SonicWalk

Technical Documentation

UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

Bachelor's Degree in Computer Science for New Media Communication

Project by: Roberto Tallarini

Year 2024

Contents

1 Overview 1

1.0.1 Desired Software and Features . . . 1

1.0.2 Hardware . . . 3

2 Python Interface 4

2.1 Interface Structure . . . 4

2.2 Execution Flow . . . 7

2.2.1 Starting Recording, Multithreading vs. Multiprocessing . . . 7

2.2.2 Execution of the Analyzer Class . . . 9

2.2.3 BPM Estimation . . . 10

2.3 Shared Objects . . . 11

2.4 Porting to Windows . . . 13

2.4.1 Multiprocessing Management (Fork vs. Spawn) . . . 13

2.4.2 Bounded-buffer problem (Producer-Consumer) . . . 14

3 Real-Time Signals Processing 16

3.1 Introduction to Signals and Analysis Choices . . . 16

3.2 Utility Methods . . . 22

3.3 Walking . . . 27

3.3.1 Test Results . . . 30

3.4 Marching in Place . . . 31

3.5 Leg Detector . . . 33

3.5.1 Gaussian Filter . . . 36

3.6 Swing . . . 37

3.6.1 stepLeg() . . . 40

3.6.2 otherLeg() . . . 42

3.7 Anterior-Posterior Load Shifting in Tandem Position . . . 44

Chapter 1

Overview

SonicWalk is a project (available on GitHub: `https://github.com/Rob00t-unimi/SonicWalk`) aimed at supporting the study promoted by the Research Laboratory in Music Therapy at the Maugeri IRCCS Clinical Scientific Institutes in Pavia, *"Sonification Techniques for Gait Training (SonicWalk)"*, registered on *clinicalTrials.gov* with the number NCT04876339.
This research aims to apply and evaluate the effectiveness of sonification in gait rehabilitation for patients with stroke, Parkinson's disease, and multiple sclerosis. To this end, SonicWalk uses two inertial sensors to analyze and store patient movements and provide real-time auditory feedback, thus facilitating the rehabilitation and monitoring process.
A previous attempt to develop similar software was made using *MATLAB*[1]. However, it did not achieve the desired results due to difficulties related to the effective and reliable processing of real-time motion data. The current version of SonicWalk has overcome these limitations and is based on a *Python*[2] interface initially developed by colleague Gabriele Esposito (available on GitHub: `https://github.com/xyzxyzxyzxy/SonicWalk`), which handles gait analysis, communication, detection, and buffering of data from Xsens MTw Awinda sensors, using Xsense *APIs*[3]. The interface has been appropriately expanded to improve *multiprocessing*[4] management, and additional algorithms have been designed to support the analysis of three new rehabilitative exercises. Moreover, modifications were made to correctly integrate the Python interface with the functionalities for managing and saving exercise recordings, implemented in the *PyQt5*[5] graphical interface of the software.

The study required the development of a robust detection software, easy to manage, and structured for large-scale use by healthcare personnel.

1.0.1 Desired Software and Features

The system was designed to have a usable and intuitive graphical interface, with as few parameters as possible to manage. It was necessary to develop a system capable of storing, modifying, searching, filtering, and deleting datasets related to patients with the different conditions involved in the study, as well as managing the execution of four different exercises: Walking, March in place, Swing, and Anteroposterior load shift in tandem position.

[1] ***MATLAB***: A high-level programming language primarily used in scientific contexts for numerical computation, data analysis, statistical analysis, visualization, and algorithm development.
[2] ***Python***: A high-level, object-oriented programming language ideal for creating distributed applications, scripts, managing numerical, statistical, and machine learning calculations.
[3] ***API***: (Application Programming Interface) interfaces that facilitate access and interaction between different software applications.
[4] ***Multiprocessing***: A programming technique that allows multiple processes to run simultaneously on a machine with multiple cores.
[5] ***PyQt5***: A library (set of modules and functions) for Python that allows the creation of graphical user interfaces (GUIs) using the Qt framework (set of tools and libraries).

Each exercise was to include three possible execution modes:

1. A first phase of gait analysis lasting 25 seconds without sonification, to estimate the BPM (Beats Per Minute) of the patient's gait.

2. A second phase of 90 seconds with playback of pre-recorded music synchronized with the previously estimated BPM.

3. A third phase of 90 seconds of real-time sonification, where the patient composes the melody by performing the exercise movements.

The first phase is used to estimate the gait speed for music playback in the second phase. The second phase trains the patient to follow the musical rhythm they will compose in the third phase. The implemented analysis algorithms were designed to be error-tolerant and dynamic in detecting events that trigger the playback of sound samples.
The software was required not only to start the selected exercise according to the different modes and manage its real-time analysis but also to save the collected signals in `.csv`[6] files, with minimal effort for the user. To achieve this, an automated local archive management system was designed. The stored recordings, related to patients, had to be viewable, deletable, or exportable at any time from the application. Each exercise also included the study of different sensitivity levels, selectable and adjustable by the healthcare operator for each patient; once an appropriate sensitivity level was selected, the system needed to save it and apply it to subsequent executions. The software was to provide voice instructions to the patient at the start of the execution and allow the selection of different sources of musical samples. Finally, it had to correctly and intuitively handle errors and *exceptions* caused by the sensors.

The software project was initially started with a Python *Linux based*[7] interface, which, utilizing Xsens APIs (API *C*[8] *wrapped*[9] for Python), enabled communication with the sensors and gait analysis. However, to facilitate large-scale management in hospitals, it was necessary to *port*[10] and continue the development of the system on Windows. This allowed the creation of an installation package in .exe format, greatly simplifying distribution and installation, avoiding the need to configure the Python execution environment on each machine and resolving issues related to *dependencies*[11].

After analyzing the software requirements and the original command-line-only interface, I decided to expand and extend the Python interface to implement all the required estimation and analysis features and manage proper communication with the Graphic User Interface (GUI) that I developed with PyQt5; a *library*[12] that is comprehensive, cross-platform, and easily manageable using the *object-oriented paradigm*[13], with excellent support for *multithreading*[14].

[6] ***Files .csv***: File format used to store tabular data in plain text. Each line of the file represents a record, and values within each record are separated by commas.
[7] ***Linux based***: Designed for Linux; an open-source operating system based on Unix
[8] ***C***: Low-level programming language widely used for system software, applications, and games development.
[9] ***Wrapping***: Process of creating an interface that allows a programming language to use an API written in another language.
[10] ***Porting***: Process of adapting software or an application to work on a platform, operating system, or environment different from the one it was originally designed for.
[11] ***Dependencies***: Software components, libraries, modules, or other packages that must be installed and configured for an application or program to function correctly.
[12] ***Library***: Collection of predefined functions and classes that can be reused in various programs.
[13] ***Object-Oriented Paradigm***: Programming method where code is organized into "objects" that represent entities with attributes and methods, promoting modularity, reusability, and code maintenance.
[14] ***Multithreading***: Technique that allows simultaneous execution of multiple threads (lightweight execution units) within a single computing process. Each thread can handle parallel tasks, improving software efficiency and responsiveness.

