《数字图像处理课程实习》

实习报告

学　　院：　遥感信息工程学院

班　　级：　2007 班&2008 班&2010 班

实习地点：　教学实验大楼 2 楼 201 机房

指导教师：　石文轩

组　　员：　2020302131339　德丽达

2020302131119　殷雨薇

2020302131208　黄淼

2020302131169　雷轶

2022 年 1 月 7 日

Hough 直线检测

1 研究内容与目标

1.1 研究内容

霍夫变换将在一个空间中具有相同形状的曲线或直线映射到另一个坐标空间的一个点上形成峰值，从而把检测任意形状的问题转化为统计峰值问题。研究包括读取图像、检测图像边缘转换图像成二值图像、核心算法 Hough 变换等在内的主要算法来实现对图像中直线的检测。

1.2 目标

霍夫变换是图像处理中的一种特征提取技术，可以识别图像中的几何形状。它将图像空间中的特征点映射到参数空间进行投票，通过检测累计结果的局部极值点得到一个符合某特定形状的点的集合。经典霍夫变换用来检测图像中的直线，后来霍夫变换扩展到任意形状物体的识别，多为圆和椭圆。它的抗噪声、抗形变能力较强。

2 算法原理

主要算法包括 Canny 算子边缘检测算法和 Hough 变换算法。

2.1 Canny 算子边缘检测

2.1.1 平滑图像，滤除噪声

边缘检测的算法主要是基于图像强度的一阶和二阶微分操作，但导数通常对噪声很敏感，边缘检测算法常常需要根据图像源的数据进行预处理操作,因此采用滤波器来改善与噪声有关的边缘检则性能，比如在进行边缘检测前，可以对原始数据先作高斯滤波处理。如果不做滤波平滑处理，原图片中不是边缘但是灰度变化频率较高的部分也容易被认为是边缘,这样会导致边缘检测性能的下降。

一维高斯分布：$$G(x)=\frac{1}{\sqrt{2\pi\sigma}}\mathrm{e}^{-\frac{\mathrm{x}^2}{2\sigma^2}}$$

二维高斯分布：$$G(x,y)=\frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}$$

对于图像来说，高斯滤波器是利用高斯核的一个 2 维的卷积算子。我们组目前选用的是 σ =1.5，模糊半径为 1 的高斯模板。利用该模板与原图像像素值进行卷积运算。

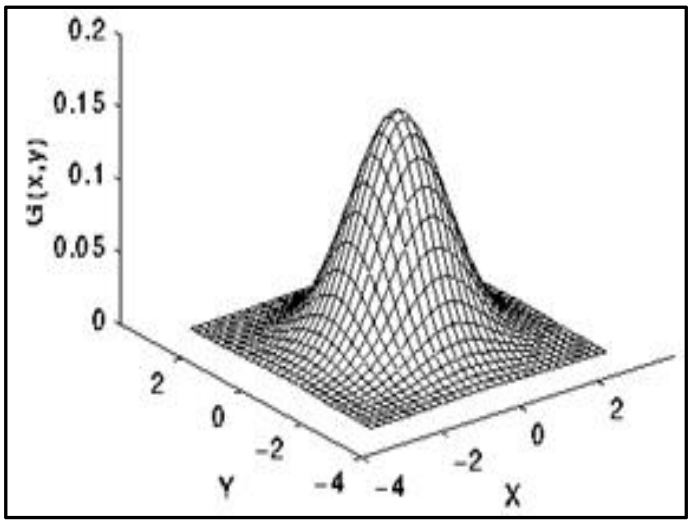


图 2.1.1 二维高斯分布图

2.1.2 计算梯度强度和方向

使用 Sobel 卷积核来计算梯度的幅度与方向。计算出的幅度与方向作为后面提取图像边缘的原始数据。Sobel 算子的优点有对边缘定位较为准确，能较好地处理灰度渐变和噪声较多的图像，计算简单，可分别计算水平和垂直边缘。

Sobel 算子:

$$Sobel_Y = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

$$Sobel_X = \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Soble 算子可以用一个平滑算子和差分算子的组合来表示。先垂直方向平滑，后水平方向差分，得到垂直边缘(dx)；先水平方向平滑，后垂直方向差分，得到水平边缘(dy)。

窗口大小	n	平滑算子（展开式系数）	差分算子
2	1	1 1	1 -1
3	2	1 2 1	1 0 -1
4	3	1 3 3 1	1 1 -1 -1
5	4	1 4 6 4 1	1 2 0 -2 -1

图 2.1.2（1） 各大小窗口对应的算子

二项式系数：

$$C_n^k = \frac{n!}{k!*(n-k)!}, k = 0,1,2,3,...n$$

差分：0 1 3 3 1 0

差分后：1-0 3-1 3-3 1-3 0-1

1 2 0 -2 -1 （差分算子）

图 2.1.2（2） 差分计算

计算梯度幅度的两种方法：

$$M（x,y)=\sqrt{d_x^2(x,y)+d_y^2(x,y)}$$

$$M（x,y)=\left|d_x(x,y)\right|+\left|d_y(x,y)\right|$$

梯度方向：

$$\theta_M=\arctan(d_y/d_x)$$

2.1.3 应用非极大值抑制，以消除边缘检测带来的杂散响应

非极大值抑制是一种边缘稀疏技术，非极大值抑制的作用在于“瘦”边。对图像进行梯度计算后，仅仅基于梯度值提取的边缘仍然很模糊。而非极大值抑制则可以帮助将局部最大值之外的所有梯度值抑制为 0，对梯度图像中每个像素进行非极大值抑制的算法是:将当前像素的梯度强度与沿正负梯度方向上的两个像素进行比较;如果当前像素的梯度强度与另外两个像素相比最大，则该像素点保留为边缘点，否则该像素点将被抑制。这样可以抑制非极大值，保留局部梯度最大的点，以得到细化的边缘。

