**The Open University of Israel**

**Department of Mathematics and Computer Science**

# A Smart Wearable Outfit System for Anomaly Detection

**Thesis (Advanced Project 22997) submitted as partial fulfilment of the requirements towards an M.Sc. degree in Computer Science**
**The Open University of Israel**
**Department of Mathematics and Computer Science**

**By**
**German Shiklov**

**Prepared under the supervision of Prof. (Dr.) Leonid Barenboim**

2023 / 5

**Abstract**

Wearables, microcontrollers, smartphones, have become more popular over the last decade. We seem to have come a long way to the revolution of wearable technologies, and yet there is still a lot to do. This area has lots of fields to improve in, whether it is the overwhelming amounts of information collected, the battery's power consumption, wiring faults, environmental effects (deviation) and many more. This study aims to portray a smart wearable outfit for the purposes of normal activity recognition and abnormal (anomalous) detection. Specifically, it investigates experiments that have been recorded for different user activities and aims to find the best usage for real-time functionality. In this context, our wearable device - smart suit, is capable of classifying whether the user is in the state of idle, walking or other activity, moreover, we clearly state if the signal is invalid or abnormal regarding normal activity. To test the hypothesis that such a smart suit can classify and differentiate normal and abnormal events, we had to validate our methodologies of LSTM, Matrix-Profile and Self-developed model to work with the acquired offline data. Only then, we base our real-time system to operate with the best fitting ideas for real-time response. In this work, we describe a software system for data acquisition via microcontrollers (Arduino or Teensy). And a real-time software system to operate the smart suit via System-on-Module controller (Jetson Nano). We discovered in our results that Artificial Intelligence models, as LSTM, are less effective in terms of time consumption. In addition, the classification is found to be less effective for anomalous detection, since anomalies have different facets. Our self developed models have shown outstanding performance in time, classification and detection. With the aforementioned key findings in regard to recorded data, we use it as the building block foundation for the design and implementation of the real-time monitoring system of the suit. This project shows it is possible to forge magic - smart wearable technology that is helpful in a broad range of applications.

*Keywords:* Wearable-IoT, innovation, engineering, artificial-intelligence.

## Abbreviations and Notations

RNN: recurrent neural network
LSTM: long short term memory network
AE/DE: autoencoder / decoder
VAE: variational autoencoder
FFT: fast fourier transform
iFFT: inverse fast fourier transform
MASS: mueen's algorithm for similarity search
TSA: time-series analysis
ANN: artificial neural network
FSR: force-sensing resistor
Piezo: piezoelectric film
IMU: inertial measurement unit
DMA: direct memory access
PCT: percent change
OOP: object oriented programming
BSN: body sensor network
FSM: finite state machine

**Table of Contents**

**List of Figures**

## List of Tables

## Introduction

Over the past decade, wearable technology has evolved dramatically, with innovations such as smartwatches and smart sport garments becoming increasingly popular. However, existing systems often lack a widely applicable software architecture. In this paper, we propose a general software system for System-on-Module (SoM) devices that can interface with various sensors and microcontrollers. We refer to the SoM as the Server Node, as it acts as the receiver for data transmitted by microcontrollers.

### Microcontrollers in Usage

The primary responsibility of microcontrollers is to acquire data from interconnected sensors via Serial communication. To enhance communication and data processing, we convert sensor data into bytes and transmit it in binary format, significantly improving efficiency compared to using string format. We employ high baud rates for data transmission, although lower rates would be sufficient for human activity recognition due to the long-term nature of action signals.

We also utilised a Myo armband, an electromyography (EMG) device, to collect real-time data through Bluetooth-Serial communication with our Jetson Nano B01 Server Node. Although we did not analyse this data extensively in the current study, it provided additional context for our research.

### Sensors in Usage

Our proposed system is compatible with various sensor types, depending on the desired application. The sensors used in this project include Force-Sensing Resistors (FSRs), Piezoelectric Films, Accelerometers, and EMG devices such as the Myo armband. These sensors provide different types of data to facilitate the analysis and classification of user activities, as well as the detection of abnormalities.

### Server Node: Jetson Nano B01

The Jetson Nano B01 is an effective SoM for AI deployment across various industries. While Nvidia has produced many SoM versions, none are currently wearable. Our server node has a power consumption of 5-10 Watts, which is sufficient for most wearable sensor applications.

In summary, this paper presents a versatile software system for SoMs that can interface with a range of sensors and microcontrollers, allowing for the development of advanced wearable technology applications.

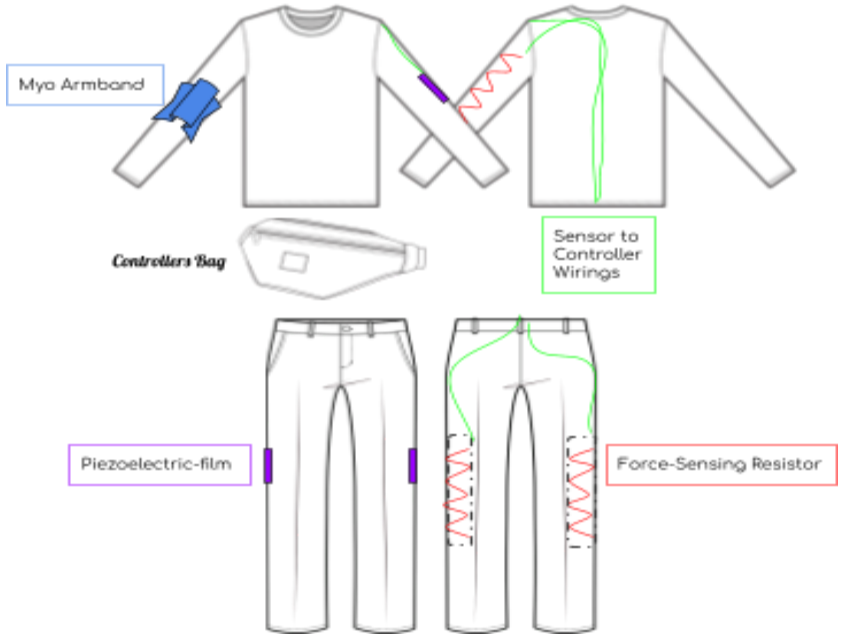


*Figure 1. A high-level depiction of the smart outfit designed and used in practice.*

**Running the Project**

Our project can be run in two modes: Real-time and Offline (Scripts). Below, you'll find instructions for setting up and running the project in each mode.

***Recorder Mode***

In order to load new code into the sub-controllers, execute the project under the PlatformIO environment, while connected with the particular USB required for Teensy or Arduino. Make sure the COM port is recognized.

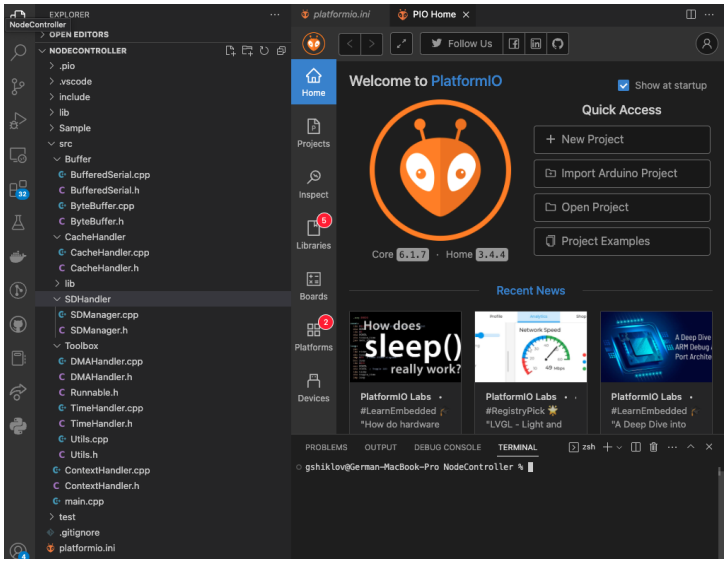


*Figure 2. Our project opened via PlatformIO in VisualStudioCode.*

Creating a new data file will require using an SD-Card in Teensy, either Arduino with SD-Card reader extension. Then, the user can perform an activity - preferred as a recurring pattern.

***Offline Mode***

In offline mode, the project does not require the suit in practice, but only data analysis. It operates using only the resources available on your local machine. Here's how you can run the project in offline mode:

1. **Setup**: Make sure you've completed the project's initial setup - by opening it in Visual Studio Code environment, including installation of Python and PlatformIO extensions and configuration. This might involve downloading necessary data or resources to your local machine while connected to the Internet, so they are accessible for offline use.
2. **Start the Project**: Open the project. Since you're running in offline mode, you won't need to login. Under *Scripts,* find the desired python to run. Make sure you have installed all required libraries in the environment, to successfully execute the scripts. Run the script files via code-cell executions in the VSC platform, similar to Jupyter Notebook.
3. **Access Local Resources**: Interact with the project using the resources (data files) stored on your local machine, and replace data files that you wish to verify by yourself. Remember that any changes made or data entered will only be saved locally, and will not be updated on the server until your commit is updated in GitHub.
4. **Exit**: When finished, simply exit the project.

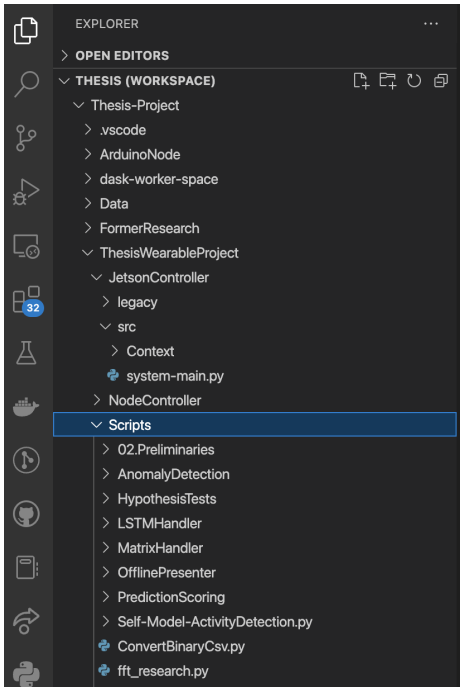


*Figure 3. Scripts, as root folder for running analysis and code on pre recorded data.*

***Real-time Mode***

In real-time mode, the project runs in the Jetson and accesses various microcontrollers connected to it via USB or Bluetooth, the sub-devices need to operate, and the control software will receive the data and evaluate the activity or anomaly in accordance with the user's pre-coding experiment. Follow the steps below to run the project in real-time mode:

1. **Ensure Peripherals are Interconnected**: Ensure your device has a stable internet connection. The project needs this to access the necessary online resources.
2. **Execute**: Open the project in terminal and enter run the *system-main.py*, under python environment. If you do not have the required libraries, install it.
3. **User-Interface**: If the project runs on a non-Jetson Nano, then the *UserInterfaceHandler* is not activated to display leds intact. Then it's the user's responsibility to tweak and present in terminal or offline csv the real-time outputs.
4. **Exit**: When you're done, you can exit the project by Control C. Any data that is available to be recorded by the system is entered into a csv, and automatically saved to the current project's folder.

