嵌入式系统高级实验结题报告

——智能“魔杖”：用运动轨迹控制小车运动

组号：I组

组员：李丰克、陈思攀、赵彬序

一、选题思路

1.1 选题背景

在智能家居日益发展的今天，怎样用越少的操作实现越复杂的功能是实现便利性的核心，这样的实现思路有很多种，典型的有基于手机操控的、基于语音的、基于手势的等等，而这些大多依托于语音和视觉。而我们组借魔杖这一概念，欲实现基于运动轨迹的控制，且通过控制一辆小车来进行直观的表现，毕竟这个选题的核心在于对运动轨迹的识别与提取并且转化为信号再传输，而接收端可以根据不同的应用场景来改变，这里选用小车是为了直观展示。魔杖这个概念在实用性上相较于其他智能家居思路也有其优势，即仅靠轨迹一个变量可以同时控制多台设备，且对比语音控制等更简单，模型也更简洁。

1.2 实现思路

开发板均选择esp32 devkit v1，传感器选择MPU6050，小车驱动选择TB6612，这些便是全部主要的硬件设施。

魔杖部分：利用MPU6050可以监测加速度和角速度的功能，获得世界坐标系下的去除重力加速度分量后的三轴加速度，要求得到的结果准确、低噪、响应速度快，然后将加速度二次积分得到三轴位移，便可以表示出世界坐标系下的三轴轨迹，再对这些轨迹按照机器学习的方法进行训练，获得能够分类轨迹的模型，再把模型部署到开发板上去，这样便能够实现对分类结果的实时运算，再将这个结果通过蓝牙传输给小车端的板子，对其控制。

小车部分：用TB6612写出基本的控制运动的代码，接收从魔杖端发送过来的指令并对其做出响应。

1.3 预期成果

预计实现四种轨迹（上下左右）的控制指令，除去小车，也可以尝试将接收信号端改为一些家具，比如将分类结果转为对应的红外线信号来控制空调，实现简单的智能家具功能，当然这些拓展是建立在完成对小车的控制上的。

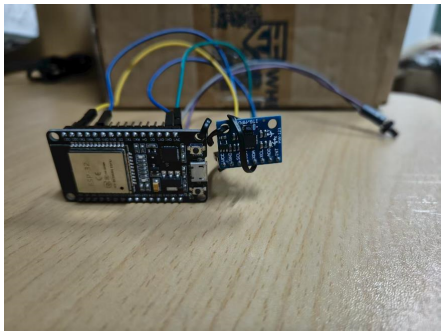
二、 魔杖部分

2.1 硬件连接

魔杖部分共用到的硬件有：esp32 devkit v1开发板，MPU6050传感器，按钮，5V锂电池，microusb公对母带开关导线，若干杜邦线。

其中MPU6050为I2C通信，有串行数据线SDA，串行时钟线SCL，均为开漏输出(OD)，而esp32仅有两个管脚支持I2C通信，依次为 D21——SDA、D22——SCL，剩下两个端口分别接3.3V和GND，来给传感器供电。

双端按钮默认常断，一端接地，一端接D13，因为D13有内置上拉电阻，否则平时浮空输入时改管脚电平不定。而电池通过microusb公对母导线与esp32连接来给开发板供电，这条导线上的按钮即为整个魔杖的上电开关。



2.2 三轴加速度的获取

我们想要把传感器的运动轨迹解算出来，即将传感器的运动轨迹用位移的数组表示出了，首先要从加速度的获取入手，获得世界坐标系下的三轴加速度再经过两次积分后便可以得到位移数组。

2.2.1 DMP库直接获取

因为我们所用的是MPU6050传感器，因此我们自然会想到用其自带的DMP姿态解算库，其内置的若干函数可以直接计算出世界坐标系下去除重力分量的三轴加速度。

核心代码如下：

```
#include "I2Cdev.h"
#include "MPU6050_6Axis_MotionApps20.h"
MPU6050 mpu;
uint8_t fifoBuffer[64];//内部缓存
Quaternion q;//四元数
VectorFloat gravity;//重力分量
VectorInt16 acc_raw;//三轴原始加速度
VectorInt16 lineacc;//去除重力的线性加速度
VectorInt16 worldacc;//世界坐标系下的
float ypr[3];//三轴角
float acc[3];//最终加速度
void setup()
{
  Wire.begin();
  Wire.setClock(400000);//I2C通信速率
  Serial.begin(115200);//波特率
  mpu.initialize();
  mpu.dmpInitialize();
  mpu.CalibrateAccel(6);
  mpu.CalibrateGyro(6);
  mpu.PrintActiveOffsets();
  mpu.setDMPEnabled(true);
}
void loop()
{
  if (mpu.dmpGetCurrentFIFOPacket(fifoBuffer))//读缓存
  {
    mpu.dmpGetQuaternion(&q,fifoBuffer);//读四元数
    mpu.dmpGetGravity(&gravity,&q);//读重力分量
    mpu.dmpGetAccel(&acc_raw,fifoBuffer);//读原始加速度
    acc_raw.x=acc_raw.x/2;
    acc_raw.y=acc_raw.y/2;
    acc_raw.z=acc_raw.z/2;
    mpu.dmpGetLinearAccel(&lineacc,&acc_raw,&gravity);//去除重力
    mpu.dmpGetYawPitchRoll(ypr,&q,&gravity);
    mpu.dmpConvertToWorldFrame(&worldacc,&lineacc,&q);//转世界坐标系
    acc[0]=worldacc.x/8192.0f;//归一化
    acc[1]=worldacc.y/8192.0f;
    acc[2]=worldacc.z/8192.0f;
  }
}
```

中间有一段是将原始加速度除以二，是因为在其原始函数中，去除重力分量的定义是这样的：

```
uint8_t MPU6050_6Axis_MotionApps20::dmpGetLinearAccel(VectorInt16 *v, VectorInt16
*vRaw, VectorFloat *gravity) {
    // get rid of the gravity component (+1g = +8192 in standard DMP FIFO packet,
sensitivity is 2g)
    v -> x = vRaw -> x - gravity -> x*8192;
    v -> y = vRaw -> y - gravity -> y*8192;
    v -> z = vRaw -> z - gravity -> z*8192;
    return 0;
}
```

