第 5 章

四旋翼飞行器图像采集处理专题

四旋翼飞行器通过读取陀螺仪的数据进行姿态稳定控制，其本质是以自身为参考坐标系来稳定控制目标，由于确定空间的绝对位置较困难，所以单纯地采用陀螺仪进行位置控制有较大的缺陷。本章单独开辟一个专题，作为前面章节的补充，介绍采用图像处理的方式实现绝对位置控制。

5.1 总体概述

为实现空间平面的精确位置控制，在四旋翼飞行器设计中加入摄像头模块，型号为 MT9V034。单片机读取摄像头的黑白图像数据，经过一系列的图像算法，最终计算出四旋翼飞行器在空间中的绝对位置。

本次实现的环境设定为在四旋翼飞行器下方有一条与背景颜色分明的直线，先对读取的原始图像数据进行自适应中值滤波处理，再通过 Sobel 边沿提取算法将处理后的图像数据进行边沿提取，使用 OTSU 二值化算法将含有边沿信息的图像数据前景与背景分开进行二值化处理，使用并行 Zhang 图像细化算法将已经二值化的边沿部分细化成单一像素，最后使用概率累加霍夫变换提取已经细化的图像数据中的直线信息，如直线在图像中的起始坐标、终止坐标、倾斜斜率等，通过这些数据就可以进行四旋翼飞行器的空间位置控制，图像处理流程如图 5-1 所示。

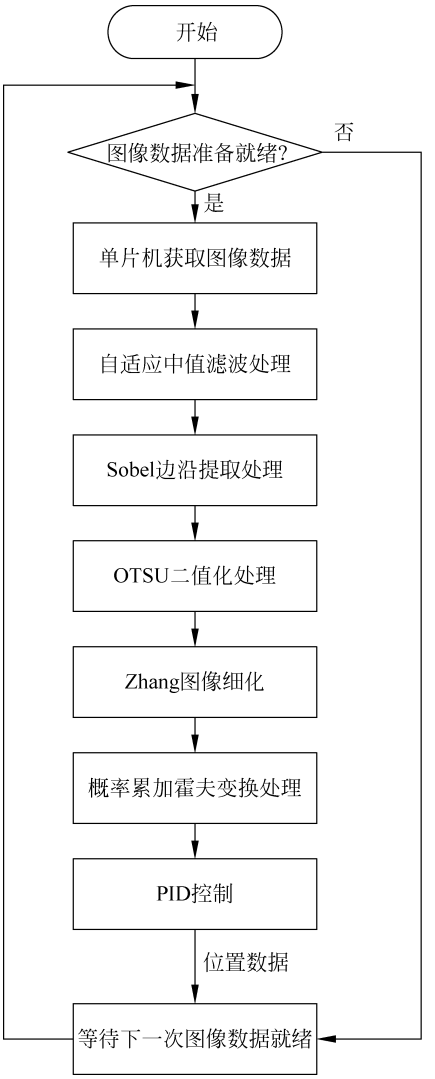


图 5-1 图像处理流程

5.2 图像算法

本节对图像处理中用到的自适应中值滤波算法、Sobel 边沿提取算法、OTSU 二值化算法、Zhang 图像细化算法以及概率累加霍夫变换算法进行说明，最终实现图像处理数据的有效使用。

5.2.1 自适应中值滤波算法

图像在成像、数值化和传输等过程中容易受到各种干扰效应形成噪声。这些噪声在图像中与物体边缘很相似。在进行图像平滑处理时，追求的目标是既能消除掉这些噪声又不使图像的边缘轮廓和线条细节变模糊。常用的图像平滑处理方法主要有邻域平均法、低通滤波法和中值滤波法等。当图像中含有大量的椒盐噪声(脉冲噪声)时，使用中值滤波法效果尤为明显。

常规中值滤波法去除脉冲噪声的性能受滤波窗口尺寸的影响较大，而且它在抑制图像噪声和保护细节两方面存在一定的矛盾：滤波窗口小，可较好地保护图像中某些细节，但滤除噪声的能力会受到限制；反之，滤波窗口大，可加强噪声抑制能力，但对细节的保护能力会减弱，有时会滤去图像中的一些细线、尖锐边角等重要细节，从而破坏图像的几何结构。这种矛盾在图像中噪声干扰较大时表现得尤为明显。根据经验，在脉冲噪声强度大于 0.2 时，常规中值滤波法的效果就不令人满意。但是，由于常规中值滤波器所使用的滤波窗口大小是固定不变的，所以，在选择窗口大小和保护细节两方面只能做到二选一，这样，矛盾就始终不能解决。因此，需要寻求其他的改进算法来解决这一矛盾。

自适应中值滤波器的滤波方式和常规的中值滤波器一样，都使用一个矩形区域的窗口 S_{xy}，不同的是在滤波过程中，自适应滤波器会根据一定的设定条件改变(即增加)滤波窗口的大小，同时当判断滤波窗中心的像素是噪声时，最终值用中值代替，否则不改变其当前像素值，这样用滤波器的输出来替代像素值。自适应中值滤波器可以处理噪声概率更大的脉冲噪声，同时能够更好地保持图像细节，这是常规中值滤波器很难做到的。