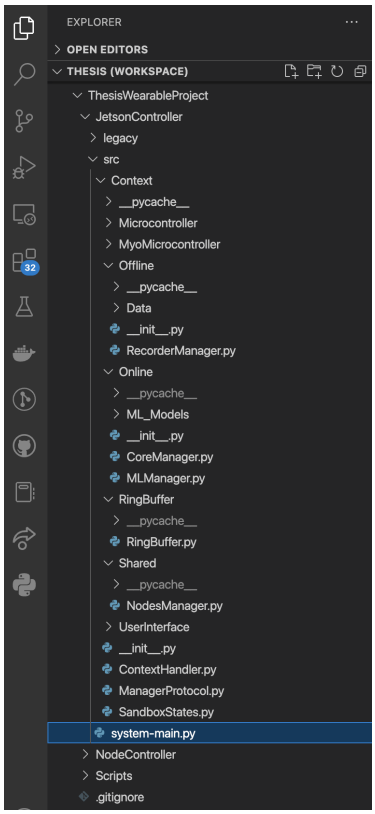


*Figure 4. VSC real-time project, execute in terminal/cmd the 'system-main.py' file.*

## Architecture

During the last decade, wearable technology has advanced nearly from zero to extent, where people wear smart-watches, smart sport garments and more. Yet, those systems did not provide any software architecture to be used in a widely spread aspect. In this paper we propose a general software system for SoM, to interconnect with any sensors and microcontrollers. Let us refer to the SoM as the *Server Node*, as it acts as the receiver from the microcontrollers.

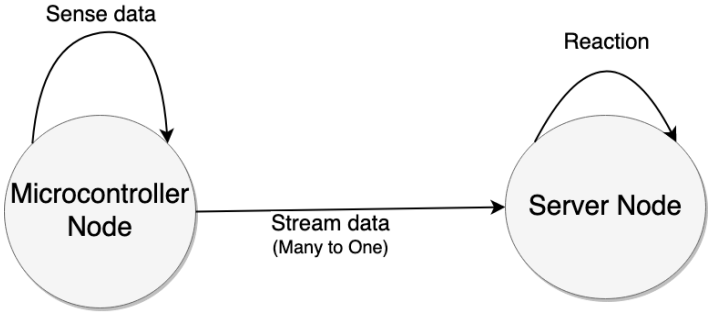


*Figure 5. A high level depiction of the node's responsibilities.*

**Table 1. A depiction of microcontroller sample rate in our project.**
*Different Sampling Rates*

| Type | Sample Rate |
|---|---|
| Myo - IMU | 50 Hz |
| Myo - sEMG | 200 Hz |
| Arduino / Teensy / Jetson | 2000 Hz |

The server node, Jetson Nano B01, delivers great performance in AI deployment across industries. Nvidia has produced many versions of SoM's, though they are not yet wearable. Our server node has a power consumption of 5 to 10 Watts which should be sufficient in most general cases of sensors via wearables.

As stated earlier, any sensor with logical attachment can be used in the proposed system. Table 2 describes more in depth of the sensors intact and their use-case scenario.

**Table 2. Sensors in depth.**
*Wearable Sensors in-depth Description and Practices.*

| Type | Description | Practically |
|---|---|---|
| Force-sensing Resistor | This sensor is a material sensor whose resistance changes in case of force, pressure or stress that is applied to it. It's a one-dimensional sensor, that is put over the smart suit, and refers as $(fsr_0, fsr_1, \ldots, fsr_k)$. | The data outputs of this sensor enabled a good analysis and classification of activity footprint. As well as abnormality recognition. |
| Piezoelectric Film | The piezo is actually crystals that generate electrostatic currencies in accordance with the direction of film being stretched. This sensor adds one additional layer to the input's dimensionality, $(p)$. | Although this sensor resembles the FSR, its sensitivity is much higher and requires more gentle user activities. |
| Accelerometer | This device uses an electromechanical sensor to measure static or dynamic acceleration. Acceleration sensing operates via three dimensional axis, $(x, y, z)$ and is able to tell distance change across time-series. | It is commonly used in a wide variety of research and acts as a ground-truth reference for the other sensors. |
| Electromyography (Myo) | It operates within eighth muscles - $(emg_0, emg_1, \ldots, emg_8)$, and thus muscle activity can be monitored and classified into the different states of the user's activity. | Wide research is done across gesture recognition via muscles being intense while showing hand gestures. |
| Gyroscope (Myo) | This device maintains measurements of rotational motion. A unit of measure is referred to as angular velocity, their units are measured in degrees per seconds or revolutions per second. A gyro simply tells the speed of rotation across $(x, y, z)$ axis. That said, the gyro should measure a recurring pattern of degree shifts for each activity. | Motion capture devices can suit the needs of smart sports and medical wearables - capture the correct motion and compare to actual input. |

## Related Work

Over the past decade, a considerable amount of research has been conducted in the area of wearable technology. While getting-started projects, as seen in different datasheets and websites by Arduino R3 (2022), NXP-Teensy (2022), and Nvidia-Jetson (2022), provide a useful starting point, the proposed embedded software is often primitive and not suitable for industry applications. As software scales during project development, code maintenance becomes challenging. Therefore, we propose a high-level approach incorporating Object-Oriented Programming (OOP) design patterns.

Mukhopadhyay (2015) presented a Body-Sensor-Network review and highlighted the significant challenge of delivering real-time data. Additionally, wearable technology has been widely applied in healthcare for monitoring and emergency medical response systems. Building on these ideas, we aim to develop software that prioritises real-time effectiveness over fault tolerance. We draw inspiration from Cho & (Ed.) (2009, ch 6), who presented key concepts for designing and building a wearable outfit, including central processing devices, wire-based networks, and signal integrity.

Kamble et al. (2022) recently proposed a fall detection system for wheelchair users utilising accelerometer and gyroscope sensors. While their system can identify falls, it still requires user verification, which may not be feasible in real-world scenarios. In contrast, our wearable outfit and algorithms can simulate detection through force application on smart fabric, adapting to different emergency medical cases.

Khan et al. (2019) also developed a patient fall detection system, achieving 95% accuracy in detecting non-fall activities using k-Nearest Neighbour and Bayes Classifier algorithms. Their accelerometer data is similar in pattern to our smart outfit's accelerometer, providing a 'ground-truth' for the Force-Sensing Resistor (FSR) sensor. However, their wearable design is not suitable for long-term use due to the uncomfortable chest band. Our smart outfit, made of cloth, offers a more comfortable and non-constraining solution for users. Despite these improvements, further considerations regarding fabric breathability are needed for long-term wearability.

## Software System - Microcontroller(slave) Nodes

The `Platform.io` framework is used for developing the embedded system via Teensy or Arduino. It contains an acquisition feature for the offline analysis and a real-time streamer for the online handler of the Master Node.

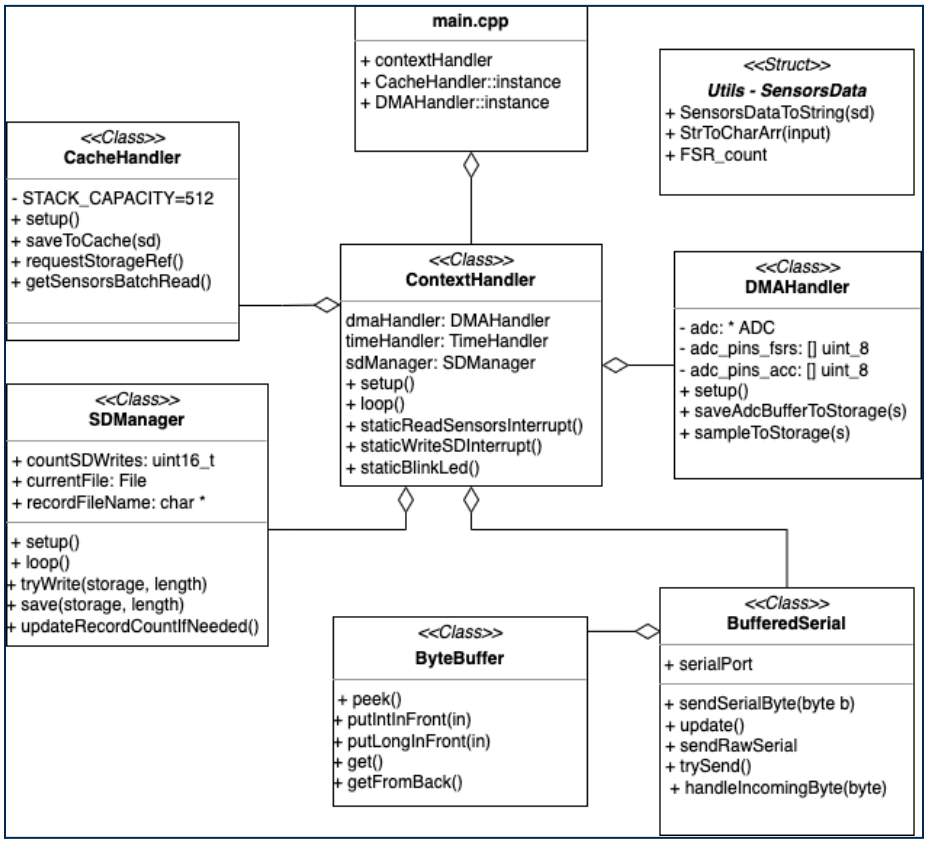


*Figure 6. A UML depiction of a Node-microcontroller.*

### Main

In both cases of Arudino and Teensy, the framework's main method is responsible for operating via `*setup*` and `*loop*` methods. In our case, a `ContextHandler` runs its setup and loop methods to comply with the hardware's life-cycle.

```
#include "ContextHandler.h"
DMAHandler* DMAHandler::instance;
CacheHandler* CacheHandler::instance;
ContextHandler contextHandler;
void setup() { contextHandler.setup(); }
void loop() { contextHandler.loop(); }
```

*Figure 7. Code of main.cpp.*

**Context Handler**

A context handler operates to set various `*Interval Timers*` which are simply interrupts to dispatch necessary functionalities. Those functions are sanity operability - a built-in blinking led on the hardware. A sensor reader, to cache the real-time stream. And two separate writing mechanisms, one for writing data in offline mode - straight to an SD-card placed on board, the other is to verbose real-time streaming into the serial port, so that the Master Node is able to decouple and predict in accordance with it.

```
void ContextHandler::staticReadSensorsInterrupt() {
   static unsigned long counter = 0;
   noInterrupts();
   SensorsData currentStorage = CacheHandler::requestStorageRef();
   DMAHandler::sampleToStorage(currentStorage);
   send_buffer.putInt(currentStorage.piezo); // update buffer with storage
   for (int index = 0; index < 8; index++) {
       send_buffer.putInt(currentStorage.fsrs[index]);
   }
   if (counter > UINT64_MAX - 2) {
       counter = 0;
   }
   counter++;
   interrupts();
}
void ContextHandler::staticWriteSDInterrupt() {
   noInterrupts();
   SensorsData* storage = CacheHandler::getSensorsBatchRead();
   if (storage != NULL) {    // Write if data available
       SDManager::tryWrite(storage, CacheHandler::STACK_CAPACITY);
   }
   interrupts();
}
void ContextHandler::staticWriteSerialInterrupt() {
   if (!serial.isBusySending()) {
       serial.sendSerialPacket(&send_buffer);
   }
}
void ContextHandler::staticBlinkLed() {
   static bool state = true;
   state = !state;
   digitalWriteFast(sanityBlinkLed, state ? HIGH : LOW);
}
void ContextHandler::setup() {
   serial.init(0, this->baudrate);
   send_buffer.init(BUFFER_SIZE);
   pinMode(sanityBlinkLed, OUTPUT);
   DMAHandler::setup();
   timeHandler.setup();
   sdManager.setup();
   CacheHandler::shared()->setup();
   PIT_readSensors.begin(staticReadSensorsInterrupt, 125);
   PIT_sdWrite.begin(staticWriteSDInterrupt, 500);
   PIT_serialWrite.begin(staticWriteSerialInterrupt, 2000);
   PIT_blink.begin(staticBlinkLed, 500000); // half second
}
void ContextHandler::loop() {
   serial.update();
   timeHandler.loop();
   sdManager.loop();
   noInterrupts();
   TimeInSec = now();
```

```
    interrupts();
    return;
}
ContextHandler::~ContextHandler() { }
```

