Cybernetic Muse Arousal Stability System *

Ana Bog

December 2025

*https://github.com/kkentia/muse_cyb_sys.git

Abstract

The Muse Arousal Stability System is a project I developed for the scope of the Cybernetics class. For this project, I used InteraXon's Muse S (Gen 1) headband for recording and streaming brain waves coming from the 4 channels AF7, AF8 (Beta waves), TP9 and TP10 (Alpha waves). The goal is to compare the alpha waves against the beta waves for getting the cleanest signal for conscious "Calm vs. Focus" mental states control. A visual sensor is helping you know if you are in a focused state or if you're drifting away. Since the controller for this system is the human participant himself, we cannot play too much with different parameters like environmental control and/or testing different types of controllers other than biological ones. For this reason I decided to implement a simulation mode, in which a digital controller (P or PID) is trying to stay on the target arousal while fighting against different external factors, and analyze their efficiency at doing so.

Contents

1 Introduction

This section introduces the Muse Arousal Stability system, detailing its background, core motivation, and overall scope.

1.1 Project Background

As previously mentioned, this small project has been developed only for the sake of the Cybernetic's course hand-in project, although why I chose this specific system as the main cybernetic loop stems from my interest in testing EEG consumer-grade hardware and experiment with it.

The rules for the cybernetic system we had to implement were the following:

Loop completeness	Sensor → Comparator → Controller → Actuator → Environment
Functionality	The loop runs and responds to changes
Disturbance handling	Disturbance implemented + tested
Error measurement	Clear, quantitative performance metric
Failure analysis	One clear failure case with explanation
Interpretation quality	Clarity, correctness, insight
Pitch quality	Delivery, visuals, conciseness

Figure 1: System Rules

Below you have a schema of the cybernetic loop used to build the Muse Arousal Stability System. The code structure also follows this diagram.

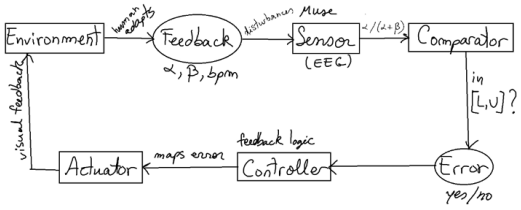


Figure 2: Cybernetic Loop

1.2 Scope of Project

The scope of this Muse Arousal Stability System is for a system (real or artificial) to try to get (and stay) in a 'focused mental state', or any other target arousal, calculated from alpha and beta brain waves (raw stream from the EEG or artificially simulated).

From now on, I am going to reference the application mode that uses the Muse hardware to stream real brainwave frequencies from a human participant as 'Real mode', and the mode in which these waves are simulated as 'Sim mode'.

1.2.1 Real Mode

In Real mode, there aren't that many features, since the only thing you can control is limited by your actual brain power, Muse's accuracy and capabilities, environmental noise and muscle artifacts, and so on. All of these are parameters that are more or less out of our control, as well as being limited by time (we cannot fast forward time to test different hypotheses regarding this system).

In this mode, the only thing that you will do is put on the Muse headband, hope for not too much noise, and then try testing if you can or not control if you can get the system's visual representation of your brainwaves stay in a viability band of a so called 'focused' state.

As such, the scope for this mode is pretty different from the scope of Simulation mode, with which we can achieve much more, but may be less accurate and subject to other problems, such as Who and How is the system controlled by?

1.2.2 Simulation Mode

In Sim mode, the scope is not so much to see whether the artificial Controller can get to move and stay in a target arousal zone, but rather how it can do so using different different types of controllers and external forces that are there to push the controller to its limits.

Energy feature: In real world situations, a system usually does not possess unlimited power resources. For the scope of this project, and since the simulation mode is trying to simulate behavior of a biological system (human), I decided to add an energy bar that would make the system behave differently depending on the energy it has left: the energy bar depletes very fast if the system's controller (current arousal index) has to make a big jump in some direction, or is fighting against external forces. Once the energy level goes below 30%, then it has much less force. When it hits 0, control is lost.

Fatigue feature: Additionally, a 'fatigue' progress bar is also added, which is separate from the energy bar. This is here to simulate over-pushing a system to its mental/biological limits and entering a 'burnt_out' state, representing system failure. This state is reached when the system's current index is above 65, from which it will start going up from 0 to 100%. Once 100% is reached, the system's state changes from its current state to 'BURNT_OUT_STATE', moment from which the system's current arousal level drops to 0 and again, control over the system is lost and entirely subject to external factors. To make it even more realistic, if the system is in the burnt out state but feedback is still turned on (the controller tries to match the target arousal), then the fatigue bar will never reset. The controllers needs to 'let go' of its efforts for the fatigue bar to drop down again all the way to 0% before its able to take control of itself again. If, instead of hitting the 100% on the fatigue bar, you choose to turn the feedback off before that happens, the system will regenerate way faster than if it reaches the end of the bar.

Thus, in this Sim mode, the goal is to experiment with and see how different controllers might behave compared to one another, and as such test their efficiency in different conditions.

2 Methodology

This section details and explains the choices I made regarding to the design of the app.

2.1 Choice of frequencies

Even though the Muse headband is able to capture delta, theta and gamma waves in addition to alphas and betas, I decided not to include them in my system for the following reasons:

- Delta (δ, 0.5-4Hz): these waves dominate during sleep or in cases of severe brain injury. Since the user is awake and engaging with the system, detecting deltas would essentially be an error state, and is irrelevant for the system.
- Theta (θ, 4-8Hz): these waves are linked to deep meditation, drowsiness or the state between waking up and sleeping. Including theta would complicate the arousal index: a user entering a Theta state is technically 'low arousal', but it's a different kind of low arousal (sleepy) compared to Alpha (calm/meditative). To keep the controllers linear and simple (calm vs. focused) I decided not to include these.
- Gamma (γ, $>$ 30Hz): these waves are associated with high-level cognition. I decided to ignore them because these frequencies overlap almost perfectly with muscle noise. If for example we clench our jaw or blink, it would generate a big electrical spike that the system would confuse with 'high concentration' states. By cutting off the filter at 30Hz (high Beta), it intrinsically filters out the vast majority of muscle artifacts, thus making the controller much more stable.

2.2 EEG Calibration

As for the initially intended mode, which was only the real EEG mode, many question arose: do I train a model for MY high and low arousal states? But that would mean that the system would correctly asses only brains that behave like mine. The other option, and the one I opted for, was having to calibrate 80 samples of incoming brainwaves from every new run, meaning different persons would get different 'good' samples that would be used to get into a personalized viability band. The issue with this is not knowing in which mental state the person is in when calibrating the hardware. Thus, the viability band would form around your current arousal, and changing state would make you go away from the viability band. This kind of inverses how you would look at a controller starting from a point a and trying to reach point b. I went for this option for 3 main reasons:

1. When trying to record samples of my brainwaves in different states, how do I actually know I am in said state?
2. Because of my hair, having constant artifacts over different runs was impossible. Something that was achievable on one of my guy friends with short hair.
3. I wanted to be able to test this with different participants, and training a model can take many hours.

2.3 Choice of controllers

This subsection is only concerned with the Sim mode. I will start by mentioning that I built this app in two big parts, v1 and v2. Both subsequent controllers are just simple algorithms.

