

MiniKame Quadruped Robot - A Desktop Robot Building Tutorial from Scratch

Documentation Introduction

This document is a complete tutorial for building a MiniKame quadruped robot from scratch, designed to help robotics enthusiasts, university students, and maker educators create their own desktop quadruped robot. The tutorial covers everything from design philosophy, hardware assembly, principle analysis to software programming, with both easy-to-understand explanations and in-depth mathematical derivations.

Target Audience:

- Robotics enthusiasts with basic Arduino experience
- University students working on course projects/graduation designs
- Maker educators conducting STEAM teaching
- Technical enthusiasts interested in quadruped robot principles

Prerequisites:

- Basic C/C++ programming knowledge
- Simple electronic circuit understanding
- Hands-on assembly ability

Learning Outcomes:

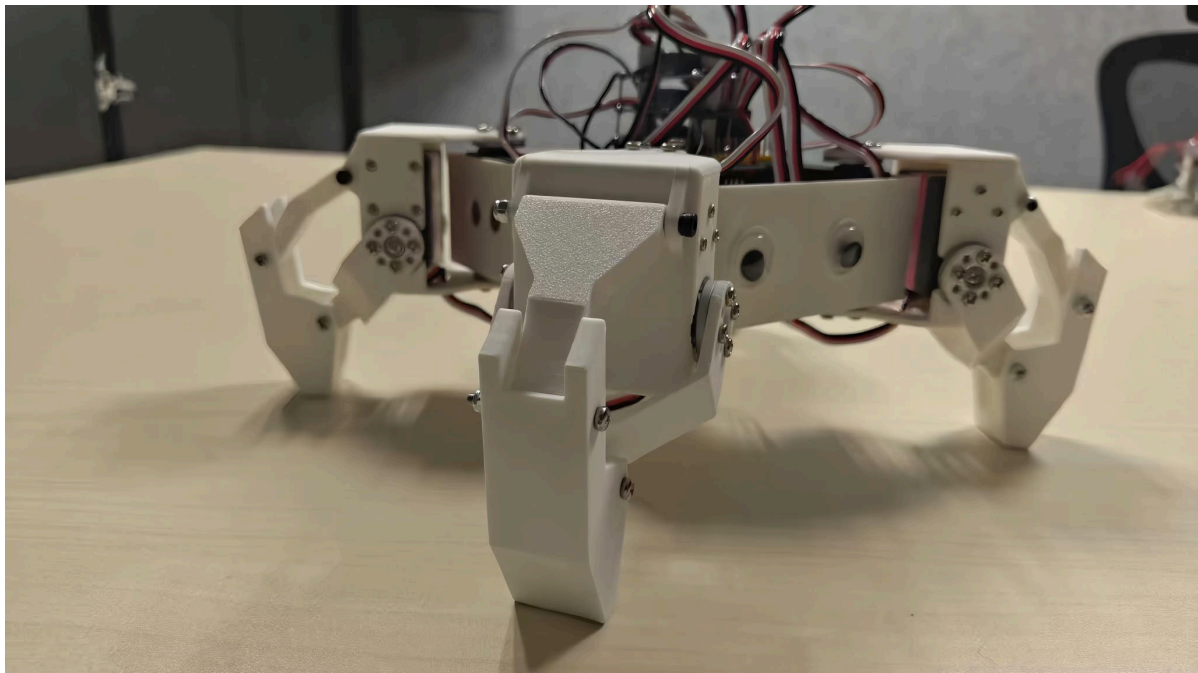
Upon completing this tutorial, you will be able to:

1. Understand the kinematics principles of quadruped robots
2. Master the application of sine oscillators in robot control
3. Independently complete the assembly, debugging, and programming of MiniKame
4. Control the robot remotely through a web interface
5. Have the capability for secondary development and innovation based on this project

Part 1: Getting to Know MiniKame

1.1 What is MiniKame?

MiniKame is an 8-degree-of-freedom (8-DOF) miniature quadruped robot based on the Arduino UNO Q (with built-in WiFi). The name "Kame" comes from the Japanese word "亀" (turtle), symbolizing stability and vitality like a turtle. This robot's design is inspired by classic spider robot forms, but with a more compact and cute appearance, perfect for desktop demonstrations and experiments.



Analogy: If we compare the robot to a small animal:

- **Arduino** is its brain, responsible for thinking and processing information
- **8 servos** are its muscles, enabling movement
- **WiFi module** is its senses, receiving external commands
- **Battery** is its energy source, providing power

Core Specifications:

- **Dimensions:** 150mm length × 120mm width × 100mm height (slightly larger than an adult's palm)
- **Weight:** Approximately 300g (about the weight of an apple)
- **DOF:** 8 (each leg has 2 joints, like human thighs and calves)
- **Control Core:** Arduino UNO Q (with built-in WiFi)
- **Actuation:** 8 35Kg.cm servos
- **Power:** 12V Lithium battery
- **Control Method:** Browser Web Interface (mobile/computer both available)

1.2 Design Philosophy: Simple but Not Simplistic

Unlike many complex quadruped robot projects, MiniKame's design philosophy is "**simple but not simplistic**". It doesn't rely on expensive sensors or complex control algorithms, but instead uses an elegant mathematical model—the sine oscillator—to simulate the biological Central Pattern Generator (CPG), achieving natural and fluid movement.

What is CPG?

Imagine when you walk, you don't consciously think "lift left foot, then step with right foot..."—these actions are automatically completed by your spinal cord. This neural system that automatically generates rhythmic movements is the CPG. MiniKame uses mathematical formulas to simulate this mechanism.

Three-Layer Separation Architecture:

Web Frontend (Browser) ↔ Python Middleware ↔ Arduino Firmware
(HTML/CSS/JS) (Command Routing) (Servo Control)

This architecture is like a restaurant:

- **Web Frontend** is the menu and ordering interface (what customers see and operate)
- **Python Middleware** is the waiter (receiving orders, conveying to kitchen)
- **Arduino Firmware** is the chef (actually preparing the dishes)

Advantages of this architecture:

- **Human-Machine Separation:** Control interface decoupled from underlying hardware, like the menu being separate from the kitchen
- **Cross-Platform:** Any device with a browser can control, like any customer can view the menu
- **Real-Time:** Socket.IO enables bidirectional communication, like a waiter keeping customers updated on order status
- **Easy to Extend:** New features can be added without affecting the bottom layer, like adding new dishes to the menu without redesigning the kitchen

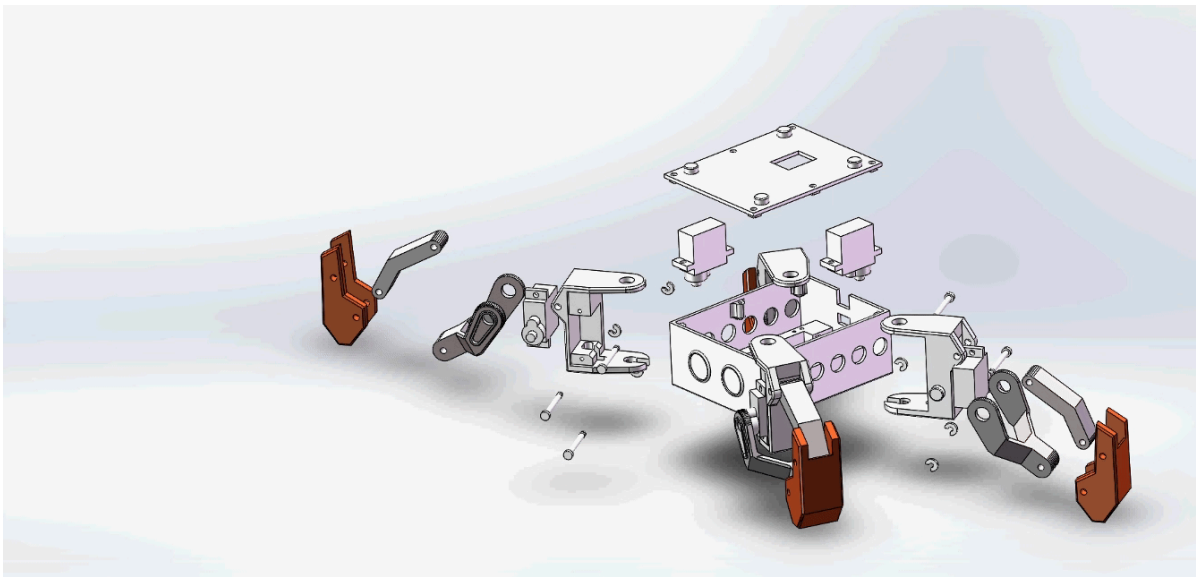
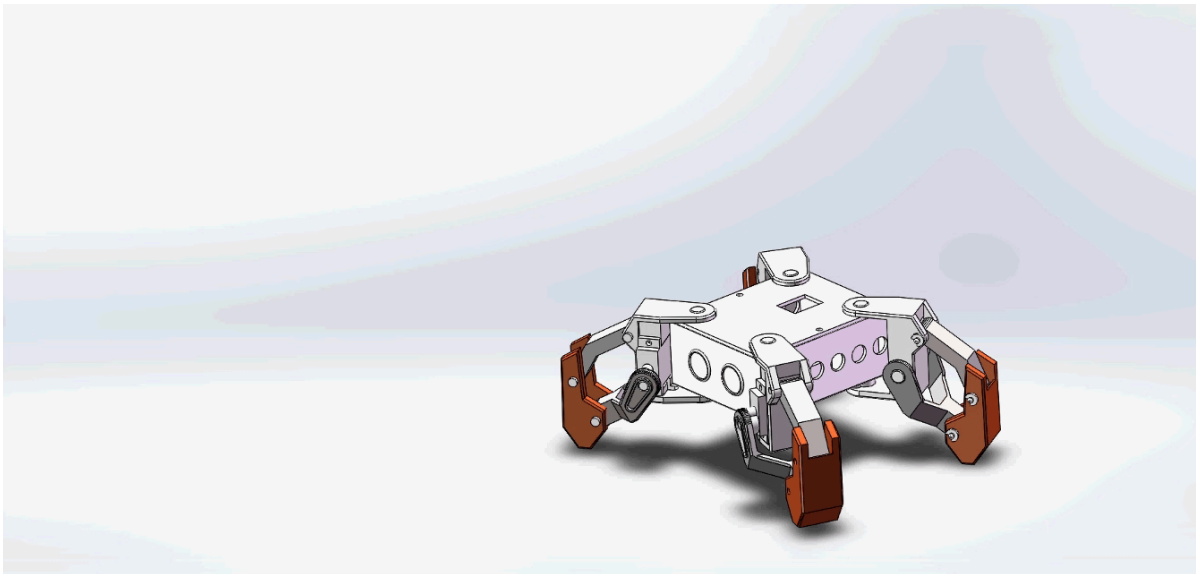
1.3 Application Scenarios

- **Robotics Enthusiast Entry Project:** Learn servo control, gait generation principles, experience the complete building process
- **University Course Projects/Graduation Designs:** Can serve as practical projects for mechatronics, embedded systems, web control courses
- **ROS/AI Frontend Platform:** Can add sensors for obstacle avoidance, line following, or target tracking
- **Maker Education/STEAM Teaching:** As a demonstration project to stimulate students' interest in robotics technology
- **Remote Laboratory:** Control physical robots remotely through web interface

1.4 Project Features

- **Pure Mathematics-Driven Motion Control:** No reliance on complex sensor feedback, generates natural gaits through sine oscillators
 - **Parametric Design:** All actions defined by four parameters: amplitude, period, phase, offset
 - **Rich Action Library:** 11 preset actions (walk, turn, dance, moonwalk, etc.)
 - **Web Remote Control:** Control through browser, no app installation required
 - **Real-Time Parameter Adjustment:** Steps (1-20) and speed (200-3000ms) sliders for real-time adjustment
 - **Cyberpunk UI:** Dark theme responsive interface, adaptive for mobile portrait mode
 - **Complete Calibration Mechanism:** Compensate mechanical assembly errors through trim and reverse parameters
-