所有加速度得到的初始值单位是LSB，也就是数模转换的灵敏度，这个取决于量程范围：对±2g，灵敏度为16384 LSB/g；对±4g，灵敏度为8192 LSB/g。但是上述函数的原理，是将重力分量乘8192再减的，但是默认量程为2g，这就导致了灵敏度换算出现差异，于是原始灵敏度要先除以二再进行运算。

但是我们最终得到的数据存在相当大的问题：z轴有非常严重的零点漂移，而且随着传感器姿态不同，漂移量也随之变化，最大可达0.3g，这就导致了我们不能仅用添加一个偏置项的方法来消除零漂。而z轴又是非常重要的，关系到上下轨迹的判断，假如静止时偏移量为向上的0.3g，而实际运动的加速度向下且小于0.3g，此时记录下来的加速度变化数据方向全部向上，会将原本向下的轨迹误判为向上。

简单判断原因可能与其内部运算逻辑有关系，重力分量是四元数运算得到的，三轴加速度是从缓存中读出来的，这差异可能就导致了结果有出入。因为DMP库是内部封装的，所以我们只能换其他算法。

零点漂移量较小时的三轴加速度

2.2.2 加速度角速度-欧拉角-旋转矩阵算法

于是我们脱离DMP库，按照一定的顺序逐步计算出来世界坐标系下的三轴加速度。

2.2.2.1 Adafruit库

Adafruit库可以直接从传感器中读出三轴加速度和三轴角速度数据，而且值准确、低噪、响应快。为方便我们定义了一个头文件。

mpu6050.h

```
#include <Adafruit_MPU6050.h>
#include <Adafruit_Sensor.h>
#include <Wire.h>
Adafruit_MPU6050 mpu;
struct MPU6050_DATA
{
  float Acc_X = 0.0;//三轴加速度m/s^2
```

```
  float Acc_Y = 0.0;
  float Acc_Z = 0.0;
  float Angle_Velocity_R = 0.0; //三轴角速度rad/s
  float Angle_Velocity_P = 0.0;
  float Angle_Velocity_Y = 0.0;
}mpu6050_data;
void Init_mpu6050(){
        ...//主要包含一些量程等初始量设置
}
void ReadMPU6050()
{
  sensors_event_t a, g, temp;
  mpu.getEvent(&a, &g, &temp);
  mpu6050_data.Acc_X = a.acceleration.x;
  mpu6050_data.Acc_Y = a.acceleration.y;
  mpu6050_data.Acc_Z = a.acceleration.z;
  mpu6050_data.Angle_Velocity_R = g.gyro.x;
  mpu6050_data.Angle_Velocity_P = g.gyro.y;
  mpu6050_data.Angle_Velocity_Y = g.gyro.z;
}
```

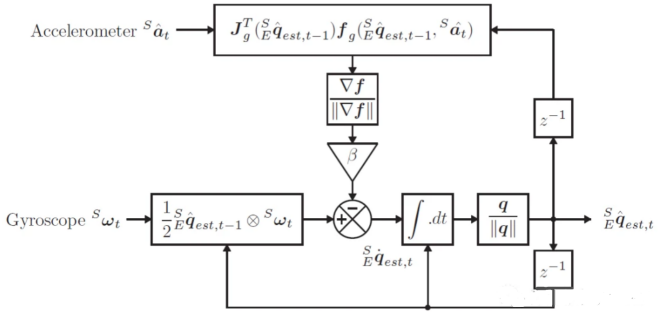
mozhang.ino（其中mpu6050_data是全局变量）

```
#include "mpu6050.h"
...
void setup() {
 Init_mpu6050();
 ...
}
void loop() {
ReadMPU6050();
...
}
```

2.2.2.2 Madgwick算法

通过Adafruit库我们获得了传感器坐标系下的三轴加速度和角速度，为转到世界坐标系我们需要由欧拉角得到的旋转矩阵，于是问题就变成了如何用获得的这六个值来解出来三个欧拉角。我们采用Madgwick算法。

Madgwick算法是一种基于四元数的六轴融合的姿态滤波算法，大致原理如下图：



但是这个算法太难了，设计的变量于公式推导都比较多，因此我们直接搬运了https://github.com/kriswiner/MPU6050这个项目的算法函数。

以下仅为部分代码，完整见附件mozhang.ion

```
void MadgwickQuaternionUpdate(float ax, float ay, float az, float gyrox, float
gyroy, float gyroz)
{
    float q1 = q[0], q2 = q[1], q3 = q[2], q4 = q[3];         // short name local
variable for readability
    float norm;                                               // vector norm
    float f1, f2, f3;                                         // objetive
funcyion elements
    float J_11or24, J_12or23, J_13or22, J_14or21, J_32, J_33; // objective
function Jacobian elements
    float qDot1, qDot2, qDot3, qDot4;
    float hatDot1, hatDot2, hatDot3, hatDot4;
    float gerrx, gerry, gerrz, gbiasx, gbiasy, gbiasz;        // gyro bias error

    // Auxiliary variables to avoid repeated arithmetic
    float _halfq1 = 0.5f * q1;
    float _halfq2 = 0.5f * q2;
    float _halfq3 = 0.5f * q3;
    float _halfq4 = 0.5f * q4;
    float _2q1 = 2.0f * q1;
    float _2q2 = 2.0f * q2;
    float _2q3 = 2.0f * q3;
    float _2q4 = 2.0f * q4;
    float _2q1q3 = 2.0f * q1 * q3;
    float _2q3q4 = 2.0f * q3 * q4;

    // Normalise accelerometer measurement
    norm = sqrt(ax * ax + ay * ay + az * az);
    if (norm == 0.0f) return; // handle NaN
    norm = 1.0f/norm;
    ax *= norm;
    ay *= norm;
    az *= norm;
    ...
}
```

```
yaw=atan2(2.0f * (q[1] * q[2] + q[0] * q[3]), q[0] * q[0] + q[1] * q[1] - q[2] *
q[2] - q[3] * q[3]);
pitch=-asin(2.0f * (q[1] * q[3] - q[0] * q[2]));
roll=atan2(2.0f * (q[0] * q[1] + q[2] * q[3]), q[0] * q[0] - q[1] * q[1] - q[2] *
q[2] + q[3] * q[3]);
```