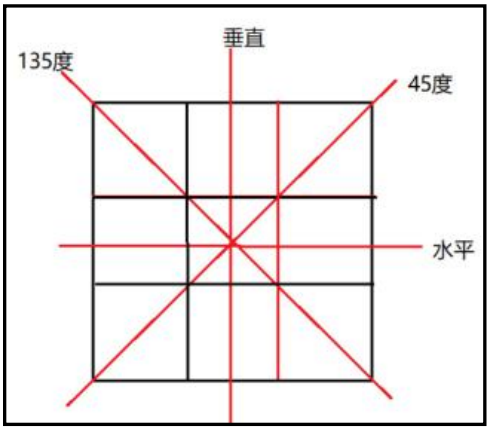


图 2.1.3 梯度划分

方法：3*3 区域内，梯度可以划分为垂直、水平、45°、135° 4 个方向。

梯度方向为垂直：

$$\theta_M \in (-22.5,22.5) \cup [-180,-157.5) \cup (157.5,180]$$ 上下方向比较

梯度方向为 45°：

$$\theta_M \in [22.5,67.5] \cup [-157.5,-112.5]$$ 右下左上方向比较

梯度方向为水平：

$$\theta_M \in (67.5,112.5] \cup (-112.5,-67.5]$$ 左右方向比较

梯度方向为-45°：

$$\theta_M \in (112.5,157.5] \cup (-67.5,-22.5]$$ 右上左下方向比较

2.1.4 双阈值检测确定边缘

在施加非极大值抑制之后，剩余的像素可以更准确地表示图像中的实际边缘。然而，仍然存在由于噪声和颜色变化引起的一些边缘像素。为了解决这些杂散响应，必须用弱梯度值过滤边缘像素，并保留具有高梯度值的边缘像素，可以通过选择高低阈值来实现。如果边缘像素的梯度值高于高阈值，则将其标记为强边缘像素；如果边缘像素的梯度值小于高阈值并且大于低阈值，则将其标记为弱边缘像素；如果边缘像素的梯度值小于低阈值，则会被抑制。阈值的选择取决于给定输入图像的内容。

方法：

（1） 根据图像选取合适的高阈值和低阈值，通常高阈值是低阈值的 2 到 3 倍。

（2） 如果某一像素的梯度值高于高阈值，则保留。

（3） 如果某一像素的梯度值低于低阈值，则舍弃。

（4） 如果某一像素的梯度值介于高低阈值之间，则从该像素的 8 邻域的寻找像素梯度值，如果存在像素梯度值高于高阈值，则保留，如果没有，则舍弃。

2.2 Hough变换直线检测

2.2.1 Hough变换定义

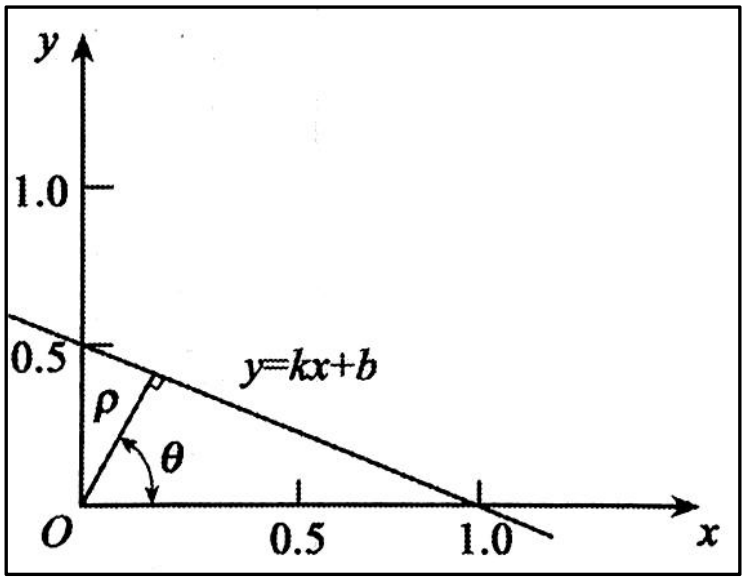


图2.2.1 y=kx+b直线

在x,y平面上，任何一条直线,y=kx+b(其中k,b分别表示斜率和截距),若 θ 是该直线与x轴的夹角,ρ 是原点到直线的距离，则这条直线可表示为：ρ =xcos θ +ysin θ 。

Hough变换就是一种直线到点的变换。即xy平面的任意一条直线，在 ρ 和 θ 定义的二维空间上对应一个点。

2.2.2 Hough变换检测直线原理

对于x,y平面的一个特定点(x_0， y_0),通过该点的直线有很多条,每一条都对应 ρ，θ 平面中的一个点。并且与x,y平面中所有这些直线对应的点在参数空间中的轨迹是一条正弦曲线,因而过x,y平面上的任意一点的所有直线对应于 ρ， θ 平面的一条正弦曲线。

如果一组边缘点位于由参数ρ_0和θ_0决定的直线上,则每个边缘点对应了ρ，θ平面的一条正弦曲线，并且所有这些正弦曲线必相交于点(ρ_0，θ_0)，这就是Hough变换检测直线的原理。