*Figure 8. Code of ContextHandler.cpp*

**DMA Handler**

A Direct-Memory-Access handler is a singleton, encapsulating in itself the functionalities necessary to set both ADC (Analog-to-Digital-Converter) for sampling at higher rates the sensors intact. It sets the resolution bits to be of 12 - maximum values of $2^{12}$ per sensor. A conversion rate and sampling speed at very high speeds. And an averaging of samplings of 32, which is considered highest as well. Finally the values are stored into a `*SensorsData*` struct which is later on used by the Context Handler.

```
class DMAHandler {
private:
    DMAHandler();
    ~DMAHandler();
    static DMAHandler* instance;
    DMAHandler(const DMAHandler &dmaHandler);
    const DMAHandler &operator=(const DMAHandler &dmaHandler);
public:
    static void setup(); // TODO: should allow once
    static DMAHandler* shared() {
        if (instance == 0) {
            instance = new DMAHandler();
        }
        return instance;
    }
    static volatile void saveAdcBufferToStorage(SensorsData &storage);
    static volatile void sampleToStorage(SensorsData &storage);
};
#endif
```

*Figure 9. Code of DMAHandler.h.*

```
volatile void DMAHandler::saveAdcBufferToStorage(SensorsData &storage) {
   float voltage = 3.3;
   uint32_t MaxValue = adc->adc0->getMaxValue();
   storage.piezo          = adc->analogRead(adc_pins_piezos[0] * voltage / MaxValue) ;
   storage.accelerometer[0] = adc->analogRead(adc_pins_acc[0] * voltage / MaxValue);
   storage.accelerometer[1] = adc->analogRead(adc_pins_acc[1] * voltage / MaxValue);
   storage.accelerometer[2] = adc->analogRead(adc_pins_acc[2] * voltage / MaxValue);
   const byte ADC_MAX_FSR = 6; // 6
   const byte StructIndent = 2; // 2
   for (byte fsrIndex = 0; fsrIndex < ADC_MAX_FSR; fsrIndex++) {
       storage.fsrs[fsrIndex] = adc->analogRead(adc_pins_fsrs[fsrIndex] * voltage /
MaxValue);
       storage.fsrs[fsrIndex+ADC_MAX_FSR + StructIndent] =
           adc->analogRead(adc_pins_fsrs[fsrIndex+ADC_MAX_FSR + StructIndent] *
voltage / MaxValue);
   }
}
volatile void DMAHandler::sampleToStorage(SensorsData &storage) {
   // Convert to voltage => value * 3.3 / adc->adc0->getMaxValue()
   float voltage = 3.3;
   uint32_t MaxValue = adc->adc0->getMaxValue();
   storage.piezo            = (uint16_t) (adc->analogRead(adc_pins_piezos[0])) ; //
   const byte ADC_MAX_FSR = 8;
   for (byte fsrIndex = 0; fsrIndex < ADC_MAX_FSR; fsrIndex++) {
       storage.fsrs[fsrIndex] = (uint16_t) (adc->analogRead(adc_pins_fsrs[fsrIndex]));
   }
}
```

*Figure 10. Code of DMAHandler.cpp*

**Time Handler**

It is a simple clock mechanism that supports more than just returning the number of seconds since the start of the Hardware. Meaning, it is processing Teensy's clock in order to return the number of seconds since 1970. Such mechanism supports offline analysis in knowing the `approximate` time the sensors were sampled. It is important to notice that even though the sampling of sensors is done swiftly and stumped with a timestamp, the actual timing between sampled sensors has a few milliseconds of drift. For the sake of this research, we ignore the miniature drifts, as they are less critical in our real-time system of recognizing activities which span in durations of seconds.

**Utils**

Utils file has useful properties for the sake of the system. First it contains the `*SensorsData*` struct, which is used widely across other objects as an object of the sensor's data. In addition, a helper function for the `*TimeLib*`, to instantiate in a proper manner. And a String to Char array converter, for the purposes of file name handling.

**Buffered Serial / Byte Buffer**

A Byte Buffer is responsible for holding an array of bytes, and has the feature of putting data into that buffer, which will represent a packet with brackets.

$Byte\ Buffer\ =\ [\ '<',\ 'b_0',\ \ldots,\ 'b_k',\ '>'\ ]$.

Buffered Serial handles the aforementioned, and brings the features of sending a packet of bytes over the serial.

**SD Manager**

An SD manager takes care of the files that are saved into the SD card. It has two methods, the first is an interrupt, controlling the writing into the file, via binary format writing the struct of `*SensorsData*`. And the latter method which takes care of updating the record file name upon the reach of an arbitrary limit.

```
class SDManager {
private:
   uint16_t countSDWrites;
   File currentFile;
   char *recordFileName;
   char *recordHeader;
   void initializeSD();
   void clearSD();
public:
   SDManager();
   void setup();
   void loop();
   static void tryWrite(SensorsData* storage, uint16_t length);
   void save(SensorsData* storage, uint16_t length);
   void updateRecordCountIfNeeded();
};
```

*Figure 11. Code of SDManager.h.*

**Cache Handler**

A class to organise the caching for the streamed data. It has a method for requesting a storage reference of the data object to be populated with, via `*DMAHandler*`. Another method when writing into the SD - a reference for the cache is returned if an arbitrary limit of cache has been reached.

```
class CacheHandler {
private:
   CacheHandler(){}
   static CacheHandler* instance;
public:
   static const uint16_t STACK_CAPACITY = 512;
   static CacheHandler* shared() {
       if (instance == 0) {
           instance = new CacheHandler();
       }
       return instance; }
   static void setup();
   static void saveToCache(SensorsData& sensorsData);
   static SensorsData& requestStorageRef();
   static SensorsData* getSensorsBatchRead();
};
```

*Figure 12, Code of CacheHandler.h.*

**Software System - SoM(Master) Node**

A System-on-Module enables high-level functionality execution of artificial-intelligence libraries via Python, and hardware that can handle the processing of slave nodes and its consequent prediction computations. The data is acquired via various short experiments which depict different activities. Figure 13 shows the different states the wearable outfit can infer - that is represented via different led colours, and a buzzer for anomaly detection.

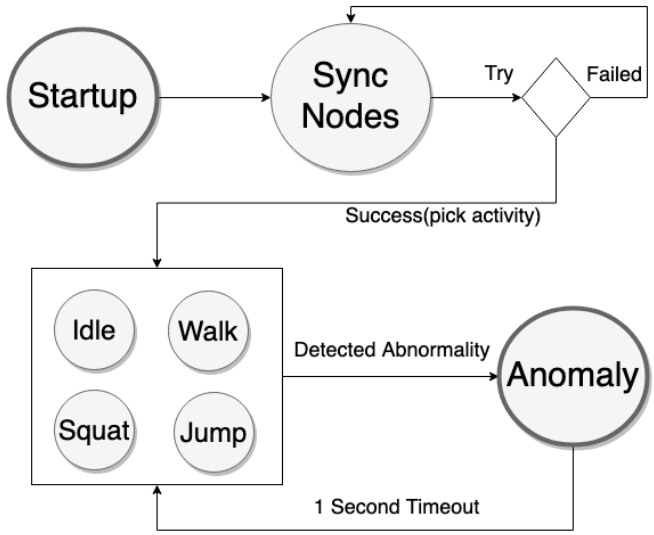


*Figure 13. An FSM depicting the wearable-system.*

Figure 14 depicts in practice Figure 13 schema.
Figure 15 shows a UML representation of the classes intact of our high-level system.

```
class SystemState(Enum):
   Startup = 'startup'
   SyncNodes = 'sync_nodes'
   Unrecognized = 'unrecognized'
   Anomaly = 'anomaly'
   Idle = 'idle'
   Walk = 'walk'
   Jump = 'jump'
   Squat = 'squat'
   def values():
       return [member.value for member in SystemState]
   def key_by(value):
       for member in SystemState:
           if member.value == value:
               return member
       return None
class JetsonNanoGPIO(IntEnum):
   buzzer_pin = 7
   blue_led_pin = 12
   red_led_pin = 16
   green_led_pin = 18
   white_led1_pin = 24
   white_led2_pin = 19
class MasterHardware(str, Enum):
   unrecognized = 'None'
```

```
    jetson = 'Jetson-Nano-B01'
    pc = 'Computer'
class NodesControllers(Enum):
    right_arm = 'Myo'
    left_leg = 'Teensy-3.6'
    left_hand = 'ArduinoUnoR3'
class TermToSensors(str, Enum):
    Teensy = 'Teensy'
    Myo_Motion = 'MyoMotion'
    Myo_EMG = 'MyoEMG'
      def values():
        return [member.value for member in TermToSensors]
class ModelType(str, Enum):
    memory_neural_network = 'LSTM-AE'
    pattern_matching = 'MASS'
    algorithmic = 'SelfMind'
    def values():
        return [member.value for member in ModelType]
```

*Figure 14. SandboxStates.py - Configurations of system.*

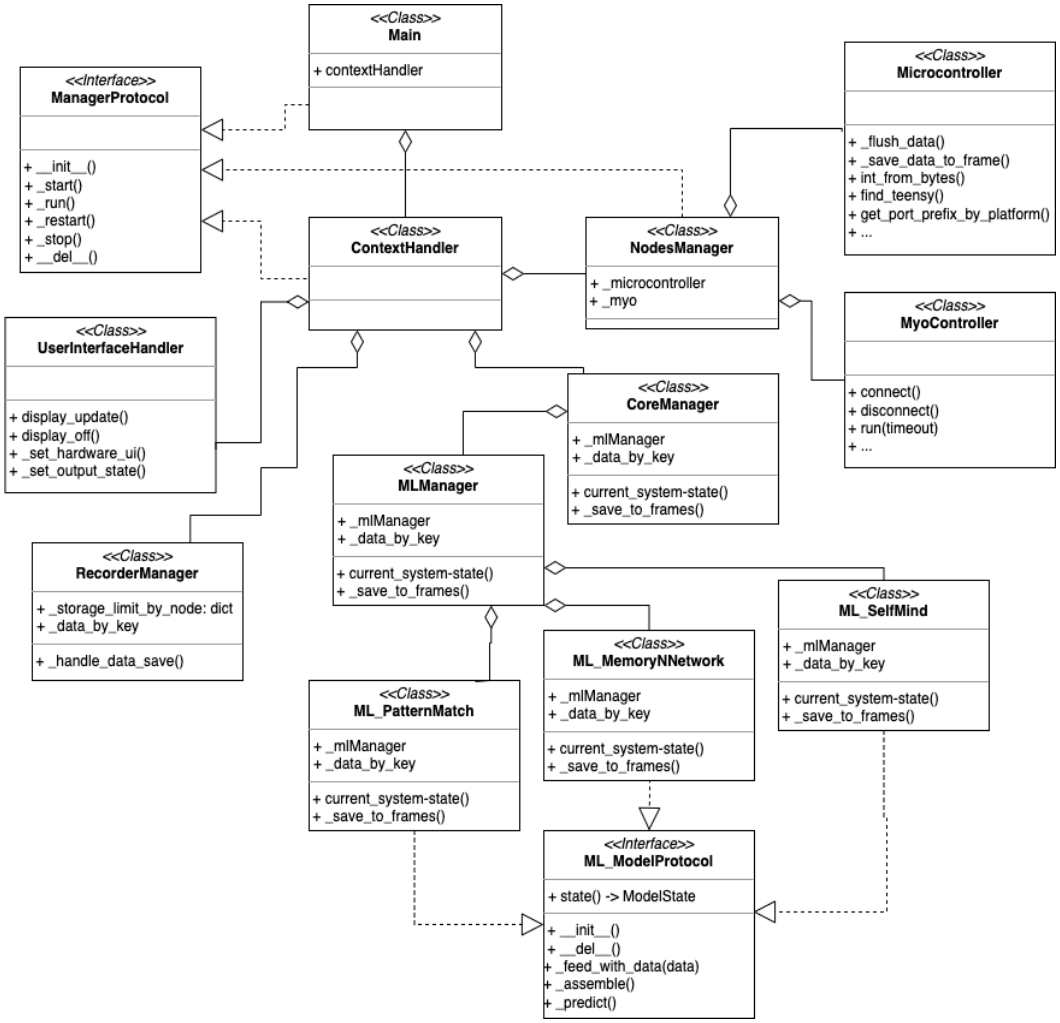


*Figure 15. A light UML of the class's interaction and relations.*

**Main Method**

The main method is the beginning of every software development process. Whereas in our case, this is the starting point for the Main class, which is a clock for our live program to operate at.

```
class Main(ManagerProtocol):
   def __init__(self):
       self.contextHandler = ContextHandler()
       self.increment = 0.001 if ExecutionState.Current() == ExecutionState.Debug else
0.001
       self._restart()
   def _start(self) -> None:
       self.next_t = time.time()
       self.i = 0
       self.done = False
       self.contextHandler._start()
   def _run(self) -> None:
       print("hello ", self.i)
       self.next_t += self.increment
       self.i += 1
       self.contextHandler._run()
       if not self.done:
           threading.Timer( self.next_t - time.time(), self._run).start()
   def _stop(self):
       self.contextHandler._stop()
       self.done = True
# End of Main.
```

*Figure 16. Code of system-main.py.*

**Main Class / Manager Protocol**

Simply, a class to organise the life-cycle of the real-time system. Figure 12 indicates a basic protocol to formulate operability across all other node handlers.