1.0.2 Hardware

The available hardware for data collection consists of two Xsens MTw Awinda inertial sensors and a Master Device USB Dongle. Inertial data are produced with a maximum *sampling frequency* of 120 Hz and transmitted wirelessly via the patented Awinda protocol to the master device with a maximum latency of 30 ms and a guaranteed maximum communication range of 10 meters. Specifically, the MTw are miniature inertial measurement units that provide 3D angular velocity via gyroscopes, 3D acceleration via accelerometers, 3D terrestrial magnetic field via magnetometers, and atmospheric pressure via the barometer. All this combined with Xsens algorithms allows for drift-free 3D orientation data[15]. The orientation information contained in each packet sent to the dongle is described as the three Euler angles (*Roll, Pitch, Yaw*[16]). The MTw are excellent measurement units for determining the orientation of human body segments, particularly because they are designed to maintain extremely precise temporal synchronization in reading data from a wireless network of multiple units. This is essential for accurately measuring joint angles. Additionally, the back of each sensor is equipped with a Velcro patch, which facilitates the attachment of the sensor to Velcro Body straps, which can be placed around the patient's ankles. The Awinda USB dongle manages the reception of synchronized data from all wirelessly connected MTw and can receive data from up to 32 sensors simultaneously [1].

Figure 1.1: Xsense Mtw Awinda Motion Tracker and USB Master Dongle. Source: [1]

Figure 1.2: Xsense Mtw Velcro and Velcro Body Straps. Source: [1]

[15] ***Drift-free Measurements***: Stable and precise measurements where readings and performance remain constant over time and do not exhibit unwanted changes.

[16] ***Roll, Pitch, Yaw***: Angles used to describe the orientation of an object in three-dimensional space defined as rotations around the x, y, z axes.

Chapter 2

Python Interface

As mentioned, the Python interface handles communication with the Master USB dongle and with the sensors using the Xsens API, a library that provides classes to interact with Xsens devices at a basic level. In this chapter, we will explore in detail the architecture, workflow, implementation choices, and expansion of this interface.

2.1 Interface Structure

The structure of the Python interface is based on the object-oriented paradigm and mainly consists of three *classes*[1]:

- **Class `MTw`**:
  The `MTw` class manages the configuration of the Master device, detection and pairing of motion trackers, sensor calibration, data acquisition, as well as the proper closing of resources. Initially, this class exposed a single public method, `mtwRecord()`, which served as the entry point for the entire sensor configuration process, recording start, and analysis. However, a new public method, `stopRecording()`, was introduced to allow stopping the recording before the preset time ends. The `mtwRecord()` method has been modified from its original version to accept additional input parameters.

```
duration = 90
samplesPath = "../sonicwalk/audio_samples/cammino_1_fase_2"

with \texttt{MTw}.MtwAwinda(120, 19, samplesPath) as \texttt{MTw}:
    data = \texttt{MTw}.mtwRecord(duration, plot=True, analyze=True,
                          exType=0, sensitivityLev=3,
                          auto_detectLegs=False, selectedLeg=True,
                          calculateBpm=False, shared_data=None,
                          setStart=None, sound=True)
```

  Originally, the parameters were only three:

  - `duration`: (`int`) Indicates the preset duration of the recording in seconds.
  - `plot`: (`bool`) Indicates whether the data should be plotted.
  - `analyze`: (`bool`) Indicates whether the data should be analyzed.

[1] ***Class***: Model that describes the properties (attributes) and behaviors (methods) of a type of object.

If the respective flags are enabled, the *plotter*[2] and the analyzer are launched as *child*[3] (*daemon*[4]) processes and read data from shared memory, where the `MTw` object inserts angular information received from the sensors. However, in the overall SonicWalk software, the plot management always happens externally to the Python interface, directly through the GUI; therefore, the `plot` parameter is always set to `False`, while `analyze` is always `True`, as every execution phase of the exercises in the software involves data analysis.

The expansion of the `MTw` class saw the introduction of new parameters to pass to the `mtwRecord()` method, including:

- `exType`: (`int`) Specifies the exercise to be performed (e.g., walking, marching in place, swing, forward-backward load shift in tandem position).
- `auto_detectLegs`: (`bool`) Specifies whether the legs should be automatically distinguished by the system or if the association between legs and sensors is manually specified.
- `selectedLeg`: (`bool`) Only if `auto_detectLegs` is `False`, specifies whether the starting leg is the right (`True`) or the left (`False`).
- `calculateBpm`: (`bool`) Specifies whether the average BPM (Beats Per Minute) trend for the current exercise should be estimated.
- `shared_data`: (`object`) When `None`, a shared data *object*[5] (data *buffer*[6]) is *instantiated*; alternatively, `shared_data` is used to pass a preallocated shared object to be used as a buffer.
- `setStart`: (`callback`) Specifies a callback function to be called when the recording is actually started (after the initial sensor setting and opening of communication).
- `sound`: (`bool`) Specifies whether real-time audio playback (Sonification) synchronized with the analysis of signals produced by the sensors should be active or not.

In addition, the `MTw` class has been extended to include advanced error and exception handling, thereby preventing the application from crashing and providing relevant information to the GUI. Additional checks and a synchronization system for starting the `analyzers` processes have been introduced, ensuring consistency, especially during leg autodetection. A BPM estimation system has been implemented, based on a robust average obtained using the *z-score* outlier removal method. Another significant change has been adapting the code to ensure compatibility with both Linux and Windows systems, thereby increasing the software's flexibility. The control and preloading of `.wav` music samples has been removed due to compatibility issues with audio *drivers* and with the multithreading support of the *`simpleaudio`* library. This has been replaced in the `MTw` class with a preliminary check based on *path*. File loading and playback are now entirely delegated to the `Analyzer` class, which uses the *`PyGame`* library and loads the samples only when they need to be played. This choice offers excellent performance and the versatility to play both `.wav` and `.mp3` files. Finally, the object returned at the end of the execution has been enriched with additional diagnostic information about the exercises, such as BPM estimation and the localization of points of interest.

[2] ***Plotter***: Tool that allows creating graphs and data visualizations.
[3] ***Processes***: Instances of a program running on a computer.
[4] ***Daemon Processes***: System processes that run in the background without direct user interaction.
[5] ***Instantiate***: Process of creating a specific instance of an object of a class.
[6] ***Buffer***: Temporary memory used to store data in transit between two devices, applications, or processes.