2.2.3 Hough变换检测直线算法

（1）初始化一个ρ,θ平面的数组。一般ρ方向上的量化数目为对角线方向像素数,θ方向上的量化间距为2°。

(2）顺序搜索图像的所有黑点，对每一个黑点，按式(1)计算ρ，θ取不同的值，分别将对应的数组元素加1。

(3）求出数组中的最大值并记录对应的ρ，θ。

(4）绘出ρ，θ对应的直线。

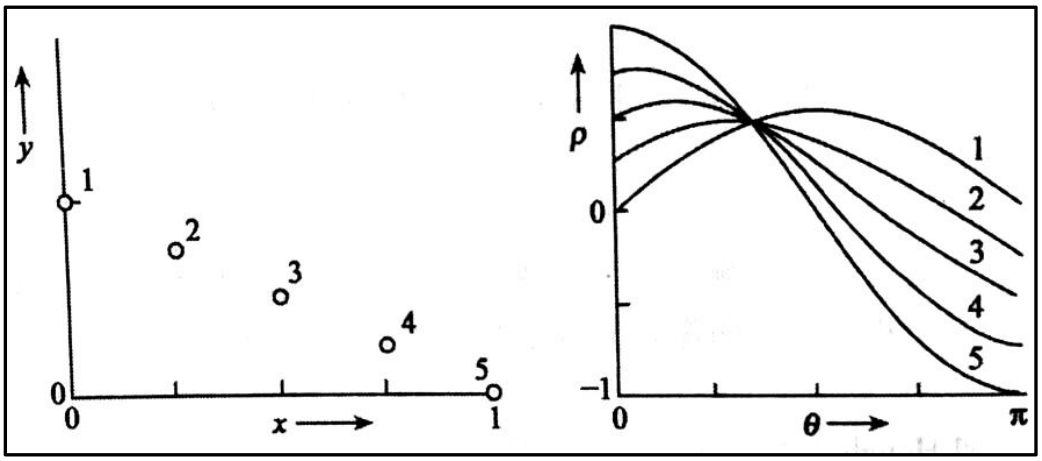


图2.2 Hough变换检测直线

2.2.4 Hough变换提取直线

如下图2.2为利用Hough变换提取一条直线的流程图。

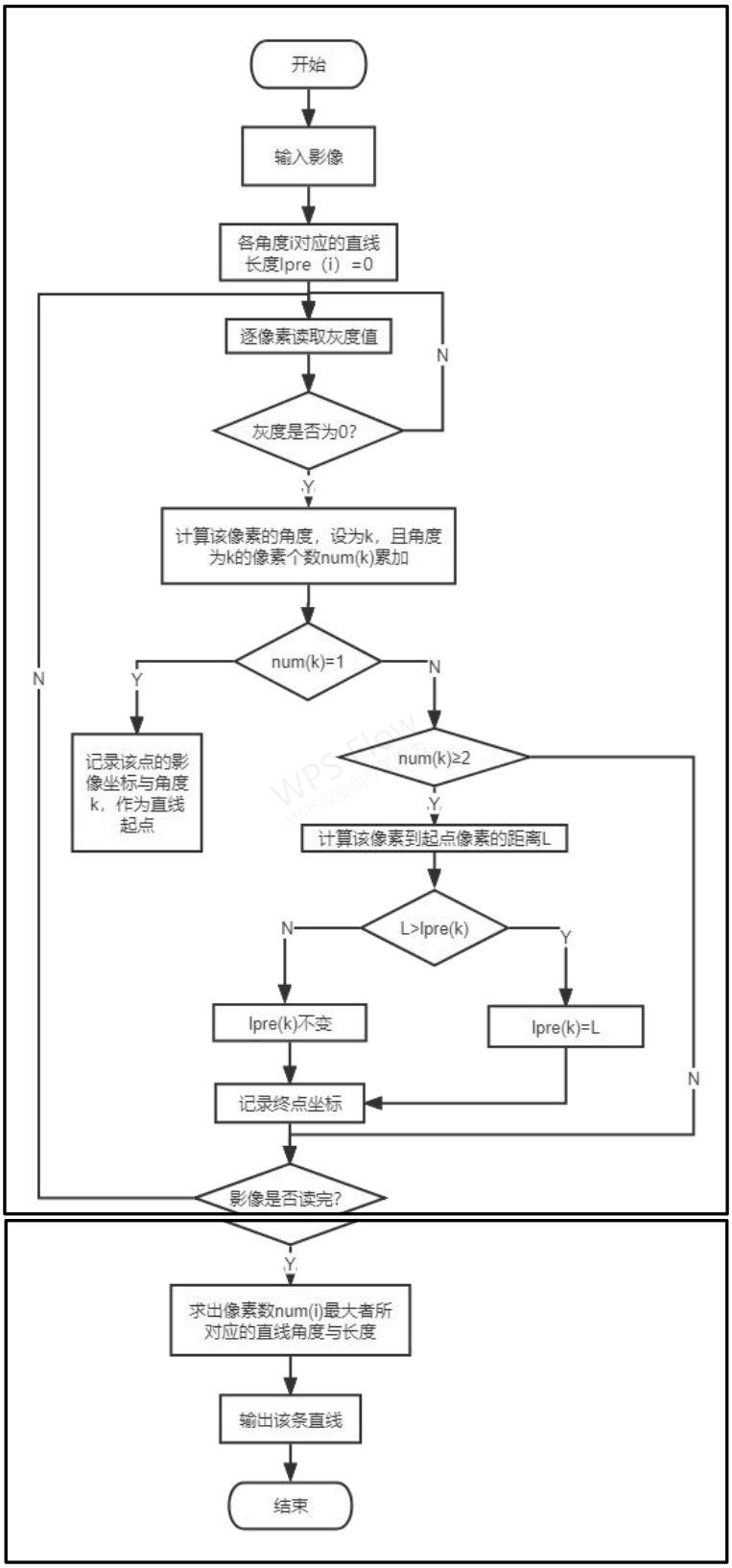


图2.2 利用Hough变换提取一条直线的流程图

3 流程设计与实现

3.1 利用 Canny 算子进行边缘检测

3.1.1 高斯平滑，滤除噪声

```
//高斯平滑
void GaussFilter(Mat grayimg)
{
    unsigned char* pimg = grayimg.data;//图像地址
    int height = grayimg.rows;
    int width = grayimg.cols;
    //为新图像处理部分分配存储空间（不包括边界点
    Mat imgnew;
    imgnew.create(height, width, CV_8UC1);
    unsigned char* pnewimg = imgnew.data;
    for (int i = 1; i < height - 1; i++)
    {
        for (int j = 1; j < width - 1; j++)
        {
            pnewimg[i * width + j] = 0;
        }
    }//初始化
```

读取高斯算子的 txt 文件，存储在数列中，再利用算子对图像除边界点外的像素点进行卷积运算：

```
 //读取保存在txt中的滤波算子
FILE* fp;
int n = 0;
fp = fopen("gausslvbo.txt", "r");

if (!fp)
{
    printf("无法读取滤波算子\n");
}

double H[9];//存储算子
for (int i = 0; i < 9; i++)
{
    fscanf_s(fp, "%lf", &H[i]);
}
```

```
//卷积运算
for (int i = 1; i < height - 1; i++)
{ //从1开始，先不处理边界点
    for (int j = 1; j < width - 1; j++)
    {
        double t = 0;
        for (int m = 0; m < 3; m++)
        {//选取所选点8-领域的点
            for (int n = 0; n < 3; n++)
            {
                t += (double)pimg[(i + m - 1) * width + j + n - 1] * H[m * 3 + n];
            }
        }
        pnewimg[i * width + j] = (unsigned char)abs(t);
        if (pnewimg[i * width + j] > 255)
        {
            pnewimg[i * width + j] = 255;
        }
    }
}
```

再对边界点做保留处理，并存储高斯平滑后的 bmp 图像：

```
//再对边界点进行处理(保留
int t = 0;//第一行
for (int q = 0; q < width; q++)
{
    pnewimg[t * width + q] = pimg[t * width + q];
}
t = height - 1;//最后一行
for (int q = 0; q < width; q++)
{
    pnewimg[t * width + q] = pimg[t * width + q];
}
int q = 0;//第一列
for (int t = 0; t < height; t++)
{
    pnewimg[t * width + q] = pimg[t * width + q];
}
q = width - 1;//最后一列
for (int t = 0; t < height; t++)
{
    pnewimg[t * width + q] = pimg[t * width + q];
}
```

```
namedWindow("高斯平滑", WINDOW_NORMAL);
imshow("高斯平滑", imgnew); //显示变换后的图像
imwrite("smoothimg.bmp", imgnew);//保存图片
waitKey();
```

