

Lesson

2

Balanced Control



Introduction:

In this lesson, there are three parts. We will teach you how to realize the balanced upright walking of the balance car.

Tumbler Balanced Control

- 1.1 Hardware Design
- 1.2 Software Design
- 1.3 Upload Validation

Preparations:

- one car (with a battery)
- one USB cable

1.1 Hardware Design

In this course, we use the MPU6050 module (as shown in figure 1.1.1~1.1.4) to measure the values related to the attitude changes, such as the angle value and the angle speed value, and then adjust and control the driving motor accurately through PID control algorithm, so as to realize the balance control function (the specific use of these values will be described in the following content).

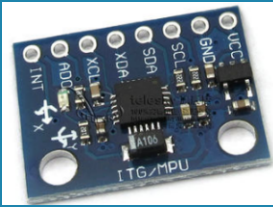


Figure 1.1.1 The real object of MPU6050

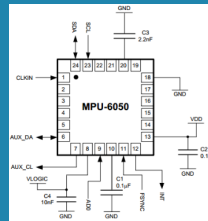


Figure 1.1.2 The structure of MPU6050

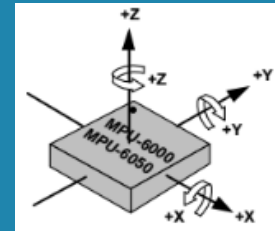


Figure 1.1.3 Orientation of Axes of Sensitivity and Polarity of Rotation

Mpu-60x0 is composed of the following key blocks and functions (As shown in figure 1.1.4):

1. Three axis MEMS rate gyroscope sensor with 16 bit ADC and signal conditioning
2. Three axis MEMS acceleration sensor with 16 bit ADC and signal conditioning
3. DMP engine
4. Main I2C and SPI (mpu-6000 only) serial communication interface
5. Auxiliary I2C serial interface for the third-party magnetometer and other sensors
6. Clock
7. Sensor data register
8. FIFO
9. Interrupt function
10. Digital output temperature sensor
11. Gyroscope and self-test function
12. Prejudice and LDO
13. Charging pump

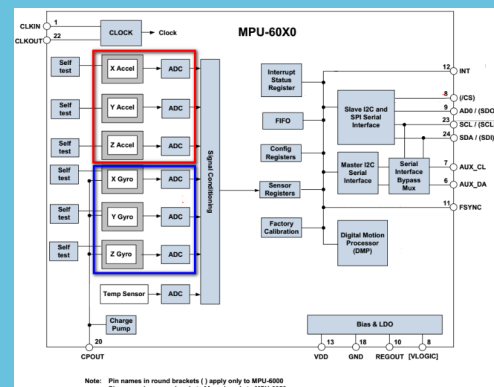


Figure 1.1.4 System structure diagram

Finally, according to the schematic (as shown in figure 1.1.5), let's see which pin of Nano is connected to the SCL and SDA of MPU6050 and start to write programs. (As shown in figure 1.1.6)

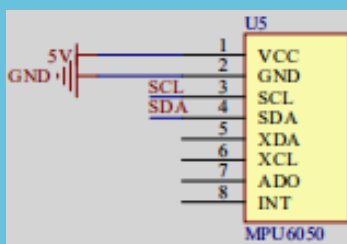


Figure 1.1.5 The schematic of the MPU6050

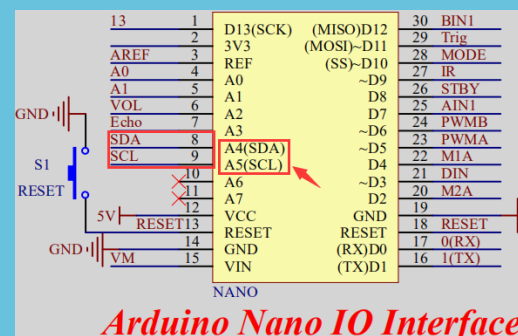


Figure 1.1.6 IIC communication is basically SDA and SCL pins

1.2 Software Design:

Please open [ELEGOO Tumbler Self-Balancing Car Tutorial](#) -> Lesson 2 Balanced Control -> Balance-> Balanced_Car->Balanced_Car.ino in the current folder.

