第二十四屆旺宏金矽獎

作品企劃書

外骨骼暨 AR 交互運動輔助裝置

Exoskeleton and AR Interactive Motion Assistance Device

參賽組別：Robotics

參賽編號：A24-192

隊　　長：賴宏達

隊　　員：沈恩佑、徐翊紘、劉芸婷

1. Project proposal

1.1. Introduction

Prolonged physical activities not only lead to muscle fatigue but may also pose safety and health risks during exercises. This reality has prompted us to consider how technology can address this issue. Consequently, we propose the development of an exoskeleton system designed to enhance strength across all age groups. This system aims to reduce the intensity of specific muscle usage and extend the duration of activities that consume physical strength, such as long-distance hiking and stair climbing. By alleviating muscle strain, we hope to enhance the safety of these activities, allowing individuals to confidently engage in various exercises, from everyday stair climbing to more rigorous physical challenges. For static activities, such as squats, the system can also monitor muscle usage to provide motion protection.

To achieve this goal, we plan to utilize AR glasses combined with a virtual coach to offer users an immersive exercise experience (**Figure 1**.) . Furthermore, we intend to integrate multiple EMG patches with the exoskeleton device, using a control chip to monitor joint angles and IMU information, and display the muscle force status in real-time on the AR interface. This design not only facilitates users in understanding their muscle condition but also activates joint restrictions and strength enhancement features of the exoskeleton in case of fatigue or incorrect movements, providing warnings. The exoskeleton is designed to be as portable as an electric scooter, integrating seamlessly into daily life. Therefore, we have focused on developing a hip joint exoskeleton, choosing the hip joint due to its significance in activities like upright walking. The so-called "hip hinge" movement, which starts from the hip joint and relies on the strength of the legs and lower back, supports the body and external weights. Our motor control strategy will employ a reinforcement learning model to adapt to the user's gait, aiming to alleviate the burden on specific muscles.

Figure 1. User exercising with AR glasses and wearing exoskeleton alongside a virtual coach.

1.2. Specification

1.2.1. Hip Exoskeleton

1.2.2. EMG Cygnus dongle and Electrode patches

1.2.3. ESP32 microcontroller

1.2.4. STM32F415 microcontroller

1.2.5. R14 Motor

1.3. Block diagram

1.4. Work theory

1.4.1. Algorithm

We utilize the Distributed Deep Deterministic Policy Gradient (D4PG) as our core strategy to predict thigh gait and enhance the power of hip joint rotation through dynamic assistance, aiming to create a product that reduces lower limb labor burden and strengthens sports protection for everyday use. In this process, we will integrate various data, including EMG signals from 8 muscle channels, angles, velocities, and accelerations of 2 joints, along with data from 3 IMU sensors. The processing of EMG signals is especially critical for real-time control and marks our biggest difference from previous exoskeleton control systems. Due to its personalized characteristics, we opt for a Deep Learning (DL) architecture, expecting it to outperform traditional dynamics models. We also hope to address the technical challenges faced by traditional whole-body dynamics models in handling rapid motion transitions. This data-driven approach, which does not rely on complex mathematical modeling (Model-free), could potentially resolve the issues that traditional dynamics models face in adapting to real-time or rapidly changing scenarios. Learning directly from data allows for more flexible adaptation to rapid changes in motion and provides more accurate control, which is our core philosophy for this algorithm.

D4PG is a recently proposed reinforcement learning strategy that extends the basic concept of Deep Deterministic Policy Gradient (DDPG) by introducing a distributed framework and a new method for estimating return distributions. Reinforcement learning (RL) has unique advantages over traditional supervised deep learning architectures. Supervised learning depends on a large dataset of labeled data for training, and the process of manual labeling can be limiting when robots learn complex, dynamically changing tasks. In contrast, reinforcement learning learns through interaction with the environment without the need for a pre-labeled dataset. It allows robots to autonomously explore and learn from successes and failures, finding the optimal strategy through a trial-and-error process.

Brief review of traditional RL methods: Policy Gradient and Q-learning

The concepts of Actor and Environment mentioned above can actually be described through the following mathematics. The Actor determines its behavior in the environment through a policy function π, which can be deterministic or stochastic. A deterministic policy provides a specific action a for each state s, acting like a function $a = \pi(s)$, whereas a stochastic policy provides a probability for each possible action, in the form $P(a \mid s) = \pi(a \mid s)$. In policy gradient methods, the probability of a trajectory:

$$P(\tau) = p(s_0)\prod_{t=0}^{T}\pi(a_t|s_t)p(s_{t+1}|s_t, a_t)$$

describing the probability of generating a complete trajectory τ under the premise that the actor follows policy π . A trajectory is a sequence of states and actions, including the environment's response to the actor's actions, i.e., the state transition probability $p(s_{t+1}|s_t, a_t)$. To maximize the expected return, there is a method called Proximal Policy Optimization (PPO) :

$$\nabla_\theta J(\theta) = E_{s\sim\rho^\pi, a\sim\pi}[Q^\pi(s,a)\nabla_\theta \log\pi\,(a|s;\theta)]$$

Where $Q^\pi(s,a)$ is the value function determined by the environment, evaluating the expected return of taking action a in state s. The process of updating weights is essentially the opposite of weight updating in traditional DL classification problems (adding a negative sign). The policy performance function $J(\theta)$ is the expected value of rewards obtained under policy π, represented by the formula:

$$\mathrm{J}(\theta) = \mathrm{E}_{\tau\sim\pi_\theta}[R(\tau)],$$

$R(\tau)$ represents the reward of the trajectory, and π_θ represents the policy defined by parameters θ. this estimated gradient is used to update the policy parameters θ through the gradient ascent method, where α is the learning rate:

$$\theta \leftarrow \theta + \alpha\nabla\theta J(\theta)$$
$$\nabla_\theta J(\theta) = E_{\tau\sim\pi_\theta}[R(\tau)\nabla_\theta log\pi_\theta(\tau)]$$

Since directly calculating the expected value is difficult, an approximation of the expected value is made by sampling:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N} R(\tau^n)\nabla_\theta log\pi_\theta(\tau^n)$$

$\nabla_\theta log\pi_\theta(\tau^n)$, n reality, this logarithmic gradient can be decomposed into the sum of the logarithmic gradients of each step's action, since the choice of policy at each step is independent. Therefore, we can rewrite this expression as the sum of the logarithmic gradients of action choices at each moment, i.e $\sum_{t=1}^{T^n}\nabla_\theta \log\pi_\theta\,(a_t^n|s_t^n)$.

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N}\left(\sum_{t=1}^{T^n} R(t)\nabla_\theta \log\pi_\theta\,(a_t^n|s_t^n)\right)$$

If each state transition implicitly assumes the Markov property defined in the Markov Decision Process(MDP):

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

$$\nabla_\theta J(\theta) = E\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta (a_t|s_t) G_t\right]$$

PPO optimizes directly in the policy space, aiming to improve the policy itself directly. However, if one first learns a value function and then derives the optimal policy based on this value function, this method, which directly estimates $Q(s,a)$ values for learning, is called Q-learning. The relationship between the value function and the Q-function can be expressed as follows: The value of a state s under policy π is equal to the Q-value obtained by taking the action recommended by policy π in state s.

$$V^\pi(s) = Q^\pi\big(s, \pi(s)\big)$$

We can prove that when

$$\pi'(s) = \arg\max_a Q^\pi (s,a), \text{ then } V^{\pi'}(s) \geq V^\pi(s) \text{ for all states}$$

$$V^\pi(s) \leq \arg\max_a Q^\pi (s,a) = Q^\pi\big(s, \pi'(s)\big), \text{ then}$$

$$\begin{aligned} V^\pi(s) &\leq E[r_{t+1} + V^\pi(s_{t+1}) | s_t = s, a_t = \pi'(s_t)] \\ &\leq E\big[r_{t+1} + Q^\pi\big(s_{t+1}, \pi'(s_{t+1})\big) \big| s_t = s, a_t = \pi'(s_t)\big] \\ &= E[r_{t+1} + r_{t+2} + \gamma V^\pi(s_{t+2}) | \ldots] \leq E\big[r_{t+1} + r_{t+2} + \gamma Q^\pi\big(s_{t+2}, \pi'(s_{t+2})\big) \big| \ldots\big] \ldots \\ &\leq V^{\pi'}(s) \end{aligned}$$

By subtracting the current estimate of $Q(s,a)$ from the Temporal Difference Error ($TD\ Error$), we are able to evaluate the discrepancy between the actual observed reward (including immediate rewards and estimated future returns) and the initially estimated reward.

$$TDError = \left(R(s,a) + \gamma \max_{a'} Q(s', a')\right) - Q(s,a)$$

And we get bellman equation that can be used to iteratively update the Q-value, thereby continuously approaching the optimal policy.

$$Q^{new}(s,a) \leftarrow Q(s,a) + \alpha \left[R(s,a) + \gamma \max_{a'} Q(s', a') - Q(s,a)\right]$$

<u>From Q-learning to Deep Deterministic Policy Gradient (DDPG)</u>

In traditional reinforcement learning approaches, the Q-function is represented in a tabular form, which becomes impractical for problems involving large state spaces. Consequently, Deep Neural Networks (DNNs) have been introduced to facilitate the mapping from high-dimensional perceptual inputs to the action space, effectively bridging complex inputs to discrete action outputs. For more sophisticated mappings that cater to continuous action spaces, the Deep Deterministic Policy Gradient (DDPG) algorithm is relied upon. This advancement allows for the handling of complex and high-dimensional environments that were previously challenging for conventional Q-learning algorithms. The mathematical structure of learnable neurons in artificial neural networks (NN) will be detailed in the "Source Code 2.1.5" section of the document. Let's first introduce the difference between Deep Q-learning and the backpropagation algorithm in a typical neural network regarding the calculation of the loss function. Neural networks can be used to approximate the value function Q of state-action pairs. In TD(0) evaluation, the loss function for this value function is:

$$J(\theta) = (TDError)^2$$

$$J(\theta) = \left(r_t + \gamma\hat{Q}(s_{t+1}, a_{t+1}; \theta^-) - \hat{Q}(s_t, a_t; \theta)\right)^2$$

DQN employs certain features, including experience replay and fixed target Q-networks, to address the instability and data sample correlation issues during the learning process. While these techniques enhance the stability and convergence of the algorithm, they also introduce additional complexity and latency, potentially impacting the real-time performance and applicability of the algorithm. By separating the actor to directly learn the policy and the critic to evaluate the value of actions, Actor-Critic can effectively handle problems with continuous action spaces. DDPG is an Actor-Critic method designed for continuous action spaces, making it applicable for tasks such as robotic control. To achieve this, we need to modify the formula stated above.