最终通过这个算法我们得到准确且响应快的欧拉角，但是仍然存在yaw角随时间慢慢偏移的现象，这个是不可避免的，这个需要加装磁力计来校正，而MPU6050没有磁力计。

2.2.2.3 旋转矩阵

用获得的欧拉角求出旋转矩阵，便可以从传感器坐标系转为世界坐标系，再减去重力分量便可以得到我们需要的世界坐标系下去除重力分量的三轴加速度。

$$WorldAcc_{(3\times1)} = R \cdot Acc_{(3\times1)}$$

旋转矩阵定义为：

$$R_x(\alpha)=\begin{bmatrix}1 & 0 & 0\\ 0 & \cos\alpha & -\sin\alpha\\ 0 & \sin\alpha & \cos\alpha\end{bmatrix}$$

$$R_y(\beta)=\begin{bmatrix}\cos\beta & 0 & \sin\beta\\ 0 & 1 & 0\\ -\sin\beta & 0 & \cos\beta\end{bmatrix}$$

$$R_z(\gamma)=\begin{bmatrix}\cos\gamma & -\sin\gamma & 0\\ \sin\gamma & \cos\gamma & 0\\ 0 & 0 & 1\end{bmatrix}$$

$$R=R_z(\gamma)R_y(\beta)R_x(\alpha)$$

$$R=\begin{bmatrix}\cos\gamma\cos\beta & \cos\gamma\sin\beta\sin\alpha-\sin\gamma\cos\alpha & \cos\gamma\sin\beta\cos\alpha+\sin\gamma\sin\alpha\\ \sin\gamma\cos\beta & \sin\gamma\sin\beta\sin\alpha+\cos\gamma\cos\alpha & \sin\gamma\sin\beta\cos\alpha-\cos\gamma\sin\alpha\\ -\sin\beta & \cos\beta\sin\alpha & \cos\beta\cos\alpha\end{bmatrix}$$

```
float R[3][3];
R[0][0]=cos(pitch)*cos(yaw);
R[0][1]=cos(yaw)*sin(roll)*sin(pitch)-cos(roll)*sin(yaw);
R[0][2]=sin(roll)*sin(yaw)+cos(roll)*cos(yaw)*sin(pitch);

R[1][0]=cos(pitch)*sin(yaw);
R[1][1]=cos(roll)*cos(yaw)+sin(roll)*sin(pitch)*sin(yaw);
R[1][2]=cos(roll)*sin(pitch)*sin(yaw)-cos(yaw)*sin(roll);

R[2][0]=-sin(pitch);
R[2][1]=cos(pitch)*sin(roll);
R[2][2]=cos(roll)*cos(pitch);

worldAccX=R[0][0]*mpu6050_data.Acc_X+R[0][1]*mpu6050_data.Acc_Y+R[0]
[2]*mpu6050_data.Acc_Z;
worldAccY=R[1][0]*mpu6050_data.Acc_X+R[1][1]*mpu6050_data.Acc_Y+R[1]
[2]*mpu6050_data.Acc_Z;
worldAccZ=R[2][0]*mpu6050_data.Acc_X+R[2][1]*mpu6050_data.Acc_Y+R[2]
[2]*mpu6050_data.Acc_Z;
worldAccZ-=9.8065;
```

以下为部分动作时的加速度响应情况：

突然向上

```
0.06,  0.04,  0.10
0.01,  0.01,  0.08
0.03,  -0.02,  -0.13
0.20,  0.26,  0.13
-1.30,  1.46,  9.93
-1.99,  -3.03,  11.95
2.99,  -0.38,  -6.46
-0.50,  1.28,  -11.45
0.60,  -0.79,  -5.50
1.25,  -2.06,  0.51
```

突然向下

```
-0.05,  -0.24,  0.06
1.13,  -0.95,  -0.30
1.61,  2.68,  -11.10
-0.70,  8.24,  -13.79
2.35,  -10.26,  11.88
-2.54,  -0.93,  9.98
-1.87,  2.33,  1.01
-0.06,  2.35,  -1.28
-0.39,  -1.27,  1.49
```

突然向左

```
0.02,  0.01,  0.21
0.04,  0.01,  -0.03
-0.03,  -0.08,  0.03
-4.27,  -2.45,  1.33
-21.16,  -8.98,  -1.21
8.99,  -0.21,  -0.70
13.69,  10.59,  1.13
3.54,  5.20,  -0.78
0.73,  1.25,  -0.80
```

突然向右

```
-0.00,  -0.00,  -0.14
-0.46,  -0.52,  -0.10
-0.72,  -0.41,  -0.44
10.09,  16.70,  0.38
12.08,  10.99,  -0.33
-13.13,  -16.07,  -3.11
-2.80,  -6.69,  2.06
2.22,  1.23,  1.28
-0.64,  0.82,  0.45
-0.69,  -0.45,  -0.62
0.90,  0.68,  0.25
```

2.3 世界坐标系的定义

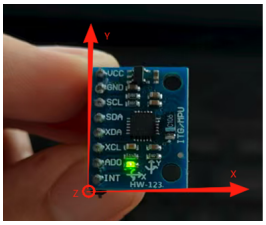
前面的一切工作都是为了获得世界坐标系下的加速度，但是世界坐标系到底是怎么定义的，直接关系到了我们对姿态的解算。

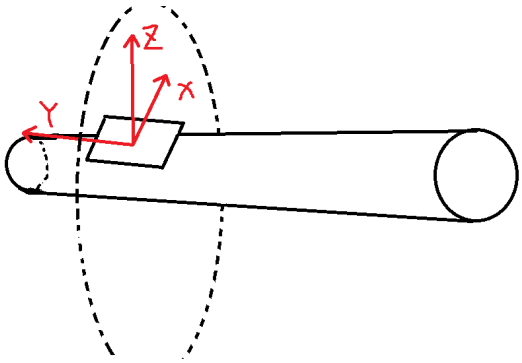
传感器坐标系的定义与传感器姿态有关，xy轴是相对于传感器确定的，z轴也因此确定下来。而定义的世界坐标系与上电时刻的传感器坐标系有关但是并不完全重合。这在之前可以看出：去除重力加速度分量时仅仅在z轴减去g便可以得到非常稳定的结果，这给我们启发，定义的**世界坐标系的z轴是始终与重力方向相反竖直向上的**，z轴便确定了，而其**xy轴则是上电时刻传感器坐标系的xy轴在垂直于z轴的水平面上的投影。**