Next, we will analyze the whole balance principle and explain the process of programming.

(As shown in figure 1.2.1)

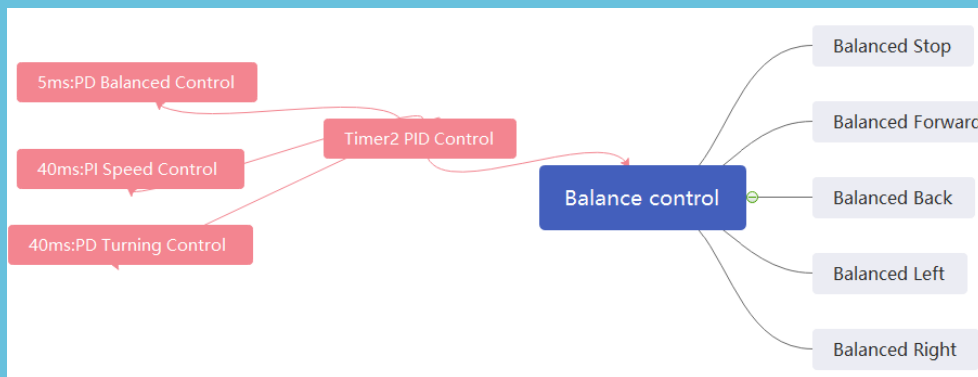


Figure 1.2.1 Programming ideas

Because the MPU6050 module transfers the data with Nano was permitted by the IIC protocol, we also need to add the "Wire.h" library.

```
//in Balanced.cpp

#include "Wire.h"
#include "MPU6050_6Axis_MotionApps20.h"
```

Class MPU6050 declaration

```
//in Balanced.h

class Mpu6050
{
public:
    void init();
    void DataProcessing();
    Mpu6050();

public:
    int ax, ay, az, gx, gy, gz;
    float dt, Q_angle, Q_gyro, R_angle, C_0, K1;
}
```

The core idea of balancing vehicle is "dynamic stability". The attitude data is obtained by MPU6050, and then the motor is adjusted by using control algorithm to achieve the balanced effect. (As shown in figure 1.2.2)

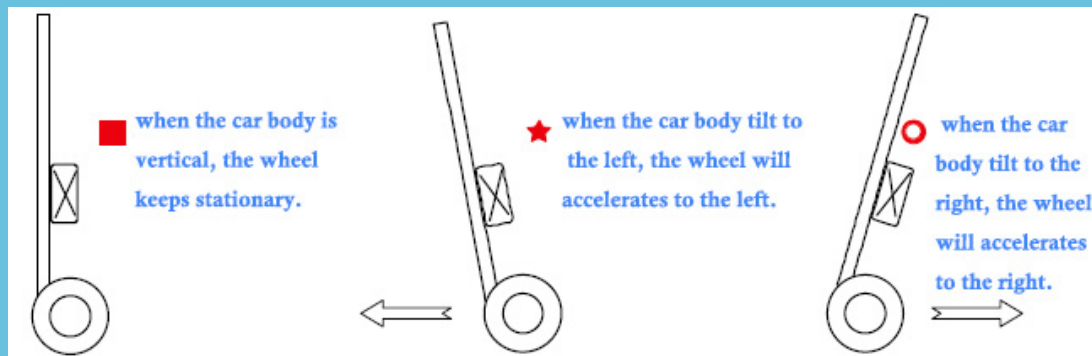


Figure 1.2.2 Balancing principle of the car

When the car body is vertical to the wheel, the body will keep still.

When the car tilts to the left, the wheels accelerate to the left, and when the speed of the wheels is greater than the speed of the car, the car body will return to a state perpendicular to the wheels.

When the car tilts to the right, the wheels accelerate to the right. When the speed of the wheels is greater than the speed of the car body, the car body will return to the state perpendicular to the wheels.

So we can divide the Tumbler motion control task into 3 basic control tasks.