For the actor net, We instead utilize the gradient of the Q-function to indirectly achieve the same objective. Since the optimization goal of the Q-function is to make $Q(s, a|\theta^Q)$, approach the actual Q^π. by maximizing the Q-function, we indirectly optimize the policy to obtain the maximum G_t. We also adopt a deterministic policy NN $\mu(s|\theta^\mu)$, where for a given state s, the policy directly outputs a specific action a, rather than providing a probability distribution over actions as in traditional policy gradient methods. This approach enables the algorithm to select actions directly in continuous action spaces without the need for sampling to determine the action to be executed.

$$\nabla_{\theta^\mu} J \approx E_{s\sim\rho^\beta}\left[\nabla_a Q(s, a|\theta^Q)|_{a=\mu(s|\theta^\mu)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)\right]$$

In addition, in DDPG, due to the use of deterministic policies, the Ornstein-Uhlenbeck (OU) process is commonly employed to generate temporally correlated noise. This noise not only facilitates exploration but also, due to its temporal correlation, can smooth out the exploration actions over time, which is particularly important for control problems, especially in exoskeleton motors where coherence between actions is required.

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t$$

- x_t represents the current noise value, indicating the noise level at time t.
- θ is the rate parameter, determining how quickly the noise returns to its long-term mean μ. A higher value results in a faster return of noise to its mean.
- μ is the long-term mean around which the Ornstein-Uhlenbeck (OU) process fluctuates. It serves as the center or equilibrium point of the noise, around which the noise values fluctuate
- σ is the scale parameter controlling the amplitude of the noise, i.e., the magnitude of noise fluctuations. A higher value leads to larger noise generation and a broader exploration range.
- dW_t is the standard Wiener process (i.e., Brownian motion), representing the stochastic component of random fluctuations. It is a random process used to simulate random fluctuations in noise.

As for the differences between the Critic and DQN, the action is derived from the actor network. Further distinctions related to iteration and sampling will be explained in the "Source Code 2.1.5".

$$L(\theta^c) = E_{(s,a,r,s')\sim D}\left[\left(r + \gamma Q_{\theta^{c-}}\left(s', \mu_{\theta^{\mu-}}(s')\right) - Q_{\theta^c}(s,a)\right)^2\right]$$

<u>Optimizing the Distribution of Expected Returns with D4PG: Enhancing Robustness and Effectiveness of DDPG in Continuous Action Spaces</u>

D4PG addresses potential issues arising from deterministic policies by introducing distributed Q functions. Such issues include insufficient exploration due to overly deterministic action selection and oversimplified estimation of environmental rewards, failing to capture the dynamics and uncertainties of the environment adequately. Below is the mathematical representation and solution approach of employing distributed value functions in D4PG. Distributed Q functions is a distribution concerning the returns. This distribution can be described by a series of probability masses on fixed support points, where each support point represents a possible return value, and the corresponding probability indicates the likelihood of that return value. The maximum and minimum values of the return will be predefined.

$$z_i = V_{\min} + i \cdot \Delta z \quad \text{for} \quad i = 0,1,\ldots,N-1$$

$$\Delta z = \frac{V_{\max} - V_{\min}}{N-1}$$

$$tz_j = \min\left(V_{\max}, \max\left(V_{\min}, r + \gamma z_j\right)\right)$$

The interpolation weights for the target distribution's support points are calculated relative to a predefined set of support points, thereby identifying the two nearest support points for accurate mapping:

$$b_j = \frac{tz_j - V_{min}}{\Delta z}$$

$$l = \lfloor b_j \rfloor, u = \lceil b_j \rceil$$

$$L(\theta^c) = \text{Distance(ProjDistr,PredDistr)}$$

$$L(\theta c) = E(s,a,r,s') \sim D[Distance\left(Proj[R(s,a) + \gamma Q\theta c - \left(s', \mu\theta\mu - (s')\right)], Q\theta c(s,a)\right)]$$

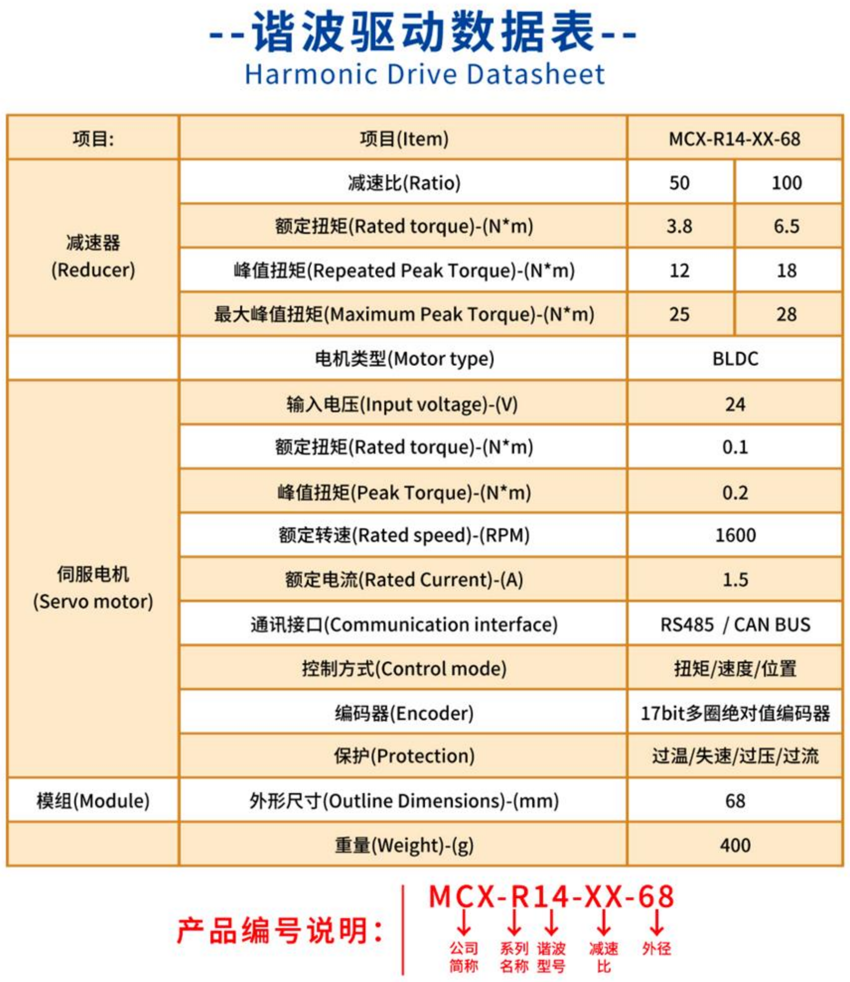
The aforementioned delineates the fundamental essence of the D4PG algorithm employed for model optimization. Within the framework of our model-free exoskeleton reinforcement system, we endeavor to directly discern the mapping relation from physiological signals to actions. This endeavor enables us to perpetually refine and ultimately synthesize precise individual gait patterns. Such an approach obviates the necessity for a priori model knowledge, thereby facilitating the provision of robust assistance. Concurrently, it facilitates the realization of personalized exoskeleton functionality. Moreover, this method adeptly circumvents the longstanding challenge encountered by conventional exoskeleton control systems—namely, their incapacity to swiftly adapt to dynamic movement transitions. The forthcoming section, "Source Code 2.1.5," will undertake an in-depth exploration of the concrete realization of these conceptual tenets, encompassing both the hierarchical architecture of neural networks and the requisite data structures essential for iterative updates.

