RM 2015 summer camp 第八组视觉方案技术报告

一、 任务分析

夏令营任务要实现自主飞行机器人的设计，在 8*8m 的场地上实现自主飞行和自主抓球和取球。因此可以提取地面二维码建立全场坐标或者巡线来实现 M100 的定位，此外还需要识别取球区、圆盘和九宫格。我们使用 opencv 开发。

二、 方案设计

以下方案全部基于摄像头装在飞机底部，向下观察场地的基础上。

目标识别方案：

1） 取球区的识别

取球区是黑色海绵加白色边框的一个盒子，此外场地其他区域均是彩色图案，因此我们可以直接使用灰度图二值化提取出盒子，之后通过腐蚀膨胀的形态学滤波滤除噪点，进而提取出轮廓后通过对轮廓面积的边长的方差进行阈值处理，提取出矩形。

但是实验中发现灰度图提取不稳定，因此，经过不断的实验和思考，我使用了 HSV 分量中的 HSV 三个分量，

（1） 利用 S 分量滤除高饱和度的杂色干扰，
（2） 对 V 分量阈值处理用以二值图，
（3） 用 H 分量滤除场地上一些深蓝色的区块，进而提取出取球区。
（4） 提取轮廓。
（5） 提取出轮廓后通过对轮廓面积的边长的方差进行阈值处理，提取出矩形。

具体实现方法如下：

```
bool ColorImg::fixStart(cv::Mat imgIn)
{
    bool hascenter = false;

    //[1]加载图像
    if (imgIn.data == NULL)
    {
        return -1;
    }

    //[1]调整大小
    resize(imgIn,imgResize,Size(160,120),0,0,INTER_LINEAR);

    cvtColor( imgResize, imgGray, CV_BGR2GRAY );
    cvtColor( imgResize, imgHSV, CV_BGR2HSV );
```

```
imgBin = imgGray < 100;/*binthreshold*/
cv::split(imgHSV,channels);
    imgH = channels.at(0);
imgS = channels.at(1);
    imgV = channels.at(2);

    imgS=imgS<120;
imgH=~((imgH>90)&(imgH<120));
imgBin = imgBin&imgS&imgH;

//[7]腐蚀
erode_ele = getStructuringElement(0, Size(3, 3));
erode(imgBin,imgSErode,erode_ele);

//找轮廓
    vector<vector<Point> > contours;
    vector<Vec4i> hierarchy;
    double area = 0;
    uint number = 0;
    Mat img2contours;
    imgSErode.copyTo(img2contours);
    findContours( img2contours, contours, hierarchy,RETR_CCOMP,CHAIN_APPROX_SIMPLE);
    for(uint i = 0;i<contours.size();i ++)
    {
        double nowarea = contourArea(contours[i]);
        if((nowarea > area)&&(nowarea > 400))
        {
            //记录面积大于500的最大的轮廓和序号
            area = nowarea;
            number= i;
        }
    }

    if(area != 0)
    {
        //轮廓
        Scalar color(0,0,255);
        drawContours(imgResize,contours,number,color,1,8,vector<Vec4i>(),0,Point());

        //矩形逼近
        RotatedRect box = minAreaRect(Mat(contours[number]));
        Point2f vertex[4];
        box.points(vertex);
```

```
            double length[5] = {0};
            length[0] = (vertex[0].x - vertex[1].x)*(vertex[0].x - vertex[1].x) +
                    (vertex[0].y - vertex[1].y)*(vertex[0].y - vertex[1].y);
            length[1] = (vertex[1].x - vertex[2].x)*(vertex[1].x - vertex[2].x) +
                    (vertex[1].y - vertex[2].y)*(vertex[1].y - vertex[2].y);
            length[2] = (vertex[2].x - vertex[3].x)*(vertex[2].x - vertex[3].x) +
                    (vertex[2].y - vertex[3].y)*(vertex[2].y - vertex[3].y);
            length[3] = (vertex[0].x - vertex[3].x)*(vertex[0].x - vertex[3].x) +
                    (vertex[0].y - vertex[3].y)*(vertex[0].y - vertex[3].y);

            //求最大边长
            for(int i = 0;i < 4;i ++)
                if(length[i] > length[5])
                    length[5] = length[i];
            //归一化
            for(int i = 0;i < 4;i ++)
                    length[i] = length[i]/length[5];
            //均值
            double ave = (length[0] + length[1] + length[2] + length[3])/4;
            double ex = 0;
            for(int j = 0;j < 4; j++)
            {
                ex += (length[j]-ave)*(length[j]-ave);
            }
            ex = ex/4;
            //方差小于1
            if(ex < 0.05)
            {
                double xita=0;
                for(int j = 0;j < 4;j ++)
                {
                    xita = atan( (vertex[(j+2)%4].x - vertex[j].x)/(vertex[(j+2)%4].y -
vertex[j].y) )/PI*180;
                    if(abs(xita)<40.0)
                    {
                        line(imgResize,vertex[j],vertex[(j+2)%4],CV_RGB(255,0,0),1,CV_AA);
                        square_angle = xita;
                    }
                        line(imgResize,vertex[j],vertex[(j+1)%4],CV_RGB(0,255,0),1,CV_AA);
                }

                square_center = box.center;
               // square_angle = box.angle;
```

```
                square_x = square_center.y-60;
                square_y = 80-square_center.x;
                square_xy_angle = square_angle;

//std::cout<<"square_angle:\t"<<square_angle<<std::endl;

                circle(imgResize, square_center, 1,Scalar(0,255,255),1);

                hascenter = true;
            }
        }

    cv::imshow("findSquare",imgResize);
    m_deal_writer<<imgResize;

    return hascenter;
}
```