The recorded data returned by the `MTwRecord` function includes various components in a complex structure that we have called `data`:

```
data0 = data[0][0]
data1 = data[0][1]
index0 = data[1][0]
index1 = data[1][1]

interestingPoints0 = data[2][0]
interestingPoints1 = data[2][1]

bpmValue = data[3]  # if calculateBpm==False it will be None
```

  - `data[0][0]` and `data[0][1]` are tuples of *NumPy* arrays containing the tilt angle buffers for the two signals. The two buffers, each of length 72,000 samples, can hold approximately 10 minutes of recording (at 120Hz), after which they are overwritten.
  - `data[1][0]` and `data[1][1]` represent the indices at which the recording was interrupted for the two signals.
  - `data[2][0]` and `data[2][1]` are tuples of `NumPy` arrays containing the approximate indices of the detected points of interest in the two signals.
  - `data[3]` contains the average BPM (beats per minute) value only if the calculation was requested and relevant data was acquired. Otherwise, it will be `False`.

- **Plotter Class**:
  The `Plotter` class manages the real-time visualization of the data produced by the sensors. It is created by the `MTw` object if the corresponding flag is specified. The plotter runs using the `multiprocessing` and `matplotlib` libraries as a separate daemon process.

- **Analyzer Class**:
  The `Analyzer` class originally handled only step detection and the cyclic activation of pre-loaded sound samples. However, it was later expanded to abstract a range of *utility methods* and support six distinct exercise detection algorithms, each specific to a particular movement or pattern in the acquired signals. When the `mtwRecord()` method is invoked and the sensors are in communication, the `Analyzer` class is instantiated twice and started in two separate daemon processes. These processes are synchronized at startup via a *semaphore* in shared memory and are responsible for analyzing the signals from the sensors located on each of the two legs.

  The analysis classes (processes) are passed various parameters and shared variables, including data buffers managed by the `MTw` class, indices storing the position of the last data entered into the buffers, the code of the selected exercise, the sensitivity level, the path to the folder containing the music samples, the leg associated with the reference sensor for each process, as well as several flags and arrays for analysis, sound, BPM estimation, and other relevant parameters.

2.2 Execution Flow

This section describes the operational and management specifications of the execution flow for recording and analyzing an exercise. For simplicity, the considered flow assumes that the exercise selection, parameter configuration, start, and consequences of recording are all actions delegated to the software's graphical interface, which will be introduced in the following chapters.

2.2.1 Starting Recording, Multithreading vs. Multiprocessing

Once the exercise, patient, and parameters are selected through the graphical interface – which runs in the main software process – it is possible to start the recording and analysis of the exercise.

It is important to note that the real-time plotting of the signals produced by the sensors during recording, in the overall software, is managed by the GUI rather than as a Daemon process, and thus also operates in the main process. To allow the plotter to easily access the buffer of recorded data, the graphical interface, running in the main process, is responsible for instantiating the shared variable to be used as a buffer. Creating the shared variable in the main process allows it to be easily passed, only forward, to the Python interface. If the Python interface receives this object instance, it is set up to use the already existing buffer, thus avoiding the creation of another one.

When the start action is triggered by the GUI, the main process creates a new instance of a class derived from the `Threading` library, in which the Python interface runs. This instance effectively becomes a child thread of the GUI's main thread. This is the only use of the `Threading` module within the software, due to the concurrency complications[7] introduced by Python's Global Interpreter Lock (GIL).

Multithreading & GIL (Global Interpreter Lock)

The Global Interpreter Lock (GIL) is a synchronization mechanism used in CPython, the most widely used implementation of the Python language, and has a significant impact on thread and parallelism management.
The GIL is a *mutex*[8] (mutual exclusion lock) that allows only one thread at a time to execute code within a process. This approach simplifies thread safety management for Python's internal data structures, avoiding concurrency and data corruption issues. Additionally, it facilitates integration with external libraries that are not designed to be *thread-safe*[9] by preventing concurrent access to Python code and ensuring that such libraries can be used without worrying about thread synchronization issues.

However, the GIL prevents true thread parallelization, limiting the ability to perform computationally heavy operations in parallel across multiple *cores*[10]. This means that, despite using multiple threads, computationally intensive operations cannot fully utilize the power of processors.

[7] ***Concurrency***: Problems that arise when multiple threads or processes simultaneously access shared resources without proper synchronization, leading to issues such as race conditions, deadlocks, and starvation.
[8] ***Mutex***: (mutual exclusion lock) A synchronization mechanism used to prevent simultaneous access to a shared resource by multiple threads or processes.
[9] ***Thread-safe***: A function or block of code is defined as thread-safe if it can be executed by multiple threads simultaneously without causing conflicts.
[10] ***Core***: Processing units within a processor that can perform operations independently.

Despite these limitations, some multithreaded operations can bypass the GIL and benefit from true parallelization. For example [2]:

- Input/output (I/O) operations: Reading from files, network requests, and other *I/O bound*[11] operations are not blocked by the GIL and can be executed in parallel across threads, improving overall efficiency.
- C/C++ Libraries: Libraries such as `NumPy` perform intensive computations in compiled languages like C or C++, which can bypass the GIL and allow for parallel execution of heavy computational operations.

Multiprocessing

To overcome the limitations imposed by the GIL and achieve true parallelism, it is possible to use the multiprocessing module. This module creates separate processes, each with its own memory space, unaffected by the GIL, allowing for parallel execution across multiple cores. However, using parallel processes also presents some complications:

- Process Creation and Management *Overhead*[12]: Creating and managing processes incurs more overhead compared to threads (particularly on Windows systems, as observed during development). This is due to the use of the *`Spawn`*[13], the only available method for creating new processes on this platform. The `Spawn` call requires complete initialization of the new process, including duplicating the execution context and importing all necessary modules, which generates significant overhead. Each process has its own memory space and separate management structures, which can increase resource consumption and the time required compared to creating threads (which share the same memory space).
- Inter-Process Communication: Separate processes do not share memory directly, so communication between them must occur through mechanisms such as *pipes*[14], *message queues*[15] or shared memory. These mechanisms can be complex to implement, introduce latency, and increase overhead.
- Synchronization: Synchronization between processes is more complex compared to threads. Inter-process synchronization mechanisms, such as *locks*[16] and semaphores, must be used to ensure safe access to shared resources.
- Data Duplication: Each process has its own memory space, meaning that data must be duplicated or copied between processes. This can lead to higher memory consumption compared to threads.

[11] ***I/O bound***: Process or system whose execution time is limited by input/output (I/O) operations rather than processing capacity.
[12] ***Overhead***: Time, resources, and additional load required beyond what is strictly necessary to perform an operation.
[13] ***Spawn***: A type of process or thread creation call predominantly used on Windows.
[14] ***Pipe***: Inter-process communication mechanism allowing one process to send data to another through a unidirectional channel.
[15] ***Queues***: Inter-process communication method allowing processes to exchange data by adding it to a queue, where it can be read later as messages.
[16] ***Lock***: Synchronization mechanism used to manage concurrent access to shared resources, ensuring that only one thread or process at a time can access them.