自适应中值滤波总体上可以分为三步：

(1) 对图像各区域进行噪声检测；

(2) 根据各区域受噪声污染的状况确定滤波窗口的尺寸；

(3) 对检测出的噪声点进行滤波。

在 S_{xy} 定义的滤波器区域内定义如下变量：

- Z_{min}：S_{xy} 中的最小灰度值；
- Z_{max}：S_{xy} 中的最大灰度值；
- Z_{med}：S_{xy} 中的灰度值的中值；
- Z_{xy}：坐标中的灰度值；
- S_{max}：S_{xy} 允许的最大尺寸。

自适应中值滤波算法包含两个进程，分别表示为进程 A 和进程 B。

进程 A：

$$A_1 = Z_{med} - Z_{min}$$

$$A_2 = Z_{med} - Z_{max}$$

说明：如果 $A_1 > 0$ 且 $A_2 < 0$，则转至进程 B 否则增大窗口尺寸；

如果窗口尺寸$\leqslant S_{max}$，则重复进程 A，否则输出 Z_{med}。

进程 B：

$$B_1 = Z_{xy} - Z_{min}$$

$$B_2 = Z_{xy} - Z_{max}$$

说明：如果 $B_1 > 0$ 且 $B_2 < 0$，则输出 Z_{xy}，否则输出 Z_{med}。

处理效果如图 5-2 所示。

(a) 原始图像　(b) 椒盐噪声图像

(c) 中值滤波图像　(d) 自适应中值滤波图像

图 5-2　自适应中值滤波算法效果

软件实现方法如下：

```
void image_RAMF(u8 * ptr,u8 * out)
{
u8 median[9] = {0},temp,median_25[25] = {0};
u8 buf[FULL_IMAGE_COLUMN_SIZE - 2] = {0};
// (1) 设置图像宽度大小为 118
for(u8 i = 0;i < FULL_IMAGE_ROW_SIZE;i++)
    {
// (2) 设置图像长度大小为 158
```

```
for(u8 j = 0;j < FULL_IMAGE_COLUMN_SIZE;j++)
{
// (3) 跳过视频图像数据边界
if(i < 2||j < 2||i >(FULL_IMAGE_ROW_SIZE3)||j >(FULL_IMAGE_COLUMN_SIZE - 3))
                  {
                 out[i * FULL_IMAGE_COLUMN_SIZE + j] = ptr[i * FULL_IMAGE_COLUMN_SIZE + j];
                  }
                  else{
// (4) 分别把 9 个像素放入 median 中
median[0] = ptr[(i1) * FULL_IMAGE_COLUMN_SIZE + (j - 1)];
median[1] = ptr[(i1) * FULL_IMAGE_COLUMN_SIZE + (j)];
median[2] = ptr[(i1) * FULL_IMAGE_COLUMN_SIZE + (j + 1)];
median[3] = ptr[(i) * FULL_IMAGE_COLUMN_SIZE + (j - 1)];
median[4] = ptr[(i) * FULL_IMAGE_COLUMN_SIZE + (j)];
median[5] = ptr[(i) * FULL_IMAGE_COLUMN_SIZE + (j + 1)];
median[6] = ptr[(i + 1) * FULL_IMAGE_COLUMN_SIZE + (j - 1)];
median[7] = ptr[(i + 1) * FULL_IMAGE_COLUMN_SIZE + (j)];
median[8] = ptr[(i + 1) * FULL_IMAGE_COLUMN_SIZE + (j + 1)];
// (5) 对 9 个数据排序,选择中值
for (u8 y = 0; y < 8; y++)
{
for (u8 x = 0; x < 8 - y; x++)
{
if (median[x] > median[x + 1])
{
    temp = median[x];
    median[x] = median[x + 1];
    median[x + 1] = temp;
   }
    }
              }
// (6) 中值和最大值或最小值一样,选用 5 × 5 的中值滤波
if((median[4] == median[0])||(median[4] == median[8]))
    {
// (7) 分别把 25 个像素放入 median-25 中
median_25[0] = ptr[(i - 2) * FULL_IMAGE_COLUMN_SIZE + (j - 2)];
median_25[1] = ptr[(i - 2) * FULL_IMAGE_COLUMN_SIZE + (j - 1)];
median_25[2] = ptr[(i2) * FULL_IMAGE_COLUMN_SIZE + (j)];
median_25[3] = ptr[(i2) * FULL_IMAGE_COLUMN_SIZE + (j + 1)];
median_25[4] = ptr[(i2) * FULL_IMAGE_COLUMN_SIZE + (j + 2)];
median_25[5] = ptr[(i1) * FULL_IMAGE_COLUMN_SIZE + (j - 2)];
median_25[6] = ptr[(i - 1) * FULL_IMAGE_COLUMN_SIZE + (j - 1)];
```

```
median_25[7] = ptr[(i1) * FULL_IMAGE_COLUMN_SIZE + (j)];
median_25[8] = ptr[(i1) * FULL_IMAGE_COLUMN_SIZE + (j + 1)];
median_25[9] = ptr[(i1) * FULL_IMAGE_COLUMN_SIZE + (j + 2)];
median_25[10] = ptr[(i) * FULL_IMAGE_COLUMN_SIZE + (j - 2)];
median_25[11] = ptr[(i) * FULL_IMAGE_COLUMN_SIZE + (j - 1)];
median_25[12] = ptr[(i) * FULL_IMAGE_COLUMN_SIZE + (j)];
median_25[13] = ptr[(i) * FULL_IMAGE_COLUMN_SIZE + (j + 1)];
median_25[14] = ptr[(i) * FULL_IMAGE_COLUMN_SIZE + (j + 2)];
median_25[15] = ptr[(i + 1) * FULL_IMAGE_COLUMN_SIZE + (j - 2)];
median_25[16] = ptr[(i + 1) * FULL_IMAGE_COLUMN_SIZE + (j - 1)];
median_25[17] = ptr[(i + 1) * FULL_IMAGE_COLUMN_SIZE + (j)];
median_25[18] = ptr[(i + 1) * FULL_IMAGE_COLUMN_SIZE + (j + 1)];
median_25[19] = ptr[(i + 1) * FULL_IMAGE_COLUMN_SIZE + (j + 2)];
median_25[20] = ptr[(i + 2) * FULL_IMAGE_COLUMN_SIZE + (j - 2)];
median_25[21] = ptr[(i + 2) * FULL_IMAGE_COLUMN_SIZE + (j - 1)];
median_25[22] = ptr[(i + 2) * FULL_IMAGE_COLUMN_SIZE + (j)];
median_25[23] = ptr[(i + 2) * FULL_IMAGE_COLUMN_SIZE + (j + 1)];
median_25[24] = ptr[(i + 2) * FULL_IMAGE_COLUMN_SIZE + (j + 2)];
// (8) 排序,原理同上
for (u8 y = 0; y < 24; y++)
{
    for (u8 x = 0; x < 24 - y; x++)
    {
        if (median_25[x] > median_25[x + 1])
            {
        temp = median_25[x];
        median_25[x] = median_25[x + 1];
        median_25[x + 1] = temp;
            }
    }
}
if((median_25[12] == median_25[0])||(median_25[12] == median_25[24]))
        {
        out[i * FULL_IMAGE_COLUMN_SIZE + j] = median_25[12];
        }
else    {
        out[i * FULL_IMAGE_COLUMN_SIZE + j] = ptr[i * FULL_IMAGE_COLUMN_SIZE + j];
        }
}
else    {
        out[i * FULL_IMAGE_COLUMN_SIZE + j] = ptr[i * FULL_IMAGE_COLUMN_SIZE + j];
        }
```

```
                    }
                }
            }
        }
```