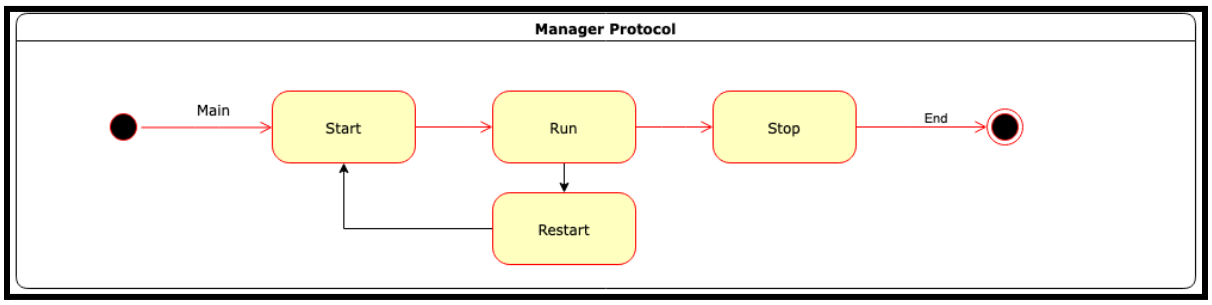


*Figure 17. A general life-cycle for the classes in use.*

Every class complies with type `*ManagerProtocol*`, in order for it to be part of the real-time life-cycle. This architecturing is biassed from microcontrollers, thus there exists a `*setup*` and `*loop*` methods - basically the `*start*` can be thought of as setting up the class, and `*run*` as a continuous loop. Now that the life-cycle is set, a context handler class operates multiple managers and takes control of the relevant functionality of a context. In other words, the recording and deciding what state is currently apparent.

```
class ManagerProtocol(abc.ABC):
    @classmethod
    def __init__(self):
        pass
    @classmethod
    @abc.abstractmethod
    def _start(self) -> None:
        pass
    @classmethod
    @abc.abstractmethod
    def _run(self) -> None:
        pass
    @classmethod
    @abc.abstractmethod
    def _restart(self) -> None:
        pass
    classmethod
    def _stop(self) -> None:
        pass
    @classmethod
    def __del__(self):
        pass
```

*Figure 18. Code of ManagerProtocol.py.*

**Context Handler**

Simply, the heart of the system. It is being run by the *Main*, so that every change of scope ,(record/classify/alert), is under its monitor. It schedules, in a round-robin old fashion way, the different managers to operate solely. Yet, if any update of data is required, it is being delegated to the *Context Handler* for treatment. A delegation of updated data can be referred to as a *Resource*, meaning the resource is shared only via this class to its related context. Unlike the interrupt-based architecture - where event-functions receive a *void* * parameter for context and have difficulty for scaling big, the *Object-Oriented* approach presents more structure and visibility of relation between objects and their responsibilities.

```
    def _start(self):
        didSyncNodes = False
        while didSyncNodes == False:
            self._systemState = SystemState.SyncNodes
            didSyncNodes = self._nodesManager._start()
        print('System State : ' + str(self._systemState))
        self._recorderManager._start()
        self._coreManager._start()
        return True
    def _run(self):
        self._nodesManager._run()
        result = self._nodesManager.get_dict_data()
        self._uiHandler.display_update(self._systemState)
        if len(result) > 0:
            print(result)
            with concurrent.futures.ThreadPoolExecutor() as executor:
                self._recorderManager._run(result)
        with concurrent.futures.ThreadPoolExecutor() as executor:
            self._coreManager._run(result)
        self._systemState = self._coreManager.current_system_state()
```

*Figure 19. Code of ContextHandler.py*

**Recorder Manager**

A class to record live data of the applied microcontrollers and their sensors. Once a certain limit has been reached, the data is output to a CSV.

***Terms to Sensors***

An enumeration of states depicting the table of sensors that are being sampled. Since every table has a different baud-rate, this separation is applied.

```
class RecorderManager(ManagerProtocol):
   def _start(self):
       self._storage_limit_by_node = dict.fromkeys([TermToSensors.Myo_EMG.value,
TermToSensors.Myo_Motion.value, TermToSensors.Teensy.value], 0)
       self._storage_limit_by_node[TermToSensors.Myo_Motion.value] = 150# 30 frames
per second
       self._storage_limit_by_node[TermToSensors.Myo_EMG.value] = 150# 30 frames per
second
       self._storage_limit_by_node[TermToSensors.Teensy.value] = 10000# 2k frames per
second
   def _run(self, nodes_data_dict):
       # For first run, initiate with new dict.
       if self._data_by_key == None:
           self._data_by_key = nodes_data_dict
           return
       for node_key in nodes_data_dict.keys():
           # if nodes_data_dict[node_key] == None:
           self._handle_data_save(node_key, nodes_data_dict[node_key])
   # Features
   def _handle_data_save(self, data_key, data):
       # First initialization for specific table
       try:
           if self._data_by_key[data_key] == None:
               self._data_by_key[data_key] = data
               return
       except:
           pass
       # Append new data
       self._data_by_key[data_key] = self._data_by_key[data_key].append(data)
       data_limit = 0
       if data_key == TermToSensors.Myo_EMG.value:
           data_limit = self._storage_limit_by_node[TermToSensors.Myo_EMG.value]
       elif data_key == TermToSensors.Myo_Motion.value:
           data_limit = self._storage_limit_by_node[TermToSensors.Myo_Motion.value]
       elif data_key == TermToSensors.Teensy.value:
           data_limit = self._storage_limit_by_node[TermToSensors.Teensy.value]

       if data_limit > 0 and len(self._data_by_key[data_key]) > data_limit:
           csv_current_path = './' + str(data_key) + '.csv'
           try:
               pd.read_csv(csv_current_path, nrows=1)
               self._data_by_key[data_key].to_csv(csv_current_path, mode='a',
index=True, header=False)
           except:
               self._data_by_key[data_key].to_csv(csv_current_path, mode='w',
index=True, header=True)
           self._data_by_key[data_key] = None
```

*Figure 20. Code of RecorderManager.py.*

**Myo Controller**

A class to work with Myo via bluetooth. Once the microcontroller is connected with Myo's dongle, two data frames are created. Each one of them is a real-time frame that is populated through the `*run*` method. Both of the frames have a limit that has to be reached in order for it to be read by the *Context Handler*.
See GitHub project code for more information.

**Teensy Microcontroller**

This controller is responsible for finding a teensy microcontroller to connect with. If the controller is not available, then the software is not responsive. Once interconnected, it starts to stream real-time data into a local variable, which is the table to be query, $Q$. It needs to be in a size of window which is considered possible to recognize. That meant to be 1k Hertz. When finished to populate the table with data, and data is requested from the shared manager, then it empties and repeats the cycle. This manager also knows how to unfold the packet received via the serial port. It converts bytes into integers that are translated into a data frame.

```
class Microcontroller(ManagerProtocol):
   # Life-cycle
   def __init__(self):
       self._columns_data = ['piezo', 'fsr-1', 'fsr-2', 'fsr-3', 'fsr-4', 'fsr-5',
'fsr-6', 'fsr-7', 'fsr-8']
       self._Cache_Limit = 2000
       self._cache = pd.DataFrame([], columns=self._columns_FSR)
       self._frame = self._cache.copy()
       self._cache_list = []
   def _start(self):
       self._targetController = self.find_teensy(baud_rate=2000000, timeout = 3)
       print('Target controller is now set:\n')
       print(self._targetController)
     def _run(self):
       result = self._unfold_packet()
       result = self._convert_packet_to_data(result)
       self._save_data_to_frame(result)
   def request_frame(self):
       result = self.__flush_data()
       return result
   # Features
   def _save_data_to_frame(self, data):
       data_len = len(data)
       if data_len == 0:
           return
       self._cache_list.extend(data)        # Append new data
```

*Figure 21. Code of Microcontroller.py*

**Core Manager**

This class is responsible for live monitoring and classification of the *SystemState.* It does that by saving the data into ring-buffers, which are later passed for processing via ML_Manager.

```
class CoreManager(ManagerProtocol):
   # Life-Cycle
   def __init__(self):
       self._mlManager = None
       self._data_by_key = None
```

```
        self._storage_limit_by_node = dict.fromkeys([TermToSensors.Myo_EMG.value,
TermToSensors.Myo_Motion.value, TermToSensors.Teensy.value], 0)
        self._storage_limit_by_node[TermToSensors.Myo_EMG.value] = 640
        self._storage_limit_by_node[TermToSensors.Myo_Motion.value] = 640
        self._storage_limit_by_node[TermToSensors.Teensy.value] = 2000
        self._detection_state = SystemState.Unrecognized
    def _start(self):
        self._mlManager = MLManager()
        self._mlManager._start()
    def _run(self, nodes_data_dict):
        # Initialize frame
        if self._data_by_key == None:
            self._data_by_key = nodes_data_dict
            return
        if len(nodes_data_dict) < 1:
            # Might compute in background
            return
        # Save to frames
        for node_key in TermToSensors:
            if nodes_data_dict[node_key] is not None:
                self._save_to_frames(node_key, nodes_data_dict[node_key])
              self._mlManager._run()
    # Features
    def current_system_state(self):  # is detected as anomalous.
        latest_classification = self._mlManager.most_relevent_classification()
        if latest_classification is not None:
            self._detection_state = latest_classification
        return self._detection_state
    def _save_to_frames(self, data_key, data):
         # First initialization for specific table
        if self._data_by_key[data_key] is None:
            self._data_by_key[data_key] =
RingBuffer(self._storage_limit_by_node[data_key])
        map(lambda row: self._data_by_key[data_key].enqueue(row), data.values.tolist())
```

*Figure 22. Code of CoreManager.py.*

**ML Model Protocol**

In order to use different model classes for classification, this protocol describes the model's operability in accordance with a typical thread state machine, via 'ModelState'. In addition with three general main functionalities to try and suit different terminologies of algorithmica, pattern matching via MASS and memory neural networks via LSTM-AE. It achieves that by providing three methods, '*_feed_with_data(data)*', *'_assemble()', '_predict()',* which are responsible for updating with new data, preprocessing it, and output scoring for different activities or anomalies.