2.3.1 P-controller

In system v1, the system only had one proportional controller (P-controller), which is the simplest form of a controller. It calculates the error between the current state and the target and applies a corrective force proportional to that error:

$$F_{effort}[n] = K_p \cdot e[n] \tag{1}$$

$$\text{Where } e[n] = \text{Target} - \text{CurrentArousal}[n]$$

It works perfectly in a noise-free environment. After playing with different parameters to see where its limits reside, I quickly found that noise, as well as other opposing forces were a big limit to this P controller.

2.3.2 PID-controller

After looking for other kinds of controllers I could implement in my system, I eventually tried the next logical option which was a PID-controller (v2). This controller extends the P-controller by taking into account the accumulated past error (Integral) and the current velocity of the error (Derivative). The discrete force calculation used in the simulation is:

$$F_{PID}[n] = K_p \cdot e[n] + K_i \sum_{k=0}^{n} e[k] + K_d \cdot (e[n] - e[n-1]) \tag{2}$$

To determine the optimal parameters (K_p, K_i, K_d) without manual guessing, I integrated the **Ziegler-Nichols (Z-N)** closed-loop tuning method. This algorithm increases the proportional gain until the system reaches stable oscillation (finding the Ultimate Gain K_u and Oscillation Period T_u), and then derives the parameters using the standard heuristic:

$$K_p = 0.60 \cdot K_u \tag{3}$$

$$K_i = \frac{2K_p}{T_u} \tag{4}$$

$$K_d = \frac{K_p T_u}{8} \tag{5}$$

But is this controller better than its simpler version, the P-controller? We will discuss that in the Results section.

2.3.3 Other controllers?

- Energy-efficient controller: When going from v1 to v2 of the app, with the limited time I had, I had to choose between a few interesting options. Before choosing to go along with a PID-controller, I had also tried introducing an energy efficient controller, but that didn't work as intended and I soon realized that it would take me too much time if I wanted to do it correctly. One of the main issues I had while implementing it was the controller would actually end up spend more energy while trying to move slower to reduce energy usage (big jumps cause big energy losses). Then, it also was very inefficient when it came to fighting against external factors, as it would prefer not doing anything since it was seeing its fatigue levels increase (which happens in high arousal states independently conscious control) and would have to spend too much energy to fight it off. It would then just end up in the burnt out state.
- Predictive controller: This would've been very interesting as it would've been the system to be the most capable of adaptive learning and the most similar to real, complex biological systems, but I had one big issue with this: for the predictions to be correct, the environment had to be predictable. Since I am only manually tuning the perturbations parameters, there is nothing predictable about the way I do it.

From the results I was able to get with only two very simple controller algorithms, I would find it very interesting to eventually find out how these two other controllers would behave in the Muse Arousal System, or in other cybernetic systems altogether.

3 Implementation

Everything is done in Python v3.9.13. The project structures follows a simple cybernetic system structure: it is divided in multiple files, each defining respectively a Controller element, an Actuator, a Processor, a Sensor and a Comparator. There are few additional files, for e.g. for connecting the Muse hardware with the program and streaming the raw EEG data.

Below is a screenshot with the project folder structure, as well as detail about each file inside.

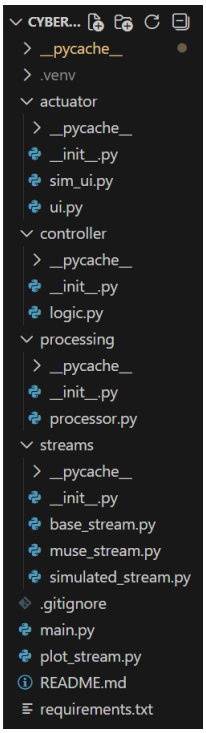


Figure 3: Project Structure

3.1 Core Components

1. **main.py**: The Orchestrator. This is the entry point of the application. It serves as the central orchestrator that manages the application lifecycle and synchronizes the different components.

   - Mode Selection: It handles the initialization and switching between "Real Mode" (Live EEG) and "Simulation Mode". It manages the global `session_state` buffers used by Streamlit to persist data across re-runs.
   - Multiprocessing Architecture: When "Live EEG" is selected, this class spawns a separate, independent process using `mp.Process`. This targets the `plot_stream.run_plot` function. Data is passed from the main thread to this plotting process via a `multiprocessing.Queue`.

- Real Mode Loop: It runs an infinite loop that polls the `MuseStream` for data, pushes it to the processor, updates the `Controller` state, and renders the `ui.py` dashboard. It also handles the dynamic calibration logic, allowing users to adjust the viability band width.
- Simulation Loop: It manages the simulation. It allows for "Extreme Trials" (e.g., Caffeine, Drowsy, Exam), which presets the simulation's internal parameters (latency, noise, threat levels) to specific hardcoded values to test the controller's robustness.

2. **simulated_stream.py**: The Simulation Plant. This class is the all-in-one engine for the Simulation Mode. It acts as the Sensor (generating data), the Plant (simulating physics), and the Controller (calculating corrective force). It models the human psyche as a resource-constrained dynamic system.

   - Arousal Physics Calculation: The arousal state x is updated every tick based on a vector sum of forces:

   $$x_{t+1} = x_t + F_{drift} + \underbrace{(F_{controller} \cdot (1 - B))}_{\text{Switched Control}} + F_{threat} + \text{Noise} \tag{6}$$

   Where F_{drift} pulls the system to a baseline, F_{threat} represents environmental stress, F_{effort} is the corrective force from the controller and where $B \in \{0, 1\}$ represents the binary Burnout State:
   $B = 1$ if Fatigue $= 100\%$, otherwise $B = 0$
   - Control Algorithms: The class implements two distinct control strategies, selectable by the user:
     - P-Controller: A simple proportional controller where the corrective force is $K_p \times$Error. It is energy-efficient but prone to steady-state error in the presence of "drift."
     - PID-Controller: A full implementation of Proportional-Integral-Derivative control. It calculates the error integral (with clamping integral values between -5 and 5 to prevent windup) and the error derivative. This allows the system to eliminate steady-state error and face more harsh conditions and noise.
   - Ziegler-Nichols auto-tuner: A built-in state machine that can automatically tune the PID parameters. It incrementally raises K_p until the system oscillates (Ultimate Gain), measures the period of oscillation, and derives the optimal K_p, K_i, K_d values.
   - Physiological Modeling:
     - Energy: A resource variable that depletes based on the magnitude of F_{effort}. If Energy drops, the controller's gain is throttled.
     - Fatigue & Burnout: A cumulative variable. If the system stays in extreme arousal states (> 0.6) for too long, fatigue accumulates. If Fatigue hits 100%, the system enters a 'Burnout State', where the current arousal drops to 0, and the controller becomes unresponsive to user targets until fatigue drops to 0%.

3. **processor.py**: The Signal Processing. This class handles the Digital Signal Processing (DSP) pipeline for the Real mode. It transforms raw voltage into the usable Arousal Index.

   - Filtering: Applies normalization and a 50Hz notch filter (using `BrainFlow`) to remove electrical noise.