- ①Balance Control (Vertical ring): To keep the Tumbler maintaining the upright balance by controlling its forward and reverse motion.
- ②Speed Control (Speed ring): To control the speed by adjusting the inclination of the Tumbler, in fact, it finally evolves to control the wheel speed by controlling the rotation speed of the motor.
- ③Direction Control (Turning ring): To control the direction change of the Tumbler by controlling the rotational differential between the two motors.

Let's take a look at the declaration of **Balanced** class in the program.

```
//in Balanced.h

class Balanced
{
public:
    Balanced();
    void Get_EncoderSpeed();
    void PD_VerticalRing();
    void PI_SpeedRing();
    void PI_SteeringRing();
    void Total_Control();

    void Motion_Control(Direction direction);
    void Stop();
    void Forward(int speed);
    void Back(int speed);
    void Left(int speed);
    void Right(int speed);

    /*Speed value*/
    double pwm_left;
    double pwm_right;
    int encoder_left_pulse_num_speed;
    int encoder_right_pulse_num_speed;
    /*Cnt*/
    int interrupt_cnt;

    /*PID parameter*/
    /*PD_VerticalRing*/
    double kp_balance, kd_balance;
    /*PI_SpeedRing*/
    double kp_speed, ki_speed;
    /*PI_SteeringRing*/
    double kp_turn, kd_turn;

    double speed_filter;
    double speed_filter_old;
    double car_speed_integral;
    double balance_control_output;
    double speed_control_output;
    double rotation_control_output;
    int setting_turn_speed;
    int setting_car_speed;

private:
    #define ANGLE_MIN -27
    #define ANGLE_MAX 27
    #define EXCESSIVE_ANGLE_TILT (kalmanfilter.angle < ANGLE_MIN || ANGLE_MAX < kalmanfilter.angle)
};
```

The three kinds of control (upright, speed and direction) of the balanced vehicle are finally superimposed as the control quantity of the motor output voltage. Of the three controls mentioned above, the vertical control is the key, so the adjustment speed should be very fast. From the point of view of the upright control of the vehicle model, the other two controls will become its interference, so the speed and direction control should be as smooth as possible to reduce interference with the upright control, so its relative adjustment speed should be set to slow.

In our program, timer2 is used to interrupt the timing of the car.

Declaration of class Timer2

```
//in Balanced.h

class Timer2
{
public:
    void init(int time);
    static void interrupt();
private:
    #define TIMER 5
};
```

The initialization setting of Timer2 is interrupted every 5ms.

```
//in Balanced_Car

void setup()
{
    Timer2.init(TIMER);
}
```

```
//in Balanced.cpp

void Timer2::init(int time)
{
    MsTimer2::set(time,interrupt);
    MsTimer2::start();
}
```

Timer 2 interrupt function

Get wheel speed(**Balanced.Get_EncoderSpeed()**) and MPU6050 data(**Mpu6050.DataProcessing()**) every 5 ms

Vertical control every 5 ms(**Balanced.PD_VerticalRing()**)

Speed control every 40 ms(**Balanced.PI_SpeedRing()**)

Steering control control every 40 ms (**Balanced.PI_SteeringRing()**)

```
//in Balanced.cpp

static void Timer2::interrupt()
{
    sei();//enable the global interrupt
    Balanced.Get_EncoderSpeed();
    Mpu6050.DataProcessing();
    Balanced.PD_VerticalRing();
    Balanced.interrupt_cnt++;
    if(Balanced.interrupt_cnt > 8)
    {
        Balanced.interrupt_cnt=0;
        Balanced.PI_SpeedRing();
        Balanced.PI_SteeringRing();
    }
    Balanced.Total_Control();
}
```

Next, let's take a closer look at the realization of three control rings (balance ring, speed ring and steering ring)
The realization of the three controls is mainly achieved by the traditional PID feedback control, PID is proportional, integral and derivative. When a certain physical quantity needs to be "stable" (such as maintaining balance, stabilizing temperature, rotating speed, etc.), PID will be of great use. (As shown in figure 1.2.3)