这样不同的上电姿态只会影响xy轴的方向，但是这也是亟待解决的，因为关系到左右轨迹的确定，如果xy轴不确定，那么左右移动时其在xy轴上的分量也是不确定的。

再者，我们所需要的轨迹，本质上都是二维轨迹，换句话说就是我们只需要用到三维轨迹在某个固定的二维平面上的投影即可。于是我们想到可以**令传感器的y轴与魔杖共线，这样xOz平面始终垂直于魔杖**，这样就实现了正常手持魔杖时无论传感器以什么姿态上电，其左右运动对应的都是x的变化。而我们用来分类的轨迹用xOz平面上的投影即可，因为xz轴都是相对于魔杖固定的。

只要挥舞时人的朝向与上电时保存一致，后续训练的模型就是可用的。





传感器坐标系　　传感器安装方法

按照这种方式安装传感器最终得到的效果如下：

左　　右　　上　　下

首次按下进入积分x:0.00,y:-0.00,z:-0.00
x:0.00,y:-0.00,z:-0.00
x:0.00,y:-0.00,z:0.00
x:0.00,y:-0.00,z:0.00
x:0.00,y:-0.00,z:0.00
x:-0.00,y:-0.00,z:0.00
x:-0.02,y:-0.00,z:-0.00
x:-0.07,y:-0.00,z:0.01
x:-0.14,y:-0.00,z:0.02
x:-0.19,y:-0.01,z:0.04
x:-0.21,y:-0.03,z:0.05
x:-0.22,y:-0.03,z:0.06

首次按下进入积分x:-0.00,y:0.00,z:-0.00
x:-0.00,y:0.00,z:-0.00
x:0.00,y:0.00,z:-0.00
x:0.00,y:-0.00,z:-0.00
x:0.02,y:0.00,z:0.00
x:0.07,y:0.02,z:0.01
x:0.14,y:0.03,z:0.03
x:0.19,y:0.05,z:0.03
x:0.21,y:0.05,z:0.04
x:0.21,y:0.04,z:0.04

首次按下进入积分x:0.00,y:0.00,z:0.00
x:0.00,y:0.00,z:0.00
x:0.00,y:0.00,z:0.00
x:0.00,y:0.00,z:0.00
x:0.00,y:0.00,z:-0.00
x:-0.00,y:-0.00,z:-0.01
x:-0.00,y:0.00,z:-0.01
x:0.00,y:0.00,z:-0.01
x:0.00,y:-0.01,z:0.05
x:0.00,y:-0.04,z:0.12
x:0.01,y:-0.06,z:0.17
x:0.01,y:-0.07,z:0.19
x:-0.00,y:-0.07,z:0.19

首次按下进入积分x:-0.00,y:0.00,z:-0.00
x:-0.00,y:0.00,z:-0.00
x:-0.00,y:0.00,z:-0.00
x:-0.00,y:0.00,z:-0.00
x:0.00,y:0.00,z:-0.02
x:0.01,y:-0.01,z:-0.07
x:0.02,y:-0.01,z:-0.14
x:0.02,y:0.00,z:-0.18
x:0.02,y:0.01,z:-0.20
x:0.03,y:0.02,z:-0.21

2.4 数据采集逻辑以及积分环节

现在我们可以获得效果非常好的数据，但是我们不能一直获取数据并运算，于是需要做一个“防误触”机制，即设置数据采集和运算的开始标志，最容易想到的便是按钮，按下开始采集，松开停止采集并且将采集到的数据进行运算。此外也可以设置一个阈值，当加速度到达某个阈值时才开始采集，但是后者不太稳定，而且阈值也不好设置，遂采用前者。

2.4.1 按键触发数据采集

直观点要实现按下按钮开始收集数据，松开按钮或者按下时间达到2s后停止记录数据，并且执行数据的一系列计算，然后再按下开启新的一段采集流程。

基本逻辑为：

1. 定义三个数组为三轴，按下按钮开始记录，每次循环中将本次数据记录到数组中；在首次按下按钮时初始化一系列变量：包括下标，时间，flag2(开始计时标志)，偏置项
2. 松开按钮或者时间计时为2s后停止记录并进行积分运算得到轨迹。同时更新flag1，flag2关闭记录通道和积分运算通道。

其中为了解决当计时2s已到但是按钮未松开导致一直在记录使数组溢出，定义flag1作为进入记录的标志；其中当按钮为松开状态时flag1为0，即只有按钮松开后才能够再次按下开始记录

为了避免平时即使没有记录也进入到积分环节，定义flag2，开始计时时才开始，积分结束后再关闭。

```
const int buttonPin=13;//按钮GPIO13
```

```
int buttonstate=0;//按钮状态
int lastbuttonstate=0;//上一次按钮状态
int i=0;//下标
unsigned long starttime=0;//开始时间
int flag1=0;//为了标志2s已到但是按钮未松开的情况的flag
int flag2=1;//为了标志开始计时的flag

void loop(){
if(buttonstate==LOW&&flag1==0){//如果按下按钮
    if(lastbuttonstate==HIGH){//首次按下按钮
      i=0;//下标更新
      starttime=millis();//起始时间更新
      flag2=0;//开始计时标志
      memset(x_x,0,sizeof(x_x));
      memset(y_x,0,sizeof(y_x));
      memset(z_x,0,sizeof(z_x));
      Serial.println("首次按下");
    }
    x_acc[i]=worldAccX;//更新加速度
    y_acc[i]=worldAccY;
    z_acc[i]=worldAccZ;
    i++;
  }

  if((millis()-starttime>=2000||
(lastbuttonstate==LOW&&buttonstate==HIGH))&&flag2==0){
    flag1=1;//关闭记录通道
    flag2=1;//关闭积分运算通道
    ...//积分运算和模型预测
    ...//串口输出和蓝牙发送等等
  }
  if(buttonstate==HIGH){flag1=0;}//如果按钮是松开的允许再按记录，如果是按到时间到了结束，
不触发flag1变化，不进记录
  lastbuttonstate=buttonstate;//记录上一次按钮状态
}
```