5.2.2 Sobel 边缘提取算法

边缘检测是为了将图像周围像素灰度有阶跃变化的像素检测出来，这些像素组成的集合就是该图像的边缘。比较常用的边缘检测方法就是考察每个像素在某个领域内灰度的变化，然后利用边缘临近一阶或二阶方向导数变化规律检测边缘，即边缘检测局部算法。

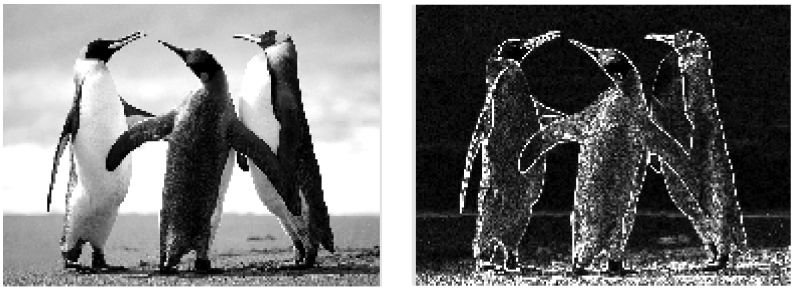
常用的边缘检测算子有 Sobel，Roberts，Prewitt，Laplace 等。Sobel 算子的检测过程是将图像的每一个点都用 Sobel 算子做卷积：一个用来检测垂直边缘，一个用来检测水平边缘，而最后两个卷积的最大值将作为该点的输出，即检测后的灰度。Sobel 算子包括两组 3×3 的矩阵，左边的表示垂直，右边的表示水平。将它与图像作平面卷积，即可分别得出垂直及水平的亮度差分近似值。Sobel 算子矩阵的一般形式如图 5-3 所示。

–1	–2	–1
0	0	0
1	2	1

–1	0	1
–2	0	2
–1	0	1

图 5-3　Sobel 算子(*Y* 方向，*X* 方向)