Multithreading Choice

In this case, it was decided to instantiate the `MTw` class in a thread rather than in a new process for four main reasons:

1. Simple and Flexible Management: Using a thread allows for simpler and more flexible management from the graphical interface. This approach facilitates the sending of signals between threads and the sharing of variables and memory, improving the integration and interaction between the graphical interface and the recording thread.

2. Minimal GIL Impact: The pseudoparallelization imposed by Python's Global Interpreter Lock (GIL) does not cause significant slowdowns or data losses in this context. During recording, it is assumed that:

   (a) The graphical interface is primarily used to monitor the patient's progress and not to perform computationally intensive operations, although other operations are not excluded.

   (b) The ordinary operations performed by the graphical interface during recording are relatively simple, such as managing Qt timers and the graphical plotter, which do not require intense CPU usage.

3. Simplicity of `MTw` Operations: The operations performed by the `MTw` class during recording are simple and mainly involve managing the data buffer from the sensors. Such operations, being I/O-bound, are not subject to the GIL limitations. Moreover, they are not complex enough to justify the overhead associated with creating and managing a new process, making multithreading a more efficient choice.

4. Parallel Execution of Xsens APIs: The internal operations of Xsens APIs are developed in C and wrapped in Python. This way, it is assumed that when multithreading is involved, the GIL is bypassed, allowing true parallelization of operations.

Once the `MTw` class is created, which handles the initial setup and the opening of radio channels, all graphical interface parameters can be passed to the `mtwRecord()` method of this class, which runs in the secondary thread.
The data buffer passed by the `MTw` class consists of two shared arrays, managed circularly, each with 1000 positions. Additionally, there are two shared values corresponding to the indices of the arrays where the last received value was inserted.

Multiprocessing Choice

The analysis class performs more complex operations (*CPU bound*[17]) compared to the `MTw` class, in real-time. To ensure true parallelization and maintain high precision without compromising performance, the `mtwRecord()` function, after calibrating the sensors by resetting the tilt angles, starts two separate child processes using the `Multiprocessing` module. These processes are dedicated to analyzing the two signals/datasets produced by the sensors.

2.2.2 Execution of the Analyzer Class

Each `analyzer` process cyclically checks, with a 3 ms interval, the termination condition (if an angle of 1000 degrees is detected in the buffer); if the condition is not met, the process extracts the last samples of size `winsize=15` from the shared memory buffer. If a new complete window is not available, the process creates a window by combining new values with a portion of previous ones. After each check and extraction of a window from the buffer, the appropriate analysis algorithm for the selected exercise is executed on it.

[17] ***CPU bound***: Process or system whose execution time is limited by processing capacity.

When points of interest within a window are detected, such as a peak or a zero crossing, various actions can be performed, including:

- Loading and orderly playing a musical sample from the samples folder and updating the shared index referring to the next sample to be played.
- Adding a *timestamp*[18] of the detection moment to a shared array.
- Adding the starting index of the window where the point of interest was detected to a shared index array. Each index approximates the position of a point of interest in the signal. Using the window index rather than the single sample index is justified by the fact that windows often overlap, thus covering negligible time intervals. Even with a complete window, the time interval is short and has a maximum time value around 125 ms. These time intervals are sufficiently short to not introduce significant variations in the signals produced by leg movements. Essentially, the vicinity of a point of interest is stable enough in such a small interval to allow the use of the window index as an approximation of the interest value index in the signal. Additionally, since these points of interest are highlighted only for *debugging*[19] purposes and are not displayed in the user interface, small errors are tolerable, as long as they provide useful indications and simplify the implementation of this mechanism.

2.2.3 BPM Estimation

The `mtwRecord()` function is designed to record pitch data for a specified time interval and return the analysis results of the collected data, in our case to the graphical interface. At the end of the recording, the function performs a series of final operations to prepare and return the processed data. The main operations include:

- Final Data Cleanup: A cleanup of the data stored in the shared arrays is performed by removing the final termination values. This ensures that only relevant data is considered for subsequent analysis.
- Timestamp Processing and BPM Calculation:
  - The two timestamp arrays, one for each signal, are merged into a single array and sorted. Subsequently, the time differences between consecutive timestamps, representing the durations of beats in seconds, are calculated.

    $$\Delta t_i = t_{i+1} - t_i \quad \text{for} \quad i = 1, 2, \ldots, (n + m - 1)$$

    The resulting array of beat durations is:

    $$\Delta t = \{\Delta t_1, \Delta t_2, \ldots, \Delta t_{n+m-1}\}$$

  - The *outliers*[20] among these time differences Δt are removed using the Z-score method with a threshold of 3. This process results in an array containing only reliable time intervals between detections, excluding any overly distant detections due to patient recovery moments or missed detections.

[18] ***Timestamp***: Temporal marker indicating the exact moment an event occurs.
[19] ***Debugging***: Process of identifying, isolating, and correcting errors or malfunctions in a program's source code.
[20] ***Outliers***: Data that significantly deviate from the rest of the dataset.

Z-score Method

The Z-score is a statistical measure that describes how much a given observation Δt_i deviates from the mean μ, in terms of *standard deviations*[21] σ. The Z-score is calculated as:

$$z_i = \frac{\Delta t_i - \mu}{\sigma} \tag{2.1}$$

where:

* Δt_i is the value of the observation,
* μ is the mean of the dataset,
* σ is the standard deviation of the dataset.

A common threshold, which has proven effective for identifying outliers, is 3. If $|z_i| > 3$, the observation Δt_i is considered an outlier and is removed from the dataset.

– If the Δt array is not empty, the average duration of a beat is calculated. By multiplying the reciprocal of the average time by 60 – that is, the conversion factor from seconds to minutes – the estimated BPM (beats per minute) value is obtained.

$$\text{BPM} = \frac{60}{\frac{1}{N}\sum_{i=1}^{N} \Delta t_i} \tag{2.2}$$

2.3 Shared Objects

Using the `multiprocessing` module, the most direct way to communicate between processes is through the use of shared objects. In this context, a main class was initially designed:

- **SharedCircularIndex**: This class manages a shared index among multiple processes and is used for managing sound samples: an array of paths contains references to the samples to be played, sorted alphabetically. The index refers to the next sample to be played and is incremented by the `analyzers` processes after each playback, safely using a lock to ensure that operations are *atomic*[22], thus avoiding *race conditions*[23]. When the index reaches the maximum value, it is reset, adopting a circular strategy that allows the samples to be played in a loop.

Subsequently, three more classes were designed to manage shared buffers, auto-detection of legs, and ensure synchronization at the start of the `analyzers` processes:

1. **SharedData**: This class was implemented to initialize shared buffers directly from the graphical interface. The class initializes and manages two arrays of 1000 elements each, used as buffers, and two indexes that store the position of the last data inserted into each array.