Part 2: Mechanical Structure Design



2.1 Leg Joint Design

MiniKame uses a typical quadruped robot leg structure, with each leg consisting of two joints, totaling 8 degrees of freedom. We can imagine each leg like a human leg:

Joint Name	Servo ID	Function Description	Human Analogy
Left Front Thigh	0	Controls left front leg forward/backward swing	Hip Joint
Right Front Thigh	1	Controls right front leg forward/backward swing	Hip Joint
Left Front Calf	2	Controls left front leg up/down lifting	Knee Joint
Right Front Calf	3	Controls right front leg up/down lifting	Knee Joint
Left Rear Thigh	4	Controls left rear leg forward/backward swing	Hip Joint

Joint Name	Servo ID	Function Description	Human Analogy
Right Rear Thigh	5	Controls right rear leg forward/backward swing	Hip Joint
Left Rear Calf	6	Controls left rear leg up/down lifting	Knee Joint
Right Rear Calf	7	Controls right rear leg up/down lifting	Knee Joint

2.1.1 Biological Insights in Joint Configuration

Observing four-legged animals in nature, their leg structures follow a common principle: **thighs generate forward power, calves adjust ground clearance.**

Real-life Examples:

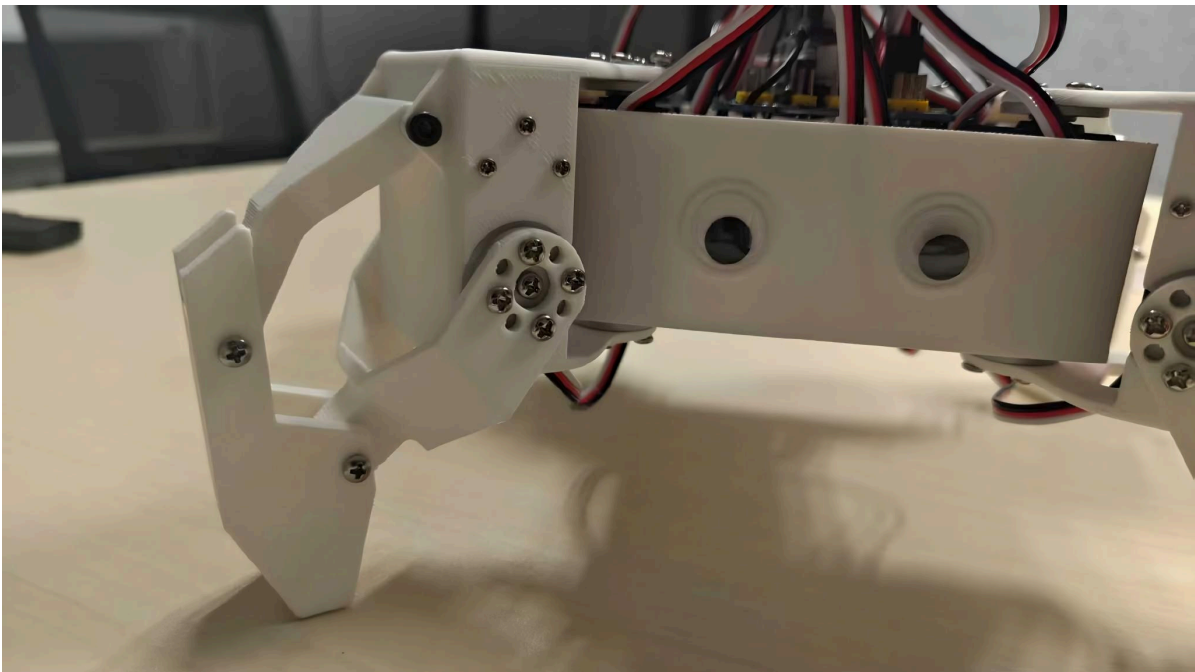
- **Thigh (Hip Joint):** Like rowing a boat, arms swinging back and forth generate forward momentum
- **Calf (Knee Joint):** Like lifting your foot when stepping over a threshold to avoid tripping

2.1.2 Unified Coordinate System Convention

For easy programming and control, we uniformly stipulate:

- **Increasing thigh angle** → Leg swings forward
- **Decreasing thigh angle** → Leg swings backward
- **Increasing calf angle** → Leg lowers (body lowers)
- **Decreasing calf angle** → Leg lifts (body raises)

Analogy: This is like agreeing that "nodding" means "yes" and "shaking head" means "no". If someone (a servo) is installed backwards and nodding becomes shaking head, we need a "translator"—this is what the `reverse[]` array does later.



2.2 Complete Assembly Guide

Assembly Steps Overview:

Step 1: Servo Initialization

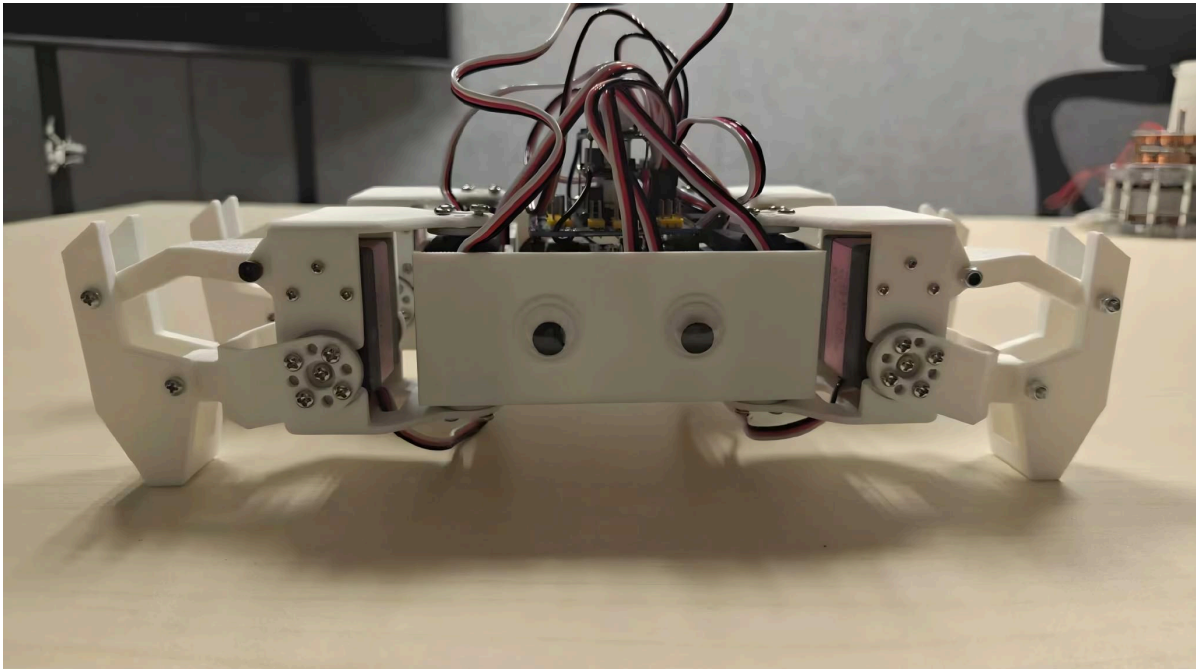
Before installation, all servos need to be set to 90 degrees (neutral position). This is like setting all clock hands to 12 o'clock before installation.

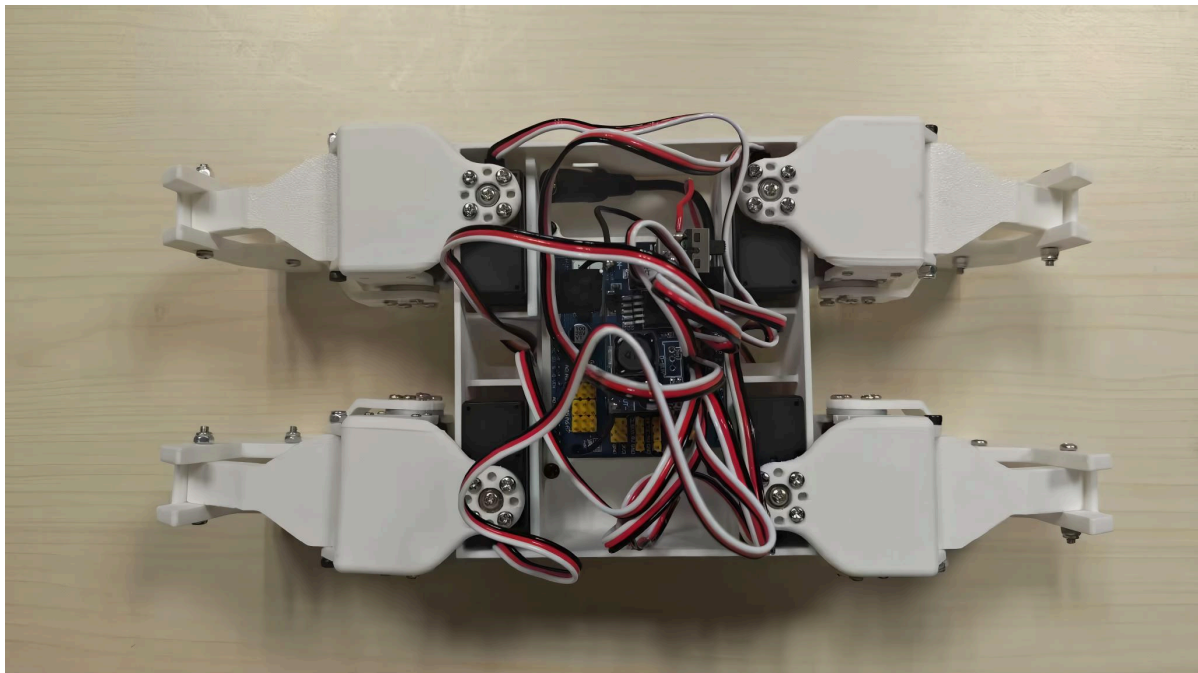
```
// Servo zeroing test program
#include <Servo.h>

Servo testServo;
int servoPins[] = {4,5,6,7,8,9,10,11};

void setup() {
    for(int i=0; i<8; i++) {
        testServo.attach(servoPins[i]);
        testServo.write(90);
        delay(500); // Give servo time to reach 90 degrees
    }
}

void loop() {
    // Stay still
}
```





Why zero before installation?

If servos aren't zeroed before installing mechanical parts, it's like installing clock hands when they're not at 12 o'clock—all times will be wrong. This can lead to:

- 1. Asymmetrical leg positions after installation
- 2. Servos stressed at extreme positions, potentially damaging gears
- 3. Extremely difficult calibration

Step 2: Body Frame Assembly

The body frame consists of upper and lower 3D printed plates connected, housing the Arduino board and battery in between. Servos are mounted on both sides of the body, connected to leg components through U-shaped brackets.