将 Sobel 算子矩阵与原始灰度图像做卷积，可得到对应的边沿图像，效果如图 5-4 所示。



(a) 原始图像　　(b) Sobel算法处理后

图 5-4　边缘提取算法效果

软件实现方法如下：

```
// (1) XY 方向的 Sobel 模板
const s8 sly[9] = { - 1 , - 2,  - 1,0 ,0 ,0,1 ,2 ,1};
const s8 slx[9] = { - 1, 0, 1, - 2 ,0, 2, - 1,0 ,1};
```

```
void image_ave( u8 const * ptr,u8 * out)
{
s16 sum_x = 0,sum_y = 0,sum = 0;;
u16 top,mid,low;
// (2) 设置图像宽度
for(u8 i = 0;i < FULL_IMAGE_ROW_SIZE;i++)
    {
        top = (i - 1) * FULL_IMAGE_COLUMN_SIZE;
        mid = i * FULL_IMAGE_COLUMN_SIZE;
        low = (i + 1) * FULL_IMAGE_COLUMN_SIZE;
// (3) 设置图像长度
for(u8 j = 0;j < FULL_IMAGE_COLUMN_SIZE;j++)
{
if(i == 0||j == 0||i == (FULL_IMAGE_ROW_SIZE - 1)||j >= (FULL_IMAGE_COLUMN_SIZE - 2))
            {
                out[i * FULL_IMAGE_COLUMN_SIZE + j] = 0;
            }
    else{
// (4) 图像分别乘 XY 方向的模板并相加
          sum_x = ((s16)ptr[top + j - 1] * slx[0])
                + ((s16)ptr[top + j] * slx[1])
                + ((s16)ptr[top + j + 1] * slx[2])
                + ((s16)ptr[mid + j - 1] * slx[3])
                + ((s16)ptr[mid + j] * slx[4])
                + ((s16)ptr[mid + j + 1] * slx[5])
                + ((s16)ptr[low + j - 1] * slx[6])
                + ((s16)ptr[low + j] * slx[7])
                + ((s16)ptr[low + j + 1] * slx[8]);
        sum_y = ((s16)ptr[top + j - 1] * sly[0])
                + ((s16)ptr[top + j] * sly[1])
                + ((s16)ptr[top + j + 1] * sly[2])
                + ((s16)ptr[mid + j - 1] * sly[3])
                + ((s16)ptr[mid + j] * sly[4])
                + ((s16)ptr[mid + j + 1] * sly[5])
                + ((s16)ptr[low - 1] * sly[6])
                + ((s16)ptr[low + j] * sly[7])
                + ((s16)ptr[low + j + 1] * sly[8]);
      sum = sum_x + sum_y;
             if(sum > 255)
                {sum = 255;}
            else if(sum < 0)
                {sum = 0;}
```

```
            out[mid + j] = sum;
                  }
            }
             }
}
```

5.2.3 OTSU 二值化算法

OTSU 算法也称最大类间差法，有时也称为大津算法，由大津于 1979 年提出，被认为是图像分割中阈值选取的最佳算法，其计算简单，不受图像亮度和对比度的影响，因此在数字图像处理上得到了广泛的应用。它是按图像的灰度特性，将图像分成背景和前景两部分。因方差是灰度分布均匀性的一种度量，背景和前景之间的类间方差越大，说明构成图像的两部分的差别越大，当部分前景错分为背景或部分背景错分为前景都会导致两部分差别变小。因此，使类间方差最大的分割意味着错分概率最小。

对于图像 $I(x,y)$，前景(即目标)和背景的分割阈值记作 T，属于前景的像素点数占整幅图像的比例记为 w_0，其平均灰度为 u_0；背景像素点数占整幅图像的比例为 w_1，其平均灰度为 u_1。图像的总平均灰度记为 u，类间方差记为 g。

假设图像的背景较暗，并且图像的大小为 $M\times N$，图像中像素的灰度值小于阈值 T 的像素个数记作 N_0，像素灰度大于阈值 T 的像素个数记作 N_1，则有：

$$w_0=\frac{N_0}{M\times N} \tag{5-1}$$

$$w_1=\frac{N_1}{M\times N} \tag{5-2}$$

$$u=w_0\times u_0+w_1\times u_1 \tag{5-3}$$

$$g=w_0\times(u_0-u)\text{^}2+w_1\times(u_1-u)\text{^}2 \tag{5-4}$$

将式(5-3)代入式(5-4)，得到等价公式：

$$g=w_0\times w_1\times(u_0-u)\text{^}2 \tag{5-5}$$

采用遍历的方法使类间方差 g 最大，此时的阈值为 T，即为所求。效果如图 5-5 所示。

(a) 原始图像　　(b) 二值算法处理后

图 5-5　OTSU 二值化算法效果

软件实现方法：用最大类间方差法求整张图像前景背景的阈值。

```
u8 otsu( u8 const * ptr)
{
    float histogram[256] = {0};
    u16 i = 0;
    float avgValue = 0;
    u8 threshold = 0;
    float maxVariance = 0;
    float w = 0, u = 0;
    float t = 0,variance = 0;
// (1) 直方图
for( i = 0;i < FULL_IMAGE_SIZE;i++)
{
    histogram[(u16)( * ptr++)]++;
}
for( i = 0; i < 256; i++)
    {
        histogram[i] = histogram[i] / FULL_IMAGE_SIZE;
    }
    for( i = 0; i < 256; i++)
    {
// (2) 设置整幅图像的平均灰度
    avgValue += i * histogram[i];
}
for( i = 0; i < 256; i++)
{
// (3) 假设当前灰度 i 为阈值, 确定 0～i 灰度的像素(假设像素值在此范围的像素叫作前景像素)
// 所占整幅图像的比例
w += histogram[i];
// (4) 灰度 i 之前的像素(0～i)的平均灰度值：前景像素的平均灰度值
    u += i * histogram[i];
    t = avgValue * w - u;
    variance = t * t / (w * (1 - w) );
    if(variance > maxVariance)
    {
        maxVariance = variance;
        threshold = i;
    }
}
// (5) 通过阈值对图像进行二值化处理
    void TT(u8 const * ptr,u8 * out,u16 threshold)
        {
        for( u16 i = 0;i < FULL_IMAGE_SIZE;i++)
            {
          if((ptr[i])> threshold)
        {
```

```
                out[i] = white;
            }
        else  { out[i] = black;}
    }
    }
```

5.2.4 并行 Zhang 图像细化算法

经过二值化后的图像由多个像素构成的直线组成,这样的图像数据不能直接给霍夫变换算法进行处理,需要进行图像细化。图像细化就是连续移除图像最外层的像素,将多像素的线条转化为单像素,在图像特征不改变的前提下减少信息的存储量,同时在图像骨架中提取图像的特征。因此,细化效果的好坏直接影响系统识别直线的性能。

(1) 细化算法一般要满足下面的要求:

(2) 细化后保证图像骨架的连通性;

(3) 细化后原图像的细节特征要保存完好;

(4) 原线条的中心线为细化的结果;

(5) 保存完好线条的端点;

(6) 交叉的线条细化后不能发生畸变;

(7) 细化的快速性。

根据是否采用迭代运算可以分为非迭代细化算法和迭代细化算法。非迭代算法不以像素为基础,一次运算就可以产生图像骨架,这种算法处理速度快但容易产生噪声点。迭代算法需要删除掉图像边缘满足一定条件的像素,使图像变为单像素带宽的骨架。根据检查像素所使用的不同方法,又把迭代细化算法分成串行细化算法和并行细化算法。在串行算法中,每次迭代都选择固定的顺序来检查像素,判断像素是否需要删除,像素是否删除取决于本次迭代中已处理过的像素点和前次迭代的结果。在并行算法中,像素点是否删除与像素图像中的顺序无关,仅取决于前次迭代的结果,因此在每次迭代中所有像素被以并行的方式独立检测。

目前已有很多种细化算法。如经典细化算法、Pavlidis 异步细化算法、Deutseh 算法、并行 Zhang 图像细化算法等。其中并行 Zhang 图像细化算法是一种实用的细化方法,它是一种 8 邻域并行细化算法,该细化算法能够较精确地保持直线、T 行交叉和拐角的特性,并且该算法需要的迭代次数少,运行速度快。但该算法仍存在丢失局部信息、细化后存在冗余像素等缺点。

P9	P2	P3
P8	P1	P4
P7	P6	P5

图 5-6 8 邻域系统

图 5-6 表示以前景二值像素 P1 为中心的 8 邻域系统,P2～P9 代表与 P1 相邻的 8 个像素点。

细化步骤如下:

(1) 循环所有前景像素点,对符合如下条件的像素点标记为删除。

① $2 \leqslant N(P1) \leqslant 6$;

② S(P1)＝1；

③ P2×P4×P6＝0；

④ P4×P6×P8＝0。

(2) 再次循环所有前景像素点，对符合如下条件的像素点标记为删除。

① 2≤N(P1)≤6；

② S(P1)＝1；

③ P2×P4×P8＝0；

④ P2×P6×P8＝0。

其中：

N(P1)表示跟 P1 相邻的 8 个像素点中为前景像素点的个数。

S(P1)表示从 P2～P9～P2 像素中出现 0～1 的累计次数，其中 0 表示背景，1 表示前景。

以上两步操作构成一次迭代，直至没有点再满足标记条件，这时剩下的点组成的区域即为细化后骨架。算法效果如图 5-7 所示。

(a) 原图像　　(b) 图像细化算法效果

图 5-7　并行 Zhang 图像细化算法处理效果

软件实现方法如下：

```
void thine(u8 * img_in)
{
u8 i = 0,j = 0,ap = 0,kk = 1;
u16 top = 0,mid = 0,low = 0,count = 0,c_count = 0,line_out[2];
u8 q2,q3,q4,q5,q6,q7,q8,q9;
while(kk)
{
    kk = 0;
    for(i = 1;i < 59;i++)
    {
        top = (i - 1) * cam_num;
        mid = (i) * cam_num;
        low = (i + 1) * cam_num;
```

```
            for(j = 1;j < 79;j++)
            {
            if(p1 == 0){continue;}
            if ((((q2 = p2) + (q3 = p3) + (q4 = p4) + (q5 = p5) + (q6 = p6) + (q7 = p7) +
(q8 = p8) + (q9 = p9)) >= 2) &&( (q2 + q3 + q4 + q5 + q6 + q7 + q8 + q9) <= 6))
                { ap = 0;
                    if (q2 == 0 && q3 == 1) {ap++; }
                    if (q3 == 0 && q4 == 1) {ap++; }
                    if (q4 == 0 && q5 == 1) {ap++; }
                    if (q5 == 0 && q6 == 1) {ap++; }
                    if (q6 == 0 && q7 == 1) {ap++; }
                    if (q7 == 0 && q8 == 1) {ap++; }
                    if (q8 == 0 && q9 == 1) {ap++; }
                    if (q9 == 0 && q2 == 1) {ap++; }
                    if ((ap == 1 )&&( (q2 * q4 * q6) == 0) && ((q4 * q6 * q8) == 0))
{
// (1) 找到需要删除的点的位置将其加入链表
                count++;
                kk = 1;
                c_count = mid + j;
                insertHeadList(&pthine,&c_count,0);
                                    }
            }
        }
    }
for(;count > 0;count -- )
{
// (2) 销毁链表将图像数据中对应的位置值置 0
    pop_del(&pthine, 1, line_out);
    img_in[line_out[0]] = 0;
        }

for(i = 1;i < 59;i++)
{
    top = (i - 1) * cam_num;mid = (i) * cam_num;low = (i + 1) * cam_num;
        for(j = 1;j < 79;j++)
        {
        if(p1 == 0){continue;}
        if ((((q2 = p2) + (q3 = p3) + (q4 = p4) + (q5 = p5) + (q6 = p6) + (q7 = p7) + (q8 =
p8) + (q9 = p9)) >= 2) &&( (q2 + q3 + q4 + q5 + q6 + q7 + q8 + q9) <= 6))
            { ap = 0;
                if (q2 == 0 && q3 == 1) {ap++; }
                if (q3 == 0 && q4 == 1) {ap++; }
                if (q4 == 0 && q5 == 1) {ap++; }
                if (q5 == 0 && q6 == 1) {ap++; }
                if (q6 == 0 && q7 == 1) {ap++; }
                if (q7 == 0 && q8 == 1) {ap++; }
```

```
                if (q8 == 0 && q9 == 1) {ap++; }
                if (q9 == 0 && q2 == 1) {ap++; }
        if ((ap == 1 )&&( (q2 * q4 * q8) == 0) && ((q2 * q6 * q8) == 0))
                    {
                    count++;
                    kk = 1;
                    c_count = mid + j;
                    insertHeadList(&pthine,&c_count,0);
                                }
                }
    }
}
        for(;count > 0;count -- )
        {
              pop_del(&pthine, 1, line_out);
              img_in[line_out[0]] = 0;
        }
}
clearList(&pthine);
}
```

