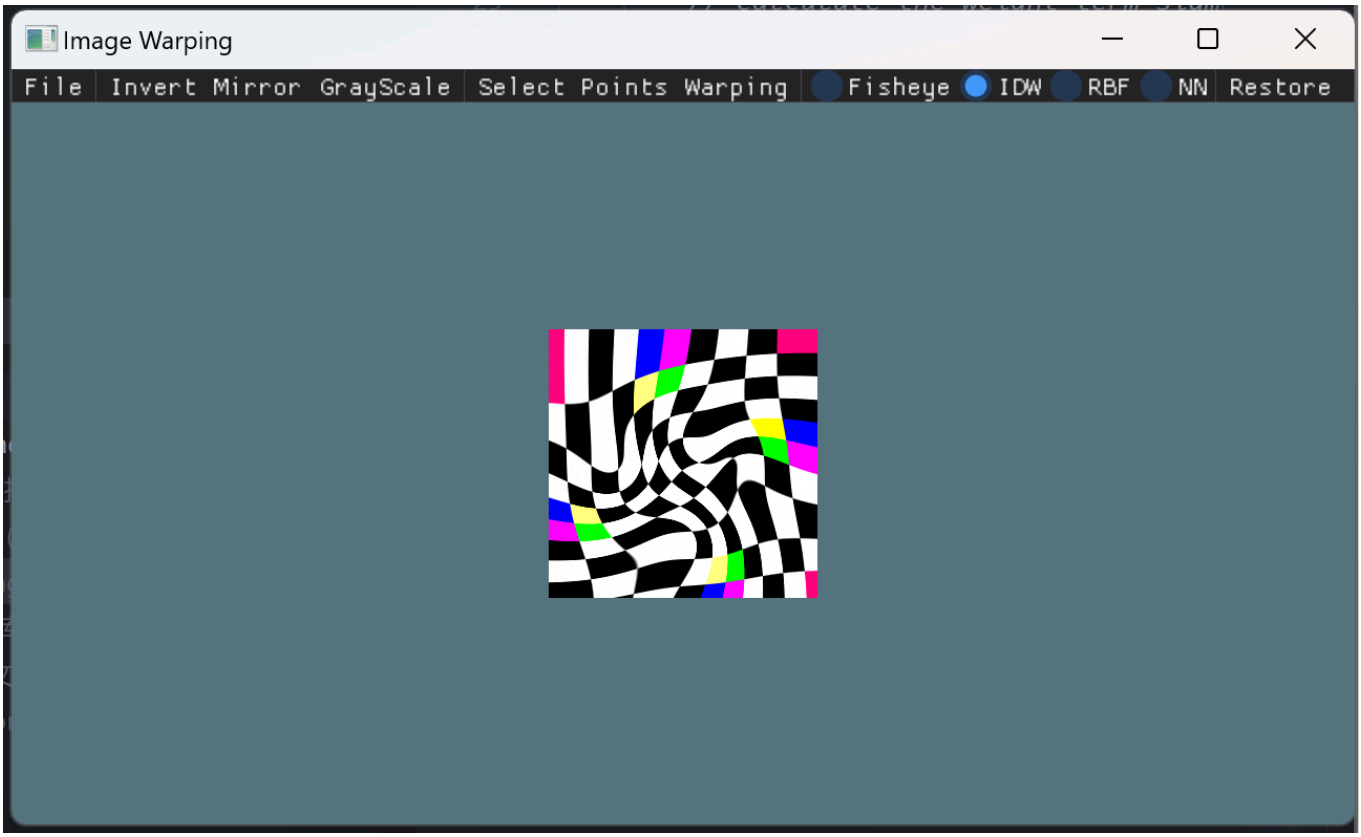


Image Warping
File
Invert
Mirror
GrayScale
Select Points
Warping
Fisheye
IDW
RBF
NN
Restore