1.4.2. Protocol

1.4.3. Flow-chart

1.5. AC/DC Specification

R14 Motor Harmonic Reducer Encoder Antriebs platine großes hohles Servo integriertes Verbindungs modul, Untersetzung verhältnis 50 oder 100



--谐波驱动数据表--
Harmonic Drive Datasheet

项目:	项目(Item)	MCX-R14-XX-68	
减速器 (Reducer)	减速比(Ratio)	50	100
	额定扭矩(Rated torque)-(N*m)	3.8	6.5
	峰值扭矩(Repeated Peak Torque)-(N*m)	12	18
	最大峰值扭矩(Maximum Peak Torque)-(N*m)	25	28
	电机类型(Motor type)	BLDC	
伺服电机 (Servo motor)	输入电压(Input voltage)-(V)	24	
	额定扭矩(Rated torque)-(N*m)	0.1	
	峰值扭矩(Peak Torque)-(N*m)	0.2	
	额定转速(Rated speed)-(RPM)	1600	
	额定电流(Rated Current)-(A)	1.5	
	通讯接口(Communication interface)	RS485 / CAN BUS	
	控制方式(Control mode)	扭矩/速度/位置	
	编码器(Encoder)	17bit多圈绝对值编码器	
	保护(Protection)	过温/失速/过压/过流	
模组(Module)	外形尺寸(Outline Dimensions)-(mm)	68	
	重量(Weight)-(g)	400	

产品编号说明: MCX-R14-XX-68

1.6. *Target System Specification*

1.7. *Timing Diagram*

1.7.1. *ESP32*

1.7.2. *Cygnus*

The Cygnus system emerges as an innovative solution designed to bridge the intricate gap between human biosignals and the responsive action of exoskeleton technologies. This integrated system, comprising both advanced software and hardware components, stands at the forefront of biomedical engineering and assistive robotics.

At its core, the Cygnus system specializes in the precise acquisition and interpretation of electromyographic (EMG) signals. These biosignals, which are electrical activities generated by muscle contractions, are captured via state-of-the-art sensors that boast high sensitivity and noise-resistant capabilities. The software suite that accompanies the Cygnus hardware is adept at processing these signals in real-time, ensuring a seamless transition from human intention to mechanical motion.

The potential applications of the Cygnus system are vast and impactful. By leveraging its system within the realm of exoskeletal development, it becomes the crucial link for individuals requiring rehabilitative support or enhanced physical capabilities. The Cygnus system provides the necessary interface to translate biological muscle signals into the precise and controlled movement of an exoskeleton, granting wearers a new level of autonomy and efficiency.

With the Cygnus system integrated into an exoskeleton framework, users can anticipate a symbiotic relationship between their movements and the device. This integration promises to deliver not only a transformative experience for the user but also paves the way for groundbreaking advancements in assistive technology.

Embrace the future of human-robotic integration with the Cygnus system, where the synergy of human will and robotic precision becomes a reality, unlocking new possibilities in mobility and empowerment.

1.7.3. Meta Quest

Embark on a journey into the next frontier of immersive experience with the Meta Quest 3, the latest innovation from the trailblazers in virtual reality. The Quest 3 is not just a VR headset; it's a versatile tool that unlocks new potentials for interaction within digital spaces.

At the heart of Quest 3's transformative experience lies its state-of-the-art passthrough technology. This feature seamlessly blends the physical and virtual realms, allowing users to engage with their environment and a digital overlay simultaneously. This breakthrough is perfect for applications requiring a harmonious mix of real-world context and virtual augmentation, such as interactive sessions with a virtual coach.

Equipped with high-fidelity sensors and advanced computing power, the Quest 3 delivers a clear, low-latency view of the world, enhancing user confidence and comfort during extended sessions of virtual training. The passthrough mode is ingeniously designed to facilitate a safe and natural experience as users navigate both their physical surroundings and virtual interfaces.

The Meta Quest 3 sets a new standard for virtual reality, offering unparalleled freedom and opportunities for creators and users alike. It is poised to revolutionize the way we learn, train, and interact with digital content, making the fusion of our reality with the virtual more intuitive than ever before.

Prepare to witness the transformation of virtual coaching and training environments with Meta Quest 3, where the boundaries of reality are reimagined, and every interaction is an opportunity for innovation and growth.

1.8. User manual

<u>Wearing exoskeletons and EMG patches</u>

For optimal deployment of the system, involving both the exoskeleton and the EMG patches, a collaborative approach is recommended, starting with the exoskeleton for the hips. This preliminary step facilitates the subsequent application of EMG patches. The exoskeleton should be worn with its sponge pads making direct contact with the user's back, ensuring that the dual motors align precisely with the hips. Upon alignment, fasten the waist belt and complete the fixation process. The device extends metal rods that must sit flush on the thighs, held firmly with elastic bands to mitigate any displacement during ambulation.

The attachment of EMG patches, the subsequent phase, necessitates meticulous placement. The patches must adhere to the identical muscle group, maintaining a set separation—commonly the span of two to three fingers. Patches are to be affixed to the anterior tibialis and gastrocnemius on both legs, plus the rectus femoris and vastus lateralis on the thighs, with a duo of patches per muscle, resulting in a total of sixteen patches. The anterior tibialis, located on the lateral aspect of the shin, and the gastrocnemius, characterized by two fan-like muscles apparent upon calf contraction, are relatively accessible for patch placement. The latter pair, situated on the thigh, require additional steps for identification; the rectus femoris is discernible as the middle muscle of the three visible when the leg is fully extended. For the vastus lateralis, the user should assume a prone position on a chair and attempt to raise the lower leg, wherein an assistant will provide counterforce to prevent elevation. This maneuver reveals the vastus lateralis on the posterior lateral aspect of the thigh.

Instructions for weaking VR glasses and using the interface

Users will don the Meta Quest 3 VR headset and activate the passthrough mode to achieve an augmented reality (AR) effect. Upon launching our Unity application, the user interface presents two exercise modes: walking on flat ground and standard squatting movements. Interaction with the interface is facilitated through the original controllers provided by the manufacturer. Users can select options by pointing the controller at buttons on the interface.For both movement modes, there are two buttons available for selection: one for action cue sounds and another for a virtual coach demonstration. The former plays correct movement cues, allowing users to pause momentarily by pressing the button, with a subsequent press resuming playback. The latter manifests a three-dimensional human model in the real world, demonstrating the correct movements. Users can observe how to perform the movements accurately by pressing the button, with another press pausing the virtual coach's demonstration.To ensure the interface does not obstruct the user’s view during exercise, we have also developed a window scaling feature. By pressing the 'B' button on the controller, users can adjust the interface to their preference.This integration aims to enhance user experience by combining physical exercise with interactive, virtual guidance, offering a seamless blend of technology and fitness.

2. *Detailed Design Data Files*

2.1. *Soure code*

2.1.1. *ESP32*

Using the Arduino framework, we've engineered a sophisticated program for an exoskeleton system. This program not only sets up Wi-Fi communication through the ESP32 S3 NodeMCU but also establishes a data exchange link with the STM32F415 microcontroller via UART(the STM32F415 controller's direct management of a crucial component of the exoskeleton—the R14 Motor Harmonic Reducer Encoder Antriebsplatine, a large hollow servo integrated connection module). The pivotal aspects of this program involve launching a server that clients can connect to over Wi-Fi and managing two concurrent tasks: one for processing commands received from Wi-Fi clients and another for fetching data from the STM32F415.

The adoption of a dual-core approach is a strategic choice, motivated by extensive testing that revealed the necessity to handle signal transmission and reception at a frequency of 20Hz. This frequency requirement necessitates a high level of responsiveness and efficiency, which is achieved by leveraging the dual-core capabilities of the ESP32 S3 NodeMCU.

Wi-Fi Server Setup

- The Wi-Fi is initialized in AP (Access Point) mode with the SSID "MyESP32AP" and password "12345678".
- The IP address is configured to 192.168.4.1, with the corresponding gateway and subnet mask settings.
- A Wi-Fi server is launched to listen for connections on port 8080.

UART Communication

The program uses the ESP32's second serial port (a hardware serial communication port), connected to the STM32F415 via GPIO 16 and GPIO 17, with the baud rate set to 115200.

Task Creation and Management

Using FreeRTOS, tasks are created on both cores of the ESP32:

- handleCommandsTask on core 0 processes commands from Wi-Fi clients
- getAndSendHipInfoTask on core 1 receives data from the STM32F415 via UART and sends it to the Wi-Fi client.

Processing Wi-Fi Client Commands

When a command is received from a Wi-Fi client, the command function reads the data and sends the received command string to the STM32F415 via UART, enabling remote control.

Receiving and Sending Data

The getHip_INFO function is responsible for receiving data from the STM32F415 and transmitting this data to connected clients over Wi-Fi, providing real-time updates on the exoskeleton's status or sensor readings.

2.1.1.1. platform.ini

```
[env:firebeetle32]
platform = espressif32
board = esp32doit-devkit-v1
framework = arduino
monitor_speed = 115200
```