3.1.2 计算图像中每个像素点的梯度幅值和方向：

先定义阶乘，再通过递归函数分别获得 Sobel 平滑算子和差分算子，再进行可分离卷积运算：

```
/*********************Sobel算子*************************/
//阶乘
int factorial(int n)
{
    int fac = 1;
    //0的阶乘
    if (n == 0)
        return fac;
    for (int i = 1; i <= n; ++i)
    {
        fac *= i;
    }
    return fac;
}

//获得Sobel平滑算子
Mat getSobelSmoooth(int wsize)
{
    int n = wsize - 1;
    Mat SobelSmooothoper = Mat::zeros(Size(wsize, 1), CV_32FC1);
    for (int k = 0; k <= n; k++)
    {
        float* pt = SobelSmooothoper.ptr<float>(0);//指向第一行第一个元素
        pt[k] = factorial(n) / (factorial(k) * factorial(n - k));
    }
    return SobelSmooothoper;
}
```

```
//获得Sobel差分算子
Mat getSobeldiff(int wsize)
{
    Mat Sobeldiffoper = Mat::zeros(Size(wsize, 1), CV_32FC1);
    Mat SobelSmoooth = getSobelSmoooth(wsize - 1);
    for (int k = 0; k < wsize; k++) {
        if (k == 0)
            Sobeldiffoper.at<float>(0, k) = 1;
        else if (k == wsize - 1)
            Sobeldiffoper.at<float>(0, k) = -1;
        else
            Sobeldiffoper.at<float>(0, k) = SobelSmoooth.at<float>(0, k) - SobelSmoooth.at<float>(0, k - 1);
    }
    return Sobeldiffoper;
}

//可分离卷积——先垂直方向卷积，后水平方向卷积
void sepConv2D_Y_X(Mat& src, Mat& dst, Mat kernel_Y, Mat kernel_X, int ddepth, Point anchor , int delta, int borderType )
{
    Mat dst_kernel_Y;
    filter2D(src, dst_kernel_Y, ddepth, kernel_Y, anchor, delta, borderType); //垂直方向卷积
    filter2D(dst_kernel_Y, dst, ddepth, kernel_X, anchor, delta, borderType); //水平方向卷积
}

//可分离卷积——先水平方向卷积，后垂直方向卷积
void sepConv2D_X_Y(Mat& src, Mat& dst, Mat kernel_X, Mat kernel_Y, int ddepth, Point anchor , int delta , int borderType )
{
    Mat dst_kernel_X;
    filter2D(src, dst_kernel_X, ddepth, kernel_X, anchor, delta, borderType); //水平方向卷积
    filter2D(dst_kernel_X, dst, ddepth, kernel_Y, anchor, delta, borderType); //垂直方向卷积
}
```

分别使用两种可分离卷积得到垂直边缘和水平边缘：

```
//Sobel算子边缘检测
//dst_X 垂直方向
//dst_Y 水平方向
void Sobel(Mat& src, Mat& dst_X, Mat& dst_Y, Mat& dst, int wsize, int ddepth, Point anchor , int delta, int borderType )
{
    Mat SobelSmooothoper = getSobelSmoooth(wsize); //平滑系数
    Mat Sobeldiffoper = getSobeldiff(wsize); //差分系数

    //可分离卷积——先垂直方向平滑，后水平方向差分——得到垂直边缘
    sepConv2D_Y_X(src, dst_X, SobelSmooothoper.t(), Sobeldiffoper, ddepth);

    //可分离卷积——先水平方向平滑，后垂直方向差分——得到水平边缘
    sepConv2D_X_Y(src, dst_Y, SobelSmooothoper, Sobeldiffoper.t(), ddepth);

    //边缘强度（近似）
    dst = abs(dst_X) + abs(dst_Y);
    convertScaleAbs(dst, dst); //求绝对值并转为无符号8位图
}

//确定一个点的坐标是否在图像内
bool checkInRang(int r, int c, int rows, int cols)
{
    if (r >= 0 && r < rows && c >= 0 && c < cols)
        return true;
    else
        return false;
}
```

```
//从确定边缘点出发，延长边缘
void trace(Mat& edgeMag_noMaxsup, Mat& edge, float TH, int r, int c, int rows, int cols)
{
    if (edge.at<uchar>(r, c) == 0)
    {
        edge.at<uchar>(r, c) = 255;
        for (int i = -1; i <= 1; ++i) //继续判断该边缘点3*3邻域内是否有大于高阈值的点
        {
            for (int j = -1; j <= 1; ++j)
            {
                float mag = edgeMag_noMaxsup.at<float>(r + i, c + j);
                if (checkInRang(r + i, c + j, rows, cols) && mag >= TH)
                    trace(edgeMag_noMaxsup, edge, TH, r + i, c + j, rows, cols);
            }
        }
    }
}
```

分别计算梯度方向和梯度幅值：

```
//计算梯度幅值
Mat edgeMag;
if (L2graydient)
    magnitude(dx, dy, edgeMag); //开平方
else
    edgeMag = abs(dx) + abs(dy); //绝对值之和近似
```

```
edgeMag.convertTo(edgeMag, CV_8UC1, 1.0, 0);//将结果转换 CV_8U
namedWindow("梯度幅值", WINDOW_NORMAL);
imshow("梯度幅值", edgeMag); //显示变换后的图像
waitKey();
```

```
//计算梯度方向 以及 非极大值抑制
Mat edgeMag_noMaxsup = Mat::zeros(rows, cols, CV_32FC1);
for (int r = 1; r < rows - 1; ++r)
{
    for (int c = 1; c < cols - 1; ++c)
    {
        float x = dx.at<float>(r, c);
        float y = dy.at<float>(r, c);
        float angle = std::atan2f(y, x) / CV_PI * 180; //当前位置梯度方向
        float mag = edgeMag.at<float>(r, c);  //当前位置梯度幅值
```

3.1.3 非极大值抑制：

分别对其垂直边缘、水平边缘、+45°边缘和+135°边缘的各方向上进行比较，非极大值抑制，从而对边缘进行细化处理：

```
//非极大值抑制
//梯度方向为水平方向-3*3邻域内左右方向比较
if (abs(angle) < 22.5 || abs(angle) > 157.5)
{
    float left = edgeMag.at<float>(r, c - 1);
    float right = edgeMag.at<float>(r, c + 1);
    if (mag >= left && mag >= right)
        edgeMag_noMaxsup.at<float>(r, c) = mag;
}

//梯度方向为垂直方向-3*3邻域内上下方向比较
if ((angle >= 67.5 && angle <= 112.5) || (angle >= -112.5 && angle <= -67.5))
{
    float top = edgeMag.at<float>(r - 1, c);
    float down = edgeMag.at<float>(r + 1, c);
    if (mag >= top && mag >= down)
        edgeMag_noMaxsup.at<float>(r, c) = mag;
}

//梯度方向为-45°-3*3邻域内右上左下方向比较
    if ((angle > 112.5 && angle <= 157.5) || (angle > -67.5 && angle <= -22.5))
{
    float right_top = edgeMag.at<float>(r - 1, c + 1);
    float left_down = edgeMag.at<float>(r + 1, c - 1);
    if (mag >= right_top && mag >= left_down)
        edgeMag_noMaxsup.at<float>(r, c) = mag;
}
```

```
        //梯度方向为+45°-3*3邻域内右下左上方向比较
            if ((angle >= 22.5 && angle < 67.5) || (angle >= -157.5 && angle < -112.5))
        {
            float left_top = edgeMag.at<float>(r - 1, c - 1);
            float right_down = edgeMag.at<float>(r + 1, c + 1);
            if (mag >= left_top && mag >= right_down)
                edgeMag_noMaxsup.at<float>(r, c) = mag;
        }
    }
}
edgeMag_noMaxsup.convertTo(edgeMag_noMaxsup, CV_8UC1, 1.0, 0);//将结果转换 CV_8U 类型
namedWindow("非极大值抑制", WINDOW_NORMAL);
imshow("非极大值抑制", edgeMag_noMaxsup); //显示变换后的图像
waitKey();
```