设置$D_i = I$



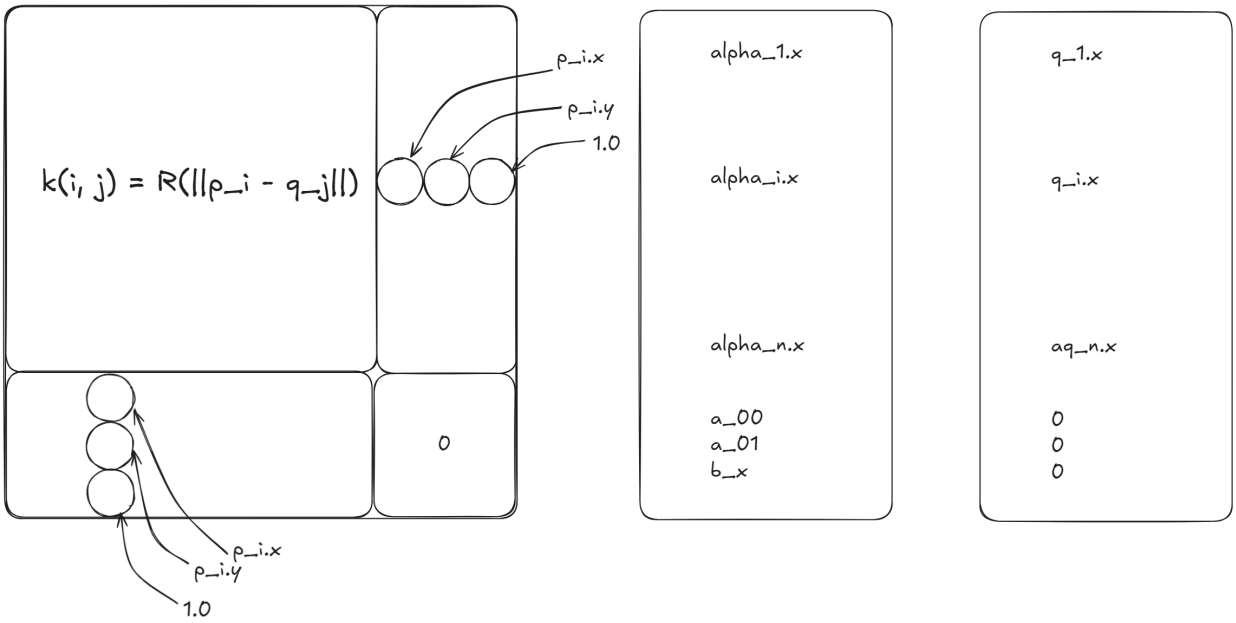

brilliant result!

# Method 2: Radial Basis Functions Interpolation

这里选择的径向基函数是$r^2 \log r$

我们需要进行一些数学变换方便我们求解

$$f(p_j) = \sum_{i=1}^{n} \alpha_i R(\|p_j - p_i\|) + Ap_j + b = q_j, \quad j = 1, \ldots, n.$$




类似可以对$y$分量写出方程

我们在这里实现了文档中提供的约束

$$\begin{pmatrix} p_1 & \cdots & p_n \\ 1 & \cdots & 1 \end{pmatrix}_{3\times n} \begin{pmatrix} \alpha_1^\top \\ \vdots \\ \alpha_n^\top \end{pmatrix}_{n\times 2} = 0_{3\times 2}.$$