- Feature Extraction: It separates the EEG channels into Posterior (TP9, TP10) and Frontal (AF7, AF8). It calculates the Power Spectral Density (PSD) ($\mu V^2/Hz$) and computes the metric: Arousal = $\log(\text{Alpha}_{posterior}) - \log(\text{Beta}_{frontal})$. EEG power values vary by huge orders of magnitude (Alpha is often much stronger than Beta). Taking the log compresses the range so they can be compared fairly, and it turns the data into a normaln (Gaussian) distribution, which is easier to process statistically.
- Artifact Rejection: Before processing, it calculates the variance of the raw signal buffer. If variance exceeds a threshold (default $10,000\mu V^2$), the sample is flagged as an artifact and discarded.
- EMA Smoothing: Applies an Exponential Moving Average ($\alpha = 0.1$) to the calculated index to reduce jitter. This is a low-pass filter.
- Calibration: Implements the `calibrate()` method, which records a baseline buffer and calculates the Interquartile Range to define the user's specific Viability Band.

4. **logic.py**: The Comparator. This class acts as the Comparator and State Logic for the Real Mode.
   - Hysteresis: It implements a hysteresis buffer to prevent the system state from flickering rapidly between states. The user must be outside the viability band for 30 consecutive samples (approx. 3 seconds) before the system updates the state variable `in_range` from True to False.
   - State Tracking: It maintains the `last_good_arousal` value.
5. **ui.py and sim_ui.py**: These classes handle the GUI rendering using Streamlit. `ui.py` is the UI for the Real mode and `sim_ui.py` the one for Simulation mode.
   - Live Dashboard: Renders the Arousal Index metric, the Viability Band visualizer (using Matplotlib) and other visual elements. In Sim mode, it also renders dynamic progress bars for Energy and Fatigue.
   - Post-Session Analysis: Both classes generate a a few graphs upon session ending. This includes calculating the Cost Function (Sum of Squared Errors), Position Error, Velocity Error, and generating Phase Space plots (PE vs. VE) using the Altair library to analyze system stability.
6. **plot_stream.py**: Plots a live vizualisation of the raw EEG data (4 channels) that is being streamed from the Muse headband.
   - Process Isolation: Because Streamlit's refresh rate is too slow for raw signal visualization, this script runs in a separate system process.
   - Matplotlib Integration: It uses the `TkAgg` backend (instead of the web backend) to create a native OS window. It manages a rolling buffer of the last 5 seconds of data and applies bandpass filters (1-45Hz) on the fly for visual clarity.
7. **muse_stream.py**: This class uses the `BrainFlow` library to treat the data from the hardware.
   - Connection: Establishes the Bluetooth connection to the Muse 2 headband.

- Data Retrieval: the `get_data()` method pulls the last chunk of data from the buffer and handles the release of the session.

8. **base_stream.py**: An abstract base class (interface) that forces `MuseStream` and `SimulatedStream` to expose the same methods.

3.2 Used Libraries

- Streamlit for the web page generation, plotting (except the live Muse EEG stream that we plot in `plot_stream.py` using Matplotlib, because refreshing a graph at high rates isn't very compatible with Streamlit who's lightweight and simple to use), buttons, selectors, all UI elements, as well as `session_state` for....
- Brainflow for retrieving the EEG raw signals from the EEG (used that way in `muse_stream.py`) as well as for treating the noise (used in `processor.py`, with a code I stole from OpenBCI opensource code on github used for their own EEGs :D)
- Matplotlib for some of the plots that had higher refresh rates or that were more complex, like the Position Error Distribution plot, and especially the separate live EEG window (running in its own process with TkAgg so Streamlit stays responsive). Also used inside the sim to draw the viability circle + moving dot (with Agg backend for Streamlit).
- Numpy for the numerics everywhere: vector ops on EEG buffers, variance for artifact detection, percentiles for calibration (IQR), EMA smoothing for arousal (took from OpenBCI), Gaussian noise, etc. (inside `processor.py` and `sim_ui.py`)
- Pandas for tracking simulation history as a DataFrame and for feeding the post simulation analysis (position, velocity, error, cost). (inside `main.py`)
- Time built-in lib: for rate control (`time.sleep(3)` for timing operations for e.g.) and pacing, as well as calibration stepping (inside `muse_stream.py`).
- Collection built-in lib: for keeping a short arousal history used for control delay to simulate latency (like there would be in real biological systems (inside `simulated_stream.py`).
- Queue built-in lib: for permitting synchronized processes (multiprocessing) between Streamlit and the separate live plotter (`plot_stream.py`), by pushing EEG chunks into a `multiprocessing.queue`. (inside `main.py`)

4 User Demonstration

This section provides a user-centric walkthrough of the Muse Arousal Stability System app, showing off the two modes and their features.
For more detailed instructions on how to deploy the app on your machine, go check out the README on the referred GitHub page.

Github repository for the project: https://github.com/kkentia/muse_cyb_sys.git

4.1 Simulation Mode

The figure below shows a screenshot of the UI of the Simulation mode. In the sidebar on the left figure the system's parameters with which you can play to simulate external forces, noise, latency and so on, as well as select and tune the used controller, the viability band width and other system parameters. The Base State will set your current arousal (default state) to the corresponding psychological state. Calm, for e.g., represents an arousal index of 0.25.
In the center of the page, you will see a live dashboard rectangle plot, which shows you the controller's current arousal (red line) and the viability band surrounding the target arousal (blue area). The plot below shows your the stream of simulated eeg data, as well as where it is located in comparison to the viability band's upper and lower limits. At the bottom of the sidebar, you can see something named 'Run Extreme Trials'. Upon selection of one of the three buttons, a specific scenario will be simulated. For example, 'Stressful Exam' sets the 'environmental distraction/threat' at 0.4 and effort amplification at 5.