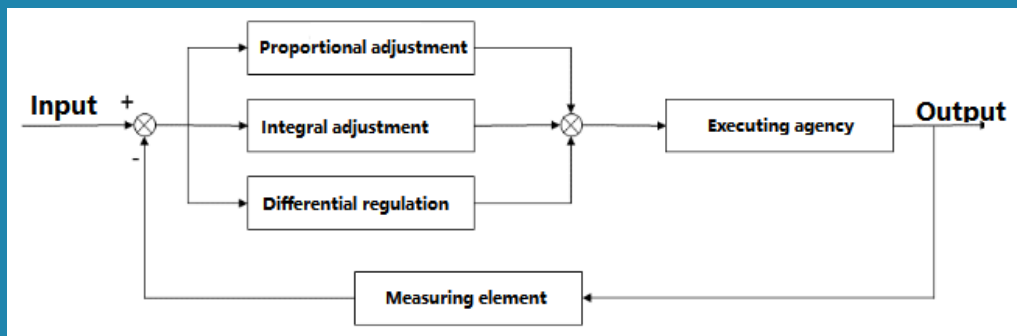


Figure 1.2.3 PID control process

First, let's give two simple examples to help you understand the concept:

① Suppose I want to control the temperature of a pot of water and keep it at 40 °C.

At this time, you may think it's very simple. If the temperature is below 40 degrees, you can heat it. If the temperature is above 40 degrees, you can cut off the power. Then why do we need calculus? Yes, it can be done under the condition of low requirements. Let's look at another example.

② Suppose what we control is a car and keep its speed at 40 km / h?

Therefore, in most cases, it is relatively simple to control a physical quantity with "switch quantity". Sometimes, it can't be stable. MCU and sensor are not infinite fast, cause collecting and controlling takes time.

Moreover, the control object has inertia. For example, if you unplug a heater, its "residual heat" (i.e. thermal inertia) may make the water temperature rise for a little higher.

We can make the following understanding simple: When we design something, we need proofing first. This sample is roughly the same as our goal, but the details are not in place. This is P. Because it is slightly different from the ideal goal, we need to modify the sample to approach the ideal goal. This is I. At the end of the period, we are not allowed to modify it at will. Even if we want to modify it, there are restrictions, and this is D.

In summary, PID is

- ① Approach to ideal goal.
- ② Prediction of the future development trend.
- ③ Eliminate the errors caused by various factors, and finally stabilize at the ideal target value.

The formula is as follows:

$$u(k) = K_p e(k) + K_i \sum_{n=0}^k e(n) + K_d (e(k) - e(k-1))$$

U (k): PID output control quantity. For example, PWM.

Kp: Kp is the scale factor, and the scale factor is the magnification K of the straight line passing through the coordinate point (0,0). The larger K is, the larger the slope of the straight line is, so it is used in $y = k * x$, where k is the scale factor KP, which is abbreviated as KP, so it is $y = Kp * X$

e (k): And X is the difference between the measured value of the current sensor and the target value. For short, error e (k), $e(k) = \text{currentValue} - \text{totalValue}$

Y is the corresponding output value u (k) of the control, so when only proportional adjustment is used, $u(k) = K_P * (\text{currentValue} - \text{totalValue})$, but at this time, the system will vibrate severely. Proportional control is like adding water to or bailing out of a water tank through a bucket. Suppose that we need to stabilize the horizontal plane in plane a, while the actual horizontal plane is in plane B, then the horizontal difference $\text{Err} = A - B$, then the amount of water we need to add to it is $K_P * \text{Err}$, and K_P is our proportional control coefficient.

If $A > B$, Err is positive, add water to the water tank; If $A < B$, Err is negative, scoop water out of the water tank. So as long as there is a difference between the expected horizontal plane and the actual horizontal plane, we will adjust the system by adding and subtracting water through buckets. At the same time, the size of K_P also has an impact on the performance of the system. If the value of K_P is large, the advantage is that the speed from plane B to plane A is fast, and the disadvantage is that when plane B is close to plane A, the system will produce relatively large oscillations. If the value of K_P is relatively small, the advantage is that the system oscillation is small when the plane B is close to the plane A, and the disadvantage is that the speed from the plane B to the plane A is slow.