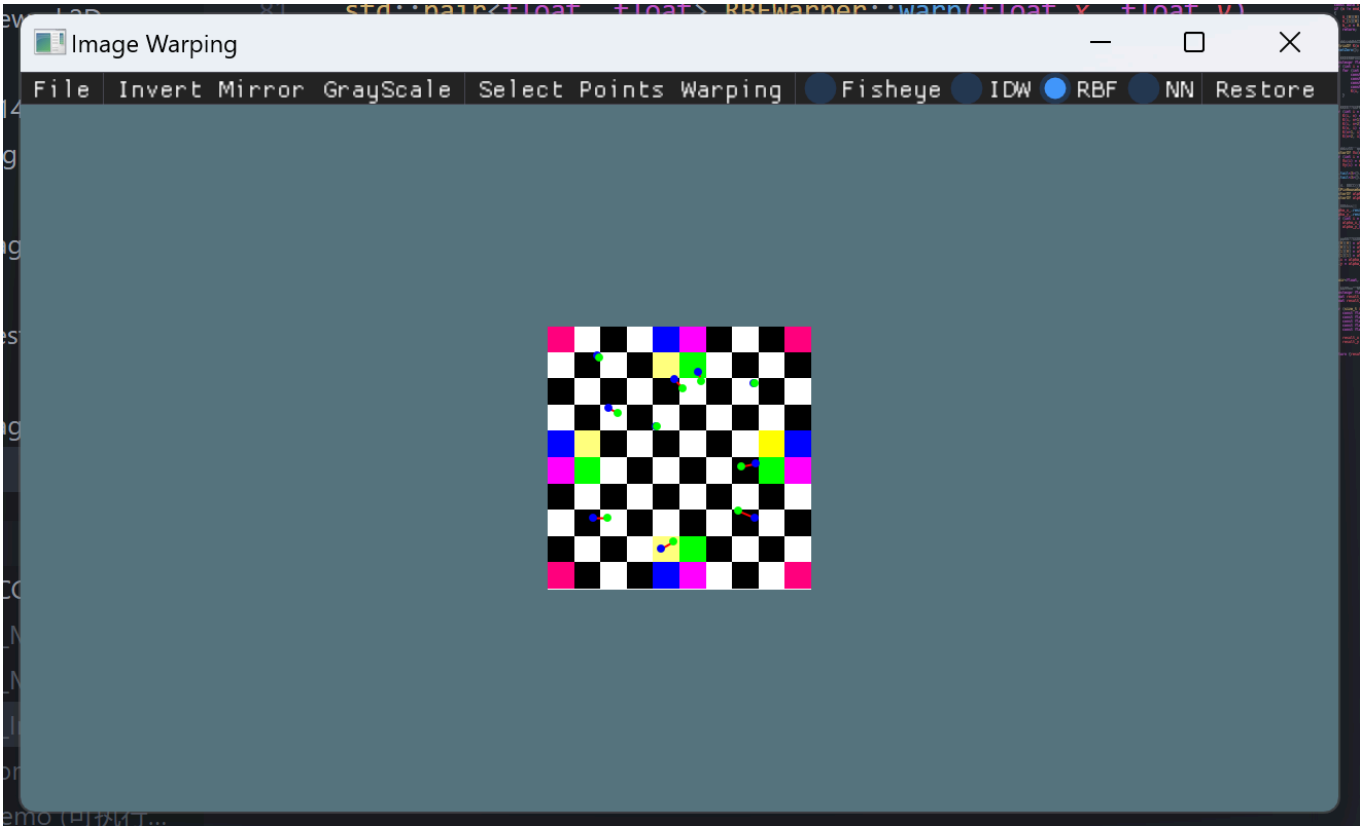
代码实现上，我们将右边的向量替换为终点和起点的位移量（为了数值稳定）

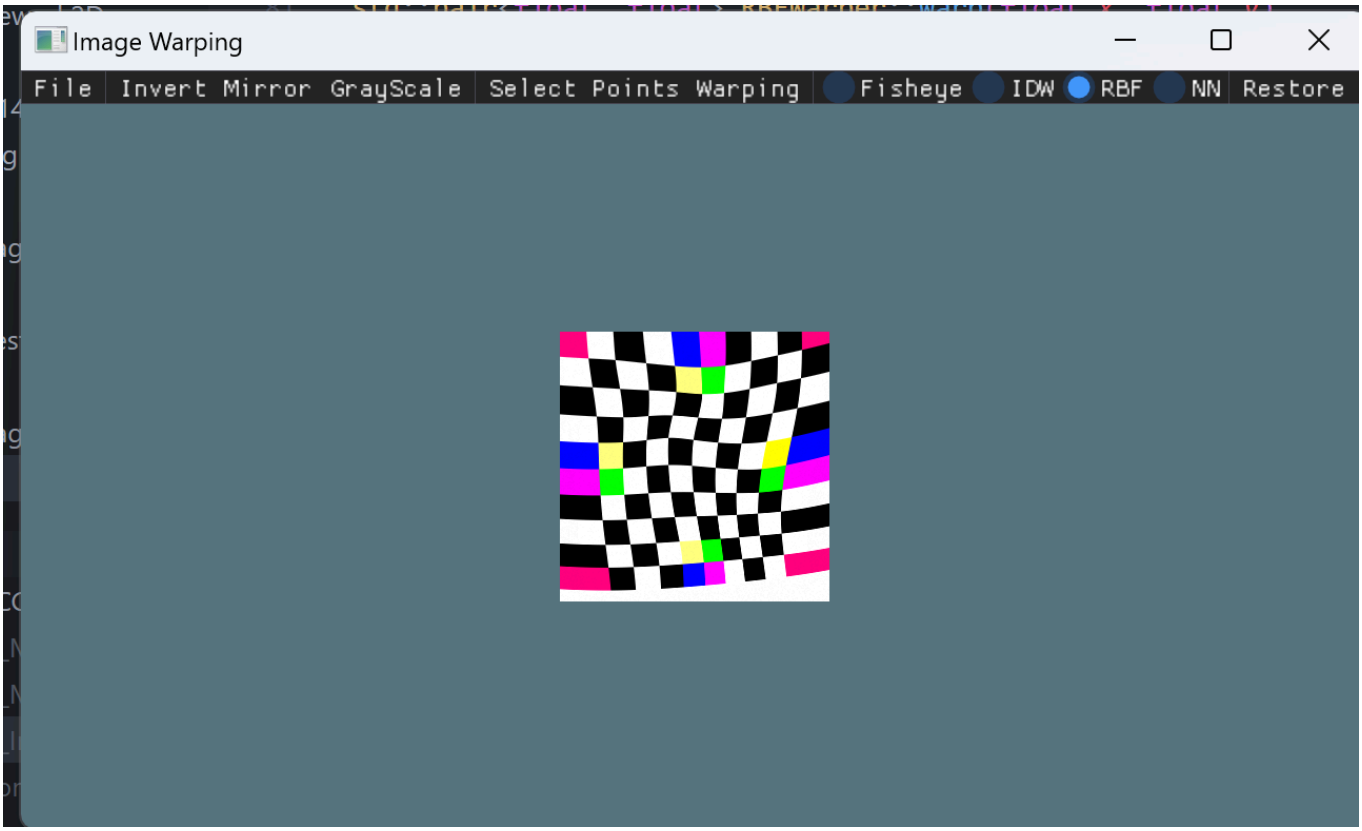
```
// 构建右侧向量 [q_x; 0] 和 [q_y; 0]
VectorXf Vx(n + 3), Vy(n + 3);