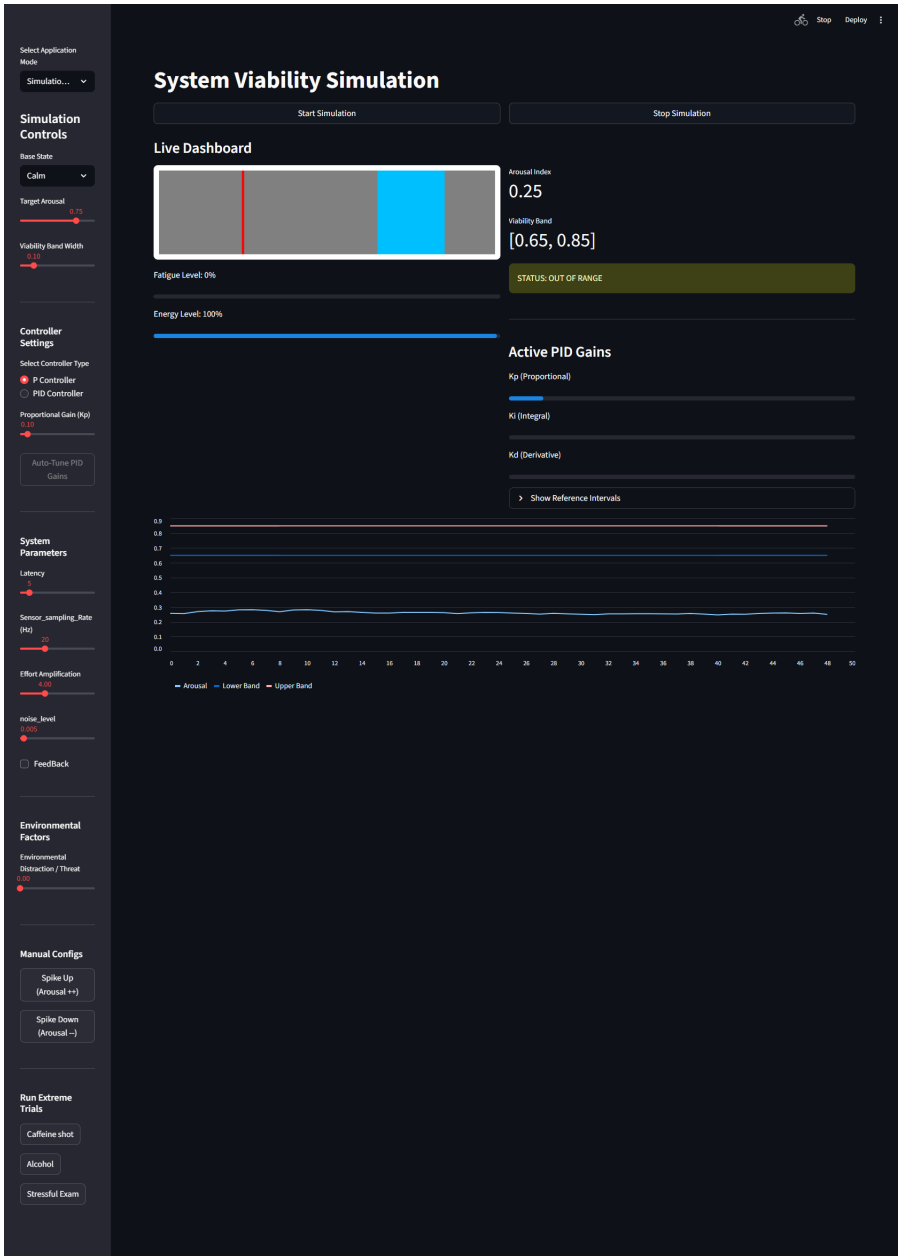


Figure 4: Simulation Dashboard

When you click on 'Stop Simulation' button, new post-analysis features will show on the page. You can also download the simulation data in CSV format.

4.2 Real Mode (EEG mode)

This mode has a simpler interface than the one before. You will not be adjusting any parameters, except for the viability band width if you wish to do so. Before running the actual cybernetic system, you will have to calibrate your EEG data. For that, you are required to sit still, relax and close your eyes until 80 clean samples are collected. Then you will see the live dashboard plot just like the one in Simulation mode.
Additionally, a separate window will pop up in which you can see the real EEG data that is being streamed from the Muse headband from 4 channels simultaneously (figure below).

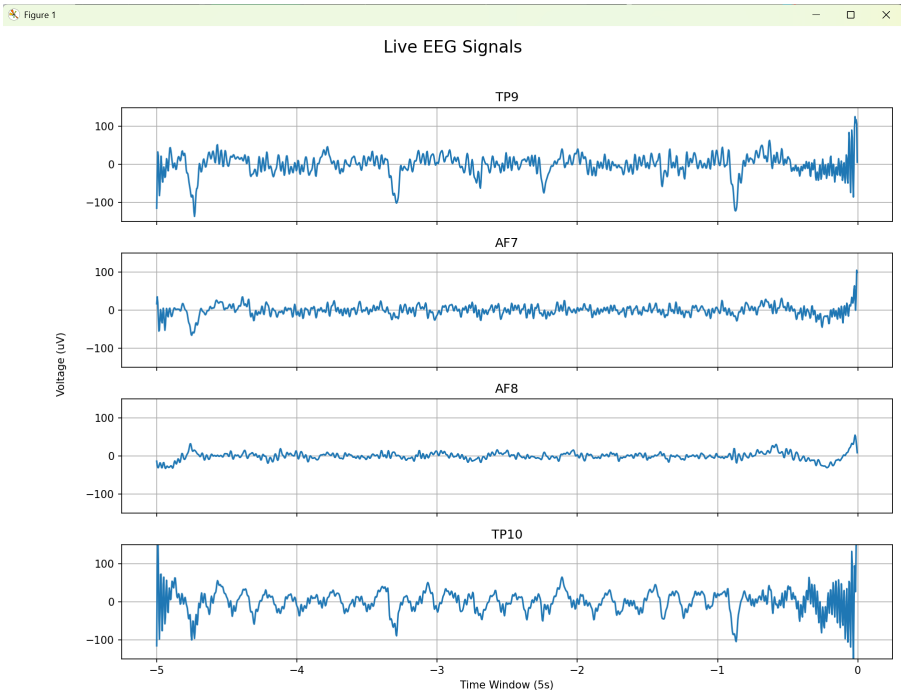


Figure 5: raw EEG data stream

5 Results

In this section, we will discuss if the application had achieved its initial goal and is working properly. Secondly, we will compare the two available controllers for the system.

5.1 Application

The app is working like it's supposed to. The interface is complete, and the post-analysis graphs are relevant. There could've been even more metrics, if the intended goal was to compare the controllers, which initially it wasn't. Selection between the 2 modes (Real mode vs. Simulation mode) works fluidly, the buttons do what they're supposed to do. Only 'issue' I had was with rendering continuous flows using Streamlit, which I discovered is not recommended since the representations flicker a lot for each update they get. Another issue I've had with Streamlit (altough very inuitive and easy to use) was updating parameters in real time. I had to add the progress bars for displaying the auto-tuned parameters for the PID controller, since it would refuse to update the manual sliding bars on the sidebar. Time was a bit tricky to keep track of since the app works in Hz and not in seconds, so that one is an approximation of time.

5.2 System

Everything works like it's supposed to. The controllers work in different conditions, as well as the auto-tune for the PID (adaptive method for controller). For the real EEG mode, there is no possible way of knowing for sure in what psychological state you are, and thus whether it is your focus drifting away that makes you leave the viability band (created around your calibrated brainwaves data) or some other thing drifting away. We just know that something in your brain has changed for the moment, making you leave the 'current state' for another. Artifacts and noise are largely taken care of in normal conditions, except for hair sometimes. Can be somewhat fixed by wetting your forehead (under the electrodes). The waves are filtered and normalized. If, after calibrations, you encounter an artifact, it doesn't get count as input, and the system knows and tells you that artifacts are detected and so the state that you will see will be the last good state you were in (hysteresis).

5.3 Controllers comparison

In this section we will look at the differences between the two controllers (P vs.PID) under different conditions, from measurements retrieved by the post-analysis of the simulation inside our app.

Let's first go through what the metrics mean.

- `SSE (Total Cost)` is the cumulative tracking error, and is calculated like so:

$$e(t) = \text{Arousal}(t) - \text{Target} \tag{7}$$

$$\text{Total Cost (SSE)} = \sum_{t=0}^{N} (e(t))^2 \tag{8}$$

- `Time to Reach Goal` is time that has passed since the start of the simulation until the current arousal reached AND stayed in the viability band for at least 3 consecutive seconds.
- Energy Remaining at Goal is self explanatory, and includes regenerated energy as well.
- `Energy Spent to Reach Goal` is the total accumulated energy used to reach goal.
- `Position Error (PE)` is the distance left between the current and target arousal.

$$PE(t) = x(t) - x_{target} \tag{9}$$

- `Velocity Error (VE)` is the rate of change of the arousal index, representing how fast the system is moving.

$$VE(t) = x(t) - x(t-1) \tag{10}$$

- Finally, the `Phase Portrait` plots PE (x axis) by VE (y axis) to show the system's trajectory and stability.