Step 3: Leg Assembly

Each leg consists of two servos (thigh and calf) and two 3D printed connecting rods. Assembly notes:

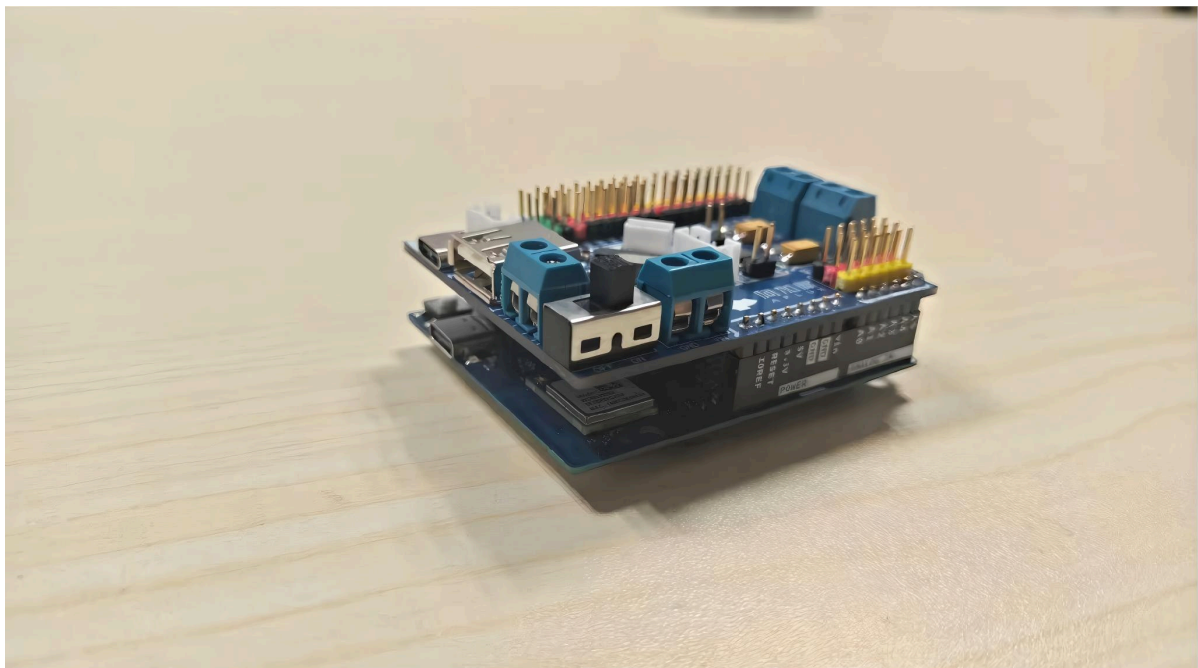
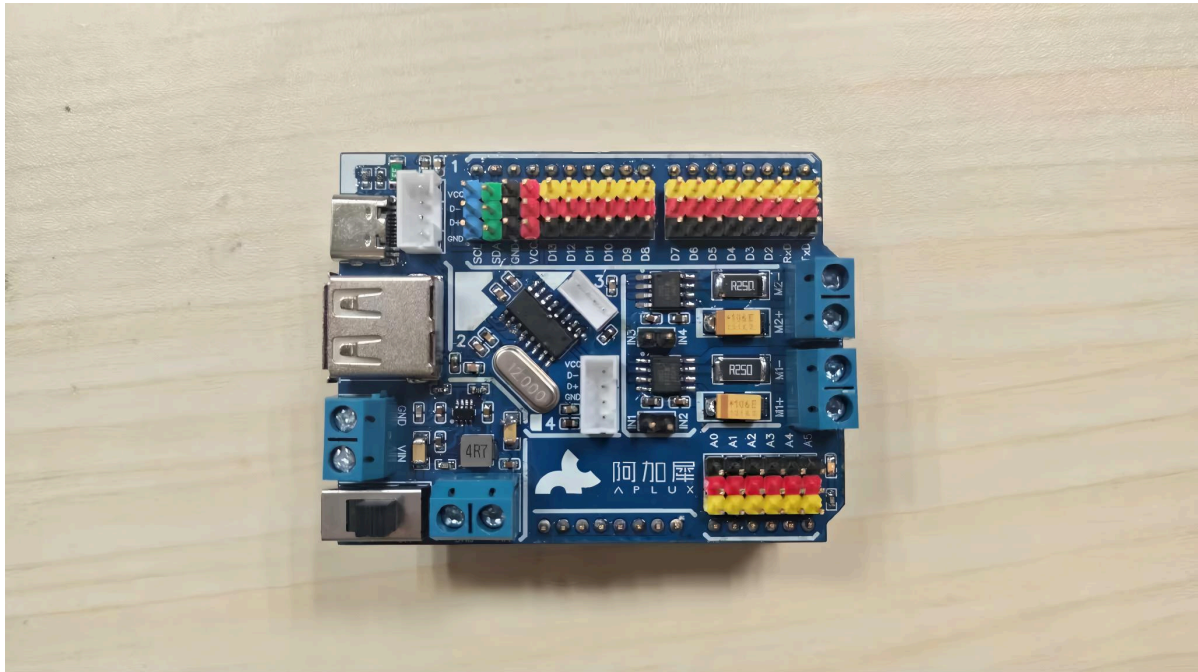
- Servo mounting direction must correspond to the `reverse[]` settings in code
- Connecting rod length directly affects movement range
- All screws should be tightened firmly but not excessively, avoiding damage to plastic parts

Part 3: Hardware System Design

3.1 Core Hardware List

Component	Model	Quantity	Description
Main Controller	Arduino UNO Q	1	Robot "brain"
Servo	Servo	8	Micro servo motors
Battery	12V Lithium Battery	1	Powers servos & board

Component	Model	Quantity	Description
Expansion Board	Custom	1	Servo power distribution and signal distribution, integrated power supply



3.2 Power System Design

3.2.1 Power Requirements Analysis

MiniKame's power system needs to meet two requirements simultaneously:

1. **Servo Power Supply:** When 8 servos work simultaneously, current requirements are:
 - Idle current: ~10mA each, total 80mA (equivalent to a small LED)
 - Normal operation: 100-200mA each, total 0.8-1.6A (equivalent to charging a phone)
 - Stall peak: up to 500-800mA each, total 4-6.4A (equivalent to a hair dryer!)

2. **Main Controller Power:** Arduino UNO Q requires 5V or 7-12V input, current ~50mA (equivalent to a small light bulb)

3.2.2 Power Solution Comparison

Solution	Advantages	Disadvantages	Application Scenario
2×18650 Li-ion in series	Large capacity, long endurance	Needs step-down to 5V, bulky	Long demonstrations
RC 2S LiPo	Strong discharge capability	Needs voltage regulation	High performance needs
USB Power Bank	Convenient, safe	5V may be unstable	Indoor demonstrations

Recommended Solution: Use 12V lithium battery, step down to 5V via voltage regulator module for servos, while also supplying Arduino's Vin pin directly.

3.2.3 Key Power Design Points

1. Current Capability:

Servos draw high current at startup, like a car consuming most fuel when starting. Power supply needs to provide peak current above 3A.

2. Voltage Regulation Accuracy:

Voltage fluctuation shouldn't exceed ±0.3V, otherwise servos may jitter. This is like people feeling dizzy with unstable blood pressure.

3. Filter Capacitor:

Connect a large capacitor (1000µF+) in parallel at servo power input to absorb transient currents. This is like adding a large water tank to a water system—when a large amount of water is suddenly needed, the tank provides buffer.

4. Power Separation:

It's recommended to separate servo power and Arduino power:

- Servos: 12V battery + 5V step-down module
- Arduino: Take power from servo supply (if step-down module has sufficient capacity)

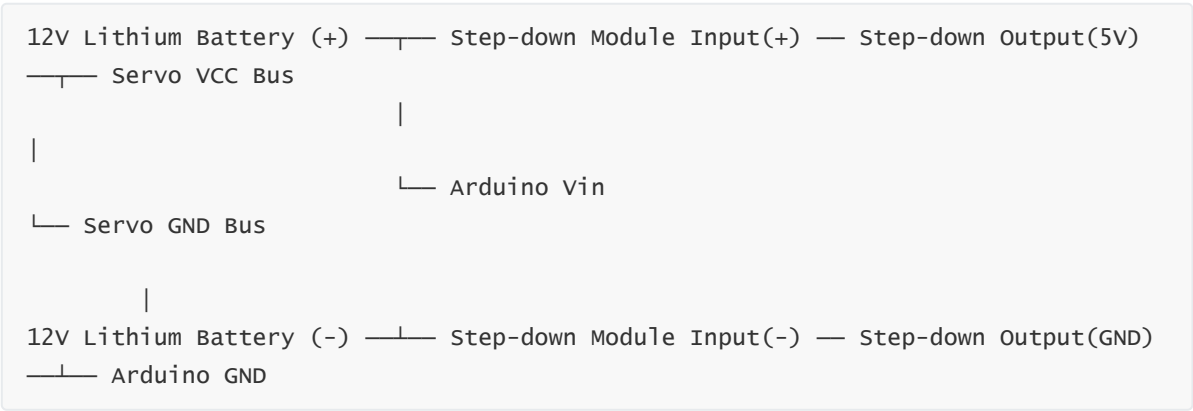
3.3 Circuit Connection Diagram

3.3.1 Servo Pin Assignment

Servo ID	Function	Arduino Pin	Wire Color (SG90 Standard)
S0	Left Front Thigh	D4	Orange (Signal), Red (VCC), Brown (GND)
S1	Right Front Thigh	D5	Orange (Signal), Red (VCC), Brown (GND)
S2	Left Front Calf	D6	Orange (Signal), Red (VCC), Brown (GND)
S3	Right Front Calf	D7	Orange (Signal), Red (VCC), Brown (GND)

Servo ID	Function	Arduino Pin	Wire Color (SG90 Standard)
S4	Left Rear Thigh	D8	Orange (Signal), Red (VCC), Brown (GND)
S5	Right Rear Thigh	D9	Orange (Signal), Red (VCC), Brown (GND)
S6	Left Rear Calf	D10	Orange (Signal), Red (VCC), Brown (GND)
S7	Right Rear Calf	D11	Orange (Signal), Red (VCC), Brown (GND)

3.3.2 Power Connection



3.3.3 Key Considerations

- Common Ground:** All power GND must be connected together, otherwise control signals can't form a complete circuit. This is like two people needing a common language to communicate.
- Signal Wires:** Servo signal wires connect directly to Arduino digital pins, no pull-up resistors needed.
- Large Capacitor:** Connect 1000 μ F electrolytic capacitor between servo power positive and negative (positive to VCC, negative to GND).
- Wire Gauge:** Servo power wires should use 22AWG or thicker, like thicker pipes allowing more water flow.

Part 4: From Biological Inspiration to Mathematical Modeling

4.1 Biological Central Pattern Generator (CPG)

In nature, quadruped walking appears simple but is extremely complex. When we watch a cat strolling leisurely, its four legs coordinate so naturally and smoothly.

An Interesting Experiment:

Scientists discovered that even when cutting spinal cord connection to the brain, animals can still produce rhythmic walking movements. This shows that basic walking patterns aren't directly controlled by the brain, but autonomously generated by a neural circuit in the spinal cord—the **Central Pattern Generator (CPG)**.

Simple Analogy for CPG:

- CPG is like an automatic metronome that keeps beat without a conductor

- The brain just needs to say "start walking" or "walk faster," and CPG automatically coordinates all limbs
- This is why we can think about other things while walking, without consciously controlling each step

CPG characteristics:

- **Autonomous Rhythmicity:** Generates periodic signals without external input
- **Adjustability:** Frequency and amplitude can be changed with simple commands
- **Coordination:** Multiple CPGs coordinate through phase coupling

4.2 Mathematical Model Construction

MiniKame's design philosophy is to use mathematical models to simulate biological CPG working principles. We use a **sine oscillator network** to let the robot autonomously generate rhythmic movements.

What is a Sine Wave?

Imagine swinging on a swing:

- From the highest point to the lowest, then to the highest on the other side—this is one period
- The higher you swing, the larger the amplitude
- The faster you swing, the shorter the period
- Two swings swinging together, one ahead of the other—this is phase difference
- The lowest point might not be exactly in the middle—this is offset

4.2.1 Mathematical Definition of a Single Oscillator

A sine oscillator's output can be expressed as:

$$\text{output}(t) = A \times \sin(2\pi \times t / T + \phi) + O$$

Let's explain each parameter with real-life examples:

- **A (Amplitude):** Like how high the swing goes. Larger A means larger leg swing range, longer stride.
- **T (Period):** Like how long one complete swing cycle takes. Smaller T means faster movement; larger T means slower movement.
- **ϕ (Phase):** Like the start time difference between two swings. If two swings start simultaneously, phase difference is 0°; if one starts while the other is already mid-swing, there's phase difference.
- **O (Offset):** Like the swing's pivot point might not be exactly centered. Offset determines the leg's resting position.

