**API for TD3 task code**

**1.Core Class API Details:**

**Motor Class:**

class Motor {

private:

PwmOut \_pwm; // PWM control signal

DigitalOut \_bip; // Direction control signal

public:

// Constructor: Initializes the PWM output pin and direction control pin

Motor(PinName pwmPin, PinName bipPin) : \_pwm(pwmPin), \_bip(bipPin) {}

// Initialize motor PWM frequency

void init(float pwmPeriod) {

\_bip = 1; // bipolar setting

\_pwm.period(pwmPeriod); // Set PWM frequency

}

// Set motor speed (speed range 0~1, representing duty cycle)

void run(float speed) {

\_pwm.write(speed);

}

};

**Controller Classes**

**PD Controller**

class PD\_controller {

// PD Controller Class (for error adjustment)

private:

double Kp, Kd; // Proportional and derivative gain

double last\_error; // Previous error

double last\_time; // Previous timestamp

public:

PD\_controller(double p, double d) : Kp(p), Kd(d), last\_error(0), last\_time(0) {}

// Compute PD control output (calculate adjustment value based on error)

double compute(double error) {

double current\_time = timer.read();

double dt = current\_time - last\_time;

last\_time = current\_time;

if (dt < 1e-6) dt = 1e-6; // Avoid division by zero

double D = (error - last\_error) / dt;

return Kp \* error + Kd \* D;

}

};

**PI Controller**

// PI Controller Class (for speed control)

class PI\_Controller {

private:

double Kp, Ki; // Proportional and integral gain

double integral; // Integral term

double last\_time; // Previous timestamp

public:

PI\_Controller(double p, double i) : Kp(p), Ki(i), integral(0), last\_time(0) {}

// Compute PI control output (calculate adjustment value based on speed error)

double compute(double error) {

double current\_time = timer.read();

double dt = current\_time - last\_time;

last\_time = current\_time;

integral += error \* dt;

return Kp \* error + Ki \* integral;

}

};

**2.Key control logic (sensor → motor control full process)**

void control\_motors() {

// Step 1: Read sensors and calculate error

double error = (-4 \* sensor1) + (-2 \* sensor2) + (0 \* sensor3)

+ (+2 \* sensor4) + (+4 \* sensor5);

// Step 2: PD correction (similar to servo control example)

double correction = PD\_sensor.compute(error);

// Step 3: PI speed adjustment (similar to PWM regulation)

float base\_speed = 0.38; // Base speed set to 38%

left\_speed = base\_speed + pi\_left.compute(...);

// Step 4: Final output (with safety constraints)

left\_motor.run(left\_speed);

}

**3**.**Troubleshooting Guide**

| **Phenomenon** | **Solution** |
| --- | --- |
| Car spins in place | 1. Swap left and right motor wiring 2. Swap encoder A/B phase |
| Sensor reading is 0 | Measure the VCC-GND voltage (should be 3.3V) |
| The motor makes high-frequency noise | Increase the PWM period in init():  motor1.init(0.001f) |

**4.complete example:**

#include "mbed.h"

#include "QEI.h"

#include "C12832.h"

Timer timer ;

QEI encoder2( PB\_8, PB\_9, NC, 256); //the original pin occupy the LCD pins

QEI encoder1(PB\_5, PB\_3, NC, 256);  // default by X2 encoding (multiply the 256 to 512 )

DigitalOut en(PA\_12);

DigitalOut dir1(PC\_10);

DigitalOut dir2(PC\_12);

C12832 lcd(D11, D13, D12, D7, D10);

AnalogIn sensor1(PC\_4);

AnalogIn sensor2(PB\_1);

AnalogIn sensor3(PC\_5);

AnalogIn sensor4(PC\_2);

AnalogIn sensor5(PC\_3); // sensor pins

class Motor {

 private:

    PwmOut \_pwm;

    DigitalOut \_bip;

 public:

    Motor(PinName pwmPin, PinName bipPin) : \_pwm(pwmPin), \_bip(bipPin) {}

    void init(float pwmPeriod) {

        \_bip = 1;

        \_pwm.period(pwmPeriod);

    }

    void run(float speed) {

        \_pwm.write(speed);

    }

};

Motor motor1(PB\_14, PB\_12);