2） 圆盘的识别

（1） 圆盘中放有黄色沙袋，可以直接使用 H 分量阈值处理

（2） 提取轮廓

（3） 滤除轮廓面积小的区域

（4） 滤除面积周长比小的周长小于 3.14 的轮廓

（5） 对剩下的轮廓椭圆拟合

具体实现方法：

```
bool ColorImg::findCircle(Mat img)
{
    bool hascenter = false;

    //调整大小
    resize(img,imgResize,cv::Size(160,120),0,0,cv::INTER_LINEAR);

    // RGB->HSV
    imgHSV  = cv::Mat::zeros(imgResize.rows,imgResize.cols,imgResize.type());
    cvtColor( imgResize, imgHSV, CV_BGR2HSV );
    //分离
    cv::split(imgHSV,channels);
//        imgH = channels.at(0);
    imgS = channels.at(1);
//        imgV = channels.at(2);

    imgS = imgS<100;

    //腐蚀
    erode_ele = getStructuringElement(0, Size(4, 4));
    erode(imgS, imgSErode, erode_ele);

    //找轮廓
    std::vector<std::vector<cv::Point> > contours;
    std::vector<cv::Vec4i> hierarchy;
    cv::Mat img2contours;
    imgSErode.copyTo(img2contours);
    findContours( img2contours, contours,
hierarchy,cv::RETR_CCOMP,cv::CHAIN_APPROX_SIMPLE );
    cv::Mat drawing = cv::Mat::zeros(imgResize.size(),CV_8UC3);

    unsigned int x_sum=0, y_sum=0;
    unsigned int cnt = 0;
    for(uint i = 0;i<contours.size();i ++)
    {
        double nowarea = contourArea(contours[i]);
        if(nowarea > 250)
        {
            cv::Scalar color(0,0,255);

drawContours(drawing,contours,i,color,1,8,std::vector<cv::Vec4i>(),0,cv::Point());

            //面积与边长的值大于75
            int length = arcLength(contours[i],true);
            if(nowarea/length > 10)
            {
                //椭圆拟合
                cv::RotatedRect ellipsein;
                ellipsein = fitEllipse((cv::Mat)contours[i]);

                    ellipse(imgResize,ellipsein,cv::Scalar(0,0,255),1);
                    x_sum += ellipsein.center.x;
                    y_sum += ellipsein.center.y;
                    circle(imgResize,ellipsein.center,1,cv::Scalar(0,0,0),2);
                cnt ++;
                hascenter = true;
            }
        }
    }

    if(hascenter)
    {
        center_point.x = x_sum/cnt;
        center_point.y = y_sum/cnt;
        circle_x = center_point.y-60;
        circle_y = 80-center_point.x;
    }

    imshow("findCircle",imgResize);
    m_deal_writer<<imgResize;
    return hascenter;
}
```