4.2.2 Oscillator Network Coupling

8 oscillators couple through phase relationships, forming a CPG network:

```
Leg1: output1(t) = A1 × sin(ωt + φ1) + O1
Leg2: output2(t) = A2 × sin(ωt + φ2) + O2
...
Leg8: output8(t) = A8 × sin(ωt + φ8) + O8
```

These oscillators share the same angular frequency ω (determined by period T), but coordinate through phase ϕ . This is like an orchestra:

- All musicians play at the same rhythm (ω)
- But different instruments enter at different times (ϕ)
- Some instruments are louder (larger A), some softer (smaller A)
- Some instruments have higher pitch (larger O), some lower (smaller O)

4.3 From Model to Code Mapping

Now let's see how this mathematical model is implemented in code.

4.3.1 Oscillator Class Implementation

The Oscillator class defined in `Octosnake.h`:

```
class Oscillator{
private:
    int _period;           // Period T: time for one complete oscillation
                           (milliseconds)
    int _amplitude;        // Amplitude A: swing range (degrees)
    int _phase;            // Phase φ: starting position in the period (degrees)
    int _offset;           // Offset O: movement center point (degrees)
    float _output;         // Currently calculated angle
    bool _stop;            // Stop flag
    unsigned long _ref_time; // Reference time: when oscillator started
    float _delta_time;     // Current time offset from reference

public:
    float refresh();       // calculate current output
    void setPeriod(int period);
    void setAmplitude(int amplitude);
    void setPhase(int phase);
    void setOffset(int offset);
    // ... other methods
};
```

Each member variable directly corresponds to a parameter in the mathematical model, implementing clear mapping from math to code.

4.3.2 refresh() Function Mathematical Implementation

```
float Oscillator::refresh(){
    if (!_stop){
        // Calculate current position in the period (0 to T-1)
        _delta_time = (millis() - _ref_time) % _period;

        // Sine wave formula:
```

```

        // output = A * sin(2π * t/T + φ) + O
        _output = (float)_amplitude *
                  sin(time_to_radians(_delta_time) +
                     degrees_to_radians(_phase)) +
                  _offset;
    }
    return _output;
}

```

Line-by-line explanation with real-life analogies:

1. `if (!_stop)`: Only calculate when oscillator is active, like only paying attention to the swing's position when it's actually swinging.
2. `(millis() - _ref_time) % _period`:
 - `millis()` is current time
 - `_ref_time` is when swinging started
 - Difference is how long we've been swinging
 - Modulo period means "no matter how many cycles completed, we only care where we are in the current cycle"
3. `time_to_radians(...)`: Convert time to radians. Why radians? Because trigonometric functions in math use radians, like thermometers using Celsius.
4. `degrees_to_radians(_phase)`: Convert phase from degrees to radians.
5. **Sine wave calculation**: This is the swing's motion formula; the computer calculates where the swing should be at this moment.

4.4 Physical Meaning of Four Core Parameters

4.4.1 Amplitude A — Determines Movement Range

Mathematical Definition: Amplitude is the maximum deviation of a sine wave, determining vertical stretch degree.

Physical Meaning: In robot motion, amplitude determines the servo's maximum swing angle. Larger amplitude means:

- Larger thigh amplitude → longer stride, faster forward speed
- Larger calf amplitude → higher leg lift, less ground friction

Analogy: Like walking—striding with large leg swings (large amplitude) vs. small steps with small swings (small amplitude).

Code Representation:

```

int x_amp = 15; // Thigh swings 15 degrees
int z_amp = 20; // Calf lifts 20 degrees

```

4.4.2 Period T — Determines Movement Speed

Mathematical Definition: Period is time for one complete oscillation, determining waveform compression/expansion on the time axis.

Physical Meaning: Period determines the robot's movement rhythm. Shorter period = faster movement; longer period = slower movement.

Analogy: Like music—allegro (short period) vs. adagio (long period).

Code Representation:

```
int period[] = {T, T, T/2, T/2, T, T, T/2, T/2};
```

Here's an interesting detail: calf period is half of thigh period. This means calves move twice as fast as thighs. Why? Because lifting needs to be quick (fast lift to avoid ground drag), while forward/backward swinging can be relatively slower.

4.4.3 Phase ϕ — Determines Movement Coordination

Mathematical Definition: Phase represents the sine wave's time-axis offset, determining waveform starting position.

Physical Meaning: Phase is the most critical parameter in gait generation. It describes time offset relationships between multiple oscillators.

Analogy:

- If four legs are dancers, phase is the choreography determining when each jumps and lands
- If all dancers jump simultaneously, phase difference is 0°
- If they jump sequentially like a wave, there are different phase differences

Code Representation:

```
int phase[] = {90, 90, 270, 90, 270, 270, 90, 270};
```

Important Phase Relationships:

- **0° difference:** Movements completely synchronized
- **90° difference:** One peaks while the other crosses zero
- **180° difference:** Movements completely opposite (one forward while other backward)

4.4.4 Offset O — Determines Neutral Position

Mathematical Definition: Offset is vertical translation of the entire waveform, determining the sine wave's center line.

Physical Meaning: Offset determines the servo's resting position. In `home()`, different legs have different offsets because the robot's body needs to stay level while leg installation angles naturally vary.

Analogy: Like chair legs on uneven ground—you need to put shims under shorter legs (offset) to keep the chair balanced.

Code Representation:

```
int offset[] = {
    90 + ap - front_x, // Left front thigh: needs more forward extension
    90 - ap + front_x, // Right front thigh: also needs forward extension
    90 - hi,           // Left front calf: needs lifting
    90 + hi,           // Right front calf: needs lowering
    // ...
};
```

Part 5: Firmware Detailed Explanation

5.1 MiniKame Class Architecture

```
class MiniKame{
private:
    oscillator oscillator[8]; // 8 oscillator instances
    servo servo[8];          // 8 servo objects
    int board_pins[8];       // Servo pin mapping
    int trim[8];              // Calibration values, compensating mechanical
errors
    bool reverse[8];         // Direction reversal flags
    unsigned long _init_time; // Initialization timestamp
    unsigned long _final_time; // End timestamp
    float _servo_position[8]; // Current servo position record

public:
    void init();
    void walk(float steps, int period);
    void turnL(float steps, int period);
    void turnR(float steps, int period);
    void dance(float steps, int period);
    void moonwalkL(float steps, int period);
    void pushup(float steps, int period);
    void hello();
    void home();
    void zero();
    // ... other methods
};
```

Roles of Each Class Component:

- **oscillator[8]**: 8 mathematical calculators, each responsible for calculating where one leg should be
- **servo[8]**: 8 physical actuators, each controlling one servo
- **board_pins[8]**: Wiring map, telling the program which pin each servo connects to
- **trim[8]**: Fine-tuning knobs, compensating assembly errors
- **reverse[8]**: Direction conversion switches, solving servo orientation issues

5.2 Initialization Function init()

```
void Minikame::init() {  
    // 1. Configure pin mapping (telling program where each servo connects)  
    board_pins[0] = 4; // Left front thigh connected to D4  
    board_pins[1] = 5; // Right front thigh connected to D5  
    board_pins[2] = 6; // Left front calf connected to D6  
    board_pins[3] = 7; // Right front calf connected to D7  
    board_pins[4] = 8; // Left rear thigh connected to D8  
    board_pins[5] = 9; // Right rear thigh connected to D9  
    board_pins[6] = 10; // Left rear calf connected to D10  
    board_pins[7] = 11; // Right rear calf connected to D11  
  
    // 2. Set calibration values (adjust based on actual assembly)  
    trim[0] = -5; // Left front thigh needs -5° counterclockwise adjustment  
    trim[1] = -5; // Right front thigh needs -5° counterclockwise adjustment  
    trim[2] = -20; // Left front calf needs -20° counterclockwise adjustment  
    trim[3] = 10; // Right front calf needs +10° clockwise adjustment  
    trim[4] = -3; // Left rear thigh needs -3° counterclockwise adjustment  
    trim[5] = -3; // Right rear thigh needs -3° counterclockwise adjustment  
    trim[6] = -15; // Left rear calf needs -15° counterclockwise adjustment  
    trim[7] = -13; // Right rear calf needs -13° counterclockwise adjustment  
  
    // 3. Default no direction reversal  
    for (int i = 0; i < 8; i++) reverse[i] = false;  
  
    // 4. Start oscillators, attach servos  
    for (int i = 0; i < 8; i++) {  
        oscillator[i].start(); // Start mathematical calculators  
        servo[i].attach(board_pins[i]); // Connect physical servos  
        delay(50); // Connect one by one to avoid current surge  
    }  
  
    // 5. Initial position zeroing  
    zero();  
}
```

Initialization Sequence Logic:

1. **Configure mapping first:** Ensure physical connections match code (like confirming sockets are plugged correctly)
2. **Set calibration values:** Prepare for subsequent movement (like tuning instruments first)
3. **Start one by one:** Avoid current surge from starting all 8 servos simultaneously (like not turning on all appliances at once)
4. **Zero last:** Ensure robot initial posture determined (like setting pointers to zero first)

5.3 Servo Control Bottom Layer Implementation

```
void Minikame::setServo(int id, float target) {  
    float final_angle = target + trim[id]; // First add calibration value  
  
    if (reverse[id]) {  
        // If reversal needed, subtract target angle from 180 degrees
```

```

        final_angle = 180 - final_angle;
    }

    // Convert angle to pulse width, send to servo
    servo[id].writeMicroseconds(angToUsec(final_angle));

    // Record current position
    _servo_position[id] = target;
}

int Minikame::angToUsec(float value) {
    // Linear mapping: 0° -> 544µs, 180° -> 2400µs
    // Like temperature: 0°C corresponds to 32°F, 100°C to 212°F
    return value / 180 * (MAX_PULSE_WIDTH - MIN_PULSE_WIDTH) + MIN_PULSE_WIDTH;
}

```

The Elegance of This Function:

1. **Calibration Value Addition:** Adding `trim[id]` before output is like putting small shims under each leg to compensate mechanical assembly errors
2. **Direction Handling:** Implementing reversal through `180 - angle`
 - Mathematical principle: If original direction increases clockwise, reversal increases counterclockwise
 - Example: Original 90° stays 90°, 120° becomes 60°, implementing direction reversal
 - This is like a translator turning "nodding" into "shaking head"
3. **Angle to Pulse Width Conversion:** Servos don't understand "angles," only "how long the high level lasts." This function is like a translator, converting angles into a language servos understand.

5.4 Unified Gait Execution Interface

```

void Minikame::execute(float steps, int period[8], int amplitude[8],
                       int offset[8], int phase[8]) {

    // Step 1: Configure 8 oscillators
    for (int i = 0; i < 8; i++) {
        oscillator[i].setPeriod(period[i]);
        oscillator[i].setAmplitude(amplitude[i]);
        oscillator[i].setPhase(phase[i]);
        oscillator[i].setOffset(offset[i]);
    }

    // Step 2: Synchronize all oscillator start times
    unsigned long global_time = millis();
    for (int i = 0; i < 8; i++)
        oscillator[i].setTime(global_time);

    // Step 3: Execute specified number of steps
    _final_time = millis() + period[0] * steps;
    while (millis() < _final_time) {
        for (int i = 0; i < 8; i++) {
            setServo(i, oscillator[i].refresh());
        }
    }
}

```

```

    }
    delay(1); // Tiny delay to avoid too frequent refreshes
  }
}

```

Three Key Steps of This Function:

1. **Configure Parameters:** Like giving each musician sheet music, telling them how to play
2. **Synchronize Time:** This is crucial! All oscillators use the same `global_time` as reference, ensuring they start simultaneously. Like a conductor's downbeat, all musicians start playing together.
3. **Loop Execution:** Continuously calculate new positions and update servos until specified steps completed. Like musicians continuously playing according to the score until the piece ends.