Some people may have doubts here. If the proportional control coefficient K_P is set to 1 directly, and then the amount of water added is directly $\text{Err} = A - B$. However, in fact, many systems can not achieve this. For example, for the temperature control system, the actual temperature is 10 degrees. I want to raise the temperature to 40 degrees by heating. Can we add 30 degrees to the system exactly at one time? Obviously, that's impossible. Then the final result of proportional control is that the value of Err tends to 0. So usually proportional control is not used alone.

$$K_i \sum_{n=0}^k e(n)$$

Ki: Ki is the integral coefficient.

$$\sum_{n=0}^k e(n)$$

That is, to integrate the error, to sum the accumulated error infinitely and takes the limit.

Why should we introduce integral ?

The integral reflects the past state.

For example, suppose we heat the thermos bottle in a very cold place to keep it stable at 50 °C, Under the action of P, the water temperature slowly rises to 45 degrees, but at this time, because the weather is too cold, the water cooling speed is the same as the P heating speed, and the water temperature will always stop at 45 degrees, less than 50 degrees. So we need to set an integral quantity. As long as the deviation exists (the temperature is not reached), the system will continue to integrate (accumulate) the deviation. At this time, the system will realize that the temperature is not reached and continue to heat. When the target temperature is reached, the integral value will not change anymore.

The integral reflects the past state.

Because the output of the proportional action is proportional to the size of the error ($y = K_P * e(k)$), the larger the error is, the larger the output is, the smaller the error is, the smaller the output is, the error is zero, and the output is zero. Because the output is zero when there is no error, it is impossible to eliminate the error completely and make the controlled U (k) value reach the given value. There must be a stable error to maintain a stable output in order to keep the U (k) value of the system stable. This is what is commonly referred to as the proportional function is poor regulation and has static difference. Strengthening the proportional function can only reduce the static difference, but not eliminate the static difference (static difference: static error, also known as steady error).

Y is the corresponding output value $u(k)$ of the control, so when only proportional adjustment is used, $u(k) = K_P * (\text{currentValue} - \text{totalValue})$, but at this time, the system will vibrate severely. Proportional control is like adding water to or bailing out of a water tank through a bucket. Suppose that we need to stabilize the horizontal plane in plane A, while the actual horizontal plane is in plane B, then the horizontal difference $\text{Err} = A - B$, then the amount of water we need to add to it is $K_P * \text{Err}$, and K_P is our proportional control coefficient.

If $A > B$, Err is positive, add water to the water tank; If $A < B$, Err is negative, scoop water out of the water tank. So as long as there is a difference between the expected horizontal plane and the actual horizontal plane, we will adjust the system by adding and subtracting water through buckets. At the same time, the size of K_P also has an impact on the performance of the system. If the value of K_P is large, the advantage is that the speed from plane B to plane A is fast, and the disadvantage is that when plane B is close to plane A, the system will produce relatively large oscillations. If the value of K_P is relatively small, the advantage is that the system oscillation is small when the plane B is close to the plane A, and the disadvantage is that the speed from the plane B to the plane A is slow.

Some people may have doubts here. If the proportional control coefficient K_P is set to 1 directly, and then the amount of water added is directly $\text{Err} = A - B$. However, in fact, many systems can not achieve this. For example, for the temperature control system, the actual temperature is 10 degrees. I want to raise the temperature to 40 degrees by heating. Can we add 30 degrees to the system exactly at one time? Obviously, that's impossible. Then the final result of proportional control is that the value of Err tends to 0. So usually proportional control is not used alone.

$$K_i \sum_{n=0}^k e(n)$$

Ki: Ki is the integral coefficient.

$$\sum_{n=0}^k e(n)$$

That is, to integrate the error, to sum the accumulated error infinitely and takes the limit.

Why should we introduce integral ?

The integral reflects the past state.

For example, suppose we heat the thermos bottle in a very cold place to keep it stable at 50 °C, Under the action of P, the water temperature slowly rises to 45 degrees, but at this time, because the weather is too cold, the water cooling speed is the same as the P heating speed, and the water temperature will always stop at 45 degrees, less than 50 degrees. So we need to set an integral quantity. As long as the deviation exists (the temperature is not reached), the system will continue to integrate (accumulate) the deviation. At this time, the system will realize that the temperature is not reached and continue to heat. When the target temperature is reached, the integral value will not change anymore.