3） 九宫格的识别

a. 普通曝光

这种情况下曝光比较高，看到的亮灯九宫格比较亮

（1） 利用 V 分量提取亮度高的区域

（2） 在亮度高的区域内用 H 分量提取颜色

（3） 求轮廓，矩形拟合，求中心

问题：

A. 相同颜色的区域可能会连在一起，从而无法区分两个格子

B. 摄像头曝光度和白平衡的影响

此方法与圆盘类似，并且我们最后也没有采用此方案，就不贴代码了。

b. 曝光降低

（1） 曝光降低后图像中只有灯管，用 a 普通曝光中所述的方式可以分别提取出黄色和红色的灯管（不能腐蚀膨胀，否则灯管也会被滤掉）

（2） 用霍夫变换提取灯管的直线

（3） 使用比较粗的线将霍夫变换提取的直线画在另一个图像中（使用比较粗的线的目的是将不连续的灯管连续起来）

（4） 腐蚀膨胀，滤除没用的图像

（5） 提取轮廓，建立轮廓的关系，提取出内部轮廓

（6） 对内部轮廓矩形拟合，求出中心

```
bool ColorImg::findSudoku(Mat imgIn)
{
            if (imgIn.data == NULL)
            {
                cout<<"no image in!"<<endl;
                exit(0);
            }

            distance_red = 0xffffffffffffffff;
            distance_yellow = 0xffffffffffffffff;

            int cnt = 0;
            int avearea = 0;
            //[1]调整大小
            resize(imgIn,imgResize,Size(320,240),0,0,INTER_LINEAR);

            //[2]RGB->HSV
            imgHSV  = Mat::zeros(imgResize.rows,imgResize.cols,imgResize.type());
            cvtColor( imgResize, imgHSV, CV_BGR2HSV );
            //[3]分离
            cv::split(imgHSV,channels);
            imgH = channels.at(0);
            imgS = channels.at(1);
            imgV = channels.at(2);

            //[4]对V分量阈值处理用以给H分量Mask
            cv::threshold(imgV,imgVthreshold,50,255,cv::THRESH_BINARY);

            //[5]V分量阈值后给H分量mask
            imgHMasked = Mat::zeros(imgResize.rows,imgResize.cols,imgResize.type());
            imgH.copyTo(imgHMasked,imgVthreshold);

            imgYellow = ((imgHMasked > 10)&(imgHMasked < 165));
            imgRed = (((imgHMasked > 165)&(imgHMasked < 190))
                     |((imgHMasked > 0)&(imgHMasked < (190 - 180))));
            erode_ele = getStructuringElement(1, Size(3, 3));
            dilate(imgRed,imgRed,erode_ele);
            dilate(imgYellow,imgYellow,erode_ele);

            vector<Vec4i> lines;//定义一个矢量结构lines用于存放得到的线段矢量集合
            HoughLinesP(imgRed, lines, 1, CV_PI/90, 50, 10, 0 );
            Mat imgLines = Mat::zeros(imgResize.rows,imgResize.cols,imgResize.type());
            for( size_t i = 0; i < lines.size(); i++ )
            {
                Vec4i l = lines[i];
                //依次在图中绘制出每条线段
                line( imgLines, Point(l[0], l[1]), Point(l[2], l[3]), Scalar(50,200,0), 5,
CV_AA);
            }
        //          imshow("imgLineR",imgLines);

            //找轮廓
            vector<vector<Point> > contoursr;
            vector<Vec4i> hierarchyr;
            Mat imgr2c;
            imgLines.copyTo(imgr2c);
            cvtColor(imgr2c,imgr2c,CV_BGR2GRAY);
            imgr2c = imgr2c > 10;
        erode_ele = getStructuringElement(1, Size(20, 20));
        dilate(imgr2c,imgr2c,erode_ele);
            imshow("img2r",imgr2c);
        findContours( imgr2c, contoursr, hierarchyr,CV_RETR_CCOMP,CHAIN_APPROX_SIMPLE );
        for(uint i = 0;i<contoursr.size();i ++)
        {
            //内轮廓
            if((hierarchyr[i][3] != -1))
            {
                double nowarea = contourArea(contoursr[i]);
                //寻找面积合理的区域
                if(nowarea > 100)
                {
                    //轮廓
    //                      Scalar color(0,0,255);
    //
drawContours(imgResize,contoursr,i,color,3,8,vector<Vec4i>(),0,Point());

                    //矩形逼近
                    RotatedRect box = minAreaRect(Mat(contoursr[i]));
                    Point2f vertex[4];
                    box.points(vertex);

                    for(int j = 0;j < 4;j ++)
                    {
                        line(imgResize,vertex[j],vertex[(j+1)%4],CV_RGB(255,0,0),2,CV_AA);
                    }
                    //求中心
                    Point2f center;