)000首次按下-0.010-0.010-0.010-0.010-0.010-0.010-0.010-0.010-0.010-0.010-0.010-0.010-0.010-0.010-0.010进入积分0000(

)0.0400.0400.0400.0400.0400.0400.0400.0400.0400.0400.0400.0400.04进入积分111111111111111

可以实现两种结束记录的方式。

其中是每隔50ms记录一次数据，理论上最多40个数据，但是为了避免溢出定义数组长度为50。至于每隔50ms没有在上述代码中体现出来，因为madgwick算法中的参数delta与时间有关，于是另外做了一个计时的变量count顺便计时50ms运行一次，就没有使用delay(50)的方法。

2.4.2 积分环节

对于获得的三个加速度数组，要对其进行两次积分，这时可以将这几个加速度数组看作分段函数，每一段的时间长度都是50ms，对其积分便是求图象上若干个长方形的面积累加，于是积分问题就转变为了对数组求累加和的问题。同时令两次积分的初始值都为0，就得到了坐标系下的相对运动轨迹。

```
float sumwhat(int i,float tool[50]){//辅助函数求tool数组的前i项和，再乘上时间间隔0.05s
  int j;
  float sum=0.0;
  for (j=0;j<=i;j++){
    sum+=tool[j]*0.05;
  }
  return sum;
}
void loop(){
    ...
    //积分环节
    Serial.println("进入积分");
    for (k=0;k<i;k++){
      x_v[k]=sumwhat(k,x_acc);
      y_v[k]=sumwhat(k,y_acc);
      z_v[k]=sumwhat(k,z_acc);
    }
    for (k=0;k<i;k++){
      x_x[k]=sumwhat(k,x_v);
      y_x[k]=sumwhat(k,y_v);
      z_x[k]=sumwhat(k,z_v);
    }
    ...
}
```

结果较为准确：

向上运动

```
首次按下进入积分x:-0.00,y:-0.00,z:0.0(
x:-0.00,y:-0.00,z:0.00
x:-0.00,y:-0.00,z:0.01
x:-0.00,y:-0.00,z:0.01
x:-0.00,y:-0.01,z:0.01
x:-0.00,y:-0.01,z:0.01
x:-0.00,y:-0.01,z:0.01
x:0.00,y:-0.01,z:0.02
x:0.01,y:-0.02,z:0.07
x:0.00,y:-0.03,z:0.16
x:-0.01,y:-0.06,z:0.25
x:-0.03,y:-0.07,z:0.31
x:-0.04,y:-0.09,z:0.34
x:-0.06,y:-0.11,z:0.35
x:-0.07,y:-0.12,z:0.36
```

向下运动

```
首次按下进入积分x:0.00,y:0.00,z:0.00
x:0.00,y:0.00,z:0.00
x:0.00,y:0.00,z:0.00
x:0.00,y:0.00,z:0.01
x:0.00,y:0.00,z:0.01
x:0.00,y:0.00,z:0.01
x:0.00,y:0.00,z:0.02
x:0.00,y:0.00,z:0.02
x:0.00,y:0.00,z:0.02
x:-0.00,y:0.01,z:0.00
x:0.00,y:0.03,z:-0.05
x:0.01,y:0.03,z:-0.12
x:0.00,y:0.03,z:-0.16
x:0.00,y:0.03,z:-0.19
x:-0.00,y:0.03,z:-0.20
x:-0.01,y:0.03,z:-0.20
x:-0.01,y:0.03,z:-0.19
```

2.5 模型训练与部署

到目前为止我们成功将运动轨迹解算成对应的相对位移数组，接下来要对上下左右四种轨迹的分类模型进行训练。

2.5.1 数据集收集

为了模拟真实魔杖挥舞场景，将传感器绑在筷子顶端，按照之前确定的方式固定，挥舞魔杖来采集数据。采集过程中也发现，按照人类的使用习惯，传感器安装在魔杖顶端，手握住末端，人手挥动时基本上只动手腕，因此不会出现纯上下左右平移的情况，而是会在Y轴上自然做出一些移动的，比如向上，人习惯向上移动时向内(-Y方向)挥动；向左，人也习惯向左时向内移动一些，这也证明了我们**忽略Y轴位移只看XOZ平面投影的处理和我们安装传感器的方法是正确的。**

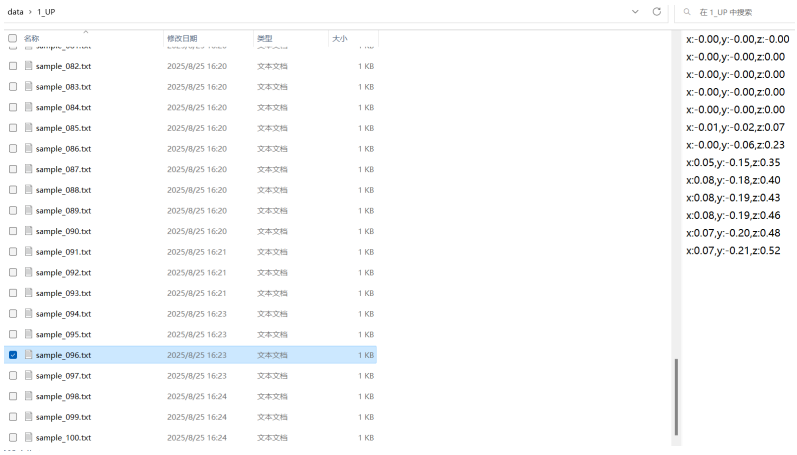
每次挥舞完成松开按钮后，将计算出来的三轴位移数据串口打印出来，再通过python的pySerial库读取串口数据并且保存在txt文档中：

```
...//在串口打印时要同时输出开始和结束标志，以方便python脚本判断应该保存的部分
    Serial.println("===START===");    // 一次动作开始
    for (k=0;k<i;k++){
      Serial.print("x:");
      Serial.print(x_x[k]);
      Serial.print(",y:");
      Serial.print(y_x[k]);
      Serial.print(",z:");
      Serial.println(z_x[k]);
      }
    Serial.println("===END===");      // 一次动作结束
...
```