for (int i = 0; i < static_cast<int>(n); ++i) {
    Vx(i) = end_points[i].x  - start_points[i].x;
    Vy(i) = end_points[i].y - start_points[i].y;
}
Vx.tail<3>().setZero();
Vy.tail<3>().setZero();
```

我们使用QR分解求解上面的矩阵方程

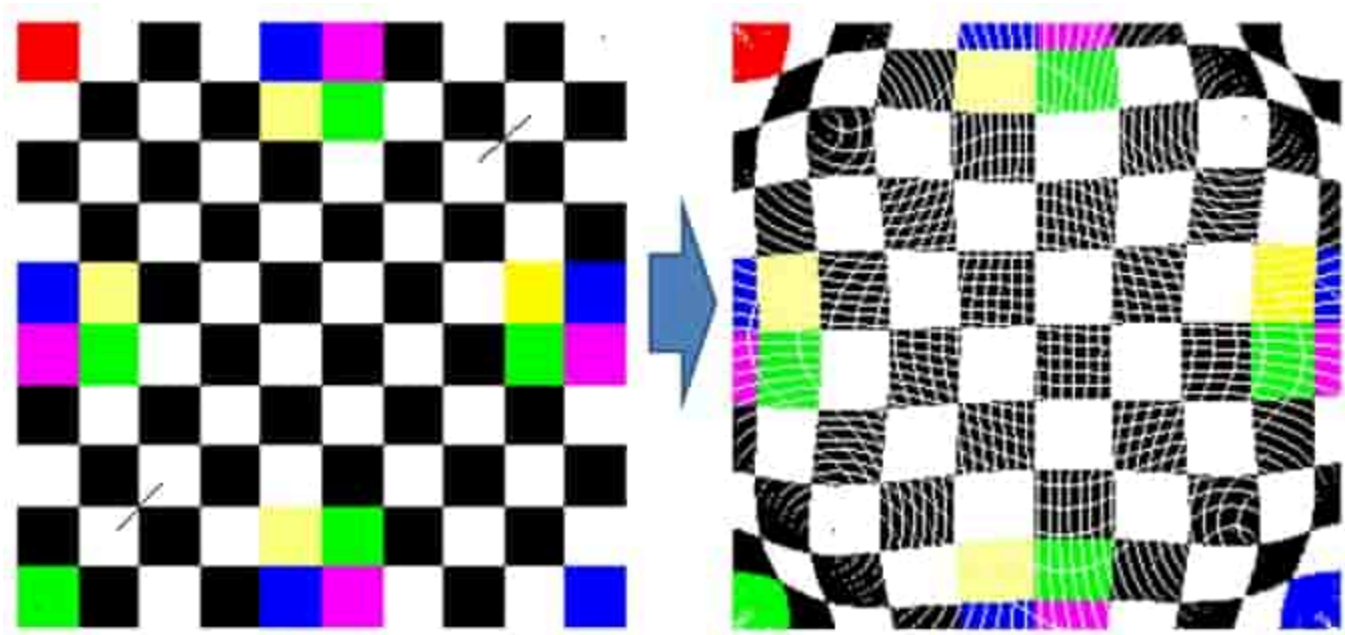
```
ColPivHouseholderQR<MatrixXf> qr(K);
VectorXf alpha_x = qr.solve(Vx);
VectorXf alpha_y = qr.solve(Vy);
```