5.2.5　概率累加霍夫变换算法

直线检测技术是要将直线在图像中的直线方程拟合出来。目前常用的算法有最小二乘法和霍夫变换，霍夫变换运用最小二乘法进行拟合，其优点就是它的速度非常快，只需遍历一次就可以计算出拟合曲线，但是它对噪声非常敏感，而且在对离散的点进行拟合时，首先必须知道那些点的情况，然后判断是采用直线，还是二次曲线或者更高次的曲线。

霍夫(Hough)直线变换是由 PVC Hough 提出的，后来 Duda 和 Hart 认为直线可以用 θ 和 ρ 参数来表示，直线的极坐标表示如图 5-8 所示，其中，ρ 表示远点到直线的垂直距离，θ 表示直线的垂线到 X 轴的夹角。

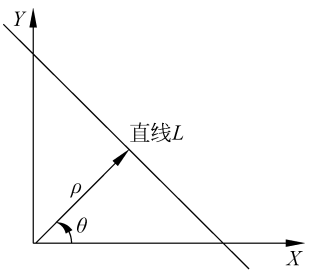


图 5-8　直线的极坐标表示

该直线的参数方程为：

$$\rho = x\cos\theta + y\sin\theta \tag{5-6}$$

这种用极坐标形式表示直线方程的优点是避免了直线点斜式表示时当斜率为 90°时，$\tan\theta$ 不存在的问题。该算法的核心是如果简单地把 ρ 和 θ 看成一个二维平面，那么对于每一个方格单元(ρ,θ)都有对应的 ACC(ρ,θ)值，它事实上是某方向点的数量。如果没有噪音的干扰，则可以用 ACC(ρ,θ)来表示该方向上直线点的多少，进而判断直线是否可靠。这样就可以较容易地确定点的共线情况。

为了解决霍夫变换耗时的问题，提出了概率累加霍夫变换(PPHT)，概率累加霍夫变换的直线拟合方法还是利用了霍夫变换，其特点是选择拟合点时采用了随机抽取的方式，并加

入了计数器，当某一条直线被拟合出的次数达到一定 ACC(ρ,θ)值时，便取出这条直线。完成一条直线的拟合。具体流程如下：

(1) 类似霍夫变换，分割参数空间，为每个区间设计累加器 ACC(ρ,θ)，其初始值为零；

(2) 检测集合 S 是否为空，如为空，则算法结束；否则随机从待处理边缘点集中取一像素点，并将此点从 S 中删除，对该像素点进行霍夫变换，在各个 ρ 值下计算相应的 θ 值，对应的累加器 ACC(ρ,θ)加 1；

(3) 判断更新后的累加器值是否大于阈值 Thr，否则返回步骤(2)；

(4) 由上一步得到的值大于 Thr 的累加器对应的参数确定一条直线，删除集合 S 中位于该直线上的点，此累加器清零；

(5) 返回步骤(2)

该算法最终可以计算出直线在图像中的起始坐标和终止坐标。起始坐标与终止坐标对应的是直线在图像中的起始点与终止点，通过这两个点可以得到直线相对于 X,Y 轴的截距，并通过进一步计算得到直线与水平方位的夹角，即直线的斜率。斜率数据可以对四旋翼飞行器的偏航轴进行控制；起始点和终止点数据可以使用 PID 算法对四旋翼飞行器进行平面位置控制。