2.1.1.2. AP_server_version.cpp

```
#include <Arduino.h>
#include <WiFi.h>
```

```
const char* ssid = "MyESP32AP";
const char* password = "12345678";
WiFiServer server(8080);
HardwareSerial MySerial(2); // second port
const int bufSize = 30;
char buf[bufSize];
int buf_index = 0;
String cmd_str;
WiFiClient client;

void command();
void getHip_INFO();
void handleCommandsTask(void* pvParameters);
void getAndSendHipInfoTask(void* pvParameters);

void setup() {
  // Init
  Serial.begin(115200);
  delay(1000);
  IPAddress IP(192, 168, 4, 1);
  IPAddress gateway(192, 168, 4, 1);
  IPAddress subnet(255, 255, 255, 0);
  WiFi.softAPConfig(IP, gateway, subnet);
  //Connecting
  WiFi.softAP(ssid, password);
  Serial.print("AP IP address: ");
  Serial.println(WiFi.softAPIP());
  server.begin();
  MySerial.begin(115200, SERIAL_8N1, 16, 17);
  Serial.println("Serial at pin is ready!");

  xTaskCreatePinnedToCore(handleCommandsTask, "HandleCommands", 10000,
NULL, 1, NULL, 0); // 核心 0
  xTaskCreatePinnedToCore(getAndSendHipInfoTask, "GetAndSendHipInfo",
10000, NULL, 1, NULL, 1); // 核心 1
  Serial.println("Multi-core ready!");
}
```

```
void loop() {
  if (!client || !client.connected()) {
    client = server.available();
    if (client) {
      Serial.println("Client connected!");
    }
  }
  vTaskDelay(pdMS_TO_TICKS(1000));
}

void handleCommandsTask(void* pvParameters) {
  while (true) {
    if (client && client.connected()) {
      command();
    }
    delay(10); // 避免 CPU 佔用率過高
  }
}

void getAndSendHipInfoTask(void* pvParameters) {
  while (true) {
    if (client && client.connected()) {
      getHip_INFO();
    }
    delay(10); // 避免 CPU 佔用率過高
  }
}

void command() {
    while (client.available()) {
        char c = client.read();
        if (c == '\0') {
            if (cmd_str.length() > 0) {
                Serial.print("received data from PC: ");
                MySerial.print(cmd_str);
                Serial.println(cmd_str);
                // client.write((const uint8_t *)cmd_str.c_str(),
cmd_str.length());
                cmd_str = "";
            }
            }
        else {
            cmd_str += c;
        }
    }
}

void getHip_INFO() {
  const int bufferSize = 1024;
  static char buffer[bufferSize];
  static int index = 0;

  while (MySerial.available()) {
    buffer[index] = MySerial.read();
    if (buffer[index] == '\n' || index == bufferSize - 2) {
      buffer[index + 1] = '\0';
      // Serial.println("From HIP: ");
      Serial.println(buffer);
      client.println(buffer);
      index = 0;
      break;
    } else {
      index++;
    }
  }
}
```

2.1.2. client_order.py:

This script, running on the PC side, is a Python script for low-level communication, designed to control and monitor a wearable exoskeleton device. Moreover, it integrates a custom library to receive data from Cygnus EMG patch sensors. The script establishes a TCP connection with the exoskeleton device's open AP through network sockets, sends control commands, and processes data returned from the exoskeleton device. Additionally, by incorporating a specific library, the script can also read EMG data in real-time. This data, collected from Cygnus EMG patch sensors worn by the user, is crucial for further analysis and improvement of the exoskeleton's control strategies.

- **Connection Establishment with the Exoskeleton Device:**

  Utilizing the connect_FREEX function, the script establishes a TCP connection with the exoskeleton device through a specified IP address and port number, enabling remote control and monitoring of the device's status.
- **Receiving and Parsing Data:**

  Through the read_line and analysis functions, the script receives and parses data from the exoskeleton device. This data includes information on the device's angle, speed, and current, which are vital for ensuring the safety and efficiency of the exoskeleton's operations.
- **Integration of Cygnus EMG Patch Data:**

  The script integrates the function to receive data from Cygnus EMG patch sensors through a custom library, emg_nonasync. In the get_INFO function, besides processing data from the exoskeleton, it also calls this library function to obtain EMG data in real-time. This data is invaluable for analyzing the wearer's muscle activity patterns and adjusting the exoskeleton's response accordingly.
- **Safety Checks:**

  Before sending any control commands, the if_not_safe function performs safety checks on the exoskeleton's current state based on predetermined safety limits, such as angle and current restrictions, to avoid potential dangers or discomfort.
- **Sending Action Commands:**

  The send_action_to_exoskeleton function decides on the appropriate action commands based on the current state of the exoskeleton and the data received from the EMG sensors. This includes adjusting the size and direction of the movements and stopping the action if necessary to ensure safety.

```python
import socket
import numpy as np
from EMG import emg_nonasync

def analysis(data):
    result = []
    if data.startswith("X"):
        parts = data[1:].strip().split()
        count = 0
        for part in parts:
            if count == 9:
                break
            clean_part = ''.join(filter(lambda x: x in '0123456789.-',
part))
            if clean_part and clean_part != '-' and not
clean_part.endswith('.'):
                try:
                    result.append(float(clean_part))
                    count += 1
                except ValueError as e:
                    print(f"Error converting '{clean_part}' to float:
{e}")
                    continue

        if len(result) == 9:
            return np.array(result), True
    # print(f"Failed to analyze data: {data}")
    return np.zeros(9), False

def FREEX_CMD(sock, mode1="E", value1="0", mode2="E", value2="0"):
    cmd_str = f"X {mode1} {value1} {mode2} {value2}\r\n\0"
    cmd_bytes = cmd_str.encode('ascii')
    try:
        sock.send(cmd_bytes)
    except Exception as e:
        FREEX_CMD(sock, "E", "0", "E", "0")
        print(f"Error when sending: {e}")

def connect_FREEX(host='192.168.4.1', port=8080):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((host, port))
    print(f"Successfully connected to {host}:{port}")
    return sock

def read_line(sock):
    try:
        data = sock.recv(1024)
        if not data:
            return None
        data = data.decode('ascii').rstrip('\r\n\0')
        return data
    except Exception as e:
        FREEX_CMD(sock, "E", "0", "E", "0")
        print(f"Error when reading_line: {e}")
        return None

def get_INFO(sock, uri, bp_parameter, nt_parameter, lp_parameter):
    while True:
        info = read_line(sock)
        if info is None or info == "":
            FREEX_CMD(sock, "E", "0", "E", "0")
            print("stucking in EXO data failed")
            continue
        # print("raw_data: ", info)
        analyzed_data, is_analyzed = analysis(info)
        if is_analyzed:
            break
        else:
            FREEX_CMD(sock, "E", "0", "E", "0")
    # print("analyzed: ", analyzed_data)
    # analyzed_data = np.random.rand(9)
    # emg
    emg_observation, bp_parameter, nt_parameter, lp_parameter =
emg_nonasync.read_specific_data_from_websocket(uri ,bp_parameter,
nt_parameter, lp_parameter)

    return analyzed_data, emg_observation, bp_parameter, nt_parameter,
lp_parameter

def if_not_safe(limit, angle, speed):
    # if (angle >= limit and speed > 0) or (angle <= -limit and speed <
0):
    if (angle >= limit) or (angle <= -limit):
        return True
    else:
        return False

last_action_was_zero = False
```

```python
left_disabled = False
right_disabled = False

def send_action_to_exoskeleton_speed(writer, action, state):
    global last_action_was_zero
    action[0] *= 10000  # Scale the action for the right side
    action[1] *= 10000  # Scale the action for the left side
    LIMIT = 10
    CURRENT_LIMIT = 50000
    R_angle, L_angle = state[0], state[3]
    R_current, L_current = state[2], state[5]
    current_action_is_zero = all(a == 0 for a in action)

    if current_action_is_zero and last_action_was_zero:
        return
    # print(f"action: {action}, angle: {R_angle}, {L_angle}, current:
{R_current}, {L_current}")
    check_R = if_not_safe(LIMIT, R_angle, action[0])
    check_L = if_not_safe(LIMIT, L_angle, action[1])

    if check_R and check_L:
        # print("both actions aborted due to safety")
        FREEX_CMD(writer, "E", "0", "E", "0")
    elif check_R:
        # print("Right action aborted due to safety")
        FREEX_CMD(writer, "E", "0", 'C', f"{action[1]}" if not check_L
else "0")
    elif check_L:
        # print("Left action aborted due to safety")
        FREEX_CMD(writer, 'C', f"{action[0]}" if not check_R else "0",
"E", "0")
    else:
        FREEX_CMD(writer, 'C', f"{action[0]}", 'C', f"{action[1]}")

    last_action_was_zero = current_action_is_zero
    print("------------------------------")
def send_action_to_exoskeleton(writer, action, state,
control_type='speed'):
    if control_type == 'speed':
        return send_action_to_exoskeleton_speed(writer, action, state)
    elif control_type == 'disable':
        pass
    else:
        raise ValueError("Unknown control_type specified.")
```

2.1.3. emg_nonasync.py

This code implements a process for real-time reception and processing of electromyography (EMG) signals from Cygnus EMG patch sensors via a WebSocket connection. It involves signal acquisition, filtering, rectification, and final analysis, aimed at providing real-time EMG feedback during muscle strength training or monitoring. Cygnus integrates data from 16 channels and connects to the PC program via Bluetooth, transmitting the data to the local network. Below are detailed explanations of the main components of the code:

WebSocket Data Reception

The read_specific_data_from_websocket function establishes a real-time WebSocket connection to a specified URI address, continuously receiving EMG data until a valid EMG array is successfully retrieved.

Data Processing

The received data is first processed by the process_data_from_websocket function. This function parses the JSON-formatted data, extracting the EMG signals and applying a series of signal processing steps to each batch of signals.

The processing includes bandpass filtering (to remove noise), notch filtering (to eliminate power line interference), full-wave rectification (to ensure the signal has only positive values), and low-pass filtering (to extract the signal envelope).

Signal Processing Functions

bandpass_filter, notch_filter, full_wave_rectification, and lowpass_filter are specific signal processing functions used to implement the aforementioned filtering and rectification operations.The process_emg_signal function integrates all the signal processing steps mentioned above, applying comprehensive preprocessing to each batch of EMG data.

Muscle Strength Level Calculation

The calculate_emg_level function calculates muscle strength levels based on the processed signals. It maps the root mean square (RMS) values of the signal to different muscle strength levels, thus providing instant muscle strength feedback. During training or monitoring, this level can be used to assess the user's muscle activity, aiding in adjusting training intensity or evaluating muscle fatigue.