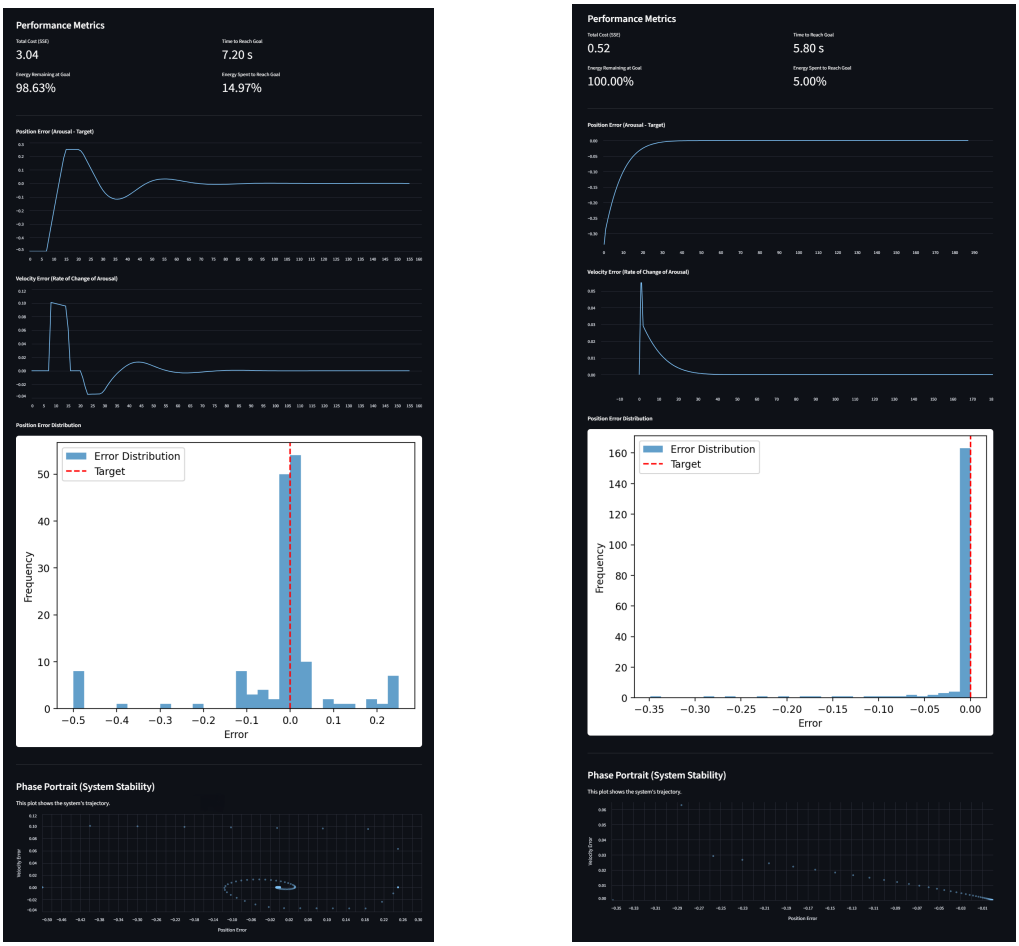
For reference, non specified parameters are all set to 0 for all runs, in order to have clear evaluation conditions. The globally set parameters are the following: 'effort amplification': 2.03, 'sampling rate': 20Hz, 'viability band width': 0.05, 'target arousal': 0.75, 'default arousal': 0.25. Simulations were run for $\approx$ 20-24 seconds each.

While the P controller's gain is manually set, the PID uses adaptive auto-tuning thanks to the Ziegler-Nichol's method.

5.3.1 Latency

Latency is here set to 7, kp(P): 0.1, and kp(PID) continuously increases.

When comparing the controllers against latency (Fig. 1, 2), we can see that both P and PID managed to stabilize in the viability band. As for the performance metrics, the PID performs better in all of them, especially in energy efficiency, where it does 3 times better than the P controller.
When looking at the PE plot, we see that while controller (a) is influenced by latency, controller (b) is completely unaffected by it, which explains its energy savings since it did not have to undergo big back and fourth oscillations. It also explains its pinpoint accuracy.



(a) P-controller (b) PID-controller

Figure 6: Latency

5.3.2 Noise

Noise is here set to 0.2, kp(P): 0.1, kp(PID): 0.102, ki(PID): 0.06, kd(PID): 0.013.

When comparing the controllers against noise (Fig. 3, 4) none managed to stay in the viability band (VB) for at least 3 consecutive seconds. This is explained by high noise and a pretty tight viability band. We see nonetheless, that during the 20 seconds in which the simulation ran, the PID (b) controller's Total Cost is significantly lower than the one of (a), meaning that again, the (b) controller is most efficient in noisy situations. If the VB were larger or noise smaller, both controllers would have managed.