5.5 Main Program and Bridge Communication

`sketch.ino` is Arduino's main program, responsible for receiving commands from Python middleware and executing corresponding actions:

```

#include "Arduino_RouterBridge.h"
#include <Servo.h>
#include "minikame.h"

Minikame robot;

// Command queue
volatile int pending_cmd = 0; // 0 = no command
volatile int cmd_steps = 5; // Default steps
volatile int cmd_period = 800; // Default period (ms)

void setup() {
  Serial.begin(9600);
  pinMode(LED_BUILTIN, OUTPUT);

  robot.init();
  delay(1000);
  robot.home();
  delay(1000);

  // Initialize Bridge, register callback functions
  Bridge.begin();
  Bridge.provide("action", set_action);
  Bridge.provide("steps", set_steps);
  Bridge.provide("period", set_period);

  Serial.println("🤖 Kame Robot Ready!");
}

void loop() {
  Bridge.update(); // Check for new commands

  if (pending_cmd != 0) {
    int cmd = pending_cmd;

```

```

    pending_cmd = 0;

    digitalWrite(LED_BUILTIN, LOW); // LED ON = executing

    // Execute corresponding action based on command ID
    switch (cmd) {
        case 1: robot.walk(cmd_steps, cmd_period);      break;
        case 3: robot.turnL(cmd_steps, cmd_period);     break;
        case 4: robot.turnR(cmd_steps, cmd_period);     break;
        case 5: robot.moonwalkL(cmd_steps, cmd_period); break;
        case 6: robot.dance(cmd_steps, cmd_period);     break;
        case 7: robot.upDown(cmd_steps, cmd_period);    break;
        case 8: robot.pushUp(cmd_steps, cmd_period);    break;
        case 9: robot.hello();                          break;
        case 11: robot.frontBack(cmd_steps, cmd_period); break;
        case 12: robot.home();                          break;
        case 13: robot.init();                          break;
    }

    // Return home after action completion (except home and init)
    if (cmd != 12 && cmd != 13) {
        robot.home();
    }

    digitalWrite(LED_BUILTIN, HIGH); // LED OFF = idle
}

delay(10);
}

// Bridge callback functions
void set_action(int action) { pending_cmd = action; }
void set_steps(int steps)   { cmd_steps = constrain(steps, 1, 20); }
void set_period(int period) { cmd_period = constrain(period, 200, 5000); }

```

Code Analysis:

1. **Bridge Communication:** `Arduino_RouterBridge` library implements communication with Python middleware, like a mail carrier delivering letters
2. **Command Queue:** Using `volatile` variables ensures interrupt safety, like having a dedicated inbox
3. **Parameter Constraints:** Steps limited to 1-20, period limited to 200-5000ms, like safety rails preventing parameters from going out of reasonable range
4. **Automatic Return Home:** All actions (except home/init) automatically return home after execution, like taking a bow after performance
5. **LED Indicator:** LED on indicates action execution, like a "working" indicator light

5.6 Action Mapping Table

Action Name	ID	Description
walk	1	Walking

Action Name	ID	Description
turnL	3	Turn Left
turnR	4	Turn Right
moonwalk	5	Moonwalk
dance	6	Dance
upDown	7	Up and Down
pushUp	8	Push-up
hello	9	Greeting
frontBack	11	Front and Back Swing
home	12	Return Home
init	13	Initialize

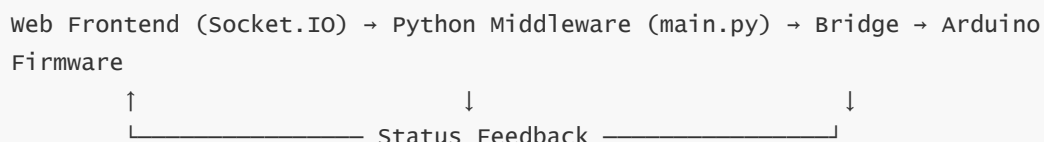
Part 6: Python Middleware Detailed Explanation

Python middleware is the "bridge" of the entire system, receiving Socket.IO messages from the web frontend and converting them to Bridge commands for Arduino.

Restaurant Analogy:

- **Web Frontend** is the menu and ordering interface (what customers see)
- **Python Middleware** is the waiter (receiving orders, conveying to kitchen)
- **Arduino Firmware** is the chef (actually preparing dishes)

6.1 Overall Architecture



6.2 main.py Complete Code Analysis

```

# main.py: Python Middleware Server

from arduino.app_utils import App, Bridge
from arduino.app_bricks.web_ui import webUI

ui = webUI()

# === Action Mapping Table ===
# Like a menu: Customer orders "steak", kitchen knows to make "dish #1"
ACTION_MAP = {
    'walk': 1,
    'turnL': 3,

```

```

    'turnR':      4,
    'moonwalk':   5,
    'dance':       6,
    'upDown':     7,
    'pushUp':     8,
    'hello':      9,
    'frontBack':  11,
    'home':       12,
    'init':       13,
}

# === Default Parameters ===
current_steps = 5      # Default steps
current_period = 800   # Default period (ms)

# === Callback Functions ===
def on_action(sid, cmd):
    """
    Handle action commands from frontend
    sid: Socket.IO session ID
    cmd: action name, e.g., 'walk'
    """
    action_id = ACTION_MAP.get(cmd, 0)
    if action_id > 0:
        print(f"🎮 Action: {cmd} (id={action_id}, steps={current_steps}, period={current_period})", flush=True)
        # Call Arduino's action function via Bridge
        Bridge.call("action", action_id)
        # Send status feedback to frontend
        ui.send_message("status", {"msg": f"Executing: {cmd}", "busy": True})
    else:
        print(f"⚠️ Unknown action: {cmd}", flush=True)

def on_steps(sid, value):
    """
    Handle steps slider changes
    value: step value (1-20)
    """
    global current_steps
    try:
        # Constrain step range, like safety rails
        current_steps = max(1, min(20, int(value)))
        # Call Arduino's steps function via Bridge
        Bridge.call("steps", current_steps)
        print(f"⚙️ Steps: {current_steps}", flush=True)
    except:
        pass

def on_period(sid, value):
    """
    Handle speed slider changes
    value: period value (200-3000ms)
    """
    global current_period
    try:
        # Constrain period range

```

```

    current_period = max(200, min(5000, int(value)))
    # Call Arduino's period function via Bridge
    Bridge.call("period", current_period)
    print(f"⚙️ Period: {current_period}ms", flush=True)
except:
    pass

# === Register Events ===
# Tell the waiter: when hearing "action", find on_action to handle
ui.on_message('action', on_action)
# When hearing "steps", find on_steps to handle
ui.on_message('steps', on_steps)
# When hearing "period", find on_period to handle
ui.on_message('period', on_period)

print("🤖 Kame Robot Web Controller Ready", flush=True)
# Start the application, open for business
App.run()

```

6.3 Middleware Workflow

1. Startup Phase:

- Initialize WebUI components (waiter reports for duty)
- Register message handling callbacks (remember who handles what)
- Wait for frontend connection (wait for customers to arrive)

2. Receive Commands:

- Frontend sends messages via Socket.IO (customers order)
- Call corresponding callback based on message type (waiter categorizes orders)
- Parameter validation and range constraint (check if customer requirements are reasonable)

3. Forward Commands:

- Convert action name to ID ("walk"→1, like "steak"→"dish #1")
- Call corresponding Arduino function via `Bridge.call()` (shout "make dish #1" to kitchen)
- Arduino executes actual action (chef starts cooking)

4. Status Feedback:

- Send status to frontend via `ui.send_message()` (waiter tells customer "your dish is being prepared")
- Frontend updates UI display (customer sees status update)

6.4 Key Design Points

1. Action Mapping Table:

```
ACTION_MAP = {
    'walk': 1,      # Frontend uses 'walk', Arduino uses 1
    'dance': 6,     # Decouple frontend and backend naming, like menu and kitchen
                    # using different numbers
}
```

2. Parameter Constraints:

```
current_steps = max(1, min(20, int(value))) # Ensure steps between 1-20
current_period = max(200, min(5000, int(value))) # Period between 200-5000ms
# Like safety rails preventing parameters from going out of reasonable range
```

3. Status Feedback:

```
ui.send_message("status", {"msg": f"Executing: {cmd}", "busy": True})
# Frontend receives and updates status display, like "your order has been
received"
```

4. Debug Output:

```
print(f"🤖 Action: {cmd} (id={action_id})", flush=True)
# flush=True ensures immediate output, convenient for debugging, like waiter
loudly repeating orders
```

Part 7: Web Frontend Detailed Explanation

The web frontend provides the graphical user interface for human-robot interaction, with responsive design supporting both mobile and desktop.

Restaurant Analogy:

- **HTML** is the menu structure (what dishes, how categorized)
- **CSS** is the menu styling (colors, fonts, layout)
- **JavaScript** is the ordering logic (button clicks, sending orders)

7.1 File Structure

```
assets/
├─ index.html      # Main page skeleton
├─ style.css       # Cyberpunk stylesheet
├─ script.js       # Interaction logic
└─ socket.io.min.js # Socket.IO library (local file, supports offline use)
```

7.2 HTML Structure (index.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```

<title>Kame Robot Controller</title>
<link rel="stylesheet" href="style.css">
<script src="socket.io.min.js"></script>
</head>
<body>
  <header>
    <div class="app-title">🤖 Kame Robot</div>
    <div class="status-badge">
      <div id="dot" class="dot"></div>
      <span id="stat">Connecting...</span>
    </div>
  </header>

  <main class="dashboard">
    <!-- Left side parameter panel -->
    <section class="panel-side" id="panel-params">
      <!-- Steps slider -->
      <div class="control-group">
        <div class="group-title">
          <span>Steps</span>
          <span id="steps-val" class="val-display">5</span>
        </div>
        <input class="slider-speed" type="range" min="1" max="20"
value="5"
          oninput="updateSteps(this.value)">
      </div>

      <!-- Speed slider -->
      <div class="control-group">
        <div class="group-title">
          <span>Period</span>
          <span id="period-val" class="val-display">800ms</span>
        </div>
        <input class="slider-gimbal" type="range" min="200" max="3000"
value="800" step="100"
          oninput="updatePeriod(this.value)">
        <div class="slider-label">
          <span>Fast</span>
          <span>Slow</span>
        </div>
      </div>

      <!-- Status panel -->
      <div class="control-group status-panel">
        <div class="group-title"><span>Status</span></div>
        <div id="status-text" class="status-text">READY</div>
      </div>
    </section>

    <!-- Center action buttons area -->
    <section class="panel-center" id="panel-actions">
      <div class="action-grid">
        <!-- Movement category -->
        <div class="action-category">
          <div class="category-title">🔍 Movement</div>
          <div class="action-buttons cols-3">

```

```

        <button class="action-btn btn-move"
onclick="doAction('walk')">
        <span class="btn-icon"> 🚶 </span>
        <span class="btn-label">walk</span>
    </button>
    <!-- More buttons... -->
</div>
</div>
<!-- More categories... -->
</div>
</section>
</main>

<script src="script.js"></script>
</body>
</html>

```

7.3 JavaScript Interaction Logic (script.js)

```

const socket = io();
const dot = document.getElementById('dot');
const stat = document.getElementById('stat');
const statusText = document.getElementById('status-text');