[21] ***Standard Deviation***: Statistical measure of the dispersion of a dataset relative to its mean.
[22] ***Atomic Operations***: Operations that are executed as an indivisible and indestructible unit, without interruptions or interference from other processes or threads.
[23] ***Race Conditions***: Two or more processes or threads access and modify a shared resource concurrently, and the outcome depends on the non-deterministic order of execution.

The arrays are implemented as `RawArray` and the indexes as `RawValue`, that is, both in *Raw*[24], allowing direct access to shared memory without the need for locks. This configuration is safe as only the `mtw` class in the secondary thread writes to the buffers, while the `analyzers` processes only read from each array and its respective index. This approach eliminates the possibility of race conditions.

2. **ProcessWaiting**: This class was designed to synchronize more precisely the startup of the two parallel `analyzers` processes, a need that arose due to the additional overhead in creating child processes on Windows, where only the `Spawn` call can be used. Specifically, this synchronization is necessary in automatic leg detection. Synchronization between processes occurs through the use of events (*Event*[25]), which act as signals between them. These events indicate when the first and second processes are ready to start. When a process is ready to start, it calls the `start` method of the `ProcessWaiting` class. The class checks if the `firstProcessStart` event has already been triggered. If the event has not been triggered, it means that the calling process is the first; therefore, the class acquires the lock, sets the `firstProcessStart` event, and puts the process into a waiting loop. If the `firstProcessStart` event has already been triggered, it means that the first process has already been put on hold, and thus the calling process is the second. In this case, the class acquires the lock, triggers the `secondProcessStart` event, and starts the second process. The first process, which was in the loop, detects that the `secondProcessStart` event has been triggered, then cleans up the events, exits the loop, and starts itself. This structure has proven, in the test system, to provide considerable synchronization accuracy for the project goals, with a time delay between the two processes of less than 0.0001 seconds.

3. **LegDetected**: In exercises where the legs perform different tasks, the configuration of the produced signals differs, as well as the analysis methods. A method was implemented to automatically detect when a leg takes the first step forward. This method, although not used in the final graphical interface for precision reasons (as detection was delicate and some patients could not correctly execute the natural initial movement), is designed to distinguish the legs from the first movement and select the most suitable analysis method for each, thus avoiding the loss of points of interest in the signals.
   The shared class `LegDetected` is used by this method to communicate between processes when the first leg performing the forward movement is detected. The class contains a *boolean variable*[26] which is protected by a lock during access and modification, ensuring that only one process at a time can read or write the value. As long as the variable is set to `false`, the auto-detection analysis continues in both processes, which cyclically check if the variable has become `true`.
   When one of the two analysis processes detects the specific hot points of the forward movement of a leg (while the variable is still `false`), it acquires the lock, sets the variable to `true`, and, knowing that it has detected the leg first, selects the correct analysis algorithm. The other process, on the next check, will find the variable already set to `true` and will understand that it needs to select the analysis method for the other leg, as the one taking the forward step has already been identified.

[24] ***Raw Format***: Representation of data in a raw state that allows direct access by multiple processes without the need for locks.
[25] ***Event***: Synchronization object used to manage communication between threads or processes.
[26] ***Boolean Variable***: Variable that can only take two states: True or False.

2.4 Porting to Windows

The Python interface initially developed on Linux required porting to Windows to facilitate the management of hospital machines and large-scale deployment. Although the graphical interface development was still in its early stages on Linux, the choice to use PyQt5 and standard libraries proved successful due to their cross-platform compatibility. However, despite the interchangeability between Linux and Windows of the GUI, the porting raised some issues:

Xsens API Compatibility

The first obstacle encountered was the availability of Xsens APIs for Python, which seemed to be limited to Linux systems. In response to this limitation, the option of replacing Xsens `MTw` Awinda sensors with Xsens Movella DOT cross-platform sensors was considered. These new sensors would have offered several advantages, including extended compatibility for both *desktop* and *mobile*[27] systems, significantly lower costs, and the integration of accelerometers. Additionally, they would have eliminated the need for the USB master dongle for communication, instead utilizing *Bluetooth*[28]. However, this solution had some drawbacks: the reliability, stability, and range of Bluetooth communication could be problematic, and adopting the new sensors would have caused delays in development. Approval from the *ethics committee*[29] would have been needed, a new batch of sensors would have to be purchased (even though the Istituti Clinici Scientifici Maugeri already had numerous Xsens `MTw` Awinda sensors), and a complete revision of the Python interface would have been required.
These concerns were resolved through a remote meeting with an Xsens expert. During the meeting, we were shown the superiority of the Xsens `MTw` Awinda sensors currently in use and were provided with Xsens APIs for Python compatible with Windows. This development allowed us to continue with the porting without the need to replace the hardware or rewrite the software from scratch, thus ensuring a smoother transition.

Code Porting

Another aspect that presented challenges was adapting the code to the specific differences between operating systems. Small, but significant, divergences emerged – described below – that required thorough analysis to ensure the correct functioning of the software on both platforms.

2.4.1 Multiprocessing Management (Fork vs. Spawn)

On *POSIX systems*[30] such as Linux (which support various process creation calls), the default system call ***`Fork`***[31] allows duplicating a process, creating a child process that inherits exactly the state of the parent process, including all global variables, open files, and other resources. All global state is left intact, making the creation of new processes much faster and lighter in terms of resources.

[27] ***Desktop and mobile systems***: Desktop systems refer to those designed to run on desktop and laptop computers, while mobile systems refer to those designed for portable devices such as smartphones and tablets.
[28] ***Bluetooth***: Short-range wireless communication technology that allows data transfer between devices via radio waves.
[29] ***Ethics Committee***: An organization composed of experts and professionals that evaluates the ethical and legal compliance of medical studies and practices.
[30] ***POSIX Systems***: Operating systems conforming to the POSIX (Portable Operating System Interface) standards, which define a set of APIs and behaviors to ensure software compatibility and portability across different Unix-like environments.
[31] ***Fork***: A type of process or thread creation call used by default on Linux.

This approach is particularly efficient, as the new process is initialized with a copy of everything in the parent process, leveraging a technique called *copy-on-write*[32], which minimizes memory overhead.

In contrast, Windows does not support `Fork`. When starting a new process with multiprocessing, Windows uses the `Spawn` method, which introduces greater overhead as it involves creating a new Python interpreter from scratch, which must reload all modules and does not automatically inherit the state of the parent process. Consequently, any *global variable*[33], resource, or state that existed in the parent process will not be available in the child process unless it is explicitly passed as an argument or recreated in the child.