软件实现方法如下：

```
void hough(u8 * img_in,u8 * line_end)
{
    u8 xflag = 0,k = 0,gap = 0,line_num = 0;
    u16 img_num = 0,count = 0,idx = 0,point[2],i = 0,j = 0;
    u8 max_val = threshold - 1,max_n = 0,goodline = 0;
    u8 line[2][2] = {0};
    s32 dx = 0,dy = 0,dx0 = 0,dy0 = 0,x = 0,y = 0,x1 = 0,y1 = 0;
    u32 x0,y0;
    float sina = 0,cosb = 0;
    memset(&att[0][0], 0, sizeof(att));
// (1) 将前景数据加入链表
for(i = 1;i < height;i++)//0.7ms
{
img_num = i * widthnum;
if(img_num >= 4800){return;}
for(j = 1;j < width;j++)
        {
        if(img_in[ img_num + j])
        {
            count++;
            / ******** 0 是 x , 1 是 y ****** /
            insertHeadList(&phou,&j,&i);
        }
    }
}
```

```
// (2) 将直线中的每一点极坐标化
for(;count > 0;count -- )
{
    memset(&line[0][0], 0, sizeof(line));                    //0.1ms
    idx = rand() % count;
    pop_del(&phou, idx, point);
    if(img_in[point[1] * widthnum + point[0]] == 0) {continue;}
    att_add(point,&max_val,&max_n);
    if( max_val < threshold ) { continue;}
    sina = - sintab[max_n];
    cosb = costab[max_n];
    if( fabs(sina) > fabs(cosb) )
        {
            xflag = 1;
            if(sina > 0) {dx0 = 1;}
            else {dx0 = - 1;}
            dy0  = round(cosb * (1 << 16)/fabs(sina));
            x0 = point[0];
            y0  = (point[1] << 16) + (1 << (16 - 1));
        }
        else
        {
            xflag = 0;
            if(cosb > 0) {dy0 = 1;}
            else    {dy0 = - 1;}
            dx0  = round(sina * (1 << 16)/fabs(cosb));
            y0 = point[1];
            x0  = (point[0] << 16) + (1 << (16 - 1));
        }  //0.1ms
// (3) 随机找到直线中的一点顺着极角的角度前后遍历整条直线
for(k = 0;k < 2;k++)                                         //0.5ms
{
      gap = 0;
      x1 = x0;
      y1 = y0;
      dx = dx0;
      dy = dy0;
    if(k)
      { dx = - dx; dy = - dy;}
      for(;; x1 += dx, y1 += dy)
      {
    if( xflag )
    {
        x = x1;
        y = y1 >> 16;
    }
    else
```

```
    {
        x = x1 >> 16;
        y = y1;
    }
if( (x <= 1) || (x >= width) || (y <= 1) ||( y >= height ))  {break;}
if(img_in[y * widthnum + x])
{
    gap = 0;
    line[k][0] = x;
    line[k][1] = y;
              }
else if(xflag == 0)
{
if(img_in[y * widthnum + x + 1])
{   gap = 0;
    line[k][0] = x + 1;
    line[k][1] = y;
    #ifdef usehalf
    img_in[y * widthnum + x + 1] = 0;
    #endif
}
    else if(img_in[y * widthnum + x - 1])
    {   gap = 0;
        line[k][0] = x - 1;
        line[k][1] = y;
        #ifdef usehalf
        img_in[y * widthnum + x - 1] = 0;
        #endif
                }
    }
        else if(xflag)                                          //左右点
            {
        if(img_in[(y + 1) * widthnum + x])
        {
        gap = 0;
        line[k][0] = x;
        line[k][1] = y + 1;
        #ifdef usehalf
        img_in[(y + 1) * widthnum + x] = 0;
        #endif
        }
        else if(img_in[(y - 1) * widthnum + x])
        {
        gap = 0;
        line[k][0] = x;
        line[k][1] = y - 1;
        #ifdef usehalf
```

```
        img_in[(y - 1) * widthnum + x] = 0;
        #endif
              }
            }
          if (gap++> lineGap)
        { break;}
        }
}
        #ifdef short_line
// (4) 直线长度不够要求的舍去,横线要 3 < x < 20 竖线要 x < 10
goodline = ( ( (abs(line[0][0] - line[1][0]) >= short_Length)&&(long_Length > abs(line[0][0] -
line[1][0])) )&&( (10 > abs(line[1][1] - line[0][1])) ));
         #else
        goodline = ((abs(line[0][0] - line[1][0]) >= lineLength) ||(abs(line[1][1] -
line[0][1]) >= lineLength)); //0.5ms
         #endif
// (5) 将已成功找到的直线从链表中删除
for(k = 0;k < 2;k++)                                          //最小 0.5ms,最大 4ms
        {
            gap = 0;
            x1 = x0;
            y1 = y0;
            dx = dx0;
            dy = dy0;
            if(k)
            {
                dx = -dx;
                dy = -dy;}
            for(;; x1 += dx, y1 += dy)
                {
                if( xflag )
            {
                x = x1;
                y = y1 >> 16;
        }
        else
        {
            x = x1 >> 16;
            y = y1;
        }
    if( (x <= 1) || (x >= width) || (y <= 1) || (y >= height) ){break;}
    if(img_in[y * widthnum + x])
                {
                if(goodline)                                  //0.1ms
                {att_reduce(x,y);}
                img_in[y * widthnum + x] = 0;
                }
```

```
                #ifndef usehalf
        else if(xflag)                                          //左右点
                {
                    if(img_in[(y+1)*widthnum+x])
                    {
                        if(goodline)
                        {att_reduce(x,y+1);}
                        img_in[(y+1)*widthnum+x]=0;
                        }
                else if(img_in[(y-1)*widthnum+x])
                {
                            if(goodline)
                            {att_reduce(x,y-1);}
                            img_in[(y-1)*widthnum+x]=0;
                  }
               }
        else if(xflag==0)
                  {
                      if(img_in[y*widthnum+x+1])
                      {
                          if(goodline)
                          {att_reduce(x+1,y);}
                          img_in[y*widthnum+x+1]=0;
                      }
                else if(img_in[y*widthnum+x-1])
                {
                if(goodline)
                          {att_reduce(x-1,y);}
                          img_in[y*widthnum+x-1]=0;
                  }
                    }
#endif
        if( (y == line[k][1])&&(x == line[k][0]) )
        {break;}

        }
      }
        if(goodline)
            {
                if(line_num>=linemax){clearList(&phou);return;}
                if(line[0][1]<=line[1][1])
                {
                line_end[line_num*4 ]=line[0][0];
                line_end[line_num*4+1]=line[0][1];
                line_end[line_num*4+2]=line[1][0];
                line_end[line_num*4+3]=line[1][1];
                line_end[29]=++line_num;
```

```
                    }
                    else
                    {
                    line_end[line_num * 4 ] = line[1][0];
                    line_end[line_num * 4 + 1] = line[1][1];
                    line_end[line_num * 4 + 2] = line[0][0];
                    line_end[line_num * 4 + 3] = line[0][1];
                    line_end[29] = ++line_num;
                    }
                        }
    }
    clearList(&phou);

    }
```