3.1.4 双阈值处理及边缘连接

双阈值处理（根据图像选取合适的高阈值和低阈值，通常高阈值是低阈值的 2 到 3 倍），大于高阈值确定为边缘，小于低阈值则被抑制，灰度值在高低阈值之间的边缘点，若与强边缘连接，保留并连接，否则抑制：

```
//双阈值处理及边缘连接
edge = Mat::zeros(rows, cols, CV_8UC1);
for (int r = 1; r < rows - 1; ++r)
{
    for (int c = 1; c < cols - 1; ++c)
    {
        float mag = edgeMag_noMaxsup.at<float>(r, c);
        //大于高阈值，确定为边缘点
        if (mag >= TH)
            trace(edgeMag_noMaxsup, edge, TH, r, c, rows, cols);
        //小于低阈值，排除为边缘点
        else if (mag < TL)
            edge.at<uchar>(r, c) = 0;
    }
}
```

3.2 Hough 变换直线检测

具体步骤：

3.2.1 定义类

声明类，类成员变量，类成员函数，包括直线检测函数的调用。

```
class LineFinder {
private:
    double delta_rho;
    double delta_theta;
    int threshold;
public:
    LineFinder() {
        delta_rho = 1;
        delta_theta = PI / 180;
        threshold = 80;
    }
    void setAccResolution(double dRho, double dTheta) {
        delta_rho = dRho;
        delta_theta = dTheta;
    }
    void setthreshold(int minv) {
        threshold = minv;
    }
    void findLines(Mat& binary, Mat& image_output) {
        hough_lines(binary, image_output, delta_rho, delta_theta, threshold);
        //HoughLines(binary, lines, delta_rho, delta_theta, threshold);
    }
};
```

3.2.2.霍夫变换

得到图像的指针，步长，图像的宽和高，霍夫变换后角度和距离的个数；分别定义地址指针，为排列数组和正余弦列表分配内存空间。

```
void hough_lines(Mat& img, Mat& image_output, float rho, float theta, int threshold) {
    AutoBuffer<int> _accum, _sort_buf;
    AutoBuffer<float> _tabSin, _tabCos;

    const uchar* image;
    int step, width, height;
    int numangle, numrho;
    int total = 0;
    int i, j;
    float irho = 1 / rho;
    double scale;

    image = img.ptr();     //得到图像的指针
    step = img.step;     //得到图像的步长
    width = img.cols;     //得到图像的宽
    height = img.rows;     //得到图像的高
    //由角度和距离的分辨率得到角度和距离的数量，即霍夫变换后角度和距离的个数
    numangle = cvRound(CV_PI / theta);
    numrho = cvRound(((width + height) * 2 + 1) / rho);

    _accum.allocate((numangle + 2) * (numrho + 2));
    //为排序数组分配内存空间
    _sort_buf.allocate(numangle * numrho);
    //为正弦和余弦列表分配内存空间
    _tabSin.allocate(numangle);
    _tabCos.allocate(numangle);
    //分别定义上述内存空间的地址指针
    int* accum = _accum, * sort_buf = _sort_buf;
    float* tabSin = _tabSin, * tabCos = _tabCos;
    //累加器数组清零
    memset(accum, 0, sizeof(accum[0]) * (numangle + 2) * (numrho + 2));
```

3.2.3

为避免重复运算，事先计算好 $\sin\theta_i/\rho$ 和 $\cos\theta_i/\rho$ 。只对图像进行非零值处理，即只对图片边缘像素进行霍夫变换。在累加器中找到距离和角点在霍夫空间中对应的位置。

```
float ang = 0;
//为避免重复运算，事先计算好sinθi/ρ和cosθi/ρ
for (int n = 0; n < numangle; ang += theta, n++)
{
    tabSin[n] = (float)(sin((double)ang) * irho);
    tabCos[n] = (float)(cos((double)ang) * irho);
}

for (i = 0; i < height; i++)
    for (j = 0; j < width; j++)
    {
        //只对图像的非零值处理，即只对图像的边缘像素进行霍夫变换
        if (image[i * step + j] != 0)
            for (int n = 0; n < numangle; n++)
            {
                int r = cvRound(j * tabCos[n] + i * tabSin[n]);
                r += (numrho - 1) / 2;
                //r表示的是距离，n表示的是角点，在累加器内找到它们所对应的位置（即霍夫空间内的位置），其值加1
                accum[(n + 1) * (numrho + 2) + r + 1]++;
            }
    }
```

3.2.4 依次对累加器中的值进行阈值判断，把符合条件的值存入缓冲器。

```
for (int r = 0; r < numrho; r++)
    for (int n = 0; n < numangle; n++)
    {
        //得到当前值在累加器数组的位置
        int base = (n + 1) * (numrho + 2) + r + 1;
        if (accum[base] > threshold &&    //必须大于所设置的阈值
            //在4邻域内进行非极大值抑制
            accum[base] > accum[base - 1] && accum[base] >= accum[base + 1] &&
            accum[base] > accum[base - numrho - 2] && accum[base] >= accum[base + numrho + 2])
            //把极大值位置存入排序数组内—sort_buf
            sort_buf[total++] = base;
    }
```

3.2.5 定义一个二维浮点数向量，将缓冲器中的极大值按顺序存储到向量中，该向量就是输出直线的序列。

```
scale = 1. / (numrho + 2);
vector<Vec2f> lines;
lines.clear();
for (i = 0; i < total; i++)
{
    Vec2f temp;
    //idx为极大值在累加器数组的位置
    int idx = sort_buf[i];
    //分离出该极大值在霍夫空间中的位置
    int n = cvFloor(idx * scale) - 1;
    int r = idx - (n + 1) * (numrho + 2) - 1;
    //最终得到极大值所对应的角度和距离
    temp[0] = (r - (numrho - 1) * 0.5f) * rho;
    temp[1] = n * theta;
    //存储到序列内
    //cout << temp[0] << endl;
    //cout << temp[1] << endl;
    lines.push_back(temp);
}
```

3.2.6 使用迭代器读取向量中的值，求得直线和图像边缘的交点。将直线上的点设置为（0,0，255），可以在图像上画出红色的线。

```
vector<Vec2f>::const_iterator it = lines.begin();
while (it != lines.end())
{
    float rho = (*it)[0];
    float theta = (*it)[1];
    if (theta < PI / 4. || theta > 3. * PI / 4.)
    {
        Point pt1(rho / cos(theta), 0);
        Point pt2((rho - image_output.rows * sin(theta)) / cos(theta), image_output.rows);
        line(image_output, pt1, pt2, Scalar(0, 0, 255), 1);
    }
    else
    {
        Point pt1(0, rho / sin(theta));
        Point pt2(image_output.cols, (rho - image_output.cols * cos(theta)) / sin(theta));
        line(image_output, pt1, pt2, Scalar(0, 0, 255), 1);
    }
    ++it;
}
```