The integral reflects the past state.

Because the output of the proportional action is proportional to the size of the error ($y = K_P * e(k)$), the larger the error is, the larger the output is, the smaller the error is, the smaller the output is, the error is zero, and the output is zero. Because the output is zero when there is no error, it is impossible to eliminate the error completely and make the controlled $U(k)$ value reach the given value. There must be a stable error to maintain a stable output in order to keep the $U(k)$ value of the system stable. This is what is commonly referred to as the proportional function is poor regulation and has static difference. Strengthening the proportional function can only reduce the static difference, but not eliminate the static difference (static difference: static error, also known as steady error).

Therefore, the introduction of integral function can reduce the error under static conditions, as close as possible to the target value, but when integrating, attention should be paid to setting the integral limit to prevent the amount of integration too large to control.

$K_d(e(k) - e(k-1))$ Kd is the differential coefficient.

Differential reaction is the rate of change of error.
Differential predicts future state.

$(e(k) - e(k-1))$ Current error minus last error

Here we also illustrate the role of differentiation:

Let's say we have a spring that's in equilibrium. Now, let's pull it and let go. Because the resistance is very small, it may oscillate for a long time before it stops in the balance position again. But if you submerge the spring in water, pull it again: in this case, the time to stop in the balance position is much shorter. Because of the "damping" effect in water, the "rate of change" of the controlled physical quantity tends to 0. And differential D works like this.

At this time, you should have known that P, I, D are three different regulating functions, which can be used alone (P, I, D), two for two (PI, PD), or three together (PID).

Now, let's go back to our program, which uses PD control in our Vertical ring.

```
//in Balanced.cpp

void Balanced::PD_VerticalRing()
{
    balance_control_output= kp_balance * (kalmanfilter.angle - 0) + kd_balance * (kalmanfilter.Gyro_x - 0);
}
```

Generally, PID of robot related devices is recommended to omit integral link, which is mainly due to the existence of integral saturation. When the static error is too large, even though the device reaches the limit stroke (such as the servo motor turning to the maximum angle position or the DC motor running to the maximum speed), the error cannot be eliminated.

We take the current balanced car as an example, assuming that the car has become stable at a certain time, and suddenly there is a problem with the sensor, which detects too large or too small data, so that the control system mistakenly thinks that the car has a great deviation from the target, resulting in a sudden increase in static error, an increase in the integral term and an acceleration of motor speed. Then the car loses balance. So in order to simplify the problem, we only used PD control.

Then, we analyze the vertical control model compared with the simple pendulum model.

Simple pendulum model are shown in the figure 1.2.5:

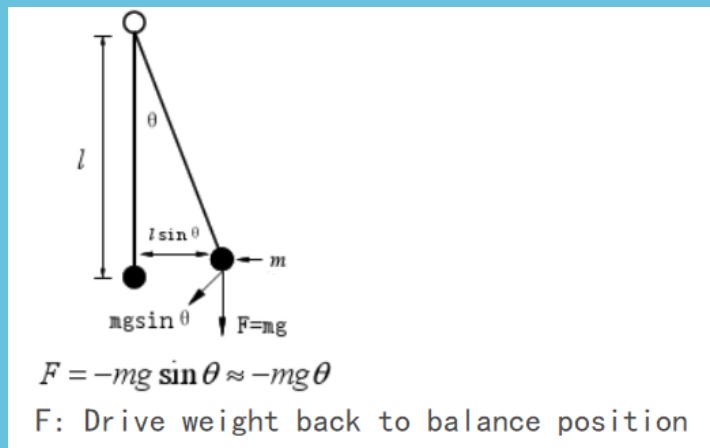


Figure 1.2.5 Simple pendulum model

Under this force, the simple pendulum will move periodically, but it will be damped by the air when it moves, so it will eventually balance.

$\sin \theta \approx \theta$. Mathematically, when θ is very small, it can be approximately equal to $\sin \theta$, which is convenient for calculation.