These differences have practical implications for the development and porting of existing code. On Windows, if a child process attempts to access a global variable initialized in the parent process, it may not exist. Additionally, unlike Linux, where with `Fork` the process state is inherited without being re-executed, on Windows everything is re-executed from scratch for each new process, except for what is enclosed in the main section:

```
if __name__ == "__main__":
```

It is therefore essential to ensure that all code that should only be executed in the main process is enclosed in this block. This includes initializing global variables, opening files, and other operations that should not be repeated in child processes, thus avoiding errors and unexpected behavior.
Finally, since the increased overhead introduced by `Spawn` on Windows significantly slows down process startup, it was necessary to develop a synchronization method between child processes. This synchronization helped to minimize delays and ensure that the processes collaborate effectively despite the limitations imposed by the operating system.

2.4.2 Bounded-buffer problem (Producer-Consumer)

The bounded-buffer problem, also known as the producer-consumer problem, is a common situation in concurrent processing systems. In this scenario, a producer process (Xsens sensor API) generates data and inserts it into a buffer, while a consumer process (the `MTw` class) retrieves the data from the buffer. Proper synchronization between the producer and the consumer is crucial to avoid data loss, buffer overflow, and resulting slowdowns.

In our case, the `MTwCallback` class acts as an intermediary between the sensors and the rest of the system. This class contains a buffer "`m_packetBuffer`" that stores the data packets received from the sensors. The buffer is managed as a queue with a maximum capacity of 300 samples. When a sensor sends a data packet to the USB dongle, the Xsens APIs automatically insert the data packet into the buffer.
The `MTw` class, on the other hand, represents the consumer of data from the `m_packetBuffer`. Within the `MTw` class, there are additional buffers shared with analysis processes and others containing the history of all recorded angles, of which `MTw` is the producer.

If the `m_packetBuffer` is empty, it might indicate that the sensors are not sending data or that the consumer (class `MTw`) is retrieving data too quickly and is likely checking for new data more often than necessary, continuously occupying resources and stealing execution time from other threads and operations.

[32] ***Copy-on-write***: An optimization strategy that reduces the need for unnecessary copies by sharing resources between processes until a modification is made. Only then is the resource actually copied.
[33] ***Global Variable***: A variable declared outside of all classes, accessible from any point in the program.

On the other hand, if the buffer is completely full, data loss and delays occur. This happens when new data packets arrive while the buffer is still occupied by previous data, suggesting that the consumer is retrieving data significantly slower than the producer's insertion speed. A full buffer therefore indicates that the consumer is not fast enough.
The goal is to keep the buffer with a reduced number of samples, while avoiding it being completely empty and thus the situations described above. To achieve this balance, it is crucial to optimize the wait time between extraction iterations. A wait time that is too long can cause buffer saturation and data loss, while a wait time that is too short can lead to excessive CPU occupation and system overload.

The `MTw` class extracts data from the `m_packetBuffer` through a loop that checks for presence every 3 ms. This interval, designed on Linux, helps to effectively synchronize the producer-consumer problem.
However, on Windows, a 3 ms wait proved to be too long. This delay caused excessive buffer filling and resulting data loss, with detection delays up to 10 seconds, likely because the consumer's data processing time was longer and combined with the wait time excessively slowed down data extraction.

To solve the problem on Windows, it was necessary to readjust the wait time. Attempts with 2 ms and 1 ms did not yield sufficient results. Completely removing `time.sleep()` from the loop, on the other hand, caused the consumer to monopolize the CPU, leading to starvation[34] issues and preventing the execution of other threads, such as the GUI thread. Setting `time.sleep(0)`, although initially counterintuitive, proved effective. Even though it does not introduce a real delay, in Python, setting `time.sleep(0)` yields control to the *scheduler*[35], allowing other threads to execute and temporarily removing the data extraction loop from execution. This can alleviate the load on a single thread, preventing the blocking of others. This approach allowed effective management of buffer space, keeping CPU usage at an acceptable level, around a maximum of 10% on Windows 11 with an AMD Ryzen 7 3700X processor and 32 GB of *RAM*[36] (Random Access Memory).

[34] ***Starvation***: A condition that occurs when a thread is unable to obtain the necessary resources to complete its work due to another thread monopolizing them.
[35] ***Scheduler***: A component of the operating system that manages resource allocation and scheduling of processes or threads.
[36] ***RAM***: (Random Access Memory) Volatile memory used by computers to temporarily store data and instructions that the processor uses during application execution.

Chapter 3

Real-Time Signals Processing

3.1 Introduction to Signals and Analysis Choices

A signal can be defined as a physical quantity that varies over time, space, or with respect to any other *independent variable*[1]. In broader terms, it represents an entity carrying information that can be extracted, transmitted, or processed [3].

Signals can be multidimensional, meaning they can vary with more than one independent variable simultaneously. However, in the current context, we are mainly interested in studying one-dimensional or "1D" signals, which vary with respect to a single independent variable, typically time. In temporal terms, signals offer a window through which one can observe and analyze the behavior of a dynamic system. They can be of *continuous* or *discrete* nature and reflect data derived from various physical phenomena, such as pressure variations, acceleration, or, in the case at hand, the angular variation of human leg movements during physical exercise.

Analog and Digital Signals

A continuous-time signal can be represented at every instant, while a discrete signal is defined only at specific moments. To be processed digitally, an *analog signal* (characterized by continuous values and continuous time) must be sampled at regular intervals and quantized, generating a sequence of discrete-time values with finite precision, which constitutes the *digital signal*. The process of discretization, or *sampling*, is crucial for the storage and digital processing of signals, as a finite computing system cannot store an infinite amount of values. Similarly, *quantization* of values is necessary, as a digital system cannot store values with infinite precision.

Figure 3.1: Sampling. Source: [3]

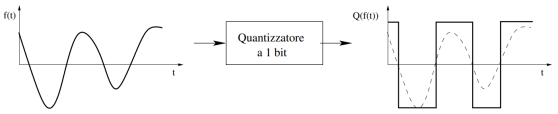

Figure 3.2: Quantization. Source: [3]

[1] ***Independent variable***: A variable that is manipulated or changed in an experiment to observe its effect on another variable (the dependent variable).

- Sampling must adhere to the *Nyquist-Shannon theorem*, which states that the *sampling frequency* must be at least twice the maximum frequency present in the signal to ensure accurate representation without information loss or *aliasing*[2].

$$f_s \geq 2f_{max} \tag{3.1}$$

- Quantization introduces an approximation in the values that can generate *harmonic distortion*[3] *periodic*[4] in signals. To reduce this distortion and obtain more accurate signals, a technique called *dithering* is used, which involves adding rounding noise. Although dithering slightly reduces signal fidelity, it eliminates harmonic distortion, which is more invasive and perceptible than the small *noise*[5] introduced. Using a greater number of bits for storage makes the effect of dithering less noticeable, as the rounding of the signal will be finer and the noise less evident.

Once digitized, signals can be analyzed with various approaches, each offering a unique perspective on the data. The two main approaches are time-domain analysis and frequency-domain analysis.