3.2.7 读取图像，转换为灰度图像，进行 Canny 边缘检测，得到二值化图像，对二值化图像进行霍夫变换，输出结果。

```
int main(int argc, char* argv[])
{
    Mat image_input = imread("C://test.bmp");
    Mat image_gray;
    Mat contours;
    cvtColor(image_input, image_gray, COLOR_RGB2GRAY);
    //Canny(image_gray, contours, 190, 300);
    Edge_Canny(image_gray, contours, 190, 300, 3, false);
    Mat image_output(contours.rows, contours.cols, CV_8U, Scalar(255));
    image_input.copyTo(image_output);
    LineFinder finder;
    finder.setthreshold(130);
    finder.findLines(contours, image_output);

    imshow("output", image_output);
    waitKey(0);
    return 0;
}
```

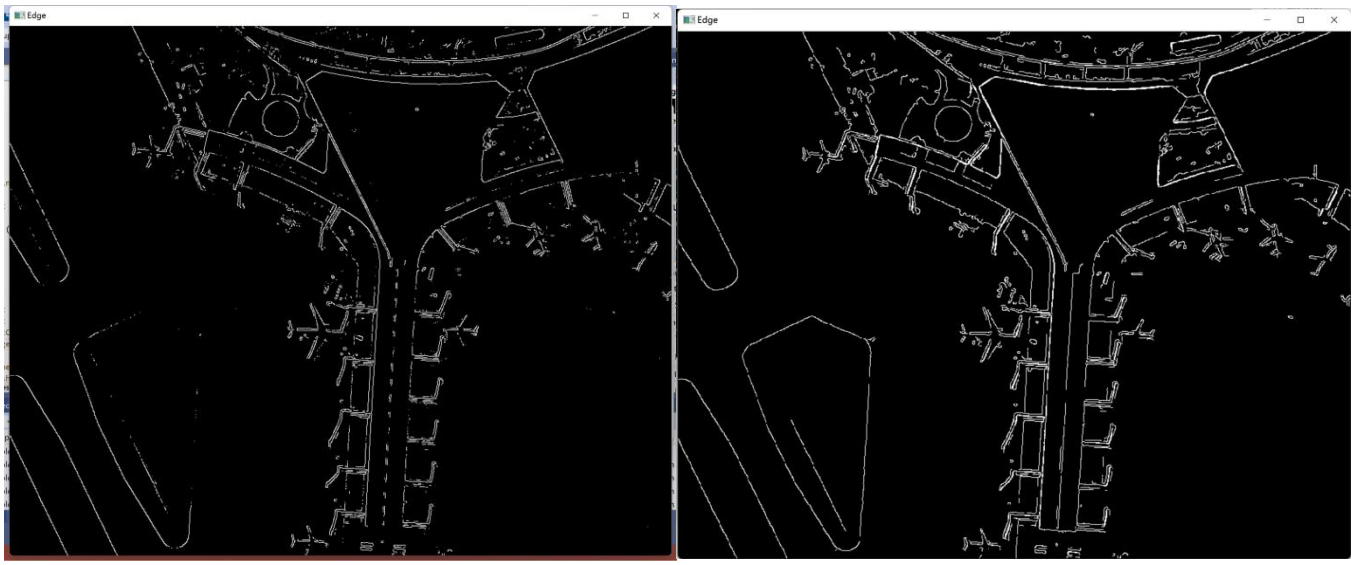
4. 结果与分析

（1）实习中遇到的问题和解决方法：

1. Hough 直线检测函数在输出直线序列时，存在问题，为了方便起见，将划线函数移至直线检测函数中，并且将原始图像一并输入，直接输出划线后图像。

```
vector<Vec2f>::const_iterator it = lines.begin();
while (it != lines.end())
{
    float rho = (*it)[0];
    float theta = (*it)[1];
    if (theta < PI / 4. || theta > 3. * PI / 4.)
    {
        Point pt1(rho / cos(theta), 0);
        Point pt2((rho - image_output.rows * sin(theta)) / cos(theta), image_output.rows);
        line(image_output, pt1, pt2, Scalar(0, 0, 255), 1);
    }
    else
    {
        Point pt1(0, rho / sin(theta));
        Point pt2(image_output.cols, (rho - image_output.cols * cos(theta)) / sin(theta));
        line(image_output, pt1, pt2, Scalar(0, 0, 255), 1);
    }
    ++it;
}
```

2. 边缘算子阈值选取的问题



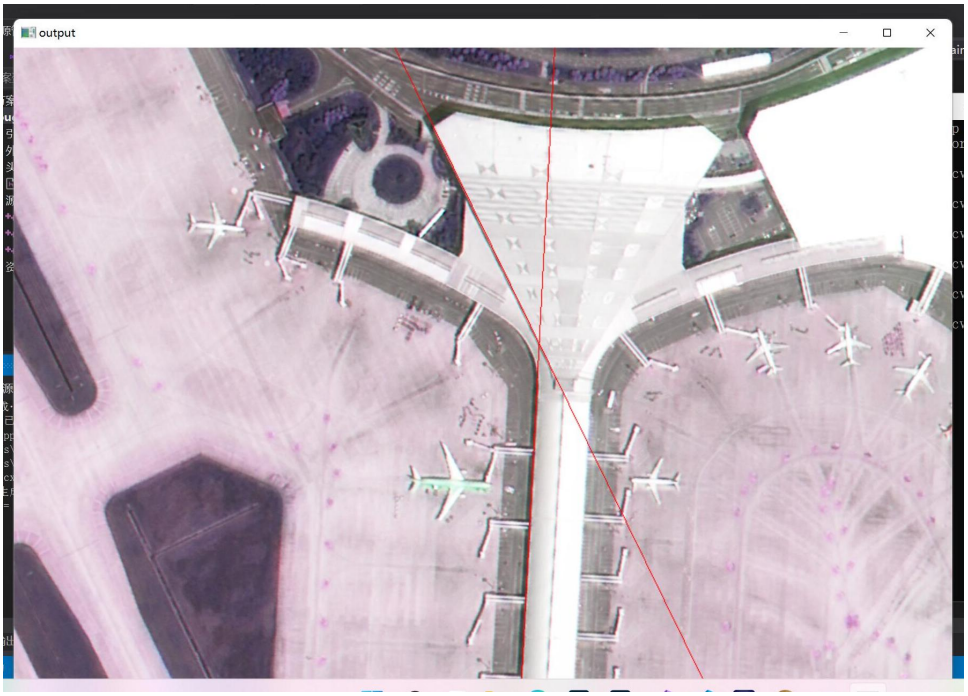
设置不同的阈值，边缘检测算子的结果会有较大的差异直线检测也会有不同的效果，小于阈值 1 被认定为不是边缘点，大于阈值 2 被认定为边缘点，介于两者之间则要追踪周围的点是否是边缘点。目前我们组设置阈值的方法还是不断尝试，然后肉眼感觉怎么设置效果会更好，日后还可以在选择阈值方面继续学习，结合自学习或其他方法找出最合适的阈值。