(a) P-controller

(b) PID-controller

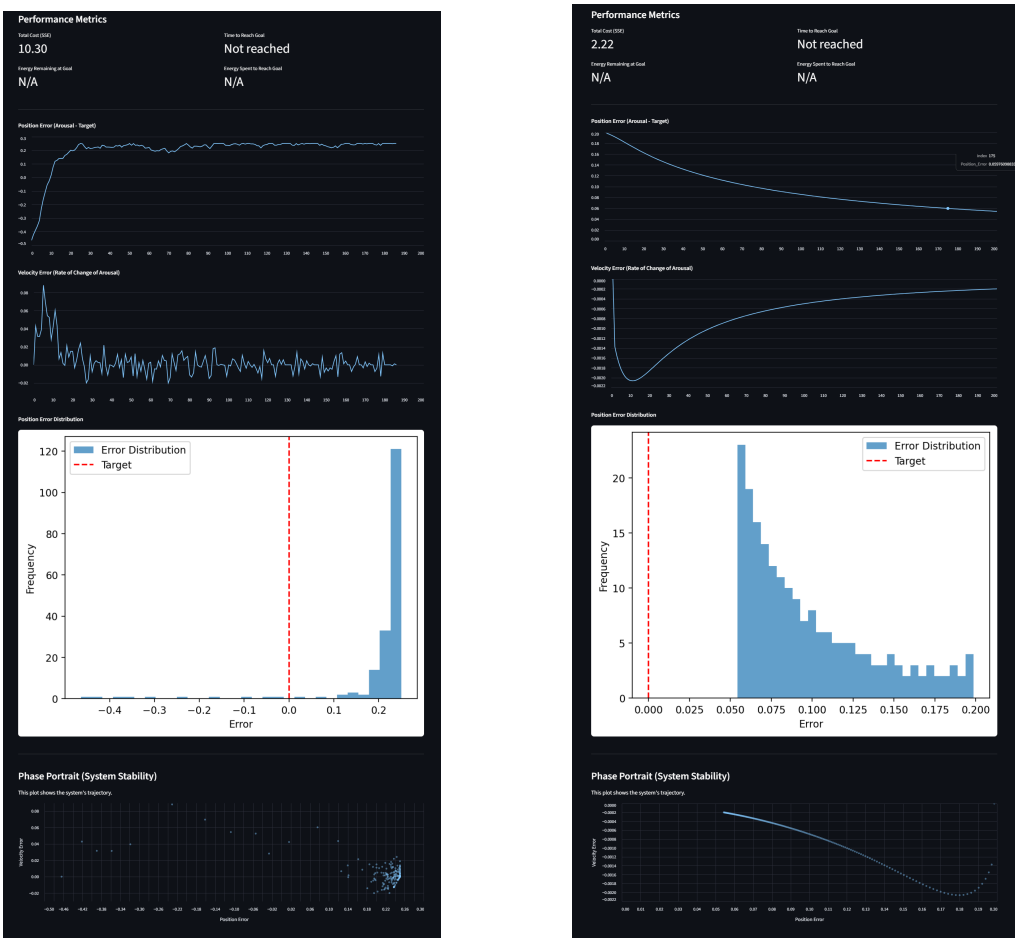
Figure 7: Noise

5.3.3 Environmental Forces

Environmental threat/force (extra high arousal push) is here set to 0.3, kp(P): 0.1, and kp(PID) continuously increases.

When comparing controllers against strong external forces (Fig. 5, 6), (a)'s gain is not strong enough to fight off this opposing force, and is constantly overshooting. As for (b), both PE and VE plots look very differently from (a). Its PE shows that it was able to fight off the external forces and got very close to the target arousal. When we look at the Position Error Distribution graph, we can see that (b) was getting increasingly closer to the target, and the error was way lesser than that of (a). Actually, if we had let the simulation run for a little longer, (b) would have finally ended up in the VB and thus succeed. Since it never got to set its `ki` and `kd` parameters because of continuous

movement, it did not adjust for its best of outcomes. Again, (b) did way better at reducing the Total Cost. The linear movement shown in (b) inside the Phase Portrait shows a convergence towards a linearly stable equilibrium, but not a dynamically rich system (no oscillation, no phase lag).



(a) P-controller

(b) PID-controller

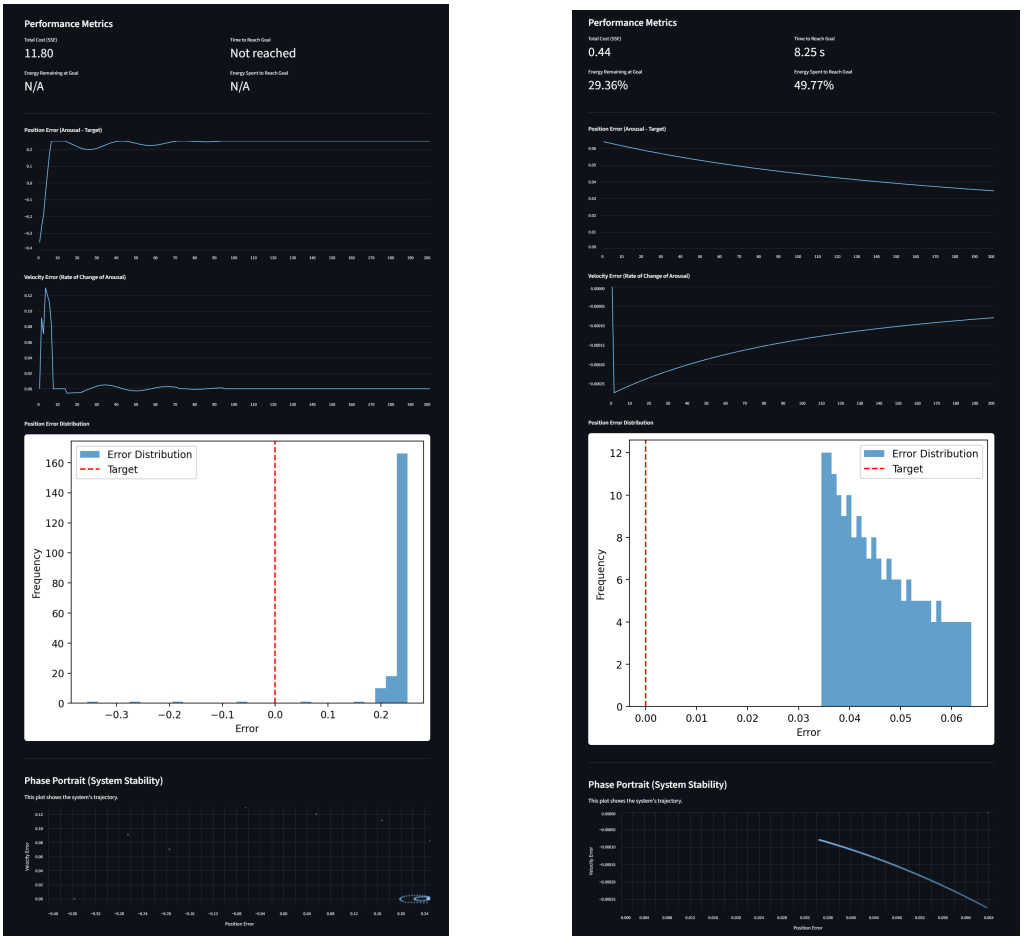
Figure 8: Environmental Threat

5.3.4 Environmental Forces with Latency

Environmental threat/force is here set to 0.3, latency: 7, kp(P): 0.1, and kp(PID) continuously increases.

When comparing controllers against strong external forces AND latency (Fig. 7, 8), (a)'s gain is not strong enough to fight off the opposing force again, and thus fails to stabilize. (b) on the other hands succeeds at this in about 8 seconds, and while also being more efficient at doing so. Even though the error distribution on the Possition Error Distribution graph is still not on the target, it is only because it is not exaclty on the target arousal, but had managed to stay inside the viability

band for at least 3 seconds. The Phase Portrait shows a smooth line moving in the direction of the target arousal. As you can see, it is in the order of the hundredths (10^{-2}) compared to (a) which is in the order of tenths (10^{-1}), meaning that (a) was quite on target, just never got to the other side of it, thus explaining the Position Error Distribution.