Incorporating data from Cygnus, which combines 16 channels and connects via Bluetooth to transmit data to the local network, enhances the utility and application range of this script, making it a comprehensive solution for real-time EMG signal analysis in applications such as muscle strength training, rehabilitation therapy, and biofeedback.

```
import numpy as np
from scipy.signal import butter, lfilter, iirnotch, lfilter_zi
import websocket
import json

def read_specific_data_from_websocket(uri, bp_parameter, nt_parameter,
lp_parameter):
        try:
            ws = websocket.WebSocket()
            ws.connect(uri)
            while True:
                data = ws.recv()
                emg_array, bp_parameter, nt_parameter, lp_parameter =
process_data_from_websocket(data, bp_parameter, nt_parameter,
lp_parameter)
                if emg_array.shape[0] != 0:
                    return emg_array, bp_parameter, nt_parameter,
lp_parameter
        except Exception as e:
            print(f"WebSocket error: {e}")
            pass
        # finally:
        #     ws.close()

def process_data_from_websocket(data, bp_parameter, nt_parameter,
lp_parameter):
    emg_values = np.zeros((8,50))
    j = 0
    try:
        data_dict = json.loads(data)
        if "contents" in data_dict:
            # 提取 serial_number 和 eeg 的值
            serial_numbers_eegs = [(item['serial_number'][0], item['eeg'])
for item in data_dict['contents']]
```

```
            # 輸出結果
            for serial_number, eeg in serial_numbers_eegs:
                # print(f"Serial Number: {serial_number}, EEG: {eeg}")
                for i in range(8):
                    emg_values[i,j] = eeg[i]      # 最新的 50 筆 emg 資料
                j+=1
            try:
                emg_array = np.empty((8, 50))
                for k in range(8):
                    #print("check2",emg_values[k],bp_parameter[k],
nt_parameter[k], lp_parameter[k])
                    emg_array[k], bp_parameter[k], nt_parameter[k],
lp_parameter[k] = process_emg_signal(emg_values[k],bp_parameter[k],
nt_parameter[k], lp_parameter[k])
                    #print("check5",emg_values[k],bp_parameter[k],
nt_parameter[k], lp_parameter[k])
                return emg_array, bp_parameter, nt_parameter, lp_parameter
            except Exception as e:
                print(f"處理信號時發生錯誤: {e}")
                return np.array([]), bp_parameter, nt_parameter,
lp_parameter
    except json.JSONDecodeError:
        print("Failed to decode JSON from WebSocket")
    except Exception as e:
        # print(f"Error processing data from WebSocket: {e}")
        return np.array([]), bp_parameter, nt_parameter, lp_parameter

# 帶通濾波器設計
def bandpass_filter(data, lowcut, highcut, fs, bp_filter_state, order=4):
    nyq = 0.5 * fs
    low = lowcut / nyq
    high = highcut / nyq
    b, a = butter(order, [low, high], btype='band')
    if bp_filter_state.all() == 0:
        bp_filter_state = lfilter_zi(b, a)
        #print("check4", bp_filter_state)
    y, bp_filter_state = lfilter(b, a, data, zi=bp_filter_state)
    return y, bp_filter_state
```

```python
# 陷波濾波器設計
def notch_filter(data, notch_freq, fs, notch_filter_state,
quality_factor=30):
    nyq = 0.5 * fs
    freq = notch_freq / nyq
    b, a = iirnotch(freq, quality_factor)
    if notch_filter_state.all() == 0:
        notch_filter_state = lfilter_zi(b, a)
        #print("check5", notch_filter_state)
    y, notch_filter_state = lfilter(b, a, data, zi=notch_filter_state)
    return y, notch_filter_state

# 全波整流
def full_wave_rectification(data):
    return np.abs(data)

# 低通濾波器設計（提取包絡）
def lowpass_filter(data, cutoff, fs, lp_filter_state, order=4):
    nyq = 0.5 * fs
    normal_cutoff = cutoff / nyq
    b, a = butter(order, normal_cutoff, btype='low', analog=False)
    if lp_filter_state.all() == 0:
        lp_filter_state = lfilter_zi(b, a)
        #print("check8", lp_filter_state)
    y, lp_filter_state = lfilter(b, a, data, zi=lp_filter_state)
    return y, lp_filter_state

# 即時信號處理函數
def process_emg_signal(data, bp_parameter, nt_parameter, lp_parameter,
fs=1000):
    # 帶通濾波
    bandpassed, bp_parameter = bandpass_filter(data, 20, 450, fs,
bp_parameter)
    # 50Hz 陷波濾波
    notch_filtered, nt_parameter = notch_filter(bandpassed, 50, fs,
nt_parameter)
    # 全波整流
    rectified = full_wave_rectification(notch_filtered)
    # 低通濾波提取包絡
    enveloped, lp_parameter = lowpass_filter(rectified, 10, fs, 
lp_parameter)

    return enveloped, bp_parameter, nt_parameter, lp_parameter
# 以下為肌力回饋
def calculate_emg_level(data, initial_max_min_rms_values, times, 
ta=20,rf=40,bf=25,Ga=15):
    #前 1 秒為暖機
    if times <= 1000:
        return 0, initial_max_min_rms_values
    # 使用第 1 秒到第 10 秒的資料來確定初始的最小、最大 RMS 值
    elif 1000 < times <= 5000:
        for i in range(8):
            rms_values = data[i]
            if initial_max_min_rms_values[i][0] == 0 or rms_values > 
initial_max_min_rms_values[i][0]:
                initial_max_min_rms_values[i][0] = rms_values
            elif initial_max_min_rms_values[i][1] == 0 or rms_values < 
initial_max_min_rms_values[i][1]:
                initial_max_min_rms_values[i][1] = rms_values
        return 0, initial_max_min_rms_values
    #每 0.05 秒傳出 reward 值
    else:
        reward = np.zeros(8)
        y = 0
        for i in range(8):
            rms_values = data[i]
            reward[i] = map_to_levels(rms_values, 
initial_max_min_rms_values[i])
        y = 
ta*reward[0]+rf*reward[1]+bf*reward[2]+Ga*reward[3]+ta*reward[4]+rf*rewar
d[5]+bf*reward[6]+Ga*reward[7]
        print("Total: ",y/200,"Reward: ",reward)
        return y/200, initial_max_min_rms_values

def calculate_rms(signal):
    "計算訊號的 RMS 值。"
    return np.sqrt(np.mean(signal**2))

def map_to_levels(value, max_min_rms_values):
    """將值映射到超出 5 到-5 級的線性值上，基於放鬆閾值和初始最大 RMS 值，
    但在上下限內分為 5 到-5 十個等級區間。"""
    # 計算每個等級的值範圍大小
    try:
        level_range = (max_min_rms_values[0] - max_min_rms_values[1]) / 10

        if value <= max_min_rms_values[1]:
            # 計算低於 min_rms_values 的值應映射到哪個級別
            level_diff = (max_min_rms_values[1] - value) / level_range
            return 5 + round(level_diff)
        elif value >= max_min_rms_values[0]:
            # 計算高於 max_rms_values 的值應映射到哪個級別
            level_diff = (value - max_min_rms_values[0]) / level_range
            return -5 - round(level_diff)
        else:
            # 線性映射到 5 到-5
            normalized_value = (value - max_min_rms_values[1]) /
(max_min_rms_values[0] - max_min_rms_values[1])
            return int(round(normalized_value * (-10))) + 5
    except Exception as e:
        print(f"計算 reward 發生錯誤: {e}, return 0")
        return 0
```

2.1.4. Env.py

To facilitate modular development in Reinforcement Learning (RL) using the `ptan` library and to save time developing frameworks such as Experience (Exp), this script wraps the functionality into an environment (ENV) compatible with the OpenAI Gym interface. It eliminates the need for asynchronous communication libraries like asyncio, proving that this level of network communication does not require such complexity. The script connects to an Exoskeleton device and Cygnus EMG patch sensors via Bluetooth, integrating the data into the local network for real-time processing. Here's how the code functions within this context:

<u>Environment Setup</u>

- Inherits from gym.Env, making it compatible with various RL algorithms and tools that utilize the Gym interface.
- Initializes communication with the Exoskeleton device through a specified IP address and WebSocket URI for real-time EMG data streaming.

Observation and Action Spaces

- Defines the observation space as a combination of device states (like angles, velocities, and currents from different joints, and IMU data) and processed EMG signals, allowing the RL model to make decisions based on a comprehensive view of the system's current state.
- The action space is defined to control the Exoskeleton device, with actions potentially representing desired movement velocities or positions for the exoskeleton limbs.

Step Function

- Handles the core logic for advancing the environment by one timestep based on an action provided by an RL agent.
- Sends control commands to the Exoskeleton device, processes incoming data (including EMG signals), calculates rewards based on the agent's actions, and checks whether the current episode has ended.

Reset Function

Resets the environment to a default state at the beginning of each episode, ensuring that learning starts from a consistent state. This includes resetting muscle power levels and re-establishing network connections if necessary.

Reward Calculation

Implements a method for calculating rewards based on the EMG signal processing, which could involve evaluating muscle activation levels, ensuring that movements are within safe and effective thresholds.

Rendering and Logging

Provides mechanisms for visualizing the state of the environment and logging detailed information about each timestep, which can be crucial for debugging and understanding the behavior of the RL agent.