3. 阈值与直线检测的敏感性有关

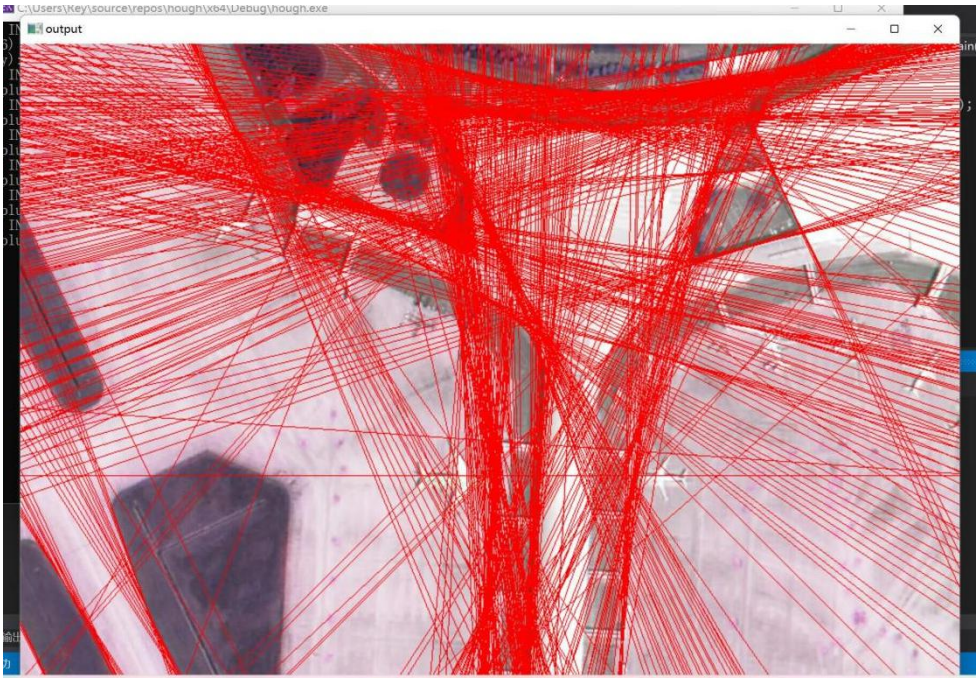
```
LineFinder finder;
finder.setthreshold(80);
finder.findLines(contours, image_output);
```

当阈值设置为 200 时，可以发现得到的直线数量过少；当阈值设置小于 50 时，可以发现得到的直线数量过多。经过测试，认为 100 是比较理想的阈值设定。

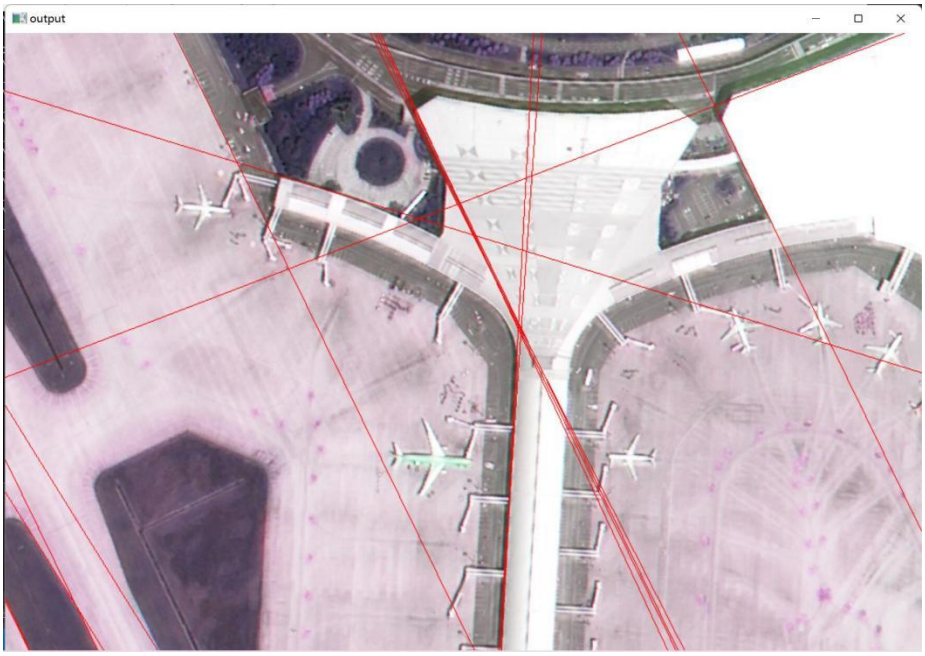
具体实例见下图。



（200）



（50）



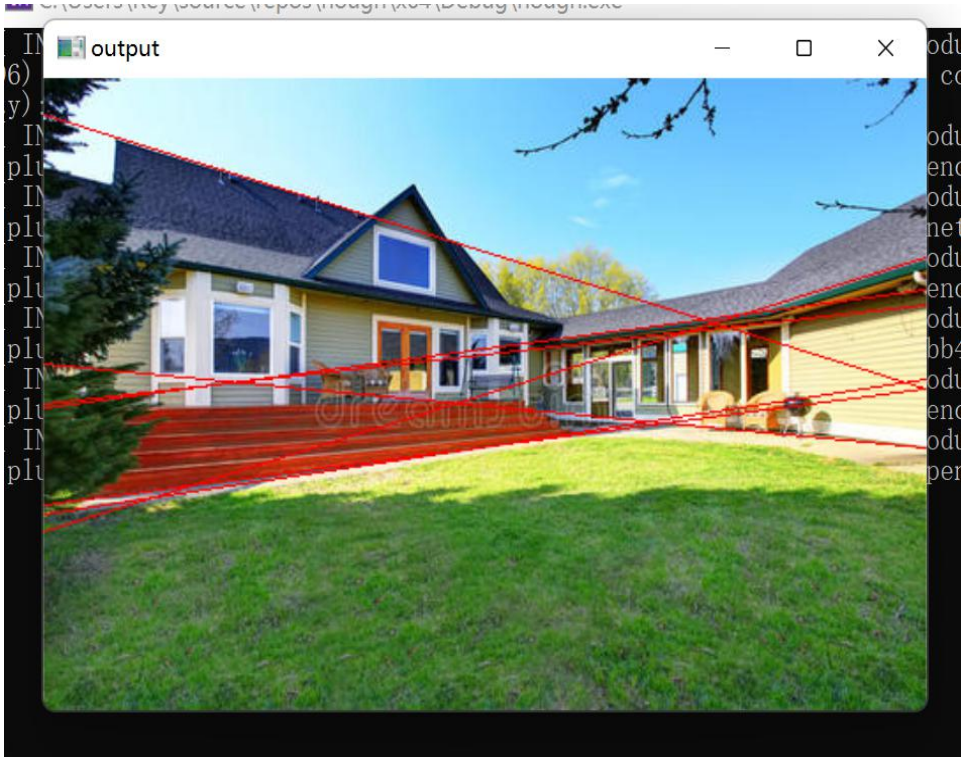
（100）

（2）结果分析

首先是边缘检测算子：高斯滤波使图像边缘更平滑，有利于后续边缘检测：噪声减少，边缘提取更完整等；Canny 边缘检测提取灰度图像边缘生成二值图像，为后续 Hough 检测打下基础。