(a) P-controller (b) PID-controller

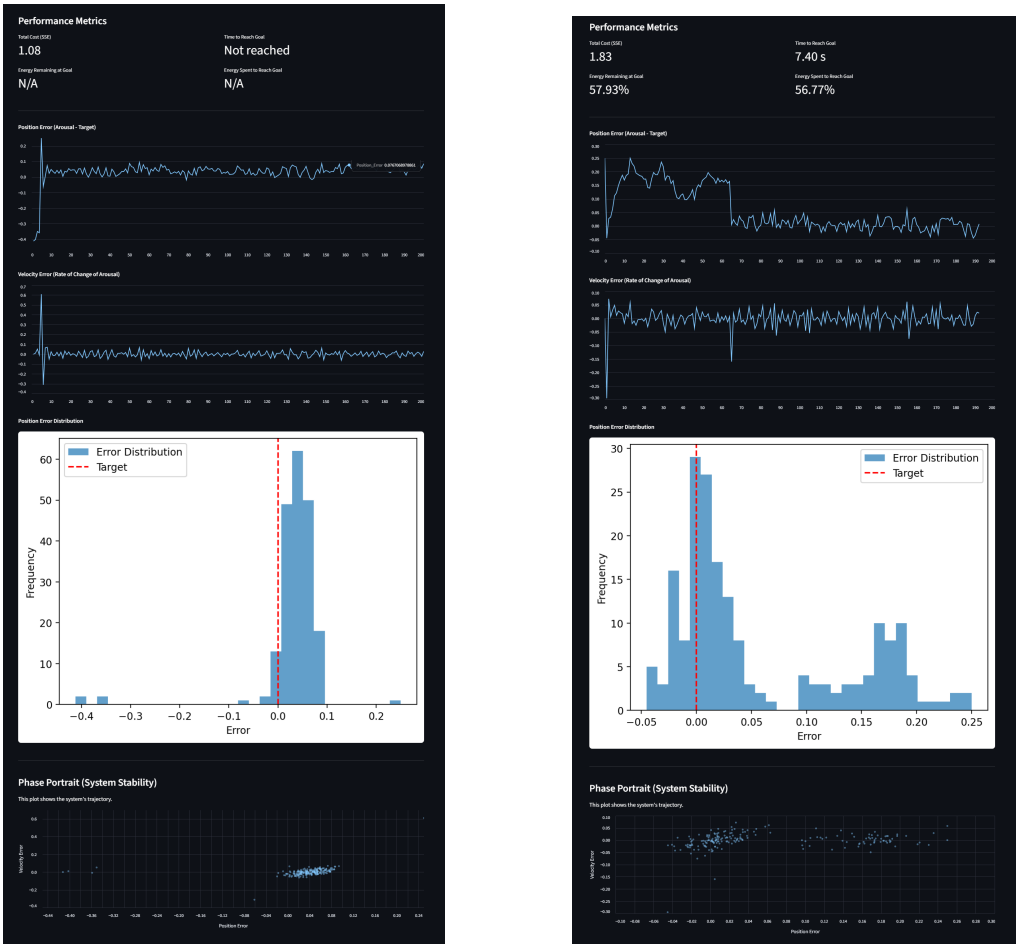
Figure 9: Environmental Threat and Latency

5.3.5 Environmental Forces with Noise

Environmental threat/force is here set to 0.3, noise: 0.02, kp(P): 1, kp(PID): 0.101, ki(PID): 0.057, kd(PID): 0.013.

When compairing the controllers against strong external forces AND noise (Fig. 9, 10), it is again only the PID controller that is able to stabilize after 7.40s, even after multiplying P's gain (kp(P))

by 10! When looking at the (b) PE and VE plots, we see that at around sample 65, there is an abrupt change in speed and distance to the target. That marks the moment where the auto-tuned parameters have been found by the model, and it is after that that the system stabilizes. Its Zieger Nichol's parameter values are very similar to those that it got when the only external perturbation was noise, meaning that it is what helps the model adjust better. Without the additional `ki` and `kd` parameters, a simple P controller (a) is not able to do anything about noisy environments, even if it was able to better work against the opposing environmental forces by giving it maximum gain.



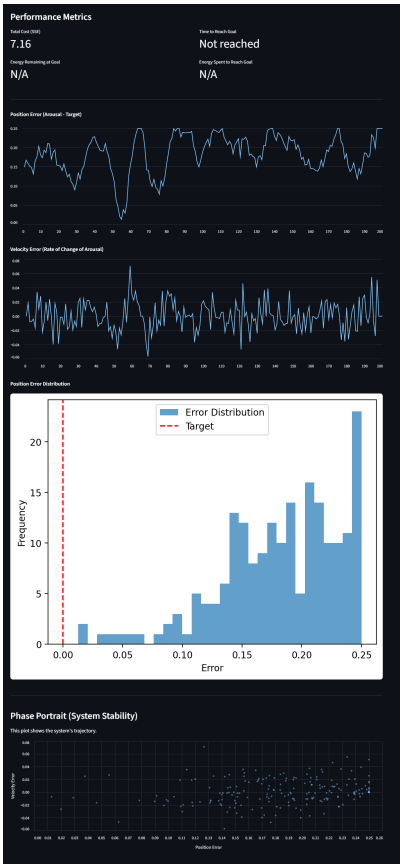
(a) P-controller (b) PID-controller

Figure 10: Environmental Threat and Noise

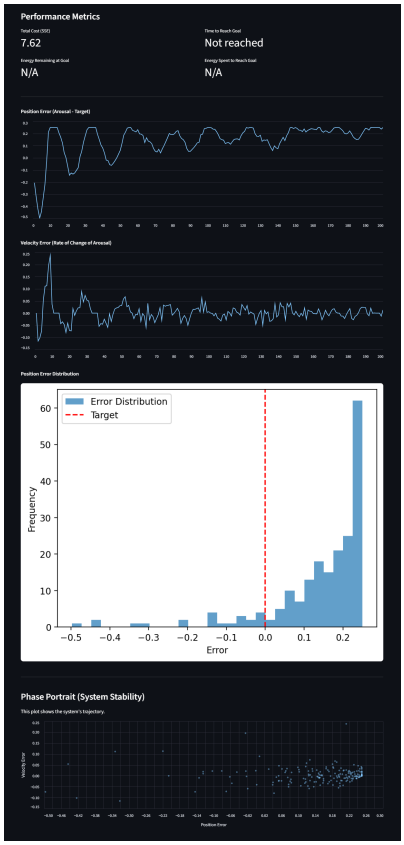
5.3.6 Environmental Forces with Latency and Noise

Environmental threat/force is here set to 0.3, noise: 0.02, latency: 5, kp(P): 0.15 (Fig. (a)), kp(P): 0.5 (Fig. (b)), kp(P): 1 (Fig. (c)), kp(PID): 0.101, ki(PID): 0.057, kd(PID): 0.013 (Fig. (d)).

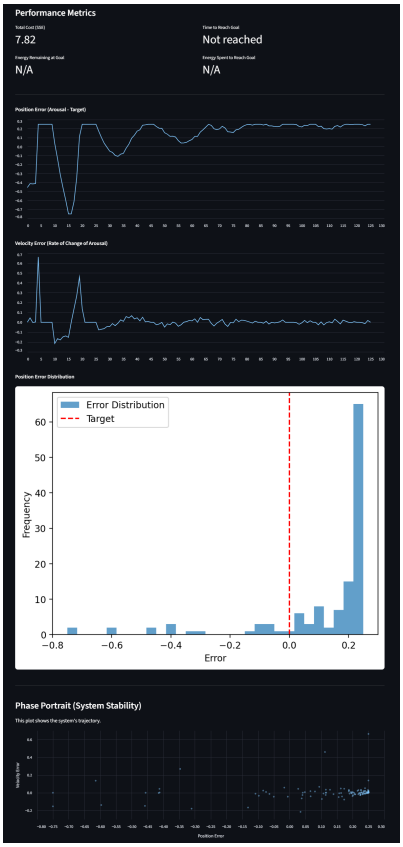
For this last trial (Fig. 11), the controllers will face all external perturbation forces at once: strong external forces, latency and noise altogether. 3 different trials are run for the P-controllers: figures (a), (b) and (c) respectively, with different kp values. All fail to stabilize. For trial (a), nothing surprising: the controller is not strong enough to fight off the high external forces. For trial (b), it is able to fight them off a little bit, but because of the latency, it ends up spending its energy faster than it was able to reach the target arousal: we can see that by looking at the PE graph, showing big oscillations at first due to overshoots caused by latency. In the VE plot we can see that the speed at which the system moves gets slower with each few samples, indicating that the system is losing energy fast. In the end, there is no energy left and the system is only subject to external high arousal forces, being shown in the Position Error Distribution plot and the Phase Portrait. Lastly, for trial (c) we see in the PE and VE plots big oscillations quickly reduced to nothing. That is the product of the high gain plus the latency and the high environmental noises, causing the system to overshoot in both directions very strongly and thus exhausting its energy ressources in two big jumps.
Again, only the PID-controller (d) has been able to successfully stabilize, even if it cost it half of its total energy to reach its goal. When looking at its Position Error Distribution and Phase Portrait, we can see that it manages to stay in a cluster of points that are spread out, but still remain inside the viability band. We can also see a smaller cluster of points in its Phase Portrait that are all the way to the right, and these points represent the system before it auto-tuned its parameters and was mostly subject to extreme environmental forces pushing its arousal index at its limits.