**ML Manager**

This class organises all models, and decides which of them to use in accordance with the developer. In addition, it includes the method *'most_relevent_classification()'*, to state the system has *'unrecognised'* in default, and yields the concrete system state according to the current model's predictions.

```
class MLManager(ManagerProtocol):
    # Life-Cycle
    def __init__(self):
        self._models = dict.fromkeys(ModelType.values())
        self._current_key_model = None
        self._current_classification = None
```

```
    def _start(self, key):
        self._current_key_model = key
        if key == ModelType.algorithmic:
            self._models[key] = ML_SelfMind()
        elif key == ModelType.pattern_matching:
            self._models[key] = ML_PatternMatch()
        elif key == ModelType.memory_neural_network:
            self._models[key] = ML_MemoryNNetwork()
        self._runnable_model = self._modes[key]
    def _run(self, dataDict=None):
        if self._current_key_model is not None:
            if dataDict is None:
                self._runnable_model._feed_with_data(dataDict)
            self._runnable_model._assemble()
            self._current_classification = self._runnable_model._predict()
    def _restart(self):
        self._current_key_model = None
        self._current_classification = None
    # Features
    def most_relevent_classification(self):
        if self._current_classification is None:
            return None
        state_score = [SystemState.Unrecognized, 0]
        state_keys = SystemState.values()
        for state_key in state_keys:
            probability_result = self._current_classification[state_key]
            if probability_result is not None and probability_result > state_score[1]:
                state_score[0] = SystemState.key_by(state_key)
                state_score[1] = probability_result
        return state_score[0]
```

*Figure 23. Code of MLManager.py*

**ML SelfMind**

A concrete model with the aspects of algorithmica in regard to expert knowledge of signal's representation and state scoring accordingly.

ML_SelfMind is a machine learning model that focuses on self-learning and understanding. It incorporates techniques from unsupervised learning and reinforcement learning to adapt to new information and situations. The model uses a combination of algorithms, probability, and optimization techniques to learn representations and make predictions. The runtime of this model depends on the complexity of the input data, the learning algorithms used, and the desired level of accuracy.

```
def scan_stream_activity_walk(self, data_frame):
        number_of_rows = data_frame.shape[0]
        fifo_length = 2000
        score_df = None
        count = 0
        all_keys = list(data_frame.columns)
        fsr_keys = list(filter(self.is_fsr_key, all_keys))
        for fifo_index in range(0, number_of_rows-fifo_length, 1):
            start = time.time()
            print('index: ' + str(fifo_index))
            fifo_frame = data_frame.iloc[fifo_index:fifo_index+fifo_length]
            filtered_fsr_keys =
self.__get_sanity_keys(fifo_frame.iloc[int(fifo_frame.shape[0]/2) +
int(fifo_frame.shape[0]/4):], fsr_keys)
```

```
            fsr_keys = filtered_fsr_keys
            seas_decom_prob_walk_by_fsr = dict(zip(filtered_fsr_keys,
zip(len(filtered_fsr_keys) * [0])))
            frame_agreement_score = 0
            frame_decompose_score = 0
            for fsr_key in filtered_fsr_keys:
                nonoutlier_frame = fifo_frame[[fsr_key]]
                nonoutlier_frame = nonoutlier_frame.rolling(window=200).mean().dropna()
                fft_frame = self.__convert_to_fft_signal(nonoutlier_frame)
                frame_agreement_score += self.__score_frame_walk(fft_frame[[fsr_key]])
                seas_decom_prob_walk_by_fsr[fsr_key] =
self.__score_seasonal_decomposition(nonoutlier_frame[[fsr_key]])
            # Final Score for all.
            frame_score = self.__probability_of_step(frame_agreement_score,
seas_decom_prob_walk_by_fsr)
            data = [[frame_score]]
            row = pd.DataFrame(data, columns=['Activity-Walk'])
            if score_df is None:
                score_df = row
            else:
                score_df = score_df.append(row)
            end = time.time()
            print("The time of execution of above program is :", end-start)
        return score_df
```

*Figure 24. Code of ML_SelfMind.py Walk activity detection.*

**ML PatternMatch**

A concrete model with the aspects of pattern recognition, specifically via MASS methodology, and scoring an activity based on signature euclidean-distance from the average.

*ML_PatternMatch* is an implementation of a pattern matching algorithm based on machine learning techniques. The model identifies patterns in the input data by processing it and comparing it to previously seen examples. By using a set of predefined rules or learned representations, the model can recognize and classify patterns. The runtime of this model varies depending on the size and complexity of the input data, the number of patterns to be matched, and the chosen pattern matching technique.

```
class ML_PatternMatch(ML_ModelProtocol):
    def __init__(self):
        self._state = ModelState.Start
        self._input = None
        self._processed_data = None
        self._predictions = dict()
        self.epsilon = 1.5
    def _feed_with_data(self, data) -> None:
        self._input = data
        self.pushups_df = pd.read_csv(data['walk_file'])
        self.pushups_df =
self.pushups_df.iloc[2000:36000].rolling(window=1000).mean().dropna()
        self.Q_df = self.pushups_df['fsr-2'].iloc[data['Q_start']:data['Q_end']]
        self.T_df = self.pushups_df['fsr-5'].iloc[data['T_start']:data['T_end']]
        self.distance_profile = stumpy.mass(self.Q_df, self.T_df)
    def _predict(self) -> dict:
        idx = np.argmin(self.distance_profile)
        Q_z_norm = stumpy.core.z_norm(self.Q_df.values)
        nn_z_norm = stumpy.core.z_norm(self.T_df.values[idx:idx + len(self.Q_df)])
```

```
        max_change = max(Q_z_norm - nn_z_norm)
        min_change = min(Q_z_norm - nn_z_norm)
        # Check if the pattern matches within the given threshold
        if max_change < self.epsilon or abs(min_change) < self.epsilon:
            self._predictions['pattern_match'] = True
        else:
            self._predictions['pattern_match'] = False
        return self._predictions
```

*Figure 25. Code of ML_PatternMatch.py.*

**ML MemoryNNetwork**

A concrete model with the aspects of Memory Neural Network. Specifically it can measure the euclidean-distance between the prediction result and the input.

This model is an implementation of a memory-augmented neural network designed to recognize patterns in sequential data. It incorporates elements from recurrent neural networks (RNNs) and memory networks to create an efficient learning mechanism. The model processes the input data and stores relevant information in its memory, which can be later used for making predictions. The runtime of this model depends on the complexity of the data, the size of the memory, and the training configurations.

```
class ML_MemoryNNetwork(ML_ModelProtocol):
    def _feed_with_data(self, data) -> None:
        self._input = data
    def _assemble(self) -> None:
        train_df, val_df, seq_len, n_features = self._prepare_dataset(self._input)
        self.model = RecurrentAutoencoder(seq_len, n_features, 256)
        self.model = self.model.to(self.device)
        self.model, _ = self._train_model(self.model, train_df, val_df, n_epochs=5)
    def _predict(self, model, dataset):
        predictions, losses = [], []
        criterion = nn.L1Loss(reduction='mean').to(self.device)
        with torch.no_grad():
            model = model.eval()
            for seq_true in dataset:
                seq_true = seq_true.to(self.device)
                seq_pred = model(seq_true)
                loss = criterion(seq_pred, seq_true)
                predictions.append(seq_pred.cpu().numpy().flatten())
                losses.append(loss.item())
        return predictions, losses
  class Encoder(nn.Module):
 def __init__(self, seq_len, n_features, embedding_dim=64):
   super(Encoder, self).__init__()
   self.seq_len, self.n_features = seq_len, n_features
   self.embedding_dim, self.hidden_dim = embedding_dim, 2 * embedding_dim
   self.rnn1 = nn.LSTM(
     input_size=n_features,
     hidden_size=self.hidden_dim,
     num_layers=1,
     batch_first=True)
   self.rnn2 = nn.LSTM(
     input_size=self.hidden_dim,
     hidden_size=embedding_dim,
     num_layers=1,
     batch_first=True)
```

*Figure 26. Code of ML_MemoryNNetwork.py*

## Activity Recognition - SelfMind

A broad offline analysis has been performed for different types of activities, but because of research scope limitations we will be going over a portion of it.

**Walk Activity Simulation**

'*def scan_stream_activity_walk(data_frame)*' - A separate simulation for walking activity, with the analysis of one second - 2k samples per window. This time we don't use the voltage convertage as it has no use for the FFT and even disqualifies the outputs such that they are of no use any longer.

A moving-average of 200 is smoothing the data, though this time it is not much effective. It is then converted into a cleaner signal via fft and an inverse of fft. Over that signal, a scoring mechanism by '*score_frame_walk(fft_signal)*', returns one or zero respectively with respect to '*validate_complete_step(df)*', which goes over the signal and tries to find the occurrence of the following arbitrary step wave.

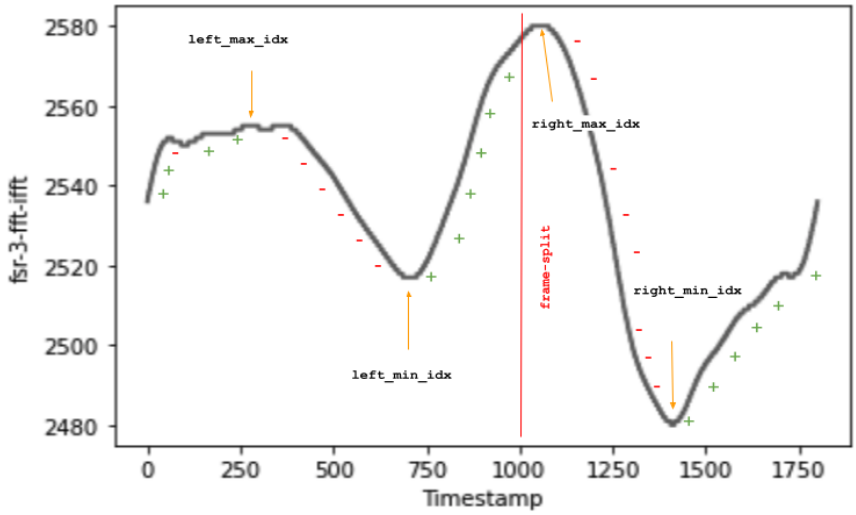


*Figure 27. Analysis of steps inside walk activity, file '04_Office_Walk.csv'.*

Eventually, for each fsr that is eligible, we count if the current frame has detected a step of walk, by detecting a frame as the above. For instance, have total of eights FSRs, where six of them ineligible, and four of them detect a step, then the score for step recognition is $\frac{4}{6} = 0.67\%$.

```
def __probability_of_step(self, count_agreement, decompose_score_by_fsr):
    count_total = len(decompose_score_by_fsr)
    probability_fsr_agreement = 0.5
    probability_ARMA = 1 - probability_fsr_agreement

    # Compute Total Decompose Score
    decompose_total_percent = probability_ARMA * 100 # 0.5 * 100 = 50%
    decompose_percent_per_fsr = decompose_total_percent / count_total
    decompose_percent_sum = 0
    for value in decompose_score_by_fsr.values():
        decompose_percent_sum += decompose_percent_per_fsr * value
```

```
        return probability_fsr_agreement * (count_agreement / count_total) +
decompose_percent_sum/100

    def __score_frame_walk(self, data_frame):
        is_step_validated = self.validate_complete_step(data_frame)
        if is_step_validated:
            return 1
        return 0
```

*Figure 28. ML_SelfMind.py - probability of step & score frame walk.*

That mentioned, we continue with another process of identifying walk via seasonal decomposition outputs, '*score_seasonal_decomposition(data)*', which takes into account the trend and residual appearance. Both of them are checked to satisfy the two criterias, stable residual (min, max, mean) are in range of $1 \pm \varepsilon$, '*is_walk_residual_multiplicative(data_resid)*'. Additionally, a detection of trend - where a walk wave should also be recognized, via one minimum(left side) and one maximum(right side), '*is_walk_trend_multiplicative(data_trend)*'. Each of the parameters has a weight of 50% in seasonal decomposition scoring, such that if only trend is available, then the score is 50%, for the particular fsr.