结论：我们小组提取机场边缘的效果不错，但是对比度不明显的边缘比如道路分界线很容易被忽略，另外许多细小的边缘没有明显的形状，不过总体来说效果不错。

其他图片实现



5. 结论

在本次实习中，我们小组成员分工协作，有条有理地布置着各项工作，最后也取得了比较令人满意的成果。首先我们完成了本次实习的研究内容，即运用 Hough 变换，识别图像中的直线。研究包括读取图像、检测图像边缘转换图像成二值图像、核心算法 Hough 变换等在内的主要算法来实现对图像中直线的检测。小组成员间互相帮助，合作解决问题，改进了部分代码。在完成既定学习任务的情况下，成员们提出并解决问题，实现了理论高度上的升华，为之后的相关学习打下了基础。

6. 小组成员分工说明

殷雨薇、雷轶主要负责代码部分，德丽达、黄淼主要负责实习报告部分，PPT 展示部分由大家共同完成。虽然大体上分成了两部分，但组员之间依旧互帮互助，一起探讨并解决遇到的问题，使实习效率大大提高。

7. 实习心得与体会

01 雷轶

能够使用 opencv，了解了图像在计算机中输入输出流的形式，加深对图像变换背后的算法的理解，加深对图像存储的数据结构的理解，在算法层掌握图像处理，和在用户界面层相比，有更高的效率，在大量图像数据以及批量处理中可以减少计算量、时间、人力等资源的占用，对学习和科研都有很大帮助。了解图像处理的本质，在使用图像处理软件时也能更加合理地使用软件的功能，获得更好的效果。

通常在图像处理中，提高图像的信噪比是主要目标，在退化图像或者极端条件下的影像中，信噪比往往很低，这时需要进行图像复原或者图像增强，使用合理的图像变换对提高信噪比至关重要，了解各种图像变换的算法原理，有利于分析各种变换的利弊，以便选择最优方案。

02 黄淼

本次实习是基于 VS2017+opencv4.5.5 环境，进行一些对图像的基本处理和操作。之前进行理论课课间实习的时候，都是通过使用 PS 或者 Matlab 等软件对图像本身进行处理，运用一些简单的代码指令就可以得到理想结果。但在这次的实习中，我们不再直接使用已有的平滑、二值化、边缘检测等指令操作，而是从图像底层的每个点的每个通道的值着手，由点到面，由局部到整体，通过自己编写代码完成这些操作，进而实现对图像的处理，极大程度上的减少了我们对代码一知半解的调用的情况。

同时，在实习过程中，我接触到了很多思路新奇、效率极高的算法，通过一系列内化转化为自己可以运用的知识，丰富了我的知识储备。我在本次实习中也认识到了自己的一些不足，独立编写算法的能力还有很大的欠缺，环境配置不规范，代码经常报错，对于读取图片时的绝对路径和相对路径问题还存在一些问题，部分代码运行成功后效果一般，仍需改进。通过本次实习，我巩固了数字图像处理的基本原理和算法，对理论课程知识有了更深的了解，也更加了解了图像处理的原理和本质，初步掌握了数字图像处理的编程实践技能，为以后更专业化的图像处理学习打下了基础，培养解决问题的综合应用能力，也进行了团队合作，锻炼了口头及书面表达与沟通能力。

03 殷雨薇

本次实习将数字图像处理的理论知识与遥感图像处理的具体实践结合起来，加深了我对数字图像处理原理与算法的理解，增强了我的编程实践技能。本次实习总时长为 6 天，共分为个人及小组合作两个部分。总体而言任务不算多，时间较为充裕，给予了我们充分思考准备及后期完善的时间。

个人部分我完成了灰度线性变换、高通滤波器、低通滤波器、中值滤波器、图像几何处理以及色彩平衡，均为数图理论课程中介绍到了的有关处理。结合课本的细致讲解、实习参考书上的代码思路以及上网查阅学习，我顺利在前三天完成了该板块任务。在思考编写代码的过程中，我掌握了 opencv 库与 Mat 类的基本用法，巩固了图像处理相关原理，对 RGB 图像存储方式产生了新的认识。

在小组合作部分，我主要承担的是 Canny 算法代码的编写及该部分的 PPT 制作与汇报，该项工作极大地提高了我对以 Canny 算法为代表的边缘检测算法的理解，锻炼了我的编程能力。我还认识到，交流合作是学习中不可缺少的一环，分工合作是在考察我们的团队协作能力，同时也体现了将复杂问题分而治之的思想。在这次实习中，我们小组拿到问题后迅速进行了明确的分工，这使我们效率较高，顺利完成了任务。

最后，感谢学院开设该门课程让我们更好地学习实践，感谢老师在实习期间给予的指导，感谢同组成员的细心帮助。总体而言本次实习收获颇丰。

04 德丽达

在本次实习中，基于 OpenCV 和 VS 环境，以个人为单位，实现了一些数字图像处理的基本操作，以小组为单位，实现了 Hough 变换检测直线较为复杂的操作。相较于以前学到的知识，通过此次的实践，对于各类算子、高斯滤波、Hough 变换等的算法原理有了更深刻的理解和掌握。这学期在数图课上学到的有些知识还是限于书本，甚至有的知识只是单纯背诵了名词解释，像卷积、二维高斯等从函数公式到代码运行的过程是在实习中反复学习理解的，这也是实习最主要的意义和目的。

我在小组任务中负责了实习报告的相关内容，首先结合指导书和资料理解清楚原理，然后思考代码实现，进行编注，有时还需要在网上搜一些代码实例的资料并分析，为己所用。期间遇到一些理解上或者编程上的问题，都通过与小组内的同学进行讨论、互查，得到了很好的解决。

总之，此次数图实习，让我们的自主编程能力、团队协作能力、知识应用能力等均有了很大的提升，也让我发现了小组其他成员的一些优点值得我去学习。