# 结果展示






# 白缝填补



# 为什么会产生缝隙呢?

第一反应是浮点数转化为整数，导致有一些像素点未被赋予非0值。检查代码`WarpingWidget::fisheye_warping`的输出

```
    int new_x = static_cast<int>(center_x + dx * ratio);
    int new_y = static_cast<int>(center_y + dy * ratio);

    return { new_x, new_y };
```

在输出结果前，将浮点数转化为了整数。起初，图像出现缝隙有一点反直觉，因为我们的映射是连续函数，非跳跃函数，直观上不该有裂缝。

考虑到连续函数作用在离散格点上，事情就变得合理。

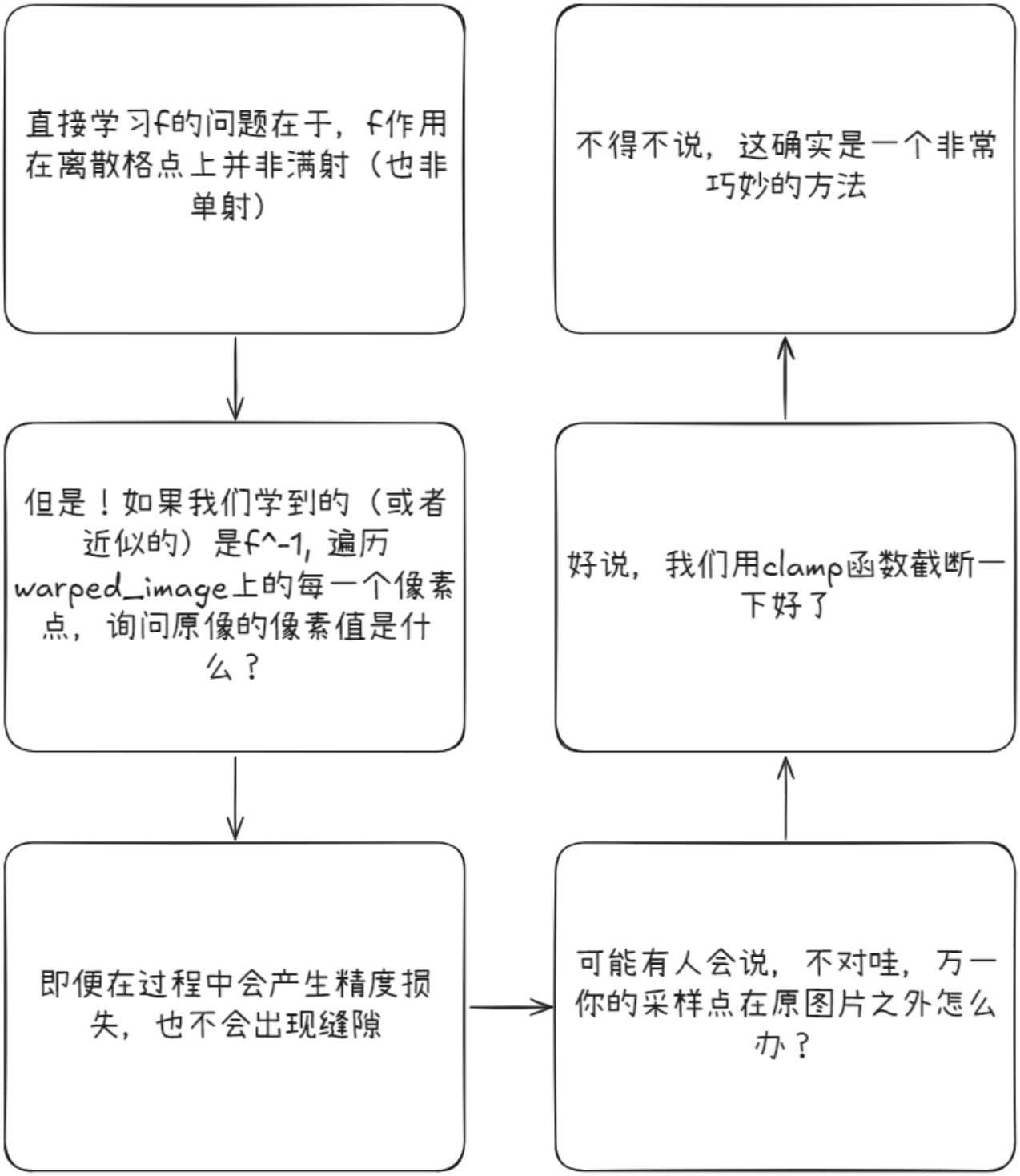
在近似格点过程中，不能保证单射，这就导致了不是满射，出现缝隙。

## 如何解决?

如何解决这个问题呢？一个直接的思路是，在像素值为0的像素点附近做最近邻搜索，将最近的非零值像素设置为该点的像素。

这个思路很简单，但还不够优雅。我们看看`deepseek`怎么说？`deepseek`强烈建议我们使用`反向映射`这一方法。

起初我以为`deepseek`完全理解错了我的意思，随着聊天的深入，`deepseek`完全不怀疑自己`反向映射`方法有问题，这使得我认真考虑这个方法。




具体的插值方法我们实现了三种：

- `bilinear_interpolation`
- `nearest_interpolation`

- `add_nearest_interpolation`：使用`Annoy`第三方库

```
std::vector<uchar> WarpingWidget::bilinear_interpolation(float& x, float& y)···

解释代码 | 注释代码 | 生成单测 | ×
std::vector<uchar> WarpingWidget::nearest_neighbor_interpolation(
    float& x,
    float& y)···

解释代码 | 注释代码 | 生成单测 | ×
std::vector<uchar> WarpingWidget::ann_nearest_neighbor_interpolation(
    float& x,
    float& y)···
}
```

  实际效果区别不大。

# Dlib

这种学一个函数的任务，怎么能少了我们的Neural Network?

我们的网络非常简单

```
You, 1小时前 | 1 author (You)
using net_type = dlib::loss_mean_squared_multioutput<dlib::fc<
    2,
    dlib::elu<dlib::fc<
        10,
        dlib::elu<dlib::fc<10, dlib::input<dlib::matrix<float>>>>>>>>;

net_type net_;
```

激活函数选的`elu`，没什么特别原因，单纯看`relu`太简单，不想用。

# 这就开始训练了吗?

直接把`start_points`，`end_points`喂给网络，就开始学习了吗?