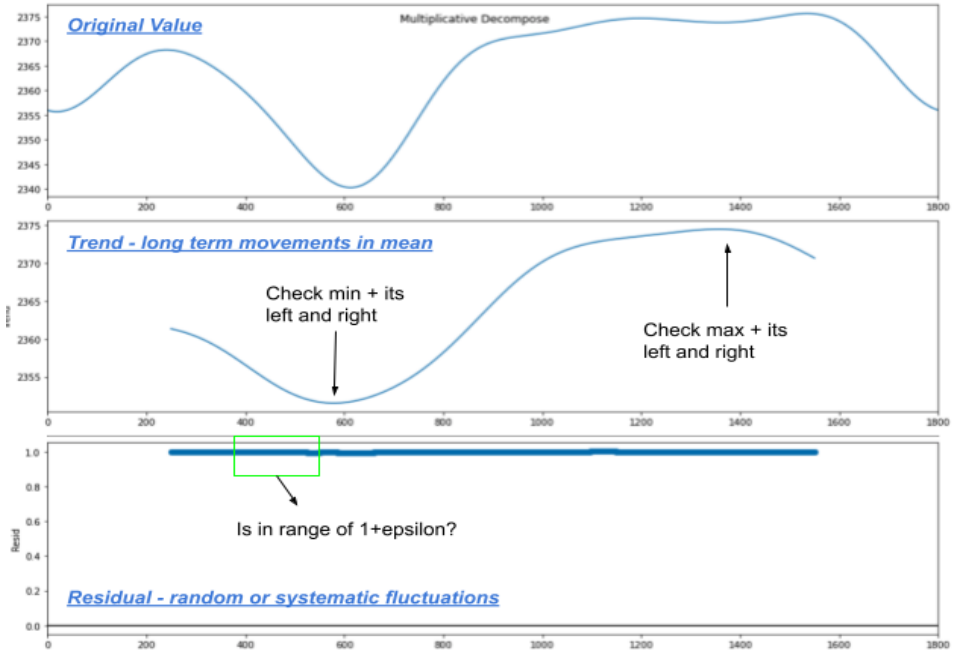


*Figure 29. A seasonal decomposition of steps in walk, 'iFFT(FFT(MA-200(input)))'.*

A summary of the above is explicit via, '*probability_of_step(frame_agreement_score, seas_decomp_prob_walk_by_fsr)*', which takes into account the recognition of a step through a preprocessed signal value, and its seasonal decomposition scores. Arbitrarily, we have chosen to score 50% weight for all FSRs finding the step pattern, and the other 50% is built from every eligible fsr agreeing on residual and trend.

$$0\% \leq \frac{1}{2} \cdot \frac{count_{agreement}}{count_{total}} + \frac{1}{2} \cdot \sum_{i=0}^{count_{total}} \widehat{x_i} \cdot \widehat{c} \leq 100\%$$

$$\widehat{x_i} = seasonal\ decomposition\ score,$$

$$\widehat{c} = partial\ percentage\ score,\ depending\ on\ eligible\ FSRs\ in\ cycle.$$

All scores are summarised into a data-frame for later analysis that has been conducted in the research paper.

**Idle Activity Simulation**

'*def scan_stream_idle(data_frame)*' - A very similar approach for idle activity detection was done, in terms of analysing a 2k sized window, but this time with two different approaches, and with a window-stride of size 10, rather than one. Later, a comparison of the two idle detection is overviewed.

**Scan Idle via PCT**

'*def scan_stream_idle(data_frame)*' - Observe the input frame as pct changes via smoothened frame, and picks the largest distance from zero, whether it is a spike up, (resistance recovery) or a spike down (stretch). Once the max change distance is determined, it is scored via exponential distribution, CDF. After cycling over all eligible FSRs, the most minimum score that was present, meaning the FSR with greatest change will affect the analysis of the current frame. It is seen right away that the areas of change are depicted with a strong fall to zero.
Figure 30 is an example of a book being dropped from different heights, the higher the distance travelled till impact, the stronger is the resistance change of the sensor - that is reflected with lower values, but this is beyond the scope of this research.

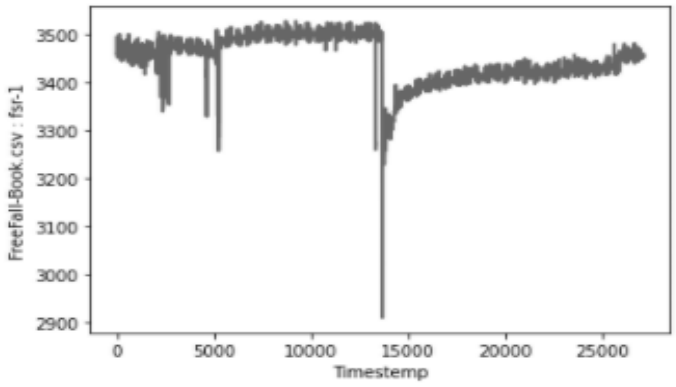


*Figure 30. A depiction of FSR reacting to book fall over pants, file 'FreeFall-Book.csv'.*

Figure 31 indicates a notable perfect binary response. It is important to take this into account because the simulation was across all eligible FSRs of the experiment and the result is astonishing in the sense of detecting occurrence and its period.

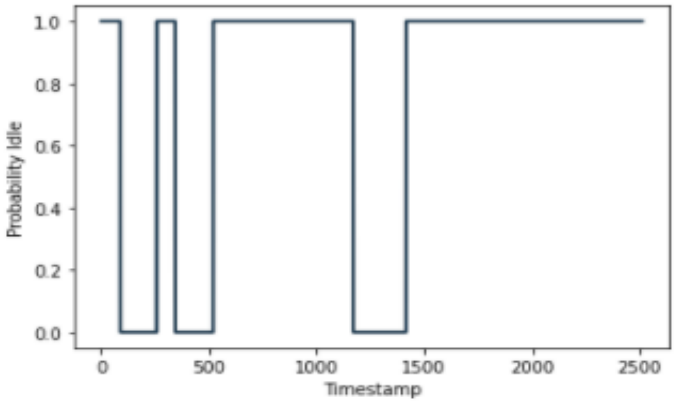


*Figure 31. Analysis of all FSRs for idle recognition via PCT, file 'FreeFall-Book.csv'.*

```
def scan_stream_idle(self, data_frame):
        number_of_rows = data_frame.shape[0]
        fifo_length = 2000
        score_df = None
        count = 0
        all_keys = list(data_frame.columns)
        fsr_keys = list(filter(self.is_fsr_key, all_keys))
        for fifo_index in range(0, number_of_rows-fifo_length, 10):
            start = time.time()
            print('index: ' + str(fifo_index))
            fifo_frame = data_frame.iloc[fifo_index:fifo_index+fifo_length]
            filtered_fsr_keys =
self.__get_sanity_keys(fifo_frame.iloc[int(fifo_frame.shape[0]/2) +
int(fifo_frame.shape[0]/4):], fsr_keys)
            fsr_keys = filtered_fsr_keys
            min_score = 1
            for fsr_key in filtered_fsr_keys:
                outlierFrame = fifo_frame[fsr_key].rolling(window=200).mean().dropna()
                pct_frame = outlierFrame.pct_change(periods=200)
                # current_score = score_activity_idle_via_pct(pct_frame)
                current_score = self.__score_activity_idle_via_MW(outlierFrame)
                min_score = min(current_score, min_score)
            data = [[min_score]]
            row = pd.DataFrame(data, columns=['Activity-Idle'])
            if score_df is None:
                score_df = row
            else:
                score_df = score_df.append(row)
            end = time.time()
            print("The time of execution of above program is :", end-start)
        return score_df

def __score_activity_idle_via_pct(self, pct_frame):
        x = np.arange(0, 1, 0.001)
        y = expon.cdf(x, 0, 0.00001)
        y = 1-y
        epsilon = 0.01
        subframe_min = abs(pct_frame.min())
        subframe_max = pct_frame.max()
        subframe_max_change = subframe_max if subframe_max > subframe_min else
subframe_min
        result = np.where(np.logical_and(x >= subframe_max_change-epsilon, x <=
subframe_max_change+epsilon))
        if len(result[0]) == 0:
```

```
            return 0
        distribution_index = result[0][0]
        return y[distribution_index]
```

*Figure 32. ML_SelfMind.py - scan and score idle stream.*

**Scan Idle via Moving-Average**

'*def score_activity_idle_via_MW(fifo_frame)*' - Simply a *Moving-Window* with applied constraints. It is an additional way of viewing the changes being made in the observed frame. By taking the *mean*, *max* and *min* of the frame, the greatest distance can be determined, plainly, thus scoring changes upon absolute value changes, in accordance to a different exponential CDF. For example, the minimal value of score is calculated along the frame, thus the minimal score among all FSRs is picked. As can be seen in Figure 18, below, the prediction probability of idle is not in the correct scale we would expect, though the sense of decreasing values close to zero when the book fell, does indicate the inverse of idle - not idle for sure.

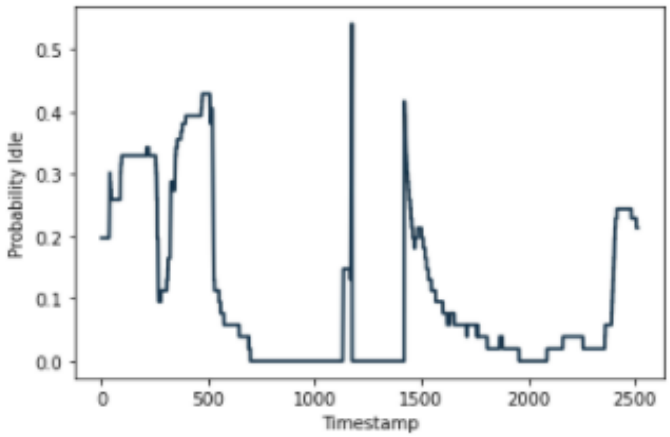


*Figure 33. Analysis of all FSRs for idle recognition via MovingWindow, file 'FreeFall-Book.csv'.*

```
def __score_activity_idle_via_MW(self, fifo_frame):
        number_of_rows = fifo_frame.shape[0]
        inner_window_length = 10
        x = np.arange(0, 100, 0.01)
        y = expon.cdf(x, 0, 50)
        epsilon = 1
        min_frame_score = 1
        for i in range(0, number_of_rows-inner_window_length, inner_window_length):
            window_frame = fifo_frame.iloc[i:i+inner_window_length]
            window_mean = window_frame.mean()
            dist_max = int(fifo_frame.max() - window_mean)
            dist_min = int(window_mean - fifo_frame.min())
            dist_greatest = dist_max if dist_max > dist_min else dist_min
            if dist_greatest >= 100:
                min_frame_score = 0
            result = np.where(np.logical_and(x >= dist_greatest-epsilon, x <=
dist_greatest+epsilon))
            if len(result[0])==0:
                continue
            distribution_index = result[0][0]
            min_frame_score = min(min_frame_score, y[distribution_index])
        return min_frame_score
```

*Figure 34. ML_SelfMind.py - score activity via MW.*

**Anomalous Detection - SelfMind**

A broad offline analysis has been performed for different types of objects that had impact with the smart outfit - while it was in a predefined state. Moreover, the aforementioned can be viewed in Appendix B.

**Real-Time Simulation**

In order for us to have legit anomaly detection in the real-time framework, we created an offline simulation that runs over offline recordings. In file `*AnomalyDetector.py*`, we first load up a data-file we're interested in, then execute `*scan_stream(data_frame)*` to observe the prediction probability of the frame. For instance, take a leg kick recording and transform it to probability of a hit. Figure 36 shows an excellent result in the recognition of anomalies. That is, if we decide a threshold with the value of 0.6, then the recognition is precise.