// === Connection Status Handling ===
socket.on('connect', () => {
    dot.classList.add('online');
    stat.innerText = "ONLINE";
    stat.style.color = "#fff";
    statusText.innerText = "READY";
    statusText.className = "status-text ready";
});

socket.on('disconnect', () => {
    dot.classList.remove('online');
    stat.innerText = "OFFLINE";
    stat.style.color = "#666";
    statusText.innerText = "DISCONNECTED";
    statusText.className = "status-text";
});

// === Receive Status Feedback ===
socket.on('status', (data) => {
    if (data.msg) {
        statusText.innerText = data.msg;
        statusText.className = data.busy ? "status-text busy" : "status-text ready";
    }
});

// === Send Action Commands ===
function doAction(cmd) {
    if(event && event.cancelable) event.preventDefault();
    if(navigator.vibrate) navigator.vibrate(20); // Haptic feedback

```

```

socket.emit('action', cmd); // Send to Python middleware
statusText.innerHTML = "⌚ " + cmd + "...";
statusText.className = "status-text busy";

// Button press animation
if(event && event.target) {
    const btn = event.target.closest('.action-btn');
    if(btn) {
        btn.classList.add('pressed');
        setTimeout(() => btn.classList.remove('pressed'), 300);
    }
}

}

// === Update Steps ===
function updateSteps(val) {
    document.getElementById('steps-val').innerHTML = val;
    socket.emit('steps', val); // Send to Python middleware
}

// === Update Period ===
function updatePeriod(val) {
    document.getElementById('period-val').innerHTML = val + "ms";
    socket.emit('period', val); // Send to Python middleware
}

// === Disable Right-Click Menu ===
document.oncontextmenu = (e) => e.preventDefault();

```

7.4 CSS Style Design Points

```

:root {
    --bg-color: #0d0d0d; /* Dark background, cyberpunk style */
    --primary-cyan: #00f2ff; /* Cyan, represents movement */
    --primary-amber: #ffae00; /* Amber, represents steps */
    --primary-green: #00e676; /* Green, represents system */
    --primary-purple: #b388ff; /* Purple, represents performance */
}

/* Responsive design: auto-adjust layout on mobile portrait */
@media (max-width: 900px) and (orientation: portrait) {
    .dashboard {
        flex-direction: column; /* Change to vertical arrangement */
    }
}

/* Button press animation */
.action-btn:active, .action-btn.pressed {
    transform: scale(0.95); /* Slightly shrink, creating pressed feeling */
    box-shadow: 0 0 20px currentColor; /* Glow effect */
}

```

7.5 Frontend-Backend Communication Flow

1. Frontend Operation → Socket.IO Message → Python Middleware
└ Click "walk" button → doAction('walk') → socket.emit('action', 'walk')
(Like customer saying "I'd like to order steak")
2. Python Middleware Processing → Bridge Call → Arduino
└ on_action('walk') → Bridge.call('action', 1)
(Waiter shouts "make one dish #1" to kitchen)
3. Arduino Execution → Completion Feedback
└ robot.walk() → Automatically returns home after execution
(Chef starts cooking)
4. Status Feedback → Socket.IO → Frontend UI Update
└ ui.send_message('status', {...}) → Frontend displays "Executing: walk"
(Waiter tells customer "your dish is being prepared")

Part 8: Complete Action Library Detailed Explanation

In this section, we will deeply analyze all of MiniKame's preset actions. Each action is implemented through carefully designed parameter combinations, including amplitude (A), period (T), phase (ϕ), and offset (O). We'll explain each action's principle and design approach in easy-to-understand ways.

8.1 Standing Posture (home)

Standing posture is the foundation of all actions and the default posture after action execution.

```
void Minikame::home() {  
    int ap = 20;    // Anteroposterior offset  
    int hi = 25;    // Height offset  
  
    int position[] = {  
        90 + ap,    // Left front thigh: forward 20°  
        90 - ap,    // Right front thigh: backward 20°  
        90 - hi,    // Left front calf: lift 25°  
        90 + hi,    // Right front calf: lower 25°  
        90 - ap,    // Left rear thigh: backward 20°  
        90 + ap,    // Right rear thigh: forward 20°  
        90 + hi,    // Left rear calf: lower 25°  
        90 - hi     // Right rear calf: lift 25°  
    };  
  
    for (int i = 0; i < 8; i++) setServo(i, position[i]);  
}
```

8.1.1 Posture Analysis

Let's use a table to understand each leg's initial position:

Leg	Thigh Position	Calf Position	Overall Effect
Left Front	Forward 20° (110°)	Lift 25° (65°)	Leg forward, up
Right Front	Backward 20° (70°)	Lower 25° (115°)	Leg backward, down
Left Rear	Backward 20° (70°)	Lower 25° (115°)	Leg backward, down
Right Rear	Forward 20° (110°)	Lift 25° (65°)	Leg forward, up

8.1.2 Why This Design?

1. Diagonal Balance Principle

Observing that left front and right rear form one group (both forward, up), right front and left rear form another (both backward, down). This is classic **diagonal balance** in quadruped robots:

- Left front-right rear diagonal: legs extend forward, body raised
- Right front-left rear diagonal: legs retract backward, body lowered

This design centers the robot's center of gravity, like a table with four legs being stable.

2. Significance of Offset Values

`ap = 20` and `hi = 25` aren't chosen randomly:

- `ap` controls front-back posture: 20° gives legs natural front-back separation
- `hi` controls body height: 25° lifts body about 2-3cm off ground—not too high to be unstable, not too low to drag

Analogy: Like standing with legs slightly apart and knees slightly bent—most stable. Standing with legs together and straight is actually more prone to falling.

8.1.3 zero(): Complete Zeroing

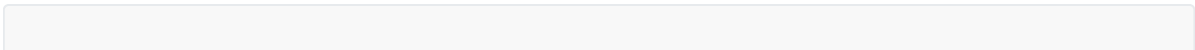
```
void Minikame::zero() {
    for (int i = 0; i < 8; i++) setServo(i, 90);
}
```

`zero()` sets all servos to 90°, the physical neutral position. Main uses:

- Verify all servos working correctly during initial debugging
- Benchmark for all movements
- Re-zero after replacing servos or reassembly

8.2 Walking Gait (walk)

Walking is the most basic and important action for quadruped robots. Let's deeply understand its implementation.



```

void MiniKame::walk(float steps, int T = 500) {
    int x_amp = 15;      // Forward/backward swing amplitude
    int z_amp = 20;      // Up/down leg lift amplitude
    int ap = 20;         // Anteroposterior offset
    int hi = 30;         // Height offset
    int front_x = 12;    // Extra forward extension for front legs

    int period[] = {T, T, T/2, T/2, T, T, T/2, T/2};
    int amplitude[] = {x_amp, x_amp, z_amp, z_amp, x_amp, x_amp, z_amp, z_amp};
    int offset[] = {
        90 + ap - front_x, // Left front thigh
        90 - ap + front_x, // Right front thigh
        90 - hi,           // Left front calf
        90 + hi,           // Right front calf
        90 - ap - front_x, // Left rear thigh
        90 + ap + front_x, // Right rear thigh
        90 + hi,           // Left rear calf
        90 - hi            // Right rear calf
    };

    int phase[] = {90, 90, 270, 90, 270, 270, 90, 270};

    // Execution logic
    for (int i = 0; i < 8; i++) {
        oscillator[i].reset();
        oscillator[i].setPeriod(period[i]);
        oscillator[i].setAmplitude(amplitude[i]);
        oscillator[i].setPhase(phase[i]);
        oscillator[i].setOffset(offset[i]);
    }

    _final_time = millis() + period[0] * steps;
    _init_time = millis();
    bool side;

    while (millis() < _final_time) {
        side = (int)((millis() - _init_time) / (period[0] / 2)) % 2;

        // Thighs always move
        setServo(0, oscillator[0].refresh());
        setServo(1, oscillator[1].refresh());
        setServo(4, oscillator[4].refresh());
        setServo(5, oscillator[5].refresh());

        // Calves alternate movement
        if (side == 0) {
            setServo(3, oscillator[3].refresh());
            setServo(6, oscillator[6].refresh());
        } else {
            setServo(2, oscillator[2].refresh());
            setServo(7, oscillator[7].refresh());
        }
        delay(1);
    }
}

```

8.2.1 Parameter Detailed Explanation

1. Amplitude

Servo	Amplitude	Function
Thighs (0,1,4,5)	15°	Forward/backward swing range
Calves (2,3,6,7)	20°	Up/down leg lift range

Why larger calf amplitude? Because lifting requires overcoming gravity, needing larger angles to ensure feet completely clear ground.

2. Period

Servo	Period	Speed	Reason
Thighs	T	Slow	Forward/backward swinging needs time to complete full stride
Calves	T/2	Fast	Lifting needs to be quick to avoid ground drag

3. Offset

Let's analyze each offset value line by line:

Left front thigh: $90 + ap - front_x = 90 + 20 - 12 = 98^\circ$

Left front thigh extends 8° forward from neutral, because front legs bear more propulsion tasks.

Right front thigh: $90 - ap + front_x = 90 - 20 + 12 = 82^\circ$

Right front thigh swings 8° backward from neutral, symmetrical with left front.

Left front calf: $90 - hi = 90 - 30 = 60^\circ$

Left front calf lifts 30°, lifting foot off ground.

Right front calf: $90 + hi = 90 + 30 = 120^\circ$

Right front calf lowers 30°, supporting body.

4. Phase

This is the most critical parameter. Let's rearrange the phase table:

Leg	Thigh Phase	Calf Phase	Phase Difference	Function
Left Front	90°	270°	180°	Calf lifts when thigh moves forward

Leg	Thigh Phase	Calf Phase	Phase Difference	Function
Right Front	90°	90°	0°	Calf lowers when thigh moves forward
Left Rear	270°	90°	180°	Calf lifts when thigh moves backward
Right Rear	270°	270°	0°	Calf lowers when thigh moves backward

8.2.2 Gait Analysis

1. Diagonal Coordination

Left front and right rear form one pair, right front and left rear another. When left front leg swings forward, right rear also swings forward, forming diagonal support. This is the classic **trot gait** of quadruped animals.

2. Calf Alternation Logic

```
side = (int)((millis() - _init_time) / (period[0] / 2)) % 2;
```

This formula determines whether it's the first or second half of the period:

- side = 0: first half, activate right front calf (3) and left rear calf (6)
- side = 1: second half, activate left front calf (2) and right rear calf (7)

3. Complete Walking Cycle

A complete walking cycle has two phases:

Phase 1 (First Half):

- Left front thigh swings forward, left front calf lifts (preparing to land)
- Right rear thigh swings forward, right rear calf lifts (preparing to land)
- Right front thigh swings backward, right front calf lowers (supporting)
- Left rear thigh swings backward, left rear calf lowers (supporting)

Phase 2 (Second Half):

- Left front thigh swings backward, left front calf lowers (supporting)
- Right rear thigh swings backward, right rear calf lowers (supporting)
- Right front thigh swings forward, right front calf lifts (preparing to land)
- Left rear thigh swings forward, left rear calf lifts (preparing to land)

This creates stable forward movement.

Analogy: Like when walking, when left leg moves forward, right arm also moves forward (diagonal coordination). If left leg and left arm both move forward together, that's walking awkwardly.

8.3 Turning Gaits

8.3.1 Turn Left (turnL)