Nono, 事情没有这么简单，如果是这么单纯的实现，你会发现网络的loss甚至会到惊人的一万+！而且即便经过一万个epoches，你的loss也很难有什么变化，这就很糟糕。

# 那我们怎么办呢?

归一化！没错，就是几乎可以出现在任何网络的任何地方的`norm`层。我们先对数据点进行归一化

```
// 计算数据范围，用于归一化
float min_x = start_points[0].x, max_x = start_points[0].x;
float min_y = start_points[0].y, max_y = start_points[0].y;

for (const auto& p : start_points) {
    min_x = std::min(min_x, p.x);
    max_x = std::max(max_x, p.x);
    min_y = std::min(min_y, p.y);
    max_y = std::max(max_y, p.y);
}

// 存储归一化参数，用于后续推断
x_scale_ = max_x - min_x > 1e-6f ? 2.0f / (max_x - min_x) : 1.0f;
y_scale_ = max_y - min_y > 1e-6f ? 2.0f / (max_y - min_y) : 1.0f;
x_offset_ = min_x;
y_offset_ = min_y;
```

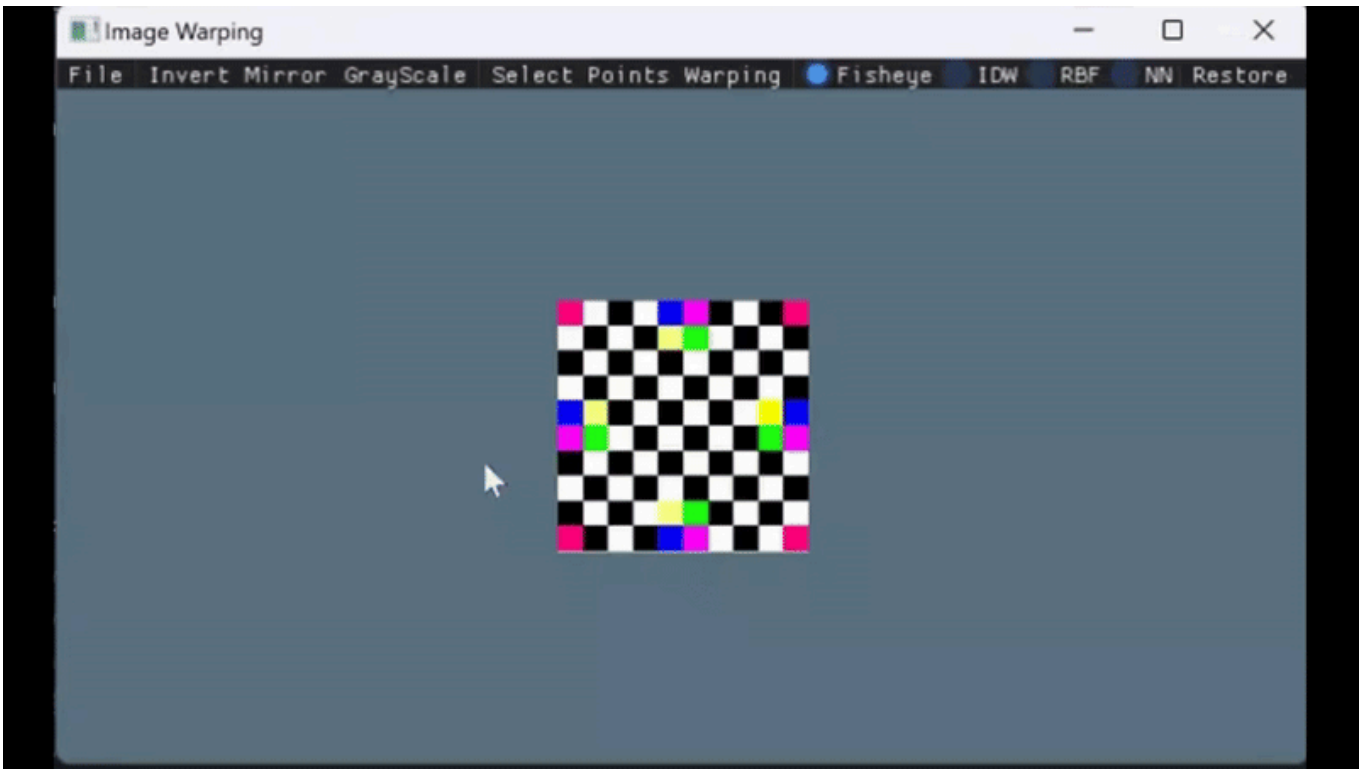
```
for (size_t i = 0; i < start_points.size(); i++)
{
    // 归一化输入数据到 [-1, 1] 范围
    dlib::matrix<float> input(2, 1);
    input(0, 0) = (start_points[i].x - x_offset_) * x_scale_ - 1.0f;
    input(1, 0) = (start_points[i].y - y_offset_) * y_scale_ - 1.0f;
    inputs.push_back(input);

    // 同样归一化输出数据
    dlib::matrix<float> target(2, 1);
    target(0, 0) = (end_points[i].x - x_offset_) * x_scale_ - 1.0f;
    target(1, 0) = (end_points[i].y - y_offset_) * y_scale_ - 1.0f;
    targets.push_back(target);
}
```