mozhang.ipynb （数据收集脚本）

```
import serial
import time
import os

SERIAL_PORT = "COM7"
BAUD_RATE = 115200
SAVE_DIR = r"D:\新建文件夹 (5)\data\RIGHT"#保存路径
os.makedirs(SAVE_DIR, exist_ok=True)

ser = serial.Serial(SERIAL_PORT, BAUD_RATE, timeout=1)
time.sleep(2)

sample_idx=1
recording=False
buffer=[]

while True:
    line = ser.readline().decode("utf-8", errors="ignore").strip()
    if not line:
        continue
    if line=="===START===":
        print(f"开始采集样本 {sample_idx}")
        buffer = []
        recording = True
        continue
    if line=="===END===":
        if recording:
            filename = os.path.join(SAVE_DIR, f"sample_{sample_idx:03d}.txt")
            with open(filename, "w") as f:
                f.write("\n".join(buffer))
            print(f"样本 {sample_idx} 保存到 {filename}")
            sample_idx+=1
            recording=False
        continue
    if recording:
        buffer.append(line)

 ser.close()
```

最终四个方向各收集到100组数据，依次打包为文件夹。具体样例如下图所示。



2.5.2 数据预处理

将得到了这些txt文档，用python脚本每个逐行遍历，并用正则表达式取出需要的xz轴的数据，再将其打包为一个2×50的矩阵，每个方向的数据整合成一个100×2×50的大矩阵，获得的四个大矩阵先整合起来再将每个小矩阵展平，最终得到一个400×100的矩阵。然后再定义标签向量，依次为100个1、2、3、4。

mozhang_train.ipynb

```
import numpy as np
import re
import glob
def process_txt(file_path, max_len=50):
    #处理单个txt文件，返回一个2x50的矩阵
    x_vals, z_vals=[],[]
    with open(file_path, 'r', encoding='utf-8') as f:
        for line in f:
            match=re.findall(r"x:([-+]?\d*\.?\d+),y:([-+]?\d*\.?\d+),z:([-+]?
\d*\.?\d+)", line)
            if match:
                x, _, z=match[0]
                x_vals.append(float(x))
                z_vals.append(float(z))
    #不够的补0
    x_vals=(x_vals+[0.0]*max_len)[:max_len]
    z_vals=(z_vals+[0.0]*max_len)[:max_len]

    return np.array([x_vals, z_vals])
# 批量读取文件
files_1=glob.glob(r"D:\新建文件夹 (5)\data\1_UP\*.txt")
files_2=glob.glob(r"D:\新建文件夹 (5)\data\2_DOWN\*.txt")
files_3=glob.glob(r"D:\新建文件夹 (5)\data\3_LEFT\*.txt")
files_4=glob.glob(r"D:\新建文件夹 (5)\data\4_RIGHT\*.txt")
```

```
matrices_1=[process_txt(f) for f in files_1]
matrices_2=[process_txt(f) for f in files_2]
matrices_3=[process_txt(f) for f in files_3]
matrices_4=[process_txt(f) for f in files_4]
#堆叠成三维数组 100×2×50
data_1=np.stack(matrices_1)
data_2=np.stack(matrices_2)
data_3=np.stack(matrices_3)
data_4=np.stack(matrices_4)
#整合
X=np.concatenate([data_1,data_2,data_3,data_4],axis=0)
y=np.array([1]*100+[2]*100+[3]*100+[4]*100)
#展平
X=X.reshape(X.shape[0],-1)
```

```
print(data_1.shape)
print(data_1[0])
```

```
(100, 2, 50)
[[-0.   -0.   -0.   -0.   -0.01 -0.02 -0.05 -0.11 -0.15 -0.16 -0.15 -0.14
  -0.12  0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.
   0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.
   0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.
   0.    0.  ]
 [ 0.    0.01  0.01  0.01  0.01  0.07  0.2   0.31  0.35  0.37  0.38  0.4
   0.41  0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.
   0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.
   0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.
   0.    0.  ]]
```

2.5.3 模型训练

接下来我们将数据集按照7：3比例分为训练集和验证集，用支持向量机SVM训练模型，得到了准确率为0.942的模型。

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score
#划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=41, stratify=y
)
#用SVM训练
clf = SVC(kernel='rbf', C=1, gamma=1.0)#此处gamma参数决定rgb核的宽度，经尝试1.0时效果最好，转c语言的时候gamma需要为定值
clf.fit(X_train, y_train)
#预测
y_pred = clf.predict(X_test)
#评估
print("准确率:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
准确率: 0.9416666666666667
              precision    recall  f1-score   support

           1       0.96      0.90      0.93        30
           2       0.97      1.00      0.98        30
           3       0.96      0.90      0.93        30
           4       0.88      0.97      0.92        30

    accuracy                           0.94       120
   macro avg       0.94      0.94      0.94       120
weighted avg       0.94      0.94      0.94       120
```

以下为将所有数据经模型运行一遍后得到的预测结果，几乎是能够完全分类正确的。

[32]:
```
B_1=data_1.reshape(data_1.shape[0],-1)
B_2=data_2.reshape(data_2.shape[0],-1)
B_3=data_3.reshape(data_3.shape[0],-1)
B_4=data_4.reshape(data_4.shape[0],-1)

a_1=clf.predict(B_1)
a_2=clf.predict(B_2)
a_3=clf.predict(B_3)
a_4=clf.predict(B_4)

print(a_1)
print(a_2)
print(a_3)
print(a_4)
```

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 3 1 1 1 2 1 1 1 1 1 1 4 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 4 1 1 1 4 1 1 1 1 1 4 1 1 1 1 1 1 1]
[2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 4 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2]
[2 3 3 3 3 3 3 3 2 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 3 4 1 4 4 3 3
 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3]
[4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 1 4 4 4 4 4 4 3 4 4 4 4 4 4 4 4 4
 4 2 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
 4 3 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4]
```