Closing the Environment

Ensures that all connections are properly closed and that the device is returned to a safe state when the environment is closed, preventing potential harm to the user or damage to the equipment.

```python
from wifi_streaming import client_order
from EMG import emg_nonasync
import asyncio
import gym
from gym import spaces
import numpy as np
```

```
from tensorboardX import SummaryWriter
import time
import keyboard

channel_names = [
    'Tibialis_anterior_right',  # 通道 1: 右腿脛前肌
    'Rectus Femoris_right',     # 通道 2: 右腿股直肌
    'Biceps_femoris_right',     # 通道 3: 右腿股二頭肌
    'Gastrocnemius_right',      # 通道 4: 右腿腓腸肌
    'Tibialis_anterior_left',   # 通道 5: 左腿脛前肌
    'Rectus Femoris_left',      # 通道 6: 左腿股直肌
    'Biceps_femoris_left',      # 通道 7: 左腿股二頭肌
    'Gastrocnemius_left'        # 通道 8: 左腿腓腸肌
]


class ExoskeletonEnv(gym.Env):
    metadata = {'render.modes': ['human']}

    def __init__(self, log_writer , device='cpu', host='192.168.4.1', url=
"ws://localhost:31278/ws", port=8080):
        super(ExoskeletonEnv, self).__init__()
        self.device = device
        self.host = host
        self.port = port
        self.uri = url
        self.observation = np.zeros(9)
        self.emg_observation = np.zeros(8)
        self.filtered_emg_observation = np.zeros((8,50))
        self.bp_parameter = np.zeros((8,8))
        self.nt_parameter = np.zeros((8,2))
        self.lp_parameter = np.zeros((8,4))
        self.initial_max_min_rms_values = np.zeros((8,2))
        self.current_step = 0
        self.init_time = 0
        self.reward = 0
        self.sock = client_order.connect_FREEX(self.host, self.port)
        self.observation_space = spaces.Box(low=-np.inf, high=np.inf,
shape=(15,), dtype=np.float32)
```

```
        self.action_space = spaces.Box(low=-1, high=1, shape=(2,),
dtype=np.float32)
        self.log_writer = log_writer

    def step(self, action):
        # 改回用 send_action_to_exoskeleton_speed 函數
        self.observation, self.filtered_emg_observation,
self.bp_parameter, self.nt_parameter, self.lp_parameter =
client_order.get_INFO(self.sock, self.uri ,self.bp_parameter,
self.nt_parameter, self.lp_parameter)
        #window.update_plot(self.filtered_emg_observation[0])
        self.emg_observation =
np.sqrt(np.mean(self.filtered_emg_observation**2, axis=1))

        client_order.send_action_to_exoskeleton(self.sock, action,
self.observation ,"speed")
        self.reward = self.calculate_reward()
        done = self.check_if_done(self.observation)
        self.current_step += 1
        self.render()
        return np.concatenate([self.observation, self.emg_observation],
axis=0), self.reward, done, {}

    def reset(self, is_recording=True):
        client_order.FREEX_CMD(self.sock, "E", "0", "E", "0")
        time.sleep(1)
        if self.sock is not None:
            self.sock.close()
            self.sock = None
        print("disconnect")
        self.sock= client_order.connect_FREEX(self.host, self.port)
        print("re-connected")
        time.sleep(2)
        client_order.FREEX_CMD(self.sock, "A", "0000", "A", "0000")
        print("reset to angle, be relaxed")
        time.sleep(2)
        client_order.FREEX_CMD(self.sock, "E", "0", "E", "0")
        time.sleep(2)
        self.emg_observation = np.zeros(8)
        self.filtered_emg_observation = np.zeros((8,50))
        self.bp_parameter = np.zeros((8,8))
        self.nt_parameter = np.zeros((8,2))
        self.lp_parameter = np.zeros((8,4))
        if is_recording:
            self.recoding_for_power_level()
        else:
            self.observation, self.filtered_emg_observation,
self.bp_parameter, self.nt_parameter, self.lp_parameter =
client_order.get_INFO(self.sock, self.uri ,self.bp_parameter,
self.nt_parameter, self.lp_parameter)
            self.emg_observation =
np.sqrt(np.mean(self.filtered_emg_observation**2, axis=1))
        print("first data recv")
        return np.concatenate([self.observation, self.emg_observation],
axis=0)  #self.emg_observation 的格式
        # return np.zeros(15)

    def recoding_for_power_level(self):
        input("Press Enter to Reset Muscle Power Level, Please walk
naturally for about 10 seconds...")
        self.initial_max_min_rms_values = np.zeros((8,2))
        self.init_time = 0
        while self.init_time <= 5000:
            self.init_time = self.init_time +
50  #len(new_emg_observation)
            self.observation, self.filtered_emg_observation,
self.bp_parameter, self.nt_parameter, self.lp_parameter =
client_order.get_INFO(self.sock, self.uri ,self.bp_parameter,
self.nt_parameter, self.lp_parameter)
            self.emg_observation =
np.sqrt(np.mean(self.filtered_emg_observation**2, axis=1))
            self.calculate_reward()
            if self.init_time % 1000 == 0:
                print("Countdown: ",10 - int(round(self.init_time/1000)))

    def calculate_reward(self):
        reward, self.initial_max_min_rms_values = 
emg_nonasync.calculate_emg_level(self.emg_observation, 
self.initial_max_min_rms_values, self.init_time)
        return reward

    def check_if_done(self, observation):
        # Implement logic to check if the episode is done
        return False

    def render(self, mode='human', close=False):
        self.log_writer.add_scalars('Joint/Angle', {'Joint1': 
self.observation[0], 'Joint2': self.observation[3]}, self.current_step)
        self.log_writer.add_scalars('Joint/Velocity', {'Joint1': 
self.observation[1], 'Joint2': self.observation[4]}, self.current_step)
        self.log_writer.add_scalars('Joint/Current', {'Joint1': 
self.observation[2], 'Joint2': self.observation[5]}, self.current_step)
        self.log_writer.add_scalars('IMU', {'Roll': self.observation[6], 
'Pitch': self.observation[7], 'Yaw':self.observation[8]}, 
self.current_step)
        self.log_writer.add_scalar('Reward', self.reward, 
self.current_step)
        filtered_emg_step = self.current_step*50
        for i in range(self.emg_observation.shape[0]):
            for j in range(50):
                self.log_writer.add_scalar(f'Filtered_EMG/{channel_names[i
]}', self.filtered_emg_observation[i][j], filtered_emg_step+j)
            self.log_writer.add_scalar(f'sqrted 
EMG/Channel_{channel_names[i]}', self.emg_observation[i], 
self.current_step)

    def close(self):
        print("closing")
        client_order.FREEX_CMD(self.sock, "A", "0", "A", "0")
        time.sleep(2)
        client_order.FREEX_CMD(self.sock, "E", "0", "E", "0")
        time.sleep(0.05)
        self.sock.close()
        self.log_writer.close()
```

2.1.5. *models.py*

This code implements the core model architecture of the Deep Deterministic Policy Gradient (D4PG) reinforcement learning algorithm using PyTorch, with the goal of simulating the action of thigh gait through reinforcement learning techniques. The code consists of several parts, including the behavior model (Actor), evaluation model (Critic), agent (Agent), and batch data processing function. Here is a detailed explanation of each component:

DDPGActor

The DDPGActor model, acting as the behavior policy, generates actions a_t for observed states s_t. It is formalized as a neural network:

- Input layer: Accepts the environmental state vector$s_t \in R^{17}$, incorporating 8 muscle signals, 3 IMU readings, and 2 each for joint angles, velocities, and accelerations.
- Hidden layer: A linear transformation followed by a Tanh activation, $\mathrm{Tanh}(W_h s_t + b_h)$, where $W_h \in R^{17\times 20}$, and $b_h \in R^{20}$
- Output layer: Produces the action vector $a_t \in R^{2}$, corresponding to the velocities of left and right motors, through $W_o h + b_o$.

D4PGCritic

The D4PGCritic serves as the evaluation mechanism, assessing potential rewards for state-action pairs:

- Observation network: Transforms the state $s \in R^{obs\_size}$ to a latent representation through $ReLU(W_{obs} s + b_{obs}\}$ with $W_{obs} \in R^{obs\_size \times 400}$
- Merging layer: Concatenates the encoded state with the action vector, $[obs; a]$, for combined processing.
- Output network: Evaluates the action's value, outputting a distribution over n_atoms discrete values representing the action value distribution, $W_{out}[obs; a] + b_{out}$, where $\boldsymbol{W_{out}} \in R^{obs\_size \times 400 \times n\_atoms}$.