```
void Minikame::turnL(float steps, int T = 600) {
    int x_amp = 15;
    int z_amp = 15;
    int ap = 15;
    int hi = 23;

    int period[] = {T, T, T, T, T, T, T, T};
    int amplitude[] = {x_amp, x_amp, z_amp, z_amp, x_amp, x_amp, z_amp, z_amp};
    int offset[] = {90 + ap, 90 - ap, 90 - hi, 90 + hi,
                   90 - ap, 90 + ap, 90 + hi, 90 - hi};
    int phase[] = {180, 0, 90, 90, 0, 180, 90, 90};

    execute(steps, period, amplitude, offset, phase);
}
```

8.3.2 Parameter Analysis

Phase Table (Turn Left):

Leg	Thigh Phase	Function
Left Front	180°	Opposite to right front
Right Front	0°	Opposite to left front
Left Rear	0°	Opposite to right rear
Right Rear	180°	Opposite to left rear

Turning Mechanics Principle:

1. **Left-Right Phase Opposition:**
 - When left side legs move forward (near 0° phase), right side legs move backward (near 180° phase)
 - This creates torque causing counterclockwise rotation
2. **Front-Rear Relationship:**
 - Left front (180°) and left rear (0°) have 180° phase difference
 - This means front legs move backward while rear move forward, further enhancing the rotation effect
3. **Consistent Calf Phases:**
 - All calves have phases around 90°
 - Ensures legs can still lift and lower during rotation

Analogy: This is like paddling only on the left side of a boat—the boat turns right. Here, left side legs moving forward (paddling) while right side moves backward (resistance) creates rotational torque.

8.3.3 Turn Right (turnR)

```
void Minikame::turnR(float steps, int T = 600) {  
    // ... same parameters ...  
    int phase[] = {0, 180, 90, 90, 180, 0, 90, 90};  
    execute(steps, period, amplitude, offset, phase);  
}
```

Compared to turn left, only the left-right leg phases are swapped:

- Left leg phase 0°, right leg phase 180° (completely opposite to turn left)
- This creates clockwise rotation

8.4 Dance Mode (dance)

```
void Minikame::dance(float steps, int T = 600) {  
    int x_amp = 0;           // Thighs don't swing  
    int z_amp = 40;          // Calves swing largely  
    int ap = 30;             // Larger offset  
    int hi = 20;  
  
    int period[] = {T, T, T, T, T, T, T, T};  
    int amplitude[] = {x_amp, x_amp, z_amp, z_amp, x_amp, x_amp, z_amp, z_amp};  
    int offset[] = {90 + ap, 90 - ap, 90 - hi, 90 + hi,  
                   90 - ap, 90 + ap, 90 + hi, 90 - hi};  
    int phase[] = {0, 0, 0, 270, 0, 0, 90, 180};  
  
    execute(steps, period, amplitude, offset, phase);  
}
```

8.4.1 Parameter Analysis

1. Special Amplitude Settings

- `x_amp = 0`: Thighs completely stationary
- `z_amp = 40`: Calf amplitude double that of walking

2. Exaggerated Offsets

- `ap = 30`: Larger than walking's 20°
- Body posture more exaggerated, presenting twisting state

3. Chaotic Phases

Leg	Phase	Description
Left Front Calf	0°	Synchronized with right front
Right Front Calf	270°	270° difference from left front
Left Rear Calf	90°	Chaotic
Right Rear Calf	180°	Chaotic

8.4.2 Source of Dance Effect

When phases no longer follow patterns, the robot presents twisting, swaying effects:

1. **Chaotic Phases:** No diagonal coordination, each leg acts independently
2. **Large Calf Swings:** Create exaggerated leg movements
3. **Exaggerated Posture:** Body twists with large amplitude

Design Philosophy: Order creates patterns, patterns create function; breaking patterns creates variation, variation creates interest.

8.5 Moonwalk (moonwalkL)

```
void Minikame::moonwalkL(float steps, int T = 5000) {  
    int z_amp = 45;           // Calves swing largely  
    int period[] = {T, T, T, T, T, T, T, T};  
    int amplitude[] = {0, 0, z_amp, z_amp, 0, 0, z_amp, z_amp};  
    int offset[] = {90, 90, 90, 90, 90, 90, 90, 90};  
    int phase[] = {0, 0, 0, 120, 0, 0, 180, 290};  
  
    execute(steps, period, amplitude, offset, phase);  
}
```

8.5.1 Moonwalk Principle

Key Design:

1. **Fixed Thighs:** All thigh amplitudes 0, remain stationary
2. **Calf Movement Only:** Only calves swing
3. **Staggered Phases:** Calf phases vary (0°, 120°, 180°, 290°)

Visual Effect:

- Fixed thighs mean body position basically unchanged
- Alternating calf movement creates wave-like visual effect
- Looks like sliding on smooth ground

Analogy: This is like Michael Jackson's moonwalk, where the body moves forward but the feet appear to slide backward. Here it's the opposite—body stationary while legs move, creating a visual illusion.

8.6 Up and Down (upDown)

```
void Minikame::upDown(float steps, int T = 5000) {  
    int x_amp = 0;           // Thighs don't swing  
    int z_amp = 25;         // Calf amplitude  
    int ap = 20;  
    int hi = 25;  
    int front_x = 0;  
  
    int period[] = {T, T, T, T, T, T, T, T};  
    int amplitude[] = {x_amp, x_amp, z_amp, z_amp, x_amp, x_amp, z_amp, z_amp};  
    int offset[] = {90 + ap - front_x, 90 - ap + front_x, 90 - hi, 90 + hi,
```

```

        90 - ap - front_x, 90 + ap + front_x, 90 + hi, 90 - hi};
int phase[] = {0, 0, 90, 270, 180, 180, 270, 90};

execute(steps, period, amplitude, offset, phase);
}

```

8.6.1 Movement Analysis

Phase Relationships:

Leg	Thigh Phase	Calf Phase	Function
Left Front	0°	90°	Thigh fixed, calf moves
Right Front	0°	270°	Opposite phase to left front calf
Left Rear	180°	270°	180° phase difference from front
Right Rear	180°	90°	180° phase difference from front

Movement Effect:

- Front calves have 180° phase difference, so one lifts while the other lowers
- Rear calves are the same, but opposite phase from front
- Result is body rising and falling as a whole

8.7 Push-up (pushUp)

```

void Minikame::pushUp(float steps, int T = 600) {
    int z_amp = 30;           // Calf amplitude 30°
    int x_amp = 60;           // Rear thighs extend backward 60°
    int hi = 20;

    int period[] = {T, T, T, T, T, T, T, T};
    int amplitude[] = {0, 0, z_amp, z_amp, 0, 0, 0, 0};
    int offset[] = {
        90, 90,                // Front thighs fixed
        90 - hi, 90 + hi,      // Front calves move up/down
        90 - x_amp,            // Left rear thigh extends backward
        90 + x_amp,            // Right rear thigh extends backward
        90 + hi, 90 - hi       // Rear calves fixed
    };
    int phase[] = {0, 0, 0, 180, 0, 0, 0, 170};

    execute(steps, period, amplitude, offset, phase);
}

```

8.7.1 Action Breakdown

1. Rear Leg Setup:

- Rear thighs set to 30° and 150°, lowering rear of body
- Like feet position when doing push-ups

2. Front Calf Movement:

- Only servos 2 and 3 (front calves) move
- Amplitude 30°, swinging within 30° range
- 180° phase difference, alternating lift and lower

3. Body Posture:

- Front legs remain upright
- Rear legs extended backward, body leaning forward
- Forms push-up ready posture

8.7.2 Movement Effect

This perfectly simulates push-up action:

1. **Preparation:** Body leans forward, center of gravity shifts forward
2. **Lowering:** Front calves bend, body lowers
3. **Rising:** Front calves straighten, body rises
4. **Repeat:** Alternating

8.8 Greeting (hello)

```
void Minikame::hello() {
    // Step 1: Sit down
    float sentado[] = {
        90 + 15, // Left front thigh forward
        90 - 15, // Right front thigh backward
        90 - 10, // Left front calf slightly lifted
        90 + 60, // Right front calf largely bent
        90 + 10, // Left rear thigh forward
        90 - 10, // Right rear thigh backward
        90 + 10, // Left rear calf lowered
        90 - 10  // Right rear calf lifted
    };
    moveServos(150, sentado);
    delay(300);

    // Step 2: Wave
    int T = 550;
    int period[] = {T, T, T, T, T, T, T, T};
    int amplitude[] = {0, 50, 0, 50, 0, 0, 0, 0};
    int offset[] = {
        90 + 15, // Left front thigh
        40,      // Right front thigh largely forward
        90 - 30, // Left front calf
        90,      // Right front calf
        90 + 20, // Left rear thigh
        90 - 20, // Right rear thigh
        90 + 10, // Left rear calf
        90 - 10  // Right rear calf
    };
    int phase[] = {0, 0, 0, 90, 0, 0, 0, 0};
```

```

execute(4, period, amplitude, offset, phase);

// Step 3: Stand up
float goingUp[] = {
    90,      // Left front thigh back to normal
    70,      // Right front thigh backward
    90,      // Left front calf back to normal
    90,      // Right front calf back to normal
    90 - 20, // Left rear thigh backward
    90 + 20, // Right rear thigh forward
    90 + 10, // Left rear calf
    90 - 10  // Right rear calf
};
moveServos(500, goingUp);
delay(200);
}

```

8.8.1 Action Sequence Design

`hello()` demonstrates combining multiple basic actions to create expressive behavior:

Phase 1: Sit Down

- Right front calf largely bent ($90+60=150^\circ$), this is key
- Other legs fine-tuned to maintain balance
- Entire robot center of gravity lowers, presenting "sitting" posture

Phase 2: Wave

- Only right front leg moves (servos 1 and 3)
- Amplitude 50° , letting leg swing largely
- Execute 4 oscillations, creating "waving" action
- Other legs remain fixed, maintaining sitting posture

Phase 3: Stand Up

- Return to standing posture
- Right front thigh returns to 70° (further back than standard posture)
- All legs adjust to final positions

8.8.2 Behavior Design Philosophy

This function shows important principles of robot behavior design:

1. **Preparation:** Sit down, attract attention
2. **Main Action:** Wave, convey information
3. **Conclusion:** Stand up, return to normal

This "prepare-execute-conclude" structure makes robot behavior more natural and expressive.

8.9 Front and Back Swing (frontBack)

```

void MiniKame::frontBack(float steps, int T = 600) {
    int x_amp = 30;           // Larger front-back amplitude
    int z_amp = 25;           // Medium up-down amplitude
    int ap = 20;
    int hi = 30;

    int period[] = {T, T, T, T, T, T, T, T};
    int amplitude[] = {x_amp, x_amp, z_amp, z_amp, x_amp, x_amp, z_amp, z_amp};
    int offset[] = {90 + ap, 90 - ap, 90 - hi, 90 + hi,
                    90 - ap, 90 + ap, 90 + hi, 90 - hi};
    int phase[] = {0, 180, 270, 90, 0, 180, 90, 270};

    execute(steps, period, amplitude, offset, phase);
}

```

8.9.1 Movement Characteristics

1. Large Amplitude: `x_amp = 30` is double walking's 15°, creating large front-back swings

2. Phase Relationships:

- Front and rear legs have same phase (front 0°, rear 0°)
- Left and right legs have opposite phase (left 0°, right 180°)

3. Movement Effect:

- All legs swing forward simultaneously, then backward simultaneously
- Robot as a whole sways front and back, like rocking

8.10 Difference Between `init()` and `home()`

Although we've analyzed `home()` in detail, it's worth explaining the difference between `init()` and `home()`:

`init()`:

- Called only once at program start
- Sets pin mapping, trim values, reverse directions
- Initializes oscillators and servos
- Equivalent to "system boot"

`home()`:

- Can be called multiple times
- Sets robot to standard standing posture
- Automatically called after action execution
- Equivalent to "return to home position"

8.11 Action Design Summary

8.11.1 Application of Four Core Parameters

Through analyzing all actions, we can see how different combinations of four parameters produce different effects:

Parameter	Function	Range	Typical Values
Amplitude A	Controls movement range	0-60°	Walk:15-20°, Dance:40°
Period T	Controls movement speed	300-5000ms	Walk:500ms, Moonwalk:5000ms
Phase ϕ	Controls coordination	0-360°	Walk:90/270°, Turn:0/180°
Offset O	Controls neutral position	60-120°	Stand:70-110°

8.11.2 Action Classification

Based on movement characteristics, actions can be classified into several categories:

1. Movement Class: walk, turnL, turnR

- Characterized by generating displacement
- Requires diagonal coordination

2. Performance Class: dance, moonwalk, hello

- Characterized by strong visual appeal
- Can break conventional patterns

3. Stunt Class: pushUp, upDown, frontBack

- Characterized by specific part movement
- Other parts remain fixed

4. System Class: init, home, zero

- Used for initialization and reset
- Doesn't involve oscillatory movement

8.11.3 Method for Designing New Actions

If you want to create new actions, you can follow this process:

1. Determine Action Intent: What do you want the robot to do? (e.g., nod, shake head, lift leg)

2. Decompose Movement Elements:

- Which legs need to move?
- What direction of movement?
- How large the movement range?
- How fast the movement speed?

3. Set Parameters:

- Stationary legs: set amplitude to 0
- Moving legs: set amplitude and period based on requirements
- Set phase to achieve coordination
- Adjust offset to maintain balance

4. Test and Adjust:

- First test each leg individually
- Then test overall effect
- Fine-tune parameters until satisfied

5. Combine Sequences:

- Multiple basic actions can be combined into complex behaviors
- Ensure smooth transitions between actions

Example: Creating a "nod" action