                    center.x = (vertex[0].x + vertex[1].x + vertex[2].x + vertex[3].x)/4;
                    center.y = (vertex[0].y + vertex[1].y + vertex[2].y + vertex[3].y)/4;
                    //在原图上绘制中心区域
                    circle(imgResize,center,2,Scalar(0,0,255),2);
                    red.push_back(center);

                    double length[5] = {0};
                    length[0] = (vertex[0].x - vertex[1].x)*(vertex[0].x - vertex[1].x) +
                            (vertex[0].y - vertex[1].y)*(vertex[0].y - vertex[1].y);
                    length[1] = (vertex[1].x - vertex[2].x)*(vertex[1].x - vertex[2].x) +
                            (vertex[1].y - vertex[2].y)*(vertex[1].y - vertex[2].y);
                    length[2] = (vertex[2].x - vertex[3].x)*(vertex[2].x - vertex[3].x) +
                            (vertex[2].y - vertex[3].y)*(vertex[2].y - vertex[3].y);
                    length[3] = (vertex[0].x - vertex[3].x)*(vertex[0].x - vertex[3].x) +
                            (vertex[0].y - vertex[3].y)*(vertex[0].y - vertex[3].y);
                        for(int k = 0;k < 4;k++)
                        {
                            length[5]+=length[i];
                        }
                        length[5]/=4;
                        avearea = (avearea*cnt + length[5]*length[5])/(cnt+1);
                        cnt++;
                    }
                }
            }
//          imshow("imgResize",imgResize);



            HoughLinesP(imgYellow, lines, 1, CV_PI/90, 50, 10, 0 );
            imgLines = Mat::zeros(imgResize.rows,imgResize.cols,imgResize.type());
            for( size_t i = 0; i < lines.size(); i++ )
            {
                Vec4i l = lines[i];
                //依次在图中绘制出每条线段
                line( imgLines, Point(l[0], l[1]), Point(l[2], l[3]), Scalar(50,200,0), 5,
CV_AA);
            }
//          imshow("imgLineY",imgLines);
            //找轮廓
            vector<vector<Point> > contoursy;
            vector<Vec4i> hierarchyy;
            Mat imgy2c;
            imgLines.copyTo(imgy2c);
            cvtColor(imgy2c,imgy2c,CV_BGR2GRAY);
            imgy2c = imgy2c > 10;
            erode_ele = getStructuringElement(1, Size(20, 20));
            dilate(imgy2c,imgy2c,erode_ele);
                imshow("img2y",imgy2c);
            findContours( imgy2c, contoursy, hierarchyy,CV_RETR_CCOMP,CHAIN_APPROX_SIMPLE );
            for(uint i = 0;i<contoursy.size();i ++)
            {
                double nowarea = contourArea(contoursy[i]);
                if((hierarchyy[i][3] != -1))
                {
                    //寻找面积合理的区域
                    if(nowarea > 100)
                    {
                        //轮廓
```

```
        //                    Scalar color(0,255,255);
        //
drawContours(imgResize,contoursy,i,color,3,8,vector<Vec4i>(),1,Point());

                        //矩形逼近
                        RotatedRect box = minAreaRect(Mat(contoursy[i]));
                        Point2f vertex[4];
                        box.points(vertex);

                        for(int j = 0;j < 4;j ++)
                        {

line(imgResize,vertex[j],vertex[(j+1)%4],CV_RGB(255,255,0),2,CV_AA);
                        }
                        //求中心
                        Point2f center;
                        center.x = (vertex[0].x + vertex[1].x + vertex[2].x + vertex[3].x)/4;
                        center.y = (vertex[0].y + vertex[1].y + vertex[2].y + vertex[3].y)/4;
                        //在原图上绘制中心区域
                        circle(imgResize,center,2,Scalar(0,255,255),2);