AgentD4PG

AgentD4PG determines actions based on the current state, incorporating exploration noise:

- It employs the DDPGActor to compute a_t from s_t, adding Gaussian noise scaled by ϵ to each action for exploration, $a_t' = a_t + \epsilon \cdot \mathcal{N}(0, I)$, and clips a_t' to $[-1, 1]$.

unpack_batch

The unpack_batch function is used for processing a batch of experiences, unpacking them into a format suitable for training. It extracts states, actions, rewards, end signals, and last states from the batch and converts them into PyTorch tensors.

```python
import ptan
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
HID_SIZE = 20

class DDPGActor(nn.Module):
    def __init__(self, obs_size, act_size):
        super(DDPGActor, self).__init__()

        self.net = nn.Sequential(
            nn.Linear(obs_size, HID_SIZE),  # 17 features to hidden layer
with 20 neurons
            nn.Tanh(),          # tanh activation function for hidden
layer
            nn.Linear(20, act_size),   # Hidden layer to 2 output values
        )

    def forward(self, x):
        return self.net(x)

class D4PGCritic(nn.Module):
    def __init__(self, obs_size, act_size,
                 n_atoms, v_min, v_max):
        super(D4PGCritic, self).__init__()

        self.obs_net = nn.Sequential(
            nn.Linear(obs_size, 400),
            nn.ReLU(),
        )

        self.out_net = nn.Sequential(
            nn.Linear(400 + act_size, 300),
            nn.ReLU(),
            nn.Linear(300, n_atoms)
        )

        delta = (v_max - v_min) / (n_atoms - 1)
        self.register_buffer("supports", torch.arange(
            v_min, v_max + delta, delta))

    def forward(self, x, a):
        obs = self.obs_net(x)
        return self.out_net(torch.cat([obs, a], dim=1))

    def distr_to_q(self, distr):
        weights = F.softmax(distr, dim=1) * self.supports
        res = weights.sum(dim=1)
        return res.unsqueeze(dim=-1)

class AgentD4PG(ptan.experience.BaseAgent):
    """
    Agent implementing noisy agent
    """
    def __init__(self, net, device="cpu", epsilon=0.3):
        self.net = net
        self.device = device
        self.epsilon = epsilon

    def __call__(self, states, agent_states):
        states_v = ptan.agent.float32_preprocessor(states)
        states_v = states_v.to(self.device)
        mu_v = self.net(states_v)
        actions = mu_v.data.cpu().numpy()
        actions += self.epsilon * np.random.normal(
            size=actions.shape)
        actions = np.clip(actions, -1, 1)
        return actions, agent_states

def unpack_batch(batch, device="cpu"):
    states, actions, rewards, dones, last_states = [], [], [], [], []
    for exp in batch:
        states.append(exp.state)
        actions.append(exp.action)
        rewards.append(exp.reward)
        dones.append(exp.last_state is None)
        if exp.last_state is None:
            last_states.append(exp.state)
        else:
            last_states.append(exp.last_state)
    states_v = ptan.agent.float32_preprocessor(states).to(device)
    actions_v = ptan.agent.float32_preprocessor(actions).to(device)
    rewards_v = ptan.agent.float32_preprocessor(rewards).to(device)
    last_states_v =
ptan.agent.float32_preprocessor(last_states).to(device)
    dones_t = torch.BoolTensor(dones).to(device)
    return states_v, actions_v, rewards_v, dones_t, last_states_v
```

2.1.6. d4pg_train_sync.py

Directly computing the expectation is often infeasible, as it requires integrating over all possible combinations of states and actions, which is computationally prohibitive in most cases. Therefore, these algorithms typically resort to sampling methods to estimate these expectations. In practice, data structures relying on experience replay buffers are employed to store experiences of interactions between the agent and the environment. By sampling from this buffer, the algorithm can estimate the expected updates of value functions or policy functions. This sampling-based approach allows the algorithm to learn from an incomplete dataset, enabling it to operate efficiently even in complex environments.

$$\text{ReplayBuffer} = \{(s_t, a_t, r_t, s_{t+1}, d_t)_i\}_{i=1}^{N}$$

Each iteration process consists of the following key steps:

2.1.6.1. Exploration and Experience Collection:

Given a state s_t, according to the current policy π, an action a_t is selected. This action is then executed, resulting in a reward r_t, and observing a new state s_{t+1}. This step involves the execution of the policy and interaction with the environment, allowing the agent to gather valuable experiences.

- The actions are selected and executed in the environment within the `while True` loop
- The instance generator `exp_source` of the class `ptan.experience.ExperienceSourceFirstLast` calls `env.step(action)`
- `env.step(action)` within the `test_net` function also performs this action in the environment.

2.1.6.2. Storage and Sampling:

The collected experiences s_t, a_t, r_t, s_{t+1} are stored in a replay buffer, and a batch (typically a batch of experiences) is randomly sampled from it for subsequent learning. This step breaks the temporal correlation between experiences, contributing to the stability and efficiency of the learning process.

- Using the `ptan.experience.ExperienceReplayBuffer``buffer.populate(1)` command, which populates the buffer with experiences.
- Sampling from the buffer to create a batch for learning is done with the `buffer.sample(BATCH_SIZE)` command.

2.1.6.3. Target and Regression:

For each sampled experience, a target value y is computed, combining the immediate reward $r_t$with an estimate of discounted future rewards (utilizing a target Q-function $\hat{Q}$ to stabilize the learning process). The Q-function parameters are then updated by regression methods to bring $Q(s_i, a_i)$ closer to the target value y. This step is pivotal to the algorithm, directly influencing the learning of the value function.

- The target Q-values are computed in the `distr_projection` function.
- With the aid of Pytorch, the regression to update the Q-function parameters is implemented in the optimization steps with `crt_opt.step()` for the critic network.

2.1.6.4. Policy Update:

The parameters of the policy π are updated by maximizing $Q(s_i, \pi(s_i))$, ensuring the selection of actions with the maximum Q-value at each state. This step embodies the core idea of policy gradient, optimizing the policy directly through gradient ascent.

- Actor's loss is calculated and then backpropagated through the network with `actor_loss_v.backward()` followed by `act_opt.step()` by using the class from Pytorch.

2.1.6.5. Periodic Update of Target Networks:

To enhance learning stability, every C steps, the target Q-function $\hat{Q}$ and target policy $\hat{\pi}$ are reset to the current Q-function and policy. This technique, known as "fixed target networks," serves to mitigate the instability encountered in traditional Q-learning, where updates to the Q values are directly based on their own predictions. It circumvents the potential for feedback loops and instability that can occur when two neural networks—target and primary—are updated too frequently in tandem during the learning process.