```
void Minikame::nod() {  
    // Nod: front legs move up/down, rear legs fixed  
    int amplitude[] = {0, 0, 20, 20, 0, 0, 0, 0};  
    int offset[] = {90, 90, 70, 110, 90, 90, 90, 90};  
    int phase[] = {0, 0, 0, 180, 0, 0, 0, 0};  
    int period[] = {500, 500, 500, 500, 500, 500, 500, 500};  
  
    execute(3, period, amplitude, offset, phase);  
}
```

By understanding the magic of these four parameters, you can create an infinite variety of actions, allowing MiniKame to showcase rich expressiveness.

Part 9: Deployment and Usage Guide

9.1 Quick Start Process

Step 1: Run the Program

```
# Execute in project root directory  
arduino-app-cli app start kame_robot
```

Like a restaurant opening for business, waiters are ready.

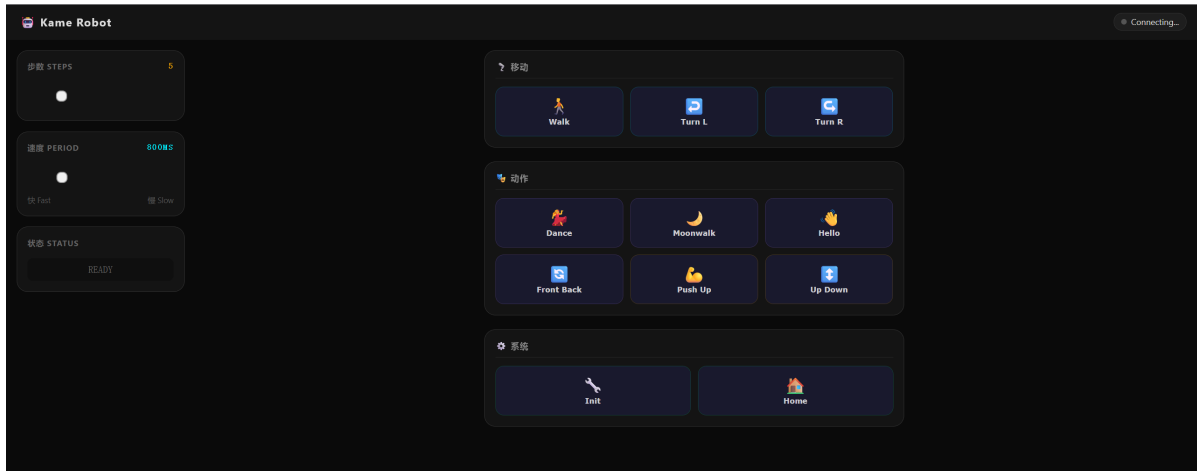
Step 2: Connect to the Same WiFi as the Robot

- Connect your phone/computer to the same WiFi as the robot
- Like entering the restaurant's exclusive area.

Step 3: Open Control Interface

- Browser access `http://<robot IP>:7000`
- Default IP can be viewed in serial monitor

- Like getting the menu, ready to order.



9.2 Interface Operation Guide

Parameter Adjustment Area:

- **Steps Slider (Amber):** 1-20 steps, controls repetition count for each action
- **Speed Slider (Cyan):** 200-3000ms, smaller value = faster action
- **Status Display:** Shows current connection and execution status

Action Button Area:

- **Movement Class (Cyan):** Walk, Turn Left, Turn Right
- **Performance Class (Purple):** Dance, Moonwalk, Hello, Front Back
- **Stunt Class (Yellow):** Push-up, Up Down
- **System Class (Green):** Initialize, Return Home

Operation Feedback:

- Button press provides haptic feedback (mobile)
- Status display updates in real-time
- LED indicator shows execution status

9.3 Configuration File Description

`app.yaml` is the Arduino App configuration file:

```
# app.yaml
name: kame_robot                # Application name
description: "Web controlled kame quadruped robot" # Description
icon: 🤖                        # App icon
ports: []                       # Network ports
bricks:
- arduino:web_ui: {}           # Used brick
```

9.4 Debugging Techniques

1. View Serial Output

```
// Add debug output in sketch.ino
Serial.println("Executing walk...");
```

2. View Python Logs

```
# Debug output already present in main.py
print(f"🤖 Action: {cmd}", flush=True)
```

3. Browser Console

```
// Add in script.js
socket.onAny((event, ...args) => {
  console.log(event, args);
});
```

4. Single-Step Debugging

Add in Arduino code:

```
while(!Serial.available()); // wait for serial input
Serial.read();              // Read and discard
```

Part 10: Calibration and Debugging

10.1 Mechanical Calibration (trim values)

Trim values compensate for mechanical assembly errors. Even with all servos set to 90°, due to assembly errors, the robot may not stand level.

Real-life Analogy:

- Trim values are like putting paper shims under table legs to stop wobbling
- Each leg may need different shim thickness
- Our goal is to find how much each leg needs

Calibration Process:

Step 1: Initial Observation

```
void setup() {
  robot.init();
  delay(2000); // Observe for 2 seconds
  robot.home();
}
```

Place the robot on a level surface and observe:

- Do all four legs touch the ground?

- Is the body level?
- Are any legs hanging or excessively pressed?

Step 2: Single Leg Debugging

```
// Debug program: adjust each leg one by one
void debugLeg(int leg) {
    for (int offset = -30; offset <= 30; offset += 5) {
        robot.setServo(leg, 90 + offset);
        delay(1000);
    }
}
```

Like putting different thickness shims under each leg one by one, observing which thickness works best.

Step 3: Record Calibration Values

Set trim values in `init()`:

```
trim[0] = -5;    // Left front thigh needs -5° counterclockwise adjustment
trim[1] = -5;    // Right front thigh needs -5° counterclockwise adjustment
trim[2] = -20;   // Left front calf needs -20° counterclockwise adjustment
trim[3] = 10;    // Right front calf needs +10° clockwise adjustment
trim[4] = -3;    // Left rear thigh needs -3° counterclockwise adjustment
trim[5] = -3;    // Right rear thigh needs -3° counterclockwise adjustment
trim[6] = -15;   // Left rear calf needs -15° counterclockwise adjustment
trim[7] = -13;   // Right rear calf needs -13° counterclockwise adjustment
```

Step 4: Verify

Re-upload the program and observe the robot's posture. Usually, 3-5 iterations are needed for satisfactory results.

10.2 Direction Calibration (reverse values)

The reverse flag handles inconsistent servo installation directions:

Determination Method:

1. Set all reverse to false
2. Call the `home()` function
3. Observe each leg's movement direction:

Observation	reverse Setting
Increasing angle moves leg forward/up	false
Increasing angle moves leg backward/down	true

Analogy:

Some people nod for "yes", some shake their head for "yes". We need to know each person's (each servo's) habit, and then "translate" accordingly.

Code Setting:

```
// Set in init()
reverse[0] = false; // Left front thigh: normal
reverse[1] = true; // Right front thigh: needs reversal
reverse[2] = false; // Left front calf: normal
reverse[3] = true; // Right front calf: needs reversal
// ...
```

10.3 Common Problems and Solutions

Q1: The robot is unstable and always leans to one side

Possible Causes:

- trim values not properly calibrated
- A leg's connecting rod installed crooked

Solutions:

1. Re-perform the calibration process
2. Check if all screws are tight

Q2: Servos jitter severely

Possible Causes:

- Insufficient power supply
- Poor ground connection

Solutions:

1. Replace with a higher current power supply
2. Add a 1000µF capacitor across the servo power
3. Ensure all GND connections are reliable

Q3: Web interface cannot connect

Possible Causes:

- WiFi not connected
- Python service not started
- Firewall blocking

Solutions:

1. Check the robot's WiFi hotspot
2. Confirm the Python service is running
3. Check if port 7000 is open

Q4: Action stops mid-execution

Possible Causes:

- No `delay()` in the `while` loop
- Watchdog timeout

Solutions:

- 1. Add `delay(1)` in long loops
- 2. Use `yield()` to feed the watchdog

Q5: Phase confusion, gait uncoordinated

Possible Causes:

- Oscillator time not synchronized
- Parameter settings are wrong

Solutions:

- 1. Ensure using `setTime(global_time)`
- 2. Check if the phase parameters are correct

Appendix

A.1 Project File Structure

```
kame_robot/
├─ sketch/                                # Arduino Firmware
│   ├── sketch.ino                        # Main program
│   ├── minikame.cpp/h                    # Minikame class implementation
│   ├── octosnake.cpp/h                   # Oscillator class implementation
│   └─ sketch.yaml                        # Dependency configuration
├─ python/
│   └─ main.py                            # Python middleware
├─ assets/                                # web frontend resources
│   ├── index.html                        # Main page
│   ├── style.css                          # Stylesheet
│   ├── script.js                         # Interaction logic
│   └─ socket.io.min.js                   # Socket.IO library
├─ app.yaml                               # Application configuration
└─ README.md                             # Project documentation
```

A.2 Pin Assignment Table

Servo	Function	Pin
S0	Left Front Thigh	D4
S1	Right Front Thigh	D5
S2	Left Front Calf	D6
S3	Right Front Calf	D7
S4	Left Rear Thigh	D8
S5	Right Rear Thigh	D9
S6	Left Rear Calf	D10

Servo	Function	Pin
S7	Right Rear Calf	D11

A.3 Action Mapping Table

Action Name	ID	Description
walk	1	Walking
turnL	3	Turn Left
turnR	4	Turn Right
moonwalk	5	Moonwalk
dance	6	Dance
upDown	7	Up and Down
pushUp	8	Push-up
hello	9	Greeting
frontBack	11	Front and Back Swing
home	12	Return Home
init	13	Initialize

A.4 Common Function Quick Reference

```
// Initialization
robot.init();

// Basic actions
robot.walk(steps, period);
robot.turnL(steps, period);
robot.turnR(steps, period);
robot.dance(steps, period);
robot.moonwalkL(steps, period);
robot.pushUp(steps, period);
robot.hello();
robot.home();
robot.zero();

// Advanced control
robot.setServo(id, target);
robot.moveServos(time, target[8]);

// Debugging
robot.getServo(id);
robot.reverseServo(id);
```

Document Information:

- Version: 2.0.0
- Last Updated: February 2026
- Author: None.
- License: MIT