                        yellow.push_back(center);



                        double length[5] = {0};
                        length[0] = (vertex[0].x - vertex[1].x)*(vertex[0].x - vertex[1].x) +
                                (vertex[0].y - vertex[1].y)*(vertex[0].y - vertex[1].y);
                        length[1] = (vertex[1].x - vertex[2].x)*(vertex[1].x - vertex[2].x) +
                                (vertex[1].y - vertex[2].y)*(vertex[1].y - vertex[2].y);
                        length[2] = (vertex[2].x - vertex[3].x)*(vertex[2].x - vertex[3].x) +
                                (vertex[2].y - vertex[3].y)*(vertex[2].y - vertex[3].y);
                        length[3] = (vertex[0].x - vertex[3].x)*(vertex[0].x - vertex[3].x) +
                                (vertex[0].y - vertex[3].y)*(vertex[0].y - vertex[3].y);
                        for(int k = 0;k < 4;k++)
                        {
                            length[5]+=length[i];
                        }
                        length[5]/=4;
                        avearea = (avearea*cnt + length[5]*length[5])/(cnt+1);
                        cnt++;
                    }
                }
            }
            if(red.size()>0)
            {
                hassudokured = true;
                for(unsigned int i = 0;i < red.size();i ++)
                {
                    double tmp;
                    tmp =     (old_point_red.x - red[i].x)*(old_point_red.x - red[i].x)
                            + (old_point_red.y - red[i].y)*(old_point_red.y - red[i].y);
                    if(tmp < distance_red)
                    {
                        distance_red = tmp;
                        old_point_red.x = red[i].y - 120;
                        old_point_red.y = 160 - red[i].x;
                    }

                }
            }
            else
                hassudokured = false;



            if(yellow.size()>0)
            {
                hassudokuyellow = true;
                for(unsigned int i = 0;i < yellow.size();i ++)
                {
                    double tmp;
                    tmp =     (old_point_yellow.x - yellow[i].x)*(old_point_yellow.x -
yellow[i].x)
                            + (old_point_yellow.y - yellow[i].y)*(old_point_yellow.y -
yellow[i].y);
                    if(tmp < distance_yellow)
                    {

                        distance_yellow = tmp;
//                        old_point_yellow = yellow[i];
                        old_point_yellow.x = yellow[i].y - 120;
                        old_point_yellow.y = 160 - yellow[i].x;
//                        cout<<"\tnew
point"<<old_point_yellow.x<<","<<old_point_yellow.y<<endl;
                    }
//                    cout<<"yellow:\t"<<yellow[i].x<<","<<yellow[i].y;
//                    cout<<"\tnow distance:\t"<<tmp<<endl;
                }
            }
            else
            {
                hassudokuyellow = false;
            }

            red.clear();
            yellow.clear();

            circle(imgResize,old_point_red,5,Scalar(255,0,255),2);
            circle(imgResize,old_point_yellow,5,Scalar(0,100,255),2);

            avesudokuarea = avearea;

            imshow("findSudoku",imgResize);
            cv::Mat img_temp;
            cv::resize(imgResize,img_temp,cv::Size(160,120) );
            m_deal_writer<<img_temp;

            int ret = 0;
            if(yellow.size()+red.size() == 0)
                ret = -1;

            return ret;
}
```

三、　结论与不足

上述算法在场地固定，光线比较固定的场景下可以稳定可靠的识别出所有目标；但是，如果摄像头曝光和白平衡已经场地光线变化比较大的时候就需要重新调整预处理的参数，这对 V 分量影响比较大，因此比赛中我们视觉方面是存在问题的。可能还需要对图像进行均衡等处理，从而提高适应性。