Motor motor2(PA\_11, PB\_2);

double get\_sensor\_error(){

     double s1,s2,s3,s4,s5;   // left is s1 and s3 , right is s2 and s5

     s1=sensor1.read();

     s2=sensor2.read();

     s3=sensor3.read();

     s4=sensor4.read();

     s5=sensor5.read();

     double sum\_values=0;

    sum\_values= -4\*s3+-2\*s1+2\*s2+4\*s5;   // use different power factors;

    if (sum\_values !=0) {

        return sum\_values;

    } else {

        return 0;  // no white detected , error is 0

    } }

class PD\_controller{

    private:

    double Kp ,Kd;

    double last\_error;

    double last\_time;

    public:

    PD\_controller(double p,double d):Kp(p),Kd(d),last\_error(0),last\_time(0){}

    double compute(double error){

        double current\_time= timer.read();

        double dt=current\_time-last\_time;

        last\_time=current\_time;

        if(dt< 1e-6) dt=1e-6; //avoid divide 0;

        double r1=error;

        double D=(error-last\_error)/dt;

        return Kp \* error + Kd \* D;

    }

};

PD\_controller PD\_sensor(0.8,0.8);                 //PD control for sensor to track the line;

class PI\_Controller {

  private:

    double Kp, Ki;

    double integral;

    double last\_time;

  public:

    PI\_Controller(double p, double i) : Kp(p), Ki(i), integral(0), last\_time(0) {}

    double compute(double error) {

        double current\_time = timer.read();

        double dt = current\_time - last\_time;

        last\_time = current\_time;

        integral += error \* dt;

        return Kp \* error + Ki \* integral;

    }

};                                             //pi control

PI\_Controller pi\_left(1.0, 0.1);  // left motor's PI

PI\_Controller pi\_right(1.0, 0.1); // right motor's PI

double get\_motor1\_error(){

    float CPR=256;

    double last\_time=0;

    int prev\_ticks1 = 0;

    int current\_ticks1=0;

    double current\_time= timer.read();

    double dt=current\_time-last\_time;

    last\_time=current\_time;     // calculate dt (sample time)

    current\_ticks1 = encoder1.getPulses();

 int delta\_ticks1 = current\_ticks1 - prev\_ticks1;

  prev\_ticks1 = current\_ticks1; // renew the prev ticks

 float rpm1 = (delta\_ticks1 / (float) CPR) \* (60.0 / dt);

  float v1=3.14\*0.07\*rpm1/60;

  double correction = PD\_sensor.compute(get\_sensor\_error());    // control to adjust the motors' speed;

  double target\_speed1;

  double constant\_speed=3;   //3cm/s

  target\_speed1=constant\_speed+correction;

    float e1= target\_speed1-v1 ;                //calculate the error between the speed and target speed;

   return  e1;

}

double get\_motor2\_error(){

    float CPR=256;

    double last\_time=0;

    int prev\_ticks2 = 0;

    int current\_ticks2=0;

    double current\_time= timer.read();

    double dt=current\_time-last\_time;

    last\_time=current\_time;     // calculate dt (sample time)

    current\_ticks2 = encoder2.getPulses();

 int delta\_ticks2 = current\_ticks2 - prev\_ticks2;

  prev\_ticks2 = current\_ticks2; // renew the prev ticks

 float rpm2 = (delta\_ticks2 / (float) CPR) \* (60.0 / dt);

  float v2=3.14\*0.07\*rpm2/60;

  double correction = PD\_sensor.compute(get\_sensor\_error());    // control to adjust the motors' speed;

  double target\_speed2;

  double constant\_speed=3;   //3cm/s

  target\_speed2=constant\_speed+correction;

    float e2= target\_speed2-v2 ;                //calculate the error between the speed and target speed;

   return  e2;

}

double motor1\_speed(){

      double s1=0.38;

      double correction =pi\_left.compute(get\_motor1\_error());

       return  s1 +=correction;

}

double motor2\_speed(){

       double s2=0.38;

       double correction =pi\_right.compute(get\_motor2\_error());

       return  s2 +=correction;

}

void control\_motors()

{

    float speed1=motor1\_speed();

    float speed2=motor2\_speed();

    if(speed1>=0.5){

        speed1=0.5;

    }

    if(speed1<=0.1){

        speed1=0.1;

    }

    if(speed2>=0.5){

        speed2=0.5;

    }

    if(speed2<=0.1){

        speed2=0.1;

    }    //limit the speed of motor 1 and motor 2

    motor1.run(speed1);

    motor2.run(speed2);

}

int main(){

    motor1.init(0.0001f);

    motor2.init(0.0001f);// set the period of both motor.

    timer.start();

    while(1){

        control\_motors();

        wait\_ms(50); }}