Frequency Domain Analysis

Frequency domain analysis involves transforming the signal, expressed as a function of time, into a *spectrum*, which is a version dependent on frequencies, typically through the *Fourier Transform*. This method is particularly useful for decomposing the signal into its frequency components, facilitating the identification of spectral patterns and periodic components. By analyzing the spectrum, it is possible to identify which frequencies are involved, recognize dominant frequencies, *harmonics*[6], design filters and control systems, distinguish background noise from significant signal components, and much more.

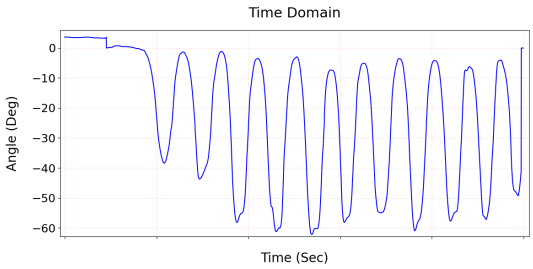

Figure 3.3: Signal in the Time Domain.

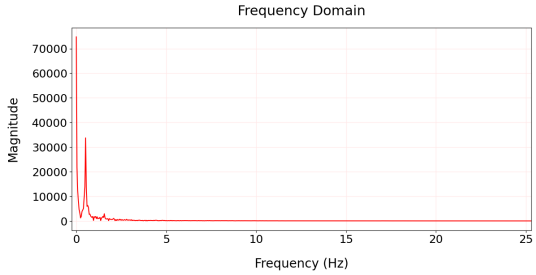

Figure 3.4: Signal in the Frequency Domain generated with the Fourier Transform.

The main mathematical tools of this approach include:

- Fourier Transform (FT): it is a mathematical transformation and can be either continuous or discrete, in continuous time or discrete time. There is also an optimized algorithmic version called *FFT (Fast Fourier Transform)*. The FT is used for decomposing a signal into a sum of *sine waves*[7] of different frequencies and is useful for moving to the frequency domain, identifying dominant frequencies, noise, and harmonics.

[2] ***Aliasing***: Frequency overlap artifacts due to sampling that does not adhere to Nyquist-Shannon.
[3] ***Harmonic distortion***: Alteration of a signal introducing unwanted harmonic frequencies.
[4] ***Periodicity***: Fundamental characteristic of a signal repeating regularly at constant time intervals.
[5] ***Noise***: Undesired interference disturbing the useful signal.
[6] ***Harmonic frequencies***: Integer multiples of the fundamental frequency of a signal.
[7] ***Sine wave***: Curve representing a mathematical sine function, characterized by regular and periodic oscillation

- *Frequency Response*: analyzes how a system responds to sinusoidal signals of various frequencies. It is essential for designing *LTI filters*[8] that attenuate or amplify specific frequencies. Essentially, it shows the relationship between the input and output of a system as a function of frequency.
- *Impulse Response*: it is the representation of a system's response to a *Dirac Delta*[9]. The impulse response fully characterizes the behavior of systems and allows representing all signals as a weighted sum of impulses. Through the operation of *convolution* between a system's impulse response and any complex signal, it is possible to calculate the system's response to the signal.
- *Z Transform*: it is a generalization of the Fourier Transform for discrete signals. It provides a representation of the signal in the *complex domain*[10], useful for analyzing and designing discrete systems and digital filters.

Despite the wide possibilities offered by frequency analysis, capturing transient or short-duration events can be challenging, as this type of analysis predominantly focuses on *stationary*[11] and periodic signals.

Time Domain Analysis

Time domain analysis, unlike frequency domain analysis, examines the signal as it was collected, observing its variations over time. This approach is particularly useful for detecting specific events, monitoring sudden or transient changes, interpreting signals without periodicity or with critical and random behaviors. Time-domain analysis allows for direct observation of characteristics that may not clearly emerge with frequency domain analysis, providing an overall view of the signal's behavior.

There are several techniques based on event detection that involve extracting *features* from signals by analyzing their behavior over time. Some of the most common features that can be detected are:

- Zero crossings: These are all the points where the signal crosses the x-axis. Useful for identifying sign changes and often used for fundamental frequency detection.
- Peaks: These are points where the signal reaches the maximum value within a specific range. Peaks are usually very significant events; for example, in an ECG, they represent heartbeats.
- Notches: These are points in the signal where a dip or significant reduction in amplitude is observed. Notches are useful for detecting attenuation or cancellation events and are often used in vibration and acoustic analysis to identify resonances or interferences.

[8] ***LTI filters***: Linear Time-Invariant systems described mathematically by differential equations or transfer functions. Their characteristics do not change over time and process signals such that the output is a linear combination of the inputs.
[9] ***Dirac Delta***: Ideal impulsive signal with an infinitely narrow spike and unit area.
[10] ***Complex domain***: Mathematical domain of complex numbers used to analyze functions and signals.
[11] ***Stationarity***: Characteristic of a signal where its statistical properties do not change over time, making it stable and predictable in the long run.

Convolution, Pattern Matching, and CNN

In addition to these techniques, to identify specific sequences and profiles in a signal, convolutional methods can be used. Convolution is a fundamental mathematical operation capable of combining two signals. In practice, a signal or a *kernel* (matrix filter) designed specifically is slid across the incoming signal, and at each position, a *linear combination*[12] of the signal values weighted by the filter is performed, creating a resulting signal that reflects the characteristics of the applied filter.

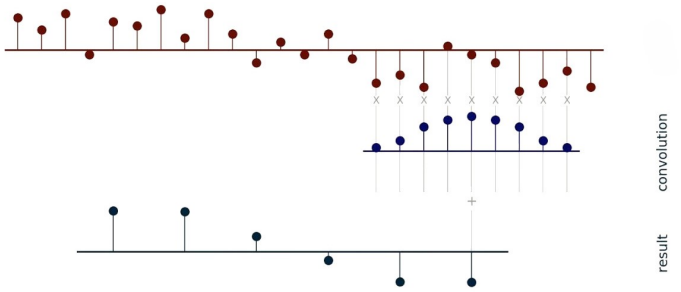

Figure 3.5: 1D Convolution.

In the context of two digital signals (composed of discrete numerical sequences) $f[n]$ and $g[n]$, discrete convolution must be applied, mathematically defined as:

$$(f * g)[n] = \sum_{k=-\infty}^{\infty} f[k] \cdot g[n-k] \tag{3.2}$$

where:

- $f[n]$ is the input sequence (original signal),
- $g[n]$ is the filter or kernel sequence,
- n is the time index of the resulting convolved sequence.

In this formula, k is the summation variable that traverses the domain of sequences, and $(f * g)[n]$ represents the convolved sequence resulting at position n.

- Through convolution, it is possible to apply the technique of *pattern matching*, a method widely used in signal processing, image recognition, and audio analysis to detect relevant events or specific sequences. Pattern matching allows for comparing a signal with a known model. In this context, the filter is a pattern or model representing the shape or sequence to be recognized and is slid and combined with the incoming signal. Points where the result of the convolution exceeds a certain threshold will indicate the presence of a match between the signal and the pattern, allowing the identification of relevant events and trends similar to known ones.
- Convolution also finds application in the frequency domain, where filters can be designed to isolate or emphasize specific spectral components of the signal to be detected.