5.3　实验结果及分析

本次实验使用 STM32F407 单片机搭载 μC/OS-Ⅲ操作系统，使用 MT9V034 摄像头和 MS5611 气压计的四旋翼飞行器，在 2m×5m 的白纸上粘有黑色直线与黑色直角弯的地面上进行实验。

图 5-9 为摄像头读取的原始灰度图像，图 5-10 为经过自适应中值滤波后的灰度图像。可以明显地看出原始图像数据的背景有很多污点，这些污点可以看作椒盐噪声，而经过自适应中值滤波处理后的图像，这些噪声大部分被去除，前景与背景可以轻易地区分出来。

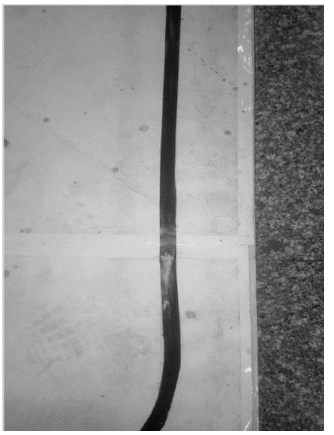


图 5-9　原始图像

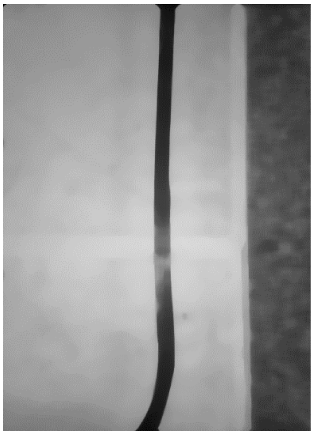


图 5-10　自适应中值滤波处理后的图像

图 5-11 为自适应中值滤波后的图像，数据经过 Sobel 算子在 X 轴与 Y 轴卷积之后的效果，对于灰度图像来说其边缘（数值明显变化处）变为白色前景，而原图中灰度数值变化不明显的区域则变为黑色背景。但此时的图像还是灰度图像，需要使用 OTSU 算法将其二值化

(见图 5-12),二值化的图像已经可以清楚地观测到我们需要检测的直线(共 3 条直线,中间两条为目标直线,右侧一条为无关直线),但此时每条直线的宽度大于 1 个像素,所以要使用并行 Zhang 图像细化算法处理,见图 5-13。

图 5-11 Sobel 算子处理后图像

图 5-12 OTSU 二值化处理后图像

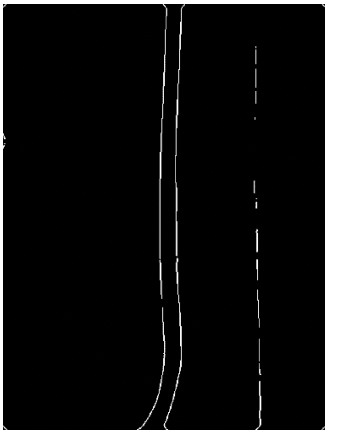


图 5-13 并行 Zhang 图像细化算法处理后图像

经过细化处理后的图像数据会经由概率累加霍夫变换处理以获取直线信息。可以看出,图 5-13 中共有三条直线,左侧两条距离较近可以通过后期软件处理进行合并,右侧的一条直线是无关数据,可通过在概率累加霍夫变换算法中添加直线有效长度去除。最终获取的控制信息,由 PID 算法对四旋翼飞行器的 X 轴、Y 轴及偏航轴进行控制。

图 5-14 是四旋翼飞行器通过算法确定 X 轴方向的位置信息而在 X 轴方向上固定顺着 Y 轴方向飞行。

图 5-15 是四旋翼飞行器通过算法确定 X 轴与 Y 轴两个方向的位置信息从而在一个固定的点飞行。

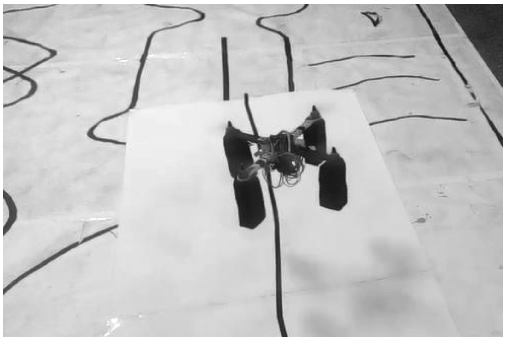


图 5-14 X 轴方向的位置信息

图 5-15 X 轴与 Y 轴两个方向的位置信息

实验证明添加摄像头后的四旋翼飞行器通过图像算法与控制算法可以实现在平面空间对四旋翼飞行器进行精确的空间位置控制，增强了四旋翼飞行器的控制能力与灵活性。但还存在一些不足，由于 STM32F407 的处理速度，每帧图像的大小不能太大，否则会延长图像算法的运行时间，从而延长每次对四旋翼飞行器位置的控制间隔，影响控制效果。