同样的，推理的时候也用归一化

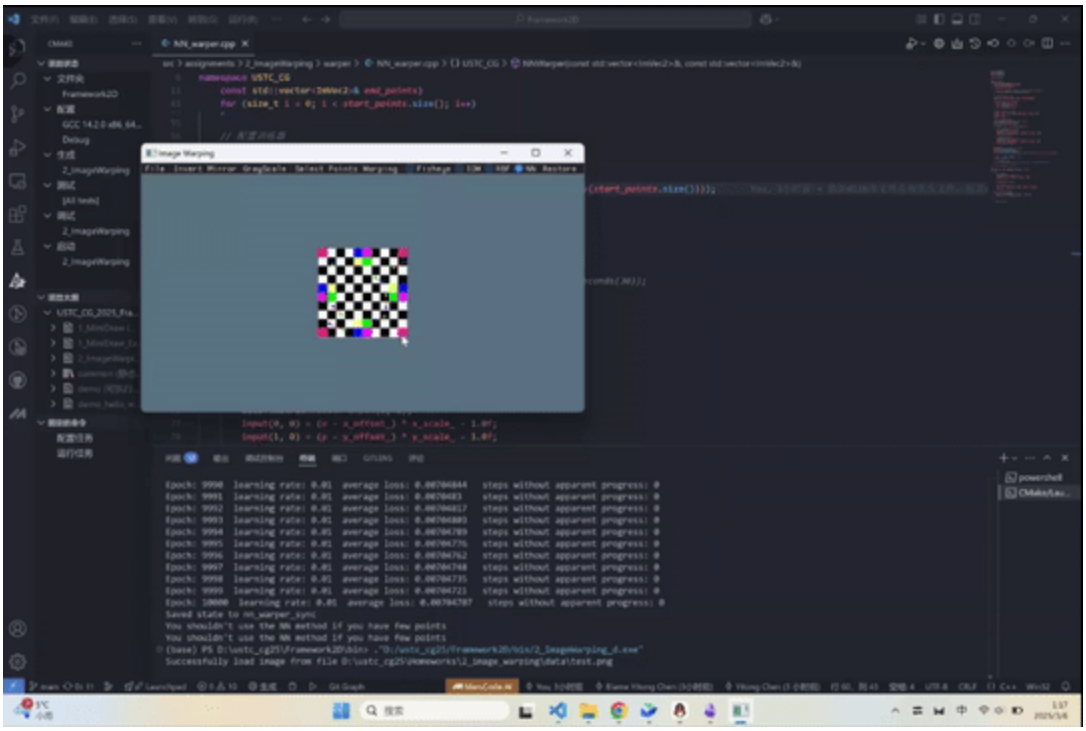
```cpp
// 归一化输入
dlib::matrix<float> input(2, 1);
input(0, 0) = (x - x_offset_) * x_scale_ - 1.0f;
input(1, 0) = (y - y_offset_) * y_scale_ - 1.0f;

// 网络前向传播
auto output = net_(input);

// 反归一化输出
float out_x = (output(0, 0) + 1.0f) / x_scale_ + x_offset_;
float out_y = (output(1, 0) + 1.0f) / y_scale_ + y_offset_;

return {out_x, out_y};
```

这样训练后，我们就可以拿到漂亮的结果。

# 结果展示




# 人像编辑



# 完整展示视频

可以从这里查看视频
密码：bs4g

# 作业中遇到的bug

# 导入Dlib

由于`Dlib`包很大，而这里只需要训练一个`mlp`层，所以自然不想要把全部的`Dlib`导入进来。

折腾了半天，发现`Dlib`层层依赖的关系还挺麻烦，索性导入了整个包。

# 导出类图

感恩`cline`, 感恩`gemini-flash-thinking`