The stress analysis of car body is shown in the figure 1.2.6:

Suppose it is a negative feedback regulation, and the wheel acceleration a is positively proportional to the inclination angle, and the proportion is K_1 .

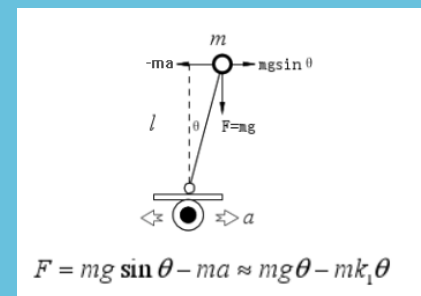


Figure 1.2.6 Simple pendulum model

In order to stabilize faster, the damping force also needs to be increased, so the formula can be changed as follows:

$$F = mg\theta - mk_1\theta - mk_2\theta'$$

Finally, the wheel acceleration control algorithm can be obtained:

$$a = k_1\theta + k_2\theta'$$

Therefore, in order to achieve vertical balance of the vehicle, it is necessary to measure the inclination angle θ and angular velocity θ' of the vehicle model.

MPU6050 is initialized. And since MPU6050 uses IIC communication, IIC (wire. Begin()) needs to be initialized as well.

```
//in Balanced.cpp

void Mpu6050::init()
{
    Wire.begin();
    MPU6050.initialize();
}
```

Because the accelerometer and gyroscope of MPU6050 have their own advantages and disadvantages, there is no accumulated error in the acceleration value of the three axes, and the inclination angle can be obtained by calculating $\tan()$, but it contains too much noise (because the acceleration will be generated when the object to be measured moves, and the vibration will be generated when the motor is running), so it can not be used directly; the gyroscope has little impact on the external vibration, with high accuracy, and it can not be used directly by calculating $\tan()$. The angle of inclination can be obtained by the integration of angular velocity, but there will be accumulated error. Therefore, in order to obtain the accurate obliquity and angular velocity values, we need to use Kalman filter.

Kalman filter parameter setting.

```
//in Balanced.cpp

Mpu6050::Mpu6050()
{
    dt = 0.005, Q_angle = 0.001, Q_gyro = 0.005, R_angle = 0.5, C_0 = 1, K1 = 0.05;
}
```

MPU6050 obtains the angle of inclination and angular velocity of the car through Kalman filter.

```
//in Balanced.cpp

void Mpu6050::DataProcessing()
{
    MPU6050.getMotion6(&ax, &ay, &az, &gx, &gy, &gz); // Data acquisition of MPU6050
    gyroscope and accelerometer
    kalmanfilter.Angletest(ax, ay, az, gx, gy, gz, dt, Q_angle, Q_gyro, R_angle, C_0, K1); //
    Obtaining Angle by Kalman Filter
}
```

Although it has been able to maintain balance under the control of the above vertical ring, due to the accuracy problem, the actual measured angle of the sensor always deviates from the angle of the vehicle model, so the vehicle model is not actually vertical to the ground, but there is an inclination angle, under the effect of gravity, the vehicle model will accelerate towards the inclined direction. So we need to introduce a speed loop to maintain the speed stability, so that the balanced car can stay still and move steadily

```
//in Balanced.cpp

void Balanced::PI_SpeedRing()
{
    double car_speed = (encoder_left_pulse_num_speed + encoder_right_pulse_num_speed) * 0.5;
    encoder_left_pulse_num_speed = 0;
    encoder_right_pulse_num_speed = 0;
    speed_filter = speed_filter_old * 0.7 + car_speed * 0.3;
    speed_filter_old = speed_filter;
    car_speed_integral += speed_filter;
    car_speed_integral += -setting_car_speed;
    car_speed_integral = constrain(car_speed_integral, -3000, 3000);

    speed_control_output = -kp_speed * speed_filter - ki_speed * car_speed_integral;
}
```

$\text{speed_control_output} = -kp_speed * \text{speed_filter} - ki_speed * \text{car_speed_integral};$

KP * speed: Since it is speed control, it is of course the speed is multiplied as a proportional parameter to maintain the speed in a stable state.