2.5.4 模型部署

目前只在python中训练出了模型，我们需要将其部署到esp32上，使其不依赖于电脑便可以运行模型实现轨迹分类。

需要用到上位机吗？并不需要，因为：

1.上下左右轨迹特征度高

2.数据量下，仅400个

3.用的是机器学习的SVM算法，比较简单

因此说明我们得到的模型相对简单，可能esp32自带的算力便可以支持。

一个模型本身可以看作是一个包含矩阵运算的复杂数学函数，只要我们能够将模型的数学形式表示出来，便可以用c语言的形式写出来，但是这种模型的参数一般都非常多而且复杂，于是我们选择使用micromlgen库，其支持将简单的python模型导出为c语言形式。

```
from micromlgen import port
c_code=port(clf)
with open(r"D:\新建文件夹 (5)\data\classifier.h", "w") as f:
    f.write(c_code)
```

用micromlgen库成功导出模型的c语言版本后，得到的结果为：（仅为部分代码，原函数见mozhang\classifier.h)

```
#pragma once
#include <cstdarg>
namespace Eloquent {
    namespace ML {
        namespace Port {
            class SVM {
                public:
                    /**
                    * Predict class for features vector
                    */
```

```
                    int predict(float *x) {
                        float kernels[131] = { 0 };
                        float decisions[6] = { 0 };
                        int votes[4] = { 0 };
                        kernels[0] = compute_kernel(x,   0.0  , 0.0  , -0.0  ,
-0.0  , 0.1  , 0.18  , 0.19  , 0.19  , 0.19  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  ,
0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0
, 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  ,
0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0
, 0.0  , -0.0  , -0.0  , 0.03  , 0.14  , 0.27  , 0.31  , 0.33  , 0.33  , 0.34  ,
0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0
, 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  ,
0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0
, 0.0  , 0.0  , 0.0  , 0.0  , 0.0  , 0.0 );
                        ...
                        decisions[0] = -0.206241160743
                        + kernels[0] * 0.06523232929
                        + kernels[6] * 0.60241767888
                        + kernels[7] * 0.239027242719
                        + kernels[8] * 0.452574668479
                        + kernels[9] * 0.31387810585
                        + kernels[11]
                        + kernels[15] * 0.786632371654
                        + kernels[17] * 0.353397278291
                        + kernels[18]
                        + kernels[19] * 0.162507380842
                        + kernels[20] * 0.252686093615
                        ...
                    }
                }
            }
        }
```

而mozhang.ino的改动如下：导入了clf模型，添加了存放输入输出的变量

```
...
#include "classifier.h"
Eloquent::ML::Port::SVM clf;//实例化模型
...
float input[100]={0};//用于跑模型的变量
int pred;//预测结果
...
void loop(){
    ...
    for (k=0;k<50;k++){
          input[k]=x_x[k];
          input[k+50]=z_x[k];
        }
    ...
    pred=clf.predict(input);
    Serial.println(pred);
}
```

经上电测试，发现上下左右的标签对应变成了0、1、2、3，但是仍然实现了轨迹分类，并且模型运行速度非常快，松开按钮后零延迟输出预测结果，说明esp32的算力是充足的。

```
entry 0x4008059c
MPU6050 Found!
首次按下
进入积分
===START===
0
===END===
首次按下
进入积分
===START===
1
===END===
首次按下
进入积分
===START===
2
===END===
首次按下
进入积分
===START===
3
===END===
```

这同时也定义了一个工作流，如果要添加新的轨迹的话，只需要按照数据收集、模型训练、模型部署的顺序更新一遍即可。

2.6 蓝牙传输

esp32自带蓝牙模块，可以直接用库函数调用。

其中魔杖的开发板作为主机，小车的开发板作为从机。依次定义名称为"ESP32_BT"和"vehicle"，但是并不能用名称连接，而是要用从机的mac地址连接，连接再用SerialBT.print()便可实现蓝牙通信。同时为了直观观察蓝牙是否连接成功，定义了一个LED灯来指示，成功连接就闪烁三下，失败就一直亮。

主机mozhang.ino

```
#include "BluetoothSerial.h"
...
BluetoothSerial SerialBT;//定义
...
void setup(){
  ...
  SerialBT.begin("ESP32_BT",true);
  uint8_t macAddr[6]={0x84, 0x1F, 0xE8, 0x15, 0xBE, 0x4E};

  if (SerialBT.connect(macAddr)) {
    Serial.println("Connected to Slave!");
    for (k=0;k<3;k++) {//蓝牙连接成功提升:灯闪三下
      digitalWrite(LED_PIN, HIGH);
      delay(300);
      digitalWrite(LED_PIN, LOW);
      delay(300);
    }
  } else {
    Serial.println("Failed to connect to Slave.");
    digitalWrite(LED_PIN, HIGH);//蓝牙连接失败提示:灯一直亮
  }
}
void loop(){
    ...
    SerialBT.println(pred);//发送
}
```

从机vehicle.ino

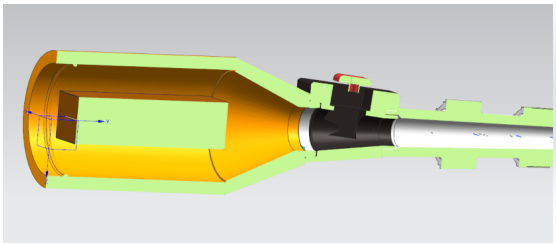
```
#include "BluetoothSerial.h"
#include "esp_bt_device.h"
```

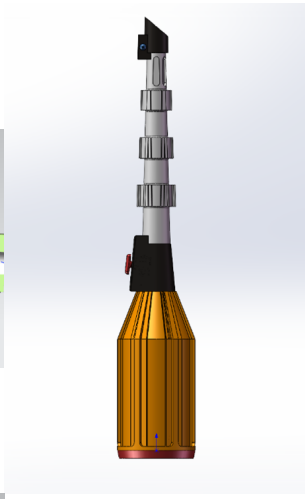
```
BluetoothSerial SerialBT;

void setup() {
  Serial.begin(115200);
  SerialBT.begin("vehicle");  //从机名称
  const uint8_t* mac = esp_bt_dev_get_address();
  Serial.printf("ESP32 Bluetooth MAC address: %02X:%02X:%02X:%02X:%02X:%02X\n",
                mac[0], mac[1], mac[2], mac[3], mac[4], mac[5]);
}
void loop() {
    if (SerialBT.available()) {
      char msg = SerialBT.read();
      //用msg控制小车
    }
  }
```