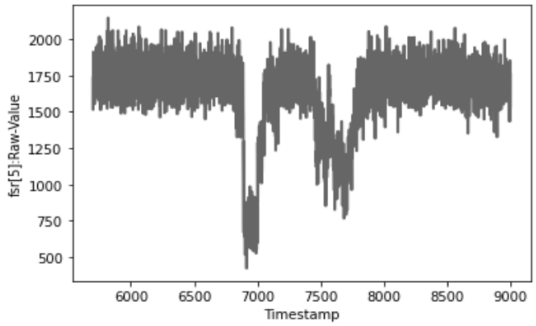


*Figure 35. A cherry-picked FSR raw data with two kicks, file 'LegKick.csv'.*

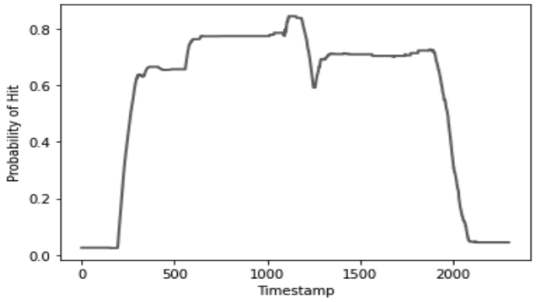


*Figure 36. A prediction analysis of two anomalies in a short time for one fsr, file 'LegKick.csv'.*

It is easily seen that a simple threshold can now decide there's a hit, though note that the other FSR sensors will not necessarily react as well as the cherry-picked one, as can be seen in Figure 18. This means the percent change of 'not well reacting' sensors, is gonna be bounded by a maximum of 10%. That can be seen due to overall values research upon hit experiments. The beginning of '*def scan_stream(data_frame)*' is catching up a frame of size 1000 data points, which is roughly half a second, and analysing all incoming FSRs sequentially. Before the analysis begins, we check if the frame is a complete noise or has saturation values, if so, the frame is dismissed from the current analysis. The method, *'def get_sanity_keys(data, keys)*' is receiving the whole data file and an array of FSR keys, and returns all keys that were qualified as not noise nor saturation signals. It is done via `*def is_signal_noise(data[key])*`,

and '*def is_signal_saturated(data[key])*'. To verify the signal is not noise, a window of size 100 is sliding and checking if the minimum and maximum values are in bound of [0, $2^{12}$-1] and the mean is not under the value of 10, though this boundary can be scaled to much higher values. In addition, it checks whether the signal has a repeating mean value, which may suggest a noise repeating itself. To verify a signal has no saturation, a sliding window of 100 is calculated of its mean value and checked if it is under 10. The disconnected signal is mainly zero with epsilon - negligible noise of the microcontroller. Figure 20 points out that even for disconnected sensors, the noise that comes from the board and transmitted to the non-connected socket, is a reaction to the outfit's resistance change - spikes from zero.

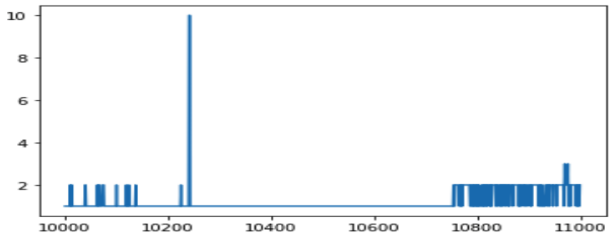


*Figure 37. An arbitrary FSR value over time, depicting a disconnected sensor, file 'LegKick.csv'.*

Once we're prepared with FSR frames that are in high chance to be correct, we convert the data into voltage for the sake of correct electrical engineering reference, voltage. That is done via '*def convert_to_voltage(data_frame, resistance)*', where *(input_voltage / teensy_res_bit) / resistance*. The resistance was checked for each FSR, and was 800kOhm. We divide the input voltage, 3.3, by the number of resolution bits set up on the teensy to sample the data, $12\ resolution\ bits$. In continuation, we use a simple heuristic for getting rid of outliers in the input signal. Remember, we are not supposed to lose the anomaly event since it is a process that affects many consequential data points in time. The method, '*def scan_stream_outliers(nonoutlier_frame)*', uses a sliding window to check if the last value of it is outside of the mean + standard deviation range. This method was found to be useful on great windows of time, as it encoded the signal into a shorter range. Right after, we continue the clean-up with *Grubb's outlier detection*. A quick recall, the outliers are the extremities of the gaussian bell. The grubb's scanning method is '*def scan_grubb_outliers(data_frame)*' which activates the function, '*def ESD_Test(input_series, alpha, max_outliers)*' over the whole window. All the indexes that are considered outliers are later excluded from a copy of the input data. A relatively small window, size of 15, is used since more outliers can be caught for small windows and less outliers for bigger windows. Considering a bigger window with the detection of less outliers and more time computation, is a bad practice for this manner. The window slides in shifts of the whole window while the previous outlier detection was using steps of size one. The final preprocessing is the execution of `*Moving-Average(200)*`. It helps to reduce the noise in a signal. A window of 200 is chosen, since it is $\frac{1}{10}$ of a second, thus 100 milliseconds is sufficient for the event's time duration for this scope of research. A data-frame built-in function, '*rolling(window=200).mean()*', is used with '*dropna()*', dropping all not available values that are

created from the process of moving average. After all this preprocessing is complete, the original window which started with the size of 1000, has decreased by roughly 300 samples.

**Table 3. Filtered values count**
*Count of items dropped per arbitrary fsr window of size 1000.*

| Methodology | Count Dropped Values |
|---|---|
| Moving-Average(200) | 200 |
| Grubb's Outlier Detection | 10 |
| Self-Clean-up | 70 |
| **Total** | **280** |

Later the shortened window is applied with '*def score_frame(data_frame)*'. This function is responsible for several important processes. First, we declare a cumulative distribution function, via exponential distribution - '*expon.cdf(x, 0, 0.3)*', where the scale parameter $\beta = 0.3, \lambda = 3.33$, were chosen after repeated experiments with different parameters. understanding the difference of activity versus anomaly in terms of percent change. The uniqueness of this distribution is it produces less than 50% for 20% stretch change. Because activities like walking are usually bounded by 10% stretch change. Peaks of anomaly were observed with more than 10% change, and even varied across different experiments - {20%, 30%, …, 70%}. Our preprocessed input is now applied with '*pct_change(periods=200)*', which produces a stream of percent change over 100ms. The change in time-series is calculated in the background via derivatives that are converted into percentages. The new output is now shortened again by 200 samples and is rated by the minimum pct change across the stream. For instance, the minimum across a half second frame was found to be $min = -\frac{1}{2} \Rightarrow result = x[|min|]$, that will be the index of exponential distribution CDF.

**Picking the Threshold**

Observing the different FSRs that are considered valid for processing, usually a subgroup of one or two FSRs will be mostly affected by the hit in the experiment's data. The great collective of fsr sensors will produce less than 10% change, and so the probability of that is less than 0.1, therefore there needs to be a factor to the FSRs that reacted drastically. A method named, '*def score_fsrs(fsrs_dict)*' handles the explained above use-case. It receives as an input an ordered by value dictionary of key-value pairs, which is the '*score_frame(data_frame)*' as value, and key of the particular analysed FSR. Arbitrary, we have conducted a score mechanism that gets as input the sorted dictionary, verifies it's not of size zero (predicts 0 probability for FSRs). If only one FSR key is available, then the value of that key in a dictionary is the result, as it is the only sensor that receives valid feedback. In case of two sensors, we sum both sensor's

scores multiplied by 50%,

$score_{frame}(fsr[i]) = x_1\ ;\ score_{frame}(fsr[j]) = x_2\ \Rightarrow\ \frac{1}{2}\cdot x_1 + \frac{1}{2}\cdot x_2)$.

Once more than two sensors are apparent, we generate the following array,

$[50\%,\ 25\%,\ 12.5\%,\ 6.25\%,\ 3.125\%,\ \ldots]$.

Since the dictionary of probabilities is sorted in reverse, the top has the largest PCT and the last value has the least change.

$$x_1 \cdot \frac{1}{2^1} + x_2 \cdot \frac{1}{2^2} + x_3 \cdot \frac{1}{2^3} + \ldots + x_k \cdot \frac{1}{2^k} = prob_{factor};\ k \geq 3\ and\ k \leq 52$$

When '*k*' reaches the end, the sum is 100, in our case the amount of sensors is less than that. Though, the sum can be less than 100%, and so we calculate the carry by subtracting the sum from 100, and splitting the carry in between the probabilities.

Finally, $prob_{carry} = (100 - prob_{factor})/keys_{length}$.

$$(prob_{carry} + x_1 \cdot \frac{1}{2^1}) + (prob_{carry} + x_2 \cdot \frac{1}{2^2}) + \ldots + (prob_{carry} + x_k \cdot \frac{1}{2^k}) \leq 100\%.$$

The result is bounded by 100%, and is returned to '*scan_stream(data_frame)*', which saves it to a dataframe output for later offline analysis. As can be seen in Figure 20, a simulation of a leg kicking the outfit depicted below 10% - idle state, whereas the hit was above 50%.

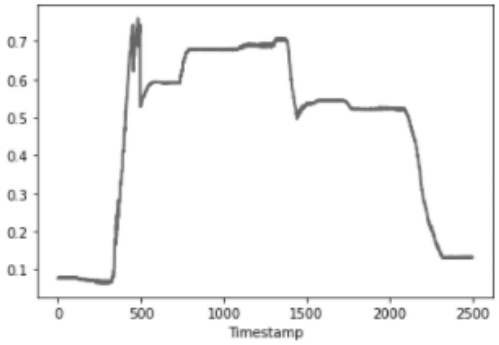


*Figure 38. A simulation result of a leg kick prediction, file 'LegKick.csv'.*

Figure 39 is an example of the hit detector not classifying the walk activity as hit activity, which is a positive signal for no false-alarm.

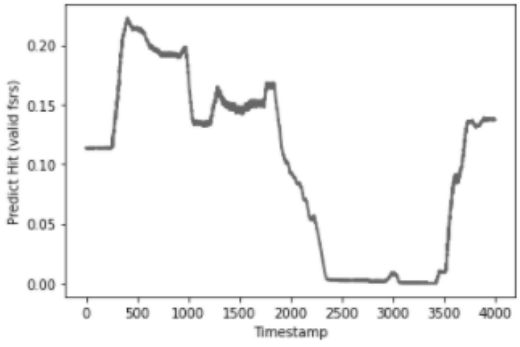


*Figure 39. A simulation of walk prediction, file '01_Walk_Home.csv'.*

## Conclusions

The primary objectives of the wearable outfit were successfully designed and achieved, as sensors were strategically integrated into the outfit, either through printing or glueing, and effectively generated valuable data. The collected signals from this outfit closely resemble those obtained in other research studies focusing on fall detection using accelerometers. A noteworthy observation when comparing the use of FSR and accelerometers is that fall detection with accelerometers is indicated across all three axes, while FSR sensors detect changes in resistance - force or stretch application - which is sufficient for discerning different user activity states, including anomalous activity.

One challenge encountered in this project was the difficulty in accurately verifying fall detection cases. To address this issue, we knew by preparing the anomalous before the start of an experiment. Additionally, we implemented methodologies to filter out inconsistent or noisy signals before performing any analysis. This approach significantly streamlined the data processing workflow.

In contrast to the conventional method of attaching gyroscopes and accelerometers to every human body joint for user monitoring, a single FSR line on a limb can effectively classify the user's motion as normal or abnormal. This innovative approach simplifies the overall system while still achieving reliable and accurate results, demonstrating the potential of our wearable outfit design for real-world applications.