- The synchronization of the target networks is facilitated by the `alpha_sync` function, a feature provided by the `ptan` library, as observed in the usage of `tgt_act_net.alpha_sync(alpha=1-1e-3)` and `tgt_crt_net.alpha_sync(alpha=1 - 1e-3)`.

```
import os
import ptan
import time
```

```python
from wifi_streaming import Env
from RL import models
import argparse
from tensorboardX import SummaryWriter
import numpy as np
import threading
from pynput import keyboard
from wifi_streaming import client_order

import torch
import torch.optim as optim
import torch.nn.functional as F

GAMMA = 0.99
BATCH_SIZE = 64
LEARNING_RATE = 1e-3
MOMENTUM = 0.9
REPLAY_SIZE = 100000
REPLAY_INITIAL = 10
REWARD_STEPS = 5 # 3~10

OBSERVATION_DIMS = 9+8
ACTION_DIMS = 2

TEST_ITERS = 160 # determines when training stop for a while
MAX_STEPS_FOR_TEST = 10

Vmax = 10
Vmin = -10
N_ATOMS = 51
DELTA_Z = (Vmax - Vmin) / (N_ATOMS - 1)


def test_net(net, env, count=10, device="cpu"):
    rewards = 0.0
    steps = 0
    obs = env.reset(is_recording=False)
    # while True:
    #         obs_v = ptan.agent.float32_preprocessor([obs]).to(device)
    #         mu_v = net(obs_v)
    #         action = mu_v.squeeze(dim=0).data.cpu().numpy()
    #         action = np.clip(action, -1, 1)
    #         obs, reward, done, _ = env.step(action)
    #         rewards += reward
    #         steps += 1
    #         if done or steps >= MAX_STEPS_FOR_TEST:
    #             print("net test1 finished")
    #             client_order.FREEX_CMD(env.sock, "E", "0", "E", "0")
    #             break
    # time.sleep(1)
    for i in range(count-1):
        steps = 0
        rewards = 0.0
        # obs = env.reset(is_recording=False)
        while True:
            obs_v = ptan.agent.float32_preprocessor([obs]).to(device)
            mu_v = net(obs_v)
            action = mu_v.squeeze(dim=0).data.cpu().numpy()
            action = np.clip(action, -1, 1)
            obs, reward, done, _ = env.step(action)
            rewards += reward
            steps += 1
            if done or steps >= MAX_STEPS_FOR_TEST:
                client_order.FREEX_CMD(env.sock, "E", "0", "E", "0")
                print(f"net test{i+2} finished")
                break
        time.sleep(1)
    return rewards / count, steps / count


def distr_projection(next_distr_v, rewards_v, dones_mask_t,
                     gamma, device="cpu"):
    # since we can't really computing tensor on cuda with numpy
    next_distr = next_distr_v.data.cpu().numpy()
    rewards = rewards_v.data.cpu().numpy()
    dones_mask = dones_mask_t.cpu().numpy().astype(np.bool_)
    batch_size = len(rewards)
    proj_distr = np.zeros((batch_size, N_ATOMS), dtype=np.float32)
    for atom in range(N_ATOMS):
        tz_j = np.minimum(Vmax, np.maximum(
            Vmin, rewards + (Vmin + atom * DELTA_Z) * gamma))
        b_j = (tz_j - Vmin) / DELTA_Z
        l = np.floor(b_j).astype(np.int64)
        u = np.ceil(b_j).astype(np.int64)
        eq_mask = u == l
        proj_distr[eq_mask, l[eq_mask]] += \
            next_distr[eq_mask, atom]
        ne_mask = u != l
        proj_distr[ne_mask, l[ne_mask]] += \
            next_distr[ne_mask, atom] * (u - b_j)[ne_mask]
        proj_distr[ne_mask, u[ne_mask]] += \
            next_distr[ne_mask, atom] * (b_j - l)[ne_mask]

    if dones_mask.any():
        proj_distr[dones_mask] = 0.0
        tz_j = np.minimum(Vmax, np.maximum(
            Vmin, rewards[dones_mask]))
        b_j = (tz_j - Vmin) / DELTA_Z
        l = np.floor(b_j).astype(np.int64)
        u = np.ceil(b_j).astype(np.int64)
        eq_mask = u == l
        eq_dones = dones_mask.copy()
        eq_dones[dones_mask] = eq_mask
        if eq_dones.any():
            proj_distr[eq_dones, l[eq_mask]] = 1.0
        ne_mask = u != l
        ne_dones = dones_mask.copy()
        ne_dones[dones_mask] = ne_mask
        if ne_dones.any():
            proj_distr[ne_dones, l[ne_mask]] = (u - b_j)[ne_mask]
            proj_distr[ne_dones, u[ne_mask]] = (b_j - l)[ne_mask]
    return torch.FloatTensor(proj_distr).to(device)

stop_event = threading.Event()
def on_press(key):
```

```python
    try:
        if key.char == 'q':
            stop_event.set()
    except AttributeError:
        pass
def start_listening():
    listener = keyboard.Listener(on_press=on_press)
    listener.start()

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--cuda", default=False, action='store_true',
help='Enable CUDA')
    parser.add_argument("-n", "--name", required=True, help="Name of the
run")
    args = parser.parse_args()
    device = torch.device("cuda" if args.cuda else "cpu")
    start_listening()
    save_path = os.path.join("saves", "d4pg-" + args.name)
    os.makedirs(save_path, exist_ok=True)

    act_net = models.DDPGActor(OBSERVATION_DIMS, ACTION_DIMS).to(device)
    crt_net = models.D4PGCritic(OBSERVATION_DIMS, ACTION_DIMS, N_ATOMS,
Vmin, Vmax).to(device)

    best_model_path = None
    best_reward = float('-inf')
    for file in os.listdir(save_path):
        if file.startswith("best_") and file.endswith(".dat"):
            # 从文件名解析奖励值
            try:
                reward_str = file.split('_')[1]
                reward = float(reward_str)
                if reward > best_reward:
                    best_reward = reward
                    best_model_path = os.path.join(save_path, file)
            except ValueError:
                pass
    if best_model_path:
        act_net.load_state_dict(torch.load(best_model_path,
map_location=device))
        print(f"Loaded best model: {best_model_path}")
    else:
        print("No best model found, starting from scratch.")

    print(act_net)
    print(crt_net)
    tgt_act_net = ptan.agent.TargetNet(act_net)
    tgt_crt_net = ptan.agent.TargetNet(crt_net)

    writer = SummaryWriter(comment="-d4pg_" + args.name)
    env = Env.ExoskeletonEnv(log_writer=writer)
    agent = models.AgentD4PG(act_net, device=device)
    exp_source = ptan.experience.ExperienceSourceFirstLast(env, agent,
gamma=GAMMA, steps_count=REWARD_STEPS)
    buffer = ptan.experience.ExperienceReplayBuffer(exp_source,
buffer_size=REPLAY_SIZE)
    act_opt = optim.SGD(act_net.parameters(), lr=LEARNING_RATE,
momentum=MOMENTUM)
    crt_opt = optim.Adam(crt_net.parameters(), lr=LEARNING_RATE)

    frame_idx = 0
    best_reward = None
    training_stopped_early = False
    with ptan.common.utils.RewardTracker(writer) as tracker:
        with ptan.common.utils.TBMeanTracker(writer, batch_size=10) as
tb_tracker:
            while True:
                if stop_event.is_set():
                    print("Training stopped by user.")
                    training_stopped_early = True
                    if best_reward is not None:
                        current_model_name = "best_%+.3f_%d.dat" %
(best_reward, frame_idx)
                    else:
                        print("you stopped training before any best reward
was achieved.")
                    current_model_path = os.path.join(save_path,
current_model_name)
                    torch.save(act_net.state_dict(), current_model_path)
                    print(f"Current model saved to {current_model_path}")
                    break

                frame_idx += 1
                buffer.populate(1)
                rewards_steps = exp_source.pop_rewards_steps()
                if rewards_steps:
                    rewards, steps = zip(*rewards_steps)
                    tb_tracker.track("episode_steps", steps[0], frame_idx)
                    tracker.reward(rewards[0], frame_idx)

                if len(buffer) < REPLAY_INITIAL:
                    continue
                if len(buffer) == REPLAY_INITIAL:
                    print("Initialization of the buffer is finished, start
training...")
                    client_order.FREEX_CMD(env.sock, "E", "0", "E", "0")
                    input("Press Enter to continue...")

                batch = buffer.sample(BATCH_SIZE)
                states_v, actions_v, rewards_v, \
                dones_mask, last_states_v = \
                    models.unpack_batch(batch, device)

                # train critic
                crt_opt.zero_grad()
                crt_distr_v = crt_net(states_v, actions_v)
                last_act_v = tgt_act_net.target_model(
                    last_states_v)
                last_distr_v = F.softmax(
                    tgt_crt_net.target_model(
                        last_states_v, last_act_v), dim=1)
                proj_distr_v = distr_projection(
                    last_distr_v, rewards_v, dones_mask,
                    gamma=GAMMA**REWARD_STEPS, device=device)
                prob_dist_v = -F.log_softmax(
                    crt_distr_v, dim=1) * proj_distr_v
                critic_loss_v = prob_dist_v.sum(dim=1).mean()
                critic_loss_v.backward()
                crt_opt.step()
                tb_tracker.track("loss_critic", critic_loss_v, frame_idx)

                # train actor
                act_opt.zero_grad()
                cur_actions_v = act_net(states_v)
                crt_distr_v = crt_net(states_v, cur_actions_v)
                actor_loss_v = -crt_net.distr_to_q(crt_distr_v)
                actor_loss_v = actor_loss_v.mean()
                actor_loss_v.backward()
                act_opt.step()
                tb_tracker.track("loss_actor", actor_loss_v,
                                frame_idx)

                tgt_act_net.alpha_sync(alpha=1 - 1e-3)
                tgt_crt_net.alpha_sync(alpha=1 - 1e-3)

                if frame_idx % TEST_ITERS == 0:
                    client_order.FREEX_CMD(env.sock, "E", "0", "E", "0")
                    print("Please prepare for a test phase by changing the
exoskeleton user, if desired.")
                    # input("Press Enter to continue after the user has
been changed and is ready...")
                    ts = time.time()
                    rewards, steps = test_net(act_net, env, count=4,
device=device)
                    print("Test done in %.2f sec, reward %.3f, steps %d" %
(
                        time.time() - ts, rewards, steps))
                    writer.add_scalar("test_reward", rewards, frame_idx)
                    writer.add_scalar("test_steps", steps, frame_idx)
                    if best_reward is None or best_reward < rewards:
                        if best_reward is not None:
                            print("Best reward updated: %.3f -> %.3f" %
(best_reward, rewards))
                            name = "best_%+.3f_%d.dat" % (rewards,
frame_idx)
                            fname = os.path.join(save_path, name)
                            torch.save(act_net.state_dict(), fname)
                        best_reward = rewards
                time.sleep(0.01)
    # except KeyboardInterrupt:
        # print("Training interrupted by keyboard.")

    # finally:
    if best_reward is None:
        print("No best reward achieved during the training.")
    elif training_stopped_early:
        print(f"Training stopped, Best reward achieved:
{best_reward:.3f}")
    try:
        env.close()
    except Exception as e:
        print(f"Error while closing resources: {e}")
```

2.2. Self-documented coding style

To ensure the readability and maintainability of the code, we prefer adopting a self-documenting coding style, combined with the following specific practices to address the specific challenges encountered in the project:

2.2.1. Clear Naming and Function Layering with Logical Decomposition:

We enhance code clarity and readability by using descriptive naming and breaking down complex logic into multiple simple boolean functions. Additionally, we distribute functions of different functionalities into multiple custom libraries, roughly categorized into communication protocols, abstraction and integration, and RL model integration.

2.2.2. Team Collaboration Using GitHub and Live Share:

We utilize GitHub for version control and collaboration, leveraging Live Share for real-time collaborative programming. We also employ shared hosts and native pip venv virtual environments to unify development environments, ensuring efficient collaboration among team members.

2.2.3. *Stability and Error Handling:*

When communicating with WiFi devices, we face challenges regarding network stability. To address this, we extensively use try-except structures with while loops in the code to handle possible exceptions and retry logic, ensuring stable communication with external devices even in unstable network conditions. For instance, when reading data from WiFi devices, we capture any exceptions and debug by printing error messages, while using loops to continuously attempt until successful data reception.

2.2.4. *Complexity of EMG Signal Processing:*

Considering the complexity of EMG signal processing, we particularly emphasize clear documentation and meticulous error handling in this part of the code. By adding detailed docstrings in signal processing functions, we explain the purpose and implementation of each step, while also employing appropriate exception handling mechanisms to address potential signal processing errors, ensuring the robustness and reliability of the signal processing workflow.

問福實

2.3. *Simulation stimulus (or test cases)*

2.4. *Simulation waveforms (or software result)*

2.5. *Emulation PCB design (if any)*

2.6. *System PCB schematic*

2.7. *Application examples*

2.8. *Application software source code (if any)*

3. *Verification Results and Materials*

3.1. *Demonstration*

3.2. *System test result*

3.3. *Lab measurement*

3.4. *Performance benchmark*

3.5. *The artwork of the project*

3.5.1. *Exoskeleton*

3.5.2. *EMG sensor*

3.5.3. *Scene through the VR goggles*

4. *Summary Report*

4.1. *Summaries of the project*

4.2. *Content index*

4.3. Cross references

4.4. Valuables

4.5. Proposal for future plan as the project: KR260