# Full Control System

*Manual 4*
Here we build the **brain** of the robot — the complete control system design.

---

# Overall Code Architecture (High Level)

A self-balancing robot runs **one tight control loop**:

```
1. Read gyro + accelerometer
2. Compute tilt angle (complementary filter)
3. Run PID controller
4. Convert PID output → motor speed + direction
5. Send PWM to motors
```

This loop must run at:

```
100 – 200 Hz   (every 5–10 ms)
```

---

# Final Complementary Filter Implementation

## Step 1 — Compute raw angle from accelerometer

```
accAngle = atan2(Ax, Az) * RAD_TO_DEG;
```

## Step 2 — Integrate gyro

```
gyroRate = Gy;      // deg/s
gyroAngle += gyroRate * dt;
```

## Step 3 — Combine with complementary filter

```
angle = 0.98 * (angle + gyroRate * dt) + 0.02 * accAngle;
```

Equivalent $\alpha = 0.98$.

**Why 0.98?**

- gyro handles 98% of fast motion
- accelerometer slowly corrects drift (2%)

---

# Time Step (dt) Calculation

Accurate dt from micros():

```
unsigned long now = micros();
dt = (now - lastTime) / 1000000.0;
lastTime = now;
```

If dt is wrong → robot will oscillate or fall.

---

# PID Controller

PID computes the correction needed to stay upright.

```
error = targetAngle - angle;
```

For balancing:

```
targetAngle = 0°
```

### PID Equation

```
P = Kp * error
I = I + Ki * error * dt
D = Kd * (error - previousError) / dt

PIDoutput = P + I + D
```

### Typical PID values (starting point)

```
Kp = 20 – 40
Kd = 0.8 – 1.5
Ki = 0.5 – 1
```

We will tune them in Manual 5.

---

# Converting PID Output to Motor Speed

PID output is in “motor command units”.

We must map it to PWM:

```
int speed = constrain(abs(PIDoutput), 0, 255);
```

Direction:

```
if (PIDoutput > 0) → move forward
if (PIDoutput < 0) → move backward
```

---

# Motor Control with L293D

L293D has:

- 2 direction pins per motor
- 1 PWM pin per motor

Example wiring:

```
Motor A:
IN1 = 5
IN2 = 6
EN1 = 9 (PWM)

Motor B:
IN3 = 10
IN4 = 11
EN2 = 3 (PWM)
```

### Function to drive motors:

```
void moveMotor(int pid)
{
    int speed = constrain(abs(pid), 0, 255);

    if (pid > 0) {
        digitalWrite(IN1, HIGH);
        digitalWrite(IN2, LOW);
        analogWrite(EN1, speed);

        digitalWrite(IN3, HIGH);
        digitalWrite(IN4, LOW);
        analogWrite(EN2, speed);
    }
    else {
        digitalWrite(IN1, LOW);
        digitalWrite(IN2, HIGH);
        analogWrite(EN1, speed);

        digitalWrite(IN3, LOW);
        digitalWrite(IN4, HIGH);
        analogWrite(EN2, speed);
    }
}
```

---

# Complete Control Loop (Concept)

Your loop() should look like:

```
loop() {
    readIMU();                    // Ax, Az, Gy
    computeAngle();               // complementary filter
    computePID();                 // error → PIDoutput
    moveMotor(PIDoutput);         // control motors
}
```

Run as fast as possible → **no delay() anywhere**
If you use delay, robot will fall immediately.

---

# Filtering for Stability

### 1. MPU6050 DLPF (Always use 21 Hz)

Internal filter that removes vibration noise.

### 2. Moving average filter (optional)

Smooths accelerometer values.

### 3. Complementary filter (main)

Removes drift and noise.

All three combined → rock solid angle output.

---

# How Everything Works Together

When robot starts to fall forward:

- angle increases
- PID output becomes positive
- motors drive forward
- robot "catches" itself

When falling backward:

- PID output negative
- motors drive backward

Your control loop must be:

```
FAST + ACCURATE + CONTINUOUS
```

If loop slows → robot falls.

---

# Summary

You now know:

- ✔ How angle is calculated
- ✔ Complementary filter implementation
- ✔ Timing (dt)
- ✔ PID control math
- ✔ Motor speed mapping
- ✔ Direction logic
- ✔ L293D control
- ✔ Main control loop architecture
- ✔ Filters needed for stability

This manual gives you the **complete working brain** of a self-balancing robot.

---