## Further Work

In this research project, we managed to build different sorts of software for a variety of necessities. Software for recording offline mechanisms by microcontrollers like Arduino, Teensy and Myo armband. More research can be done in exploring the Myo different sensors, on the same activities that were presented here. In addition we've added the ability for those microcontrollers to communicate via serial, and present real-time monitoring. A separate, high-level software for the Jetson B01 is implementing real-time read on the serial, and classification of activities and anomaly detection. Further improvements can be done to optimise the computability of functions - changing the parameters of the stream's scan. An alert mechanism for intrusion and interaction with the environment, for example our suit can have an additional led panel to depict the user activity recognition while using the suit. More about that in Appendix C.

### Futuristic Vision

In this period of fast growing technology, our vision is that smart suits will become part of our daily-use products, that can communicate with the user and assist in various scopes, just like Smart Watches. Additionally, an important aspect of putting sensors on the user's cloth, will need to be printed on it to permit more breathability of the fabric while the user is wearing it. Lastly, adaptability to an application that a user can use on his smartphone, will present the user with more accessibility to the device itself.

## References

Arduino Uno R3 Datasheet.(2022). Arduino.cc. https://docs.arduino.cc/hardware/uno-rev3

Jetson Nano B01 Datasheet.(2022). Nvidia. https://developer.download.nvidia.com/assets/embedded/secure/jetson/Nano/docs/JetsonNano_DataSheet_DS09366001v1.1.pdf?iCsjR0Mp3QI1bG2SlbaKUo_BJPhTpM_KttZIPezSnYSyiJL3UXMcNr1AJzIUYMU-tfPRZZx6x5unreHQZqZwq9MJtFE0fE8lVHnuox1q3Kj_rako6nam6smE7zBSAErgxaR7TituloiNACClxJ_5NhVPbMsk8K50En9TrkcdMgFmZ7oY5EY9N6sC-liHXQ==

Teensy 3.6 Datasheet.(2016). NXP. https://www.pjrc.com/teensy/K66P144M180SF5V2.pdf

Visconti,P.,Gaetani,F.,Zappatore,G. & Primiceri,P.(2018).Technical Features and Functionalities of Myo Armband: An Overview on Related Literature and Advanced Applications of Myoelectric Armbands Mainly Focused on Arm Prostheses. International Journal on Smart Sensing and Intelligent Systems,11(1) 1-25. https://doi.org/10.21307/ijssis-2018-005

Mukhopadhyay, S. C.(2015). Wearable sensors for human activity monitoring: A review. *IEEE Sensors Journal*, *15*(3), 1321-1330. [6974987]. https://doi.org/10.1109/JSEN.2014.2370945

Kammerer K, Hoppenstedt B, Pryss R, Stökler S, Allgaier J, Reichert M.(2019). Anomaly Detections for Manufacturing Systems Based on Sensor Data—Insights into Two Challenging Real-World Production Settings. *Sensors*. 19(24):5370. https://doi.org/10.3390/s19245370

Khan, S., Qamar, R., Zaheen, R., Al-Ali, A. R., Al Nabulsi, A., & Al-Nashash, H. (2019). Internet of things based multi-sensor patient fall detection system. *Healthcare technology letters*, *6*(5), 132–137. https://doi.org/10.1049/htl.2018.5121

Kamble, S. M., Kawle, P. R., Mohile, S. S., Meshram, M. R., Mam, Prof. G. P., & Mam, N. G. (2022). IOT Based Person/Wheelchair Fall Detection System. International Journal for Research in Applied Science and Engineering Technology, 10(2), 258–263. https://doi.org/10.22214/ijraset.2022.40243

Raja, J. M., Elsakr, C., Roman, S., Cave, B., Pour-Ghaz, I., Nanda, A., Maturana, M., & Khouzam, R. N. (2019). Apple Watch, Wearables, and Heart Rhythm: where do we stand? Annals of Translational Medicine, 7(17), 417–417. https://doi.org/10.21037/atm.2019.06.79

Cho, G. (Ed.). (2009). Smart Clothing: Technology and Applications (1st ed.). CRC Press. https://doi.org/10.1201/9781420088533

Abdulla, Raed & Kumar Selvaperumal, Assoc. Prof. Dr. Sathish & Nataraj, Chandrasekharan. (2020). WHEELCHAIR-PERSON FALL DETECTION WITH INTERNET OF THINGS. Solid State Technology. 63. 911-922. https://www.researchgate.net/publication/344590832_WHEELCHAIR-PERSON_FALL_DETECTION_WITH_INTERNET_OF_THINGS

Project GitHub: https://github.com/Jeremaiha/Thesis-Project-Anomaly-Detection

## Appendix A

### General Wearable Outfit

A wearable outfit consists of a node list and a server node to conclude operations in real-time. Generally meaning, there shall be $N \geq 1$ ; $N = nodes$. A server node is referred as a node, in case $N = 1$, that node shall be the server. Figure 1 is an example of an outfit we built and used for those purposes.

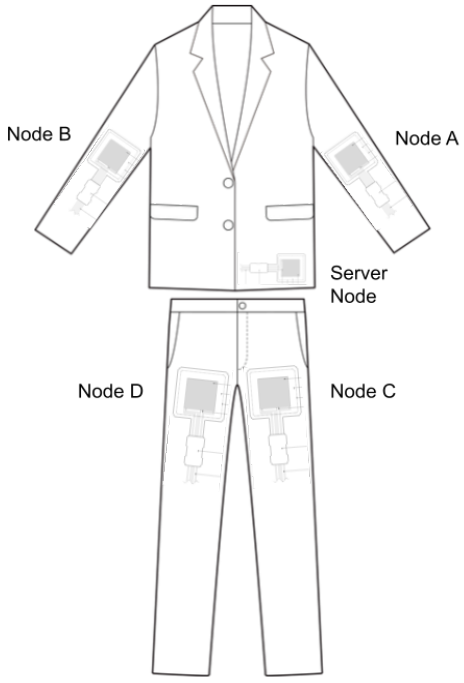


*Figure A1. A high-level general wearable outfit with $N = 5$, a depiction of our implementation.*

A server node is responsible for all incoming traffic, Thus triggers an activation in case of discord detection. Each node is a streamer of sensor data.
$Server\ Node \equiv System - on - Module$ ; $Node \equiv Microcontroller$

**Hardware in Practice**

We use a powerful System-on-Module, Jetson nano B01, since it has an extension pack for battery(long term use), and a Unix software that runs python scripts. This SoM is monitoring its serial USB ports, and activates logic accordingly. Figure 2 through 5 are examples of devices we worked with for this suit system.

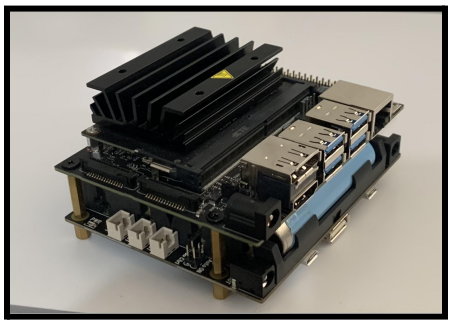

*Figure A2. A System-on-Module, Jetson Nano B01 with T200 shield(battery pack).*


*Figure A3. A Microcontroller, Myo-armband. used for emg signal collection.*

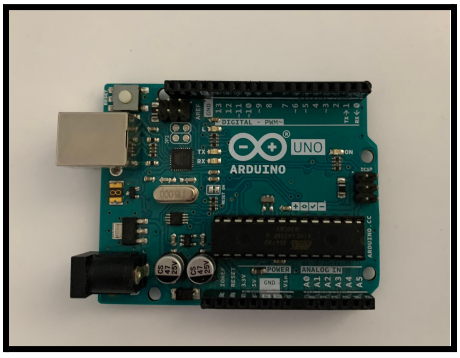


*Figure A4. A microcontroller, Arduino Uno R3, used for fsr, piezo signal collection.*

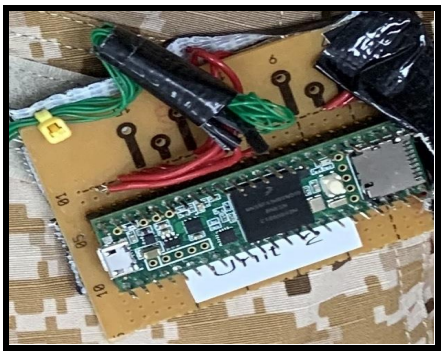

*Figure A5. A microcontroller, Teensy 3.6, used for fsr, piezo signal collection.*

*Outfit and Sensors.* The smart outfit can be any cloth part, with sensors on it. For instance, in Figure A6, A7, we used garments provided by Xmetix. A printed fsr sensor is in Figure A8.

*Figure A6, A7. A top and bottom outfit with sensors.*

*Figure A8. A printed force-sensing resistor on pants.*

## Appendix B

### Ring Buffer

This class is a common implementation of a design pattern named ‘CircularQueue’, and is an effective real-time data structure, as it keeps you with the most recent data depending on the queue’s size. It consists of *‘enqueue(new_value)’, ‘dequeue()’, ‘get_list()’, ‘get_at(index)’, ‘is_full()’.*

```
""" A class for Circular-Queue / Ring Buffer """

class RingBuffer():

   def enqueue(self, new_value):

       if self.is_full():

           self._data[self._current] = new_value

           self._current = (self._current+1) % self._capacity

       else:

           self._data.append(new_value)

           if len(self._data) == self._capacity:

               self._current = 0

   def dequeue(self):

       value = self._data[self._current]

       self._current = (self._current-1) % self._capacity

       return value

   def get_list(self):

       return self._data[self._current:] + self._data[:self._current] if
self.is_full() else self._data

   def get_at(self, index):

       if index < 0 or index >= self._capacity:

           return None

       return self._data[index]
```

*Figure B1. Code of RingBuffer.py*

## Appendix C

A futuristic implementation of the smart suit has to have an accessibility to the environment, in order to notify the user of what is going on. An idea of a buzzer activation once an abnormality is detected. And a led panel to present the user is present in some activity either with anomaly detection.

The infrastructure is prepared in *UserInterfaceHandler*, and it is up for the user to choose whether he wants to test it in a terminal environment, or display the outputs through a visual led panel.

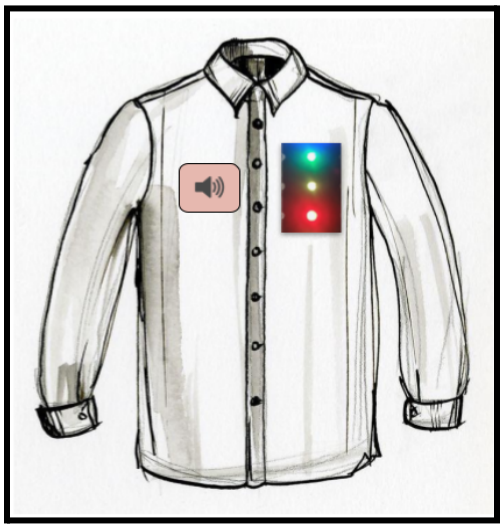

*Figure C1. A high-level User-Interface of the smart-suit. A buzzer and a led state panel.*

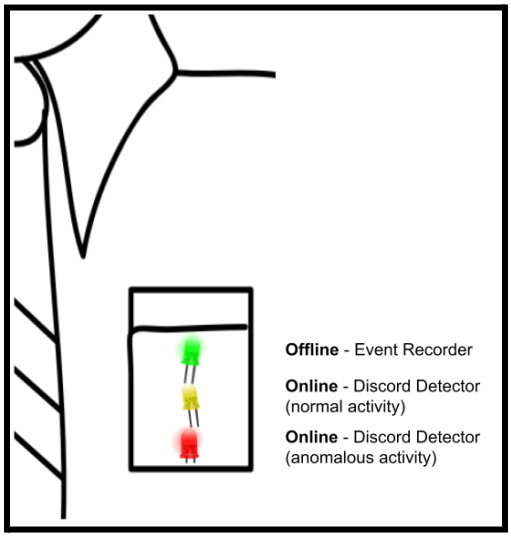


*Figure C2. An example of different states for a User-Interface led-panel.*