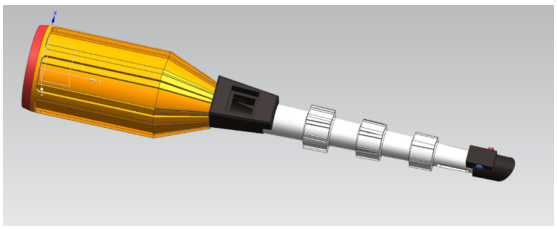
```
load:0x40078000,len:15672
load:0x40080400,len:3152
entry 0x4008059c
ESP32 Bluetooth MAC address: 84:1F:E8:15:BE:4E
```

2.7 魔杖建模

魔杖制作完成后，我们用solidworks进行了建模，其中包括电池存放的尾部，卡住传感器的顶端以及为电源开关和按钮预留了开口，但是因为3D打印的价格太贵了，就没有打印出来，而是继续用筷子来模拟魔杖的形状。







三、小车部分

2.1 硬件

小车所用的硬件有TB6612稳压板和带光电编码器的直流电机，以及控制他们的esp32开发板。

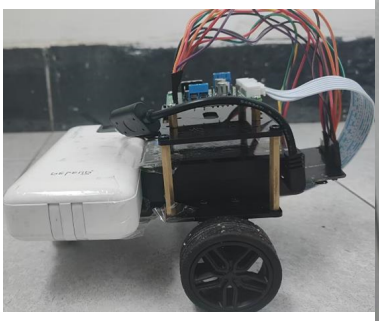
在TB6612稳压板上，其对电机的控制通过七个管脚：STBY、PWMA/PWMB、AIN1/AIN2、BIN1/BIN2。其中PWM控制速度，IN1/IN2控制方向，只有STBY高电平时才会激活控制。

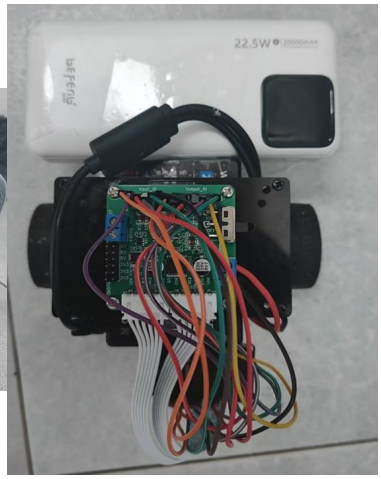
编码器的相关管脚为E1A/E1B，E2A/E2B，依次表示左右轮的A相和B相，A相用来判断速度，用固定时间内的脉冲数来换算；B相用来判断方向，通过B相相对于A相是超前还是落后来判断。本来用的是双轮平衡小车，想要通过调节PID来使其直立平衡和运动平衡，但是控制需要得到双轮转速来进行反馈调节，每转脉冲数600，轮径4.5cm，轮距10cm都已经测出。但是右轮电机的光电编码器是坏的，不能准确测出脉冲数，因此不能测出速度，不能实现双轮平衡。

于是我们采取添加拓展版加牛眼轮的方法使小车能够平衡。

管脚连接为：

```
#define PWMA 12
#define AIN1 14
#define AIN2 27
#define PWMB 13
#define BIN1 33
#define BIN2 32
#define STBY 26

#define ENCODER_L   19   //左轮A相
#define DIRECTION_L 22   //左轮B相
#define ENCODER_R   18   //右轮A相
#define DIRECTION_R 23   //右轮B相
```





2.2 控制逻辑

因为测速有问题，并不能实现过于精确的逻辑控制，于是仅仅做了简单的控制逻辑：

```
void loop()
{

  if (SerialBT.available()) {
    msg = SerialBT.read();
    Serial.println(msg);
    if(msg=='0'){//前进
      SetPWM(1, 70);
      SetPWM(2, 70);
    }
    if(msg=='1'){//停止
        SetPWM(1, 0);
        SetPWM(2, 0);
    }
```

```
    if(msg=='2'){//左转
      SetPWM(1,40);
      SetPWM(2,70);
    }
    if(msg=='3'){//右转
      SetPWM(1,70);
      SetPWM(2,40);
    }
  }
}
```

如果测速功能正常，可以按照当前状态是否静止，或者当前是前进还是后退，用if-else和switch-case语句来实现更多样的控制。

四、成果、可改进之处和小组分工

4.1 成果

我们最终实现了用魔杖的上下左右四个轨迹来对小车进行控制。总体上实现了预期成果的基本要求。

而且也实现了一整套训练轨迹的工作流，相当于**搭建起了一套具有通用性的算法基础框架**，基于此基础可以自由更改轨迹、被控对象以及控制逻辑，仅需要按照先前的工作流重新训练一遍或者在基础上更新一些不必要的参数即可实现自由改变。比如想要添加圆形轨迹，或者是将被控对象改为空调，均不需要大改代码底层算法，只需要微调那些非必要的变量即可。

成果视频见附件。

所有代码均已经上传github

https://github.com/DKESTXD/qiangao-mozhang

4.2 可改进之处

世界坐标系定义问题。前文中说到世界坐标系在上电瞬间就确定下来了，因此在使用魔杖时只能始终面朝上电方向，或者说这时左右变成了一个绝对的定义而不是相对人身体的定义，比如如果上电后转身180°再使用的话，左右轨迹的判断就反过来了

小车的平衡问题。因为右轮电机的编码器问题，没能实现优雅的双轮平衡小车，而是加了辅助轮。

还可以再拓展。我们只是做完了最基础的框架，而且用小车这一被控对象来直观表现，没有实现预期成果中的智能家具的尝试。此外我们的轨迹只考虑对xOz平面上的轨迹二维投影，而且使用的模型也很简单，我们**也可以考虑将y轴考虑进去，用3D轨迹，用更大更复杂的模型训练，使得轨迹判断更精确，甚至能够解决世界坐标系定义导致的使用魔杖时转身导致不准的问题。**

4.3 小组分工

组员及分工情况：

李丰克：魔杖功能实现，小车控制代码，结题答辩ppt，报告撰写

陈思攀：魔杖建模，硬件调试，开题答辩ppt，结题pre

赵彬序：小车组装，硬件调试，开题pre