Ki * displacement: Ki is the integral link, and the velocity integral is the displacement, so the displacement control parameters are selected.

```
speed_filter = speed_filter_old * 0.7 + car_speed * 0.3
```

Carry out low-pass filtering to slow down the speed difference and disturb the upright.

```
speed_filter_old = speed_filter;  
car_speed_integral += speed_filter;  
car_speed_integral += -setting_car_speed;  
Integrating speed
```

```
car_speed_integral = constrain(car_speed_integral, -3000, 3000);
```

Limit the integral value to determine the maximum upper limit of the speed.

Finally, we will learn about the steering ring. The realization of the steering ring is mainly to obtain the data of z-axis gyroscope as the steering speed deviation for P control. The goal is to keep the steering speed as the set value.

```
//in Balanced.cpp  
  
void Balanced::PI_SteeringRing()  
{  
    rotation_control_output = setting_turn_speed + kd_turn * kalmanfilter.Gyro_z;////control  
    with Z-axis gyroscope  
}
```

Finally, we just need to add the control output quantity of the three ring together.

```
//in Balanced.cpp  
  
void Balanced::Total_Control()  
{  
    pwm_left = balance_control_output - speed_control_output -  
    rotation_control_output;////Superposition of Vertical Velocity Steering Ring  
    pwm_right = balance_control_output - speed_control_output +  
    rotation_control_output;////Superposition of Vertical Velocity Steering Ring  
  
    pwm_left = constrain(pwm_left, -255, 255);  
    pwm_right = constrain(pwm_right, -255, 255);  
  
    while(EXCESSIVE_ANGLE_TILT || PICKED_UP)  
    {  
        Mpu6050.DataProcessing();  
        Motor.Stop();  
    }  
  
    (pwm_left < 0) ? (Motor.Control(AIN1,1,PWMA_LEFT,-pwm_left)):  
        (Motor.Control(AIN1,0,PWMA_LEFT,pwm_left));  
  
    (pwm_right < 0) ? (Motor.Control(BIN1,1,PWMB_RIGHT,-pwm_right)):  
        (Motor.Control(BIN1,0,PWMB_RIGHT,pwm_right));  
}
```

This is to stop the balanced car when it is picked up or tilted too much.

Finally, we can modify the parameters according to the actual situation in order to get better balancing effect.

In the end, under the premise of balance, we only need to modify the target speed and the target steering speed to control the movement of the balanced car at will.

```
//in Balanced.cpp
```

```
while(EXCESSIVE_ANGLE_TILT || PICKED_UP)
{
    Mpu6050.DataProcessing();
    Motor.Stop();
}
```

```
//in Balanced.cpp
```

```
Balanced::Balanced()
{
    kp_balance = 55, kd_balance = 0.75;
    kp_speed = 10, ki_speed = 0.26;
    kp_turn = 2.5, kd_turn = 0.5;
}
```

```
//in Balanced.cpp
```

```
void Balanced::Motion_Control(Direction direction)
{
    switch(direction)
    {
        case STOP:
            Stop();break;
        case FORWARD:
            Forward(40);break;
        case BACK:
            Back(40);break;
        case LEFT:
            Left(50);break;
        case RIGHT:
            Right(50);break;
        default:
            Stop();break;
    }
}
```

```
void Balanced::Stop()
{
    setting_car_speed = 0;
    setting_turn_speed = 0;
}
```

```
void Balanced::Forward(int speed)
{
    setting_car_speed = speed;
    setting_turn_speed = 0;
}
```

```
void Balanced::Back(int speed)
{
    setting_car_speed = -speed;
    setting_turn_speed = 0;
}
```

```
void Balanced::Left(int speed)
{
    setting_car_speed = 0;
    setting_turn_speed = speed;
}
```

```
void Balanced::Right(int speed)
{
    setting_car_speed = 0;
    setting_turn_speed = -speed;
}
```

1.3 Upload Validation

After uploading the program, you will see that the car will cycle as follows: move forwards in balance for 5 seconds, move backwards in balance for 5 seconds, move to the left in balance for 5 seconds, move to the right in balance for 5 seconds and then stay static for 5 seconds.