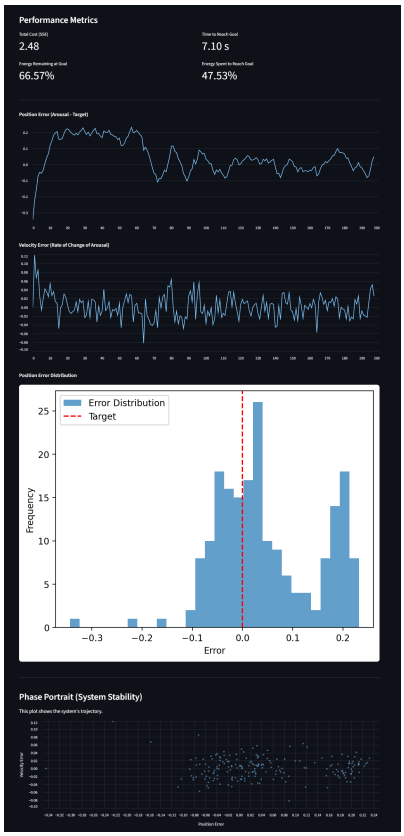
(a) P-controller, kp=0.15



(b) P-controller, kp=0.5



(c) P-controller, kp=1



(d) PID-controller

Figure 11: Environmental Threat, Noise and Latency

5.3.7 High Latency P-controller

As a bonus, below is a screenshot of the oscillations generated by the PE and VE of the P-controller under high latency conditions, and nothing else

Latency is here set to 10, kp(P): 0.1.

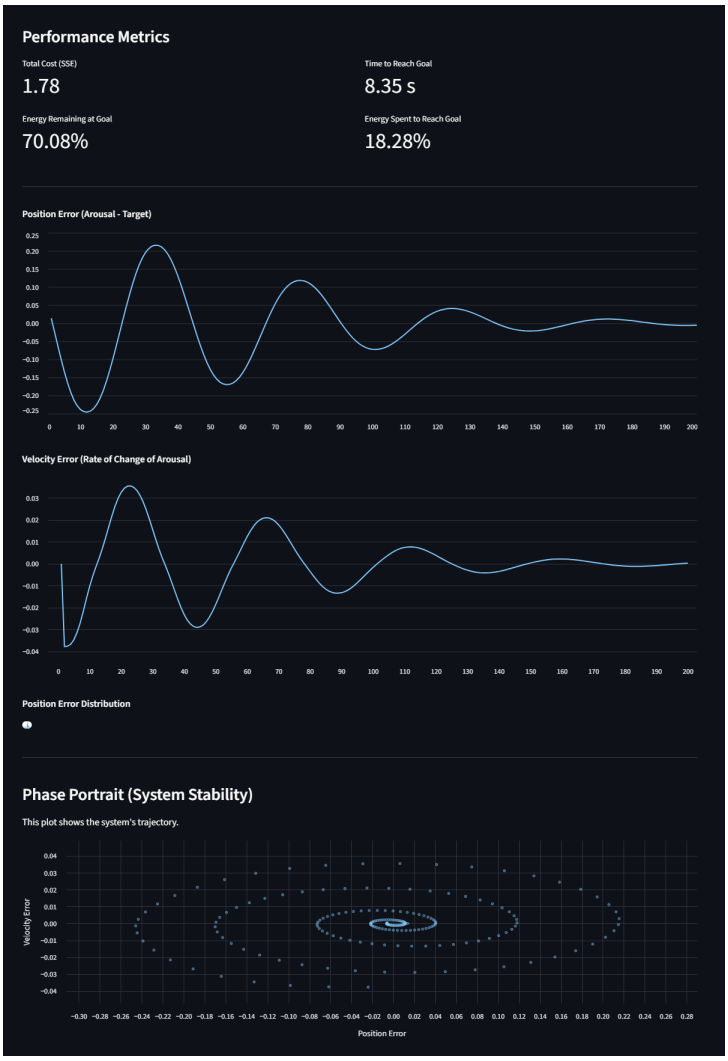


Figure 12: Bonus beautiful phase portrait

The PID controller is clearly better suited for the most harsh environments. Next step is comparing this controller to other advanced ones.

6 Conclusion

All in all, this project has been pretty interesting. I got to understand feedback systems more and their role in all kinds of different stable systems. I have become intrigued in the oscillations that different aimed movements create, and how superposing these oscillations sometimes can create patterns, which are related to a system's stability, just like how we find them in nature. One famous example is the Fibonacci spiral, that I found out appears when certain specific conditions are met:

- Hysteresis: the system has memory, and it exists through a time continuum.
- Local interaction rules: each new addition depends only on nearby or recent structured and not global optimization.
- Discrete incremental growth steps: new elements are added sequentially.
- Constrained environment: limited area forces efficient behavior.
- Scale invariance: same rule applies across growth scales.
- Feedback Loops: continuous feedback between structure and growth.
- Golden angle: ≈ 137.5 degrees of (most practical) irrational angular offset prevents overlap and periodic locking.
- Rules: well constructed rules create beautiful things. Efficiency emerges from then rather than being directly enforced.

Now back to more relevant stuff.

This project started for me as a way to experiment with EEG signals, but it evolved into a deeper look at why maintaining stability (focus in our example) is so difficult both for computers and humans.
In Real Mode, I found that while the system works, it is incredibly sensitive. Because every day is different (and every calibration varies), relying on relative changes between Alpha and Beta waves was a good choice, but it still feels a bit like a "black box." I realized that without a good amount of data to train a model, simple biological feedback has its limits.
The Simulation Mode was where I could actually play with different parameters and test different hypotheses. The results showed that the PID controller is definitely superior to the simple P-controller when things get stressful (high noise or threat), but it all comes at a price. The most interesting finding was seeing the "burnout" mechanics: if I tuned the PID gains too high to get "perfect" stability, the system just burned through its energy and crashed, resulting in total system failure.

Overall, this project showed me that the best controller isn't necessarily the one with the lowest error, but the one that manages its energy best so it doesn't break down in the long run, which was expected. Time was the only limit to trying out even more different approaches, which I found to be quite fun/interesting since there wasn't much to do with the Real EEG mode.