[12] ***Linear combination***: Weighted sum of multiple vectors or functions, where each vector or function is multiplied by a scalar coefficient.

- Even in the context of *neural networks*[13], convolution plays a crucial role, particularly in *Convolutional Neural Networks* (CNNs). CNNs are neural network architectures particularly effective for analyzing multidimensional structured data in matrix form. Filters, or kernels, are applied to the data through convolution to extract *feature maps*[14]. Each convolutional layer of the network applies different filters, each designed to recognize specific features. The training of CNNs involves optimizing the parameters of the convolutional filters through *backpropagation*[15] and *gradient descent*[16]. This process allows the network to automatically learn the most effective filters for solving specific problems. However, once training is complete, how the neural network processes and recognizes patterns in data can be somewhat obscure. This phenomenon is often described as the "black box" problem.

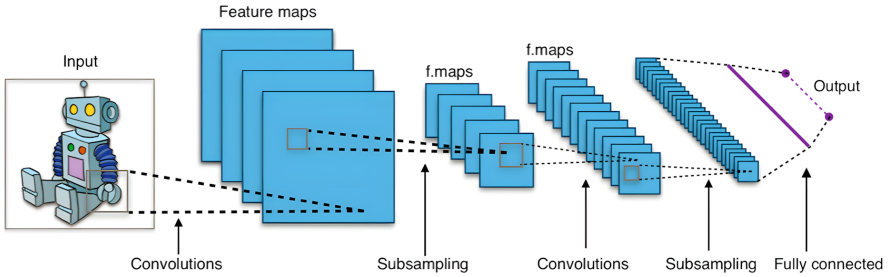

Figure 3.6: Fully connected neural network architecture, typical of CNNs. Source: `https://www.wikipedia.org`

Time-Frequency Analysis

Although transient phenomena are better represented in the time domain, some techniques can represent signals in both frequency and time terms, such as the *Short Time Fourier Transform* (STFT), useful for generating *spectrograms*[17], or the *Wavelet Transform*, useful for analyzing signals at different resolution scales.

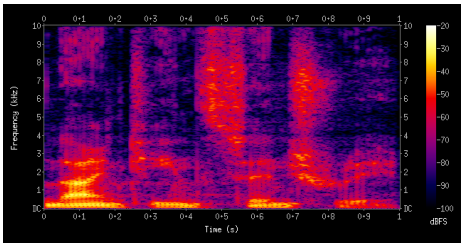

Figure 3.7: Spectrogram: the horizontal axis represents time, the vertical axis represents frequency, and the color intensity indicates the signal amplitude for each frequency and time instant.
Source: `https://www.wikipedia.org`

[13] ***Neural networks***: Computational models composed of perceptrons inspired by the functioning of the human brain, used in the field of artificial intelligence.
[14] ***Feature maps***: Intermediate representations of an image or signal obtained from CNNs, where each map highlights specific features extracted from the data.
[15] ***Backpropagation***: Learning algorithm for neural networks that calculates the error between the predicted and actual output and propagates it backward through the network to update the weights and improve model accuracy.
[16] ***Gradient descent***: Optimization method that updates the parameters of the neural network model in the opposite direction of the gradient of the loss function, in order to minimize the error and improve performance.
[17] ***Spectrogram***: Visual representation of the frequency variation of a signal over time.

In conclusion, the choice between time-domain analysis, frequency-domain analysis, or the application of neural networks depends on the nature of the signal, the information to be extracted, and the available computational resources.

Choices for Real-Time Analysis

The signals produced by the lower body joint movements during the execution of repetitive rehabilitative exercises generate *pseudo-periodic* signals, as the general trend recalls periodicity, but with a good dose of uncertainty and unpredictability due to the precision of execution, balance management, and individual variations in movements. The need to identify specific events that occur rapidly and in a pseudo-periodic manner implies that the system must be able to detect changes in the signal with high temporal precision. The ability to respond immediately to these events is crucial for software feedback, which must provide timely and accurate responses to effectively support the rehabilitation process. Managing such precision and reactivity requires the implementation of advanced algorithms and real-time analysis techniques to minimize error and optimize the system's response to dynamic signal variations.

In this context, *event-based analysis* in the time domain represents the best choice. With a sampling frequency of 120 Hz, the sensors offer high temporal resolution, allowing rapid and transient events to be detected with great flexibility. Time-domain operations are less computationally expensive and offer faster responses, essential for providing real-time software feedback. This approach allows for the identification of specific events without the added complexity of frequency transformations and better adapts to irregularities. Applying analysis techniques directly to data windows in the time domain can help maintain a continuous and progressive view of the signal, enabling timely event detection. It is important to note that this methodology does not directly provide information about the signal's frequency components, which might be useful if one wishes to analyze periodic patterns or identify specific frequency components and separate them from noise. In this context, it is useful to consider that the data from the sensors are very stable and accurate due to the pre-filtering performed by the Xsens algorithms. Consequently, it is not strictly necessary to further reduce noise. It is often simpler and more immediate to detect points of interest as they occur, rather than using the frequency domain to analyze the periodicity of a potentially irregular signal one window at a time. In fact, with windows of only 15 samples, which are necessarily short to maintain real-time perception and respond promptly, the frequency resolution is limited to the single window. Even with the Short Time Fourier Transform (STFT), frequency analysis might not be sufficient to identify fine details of spectral components. Additionally, such short windows can introduce *leakage effects*[18] and only address a small real-time segment, failing to describe and analyze the global periodicity of the signal. Accumulating windows through buffering and analyzing everything that has arrived previously could be counterproductive, as it would require performing global calculations with each new window. Moreover, FFT and frequency operations often involve a greater computational load compared to time-domain calculations.

Therefore, in order to maintain efficiency and timeliness in event detection, it is preferable to use time-domain analysis, reserving spectral analysis only for cases where it is strictly necessary for complete signal interpretation.

[18] ***Leakage effect***: Distortion that occurs in spectral analysis when a signal is not periodic with respect to the observation window, causing the signal's energy to spread over adjacent frequencies and compromising the accuracy of the spectral estimate.

Neural networks, although very powerful tools, may be excessive for analyzing one-dimensional signals where the main features are already easily identifiable through simpler techniques. Although their use can improve the detection system, their implementation requires a significant amount of computational resources for a lengthy and complex training phase, which might not justify the incremental improvement over traditional methods. Moreover, neural networks might not offer a significant advantage in scenarios where signals need to be analyzed in real time, due to the high computational load and memory and storage space they require during execution and installation. This can not only compromise real-time performance but also limit the machines on which they can be run, making them less applicable and distributable on common hospital management computers. In real-world contexts, this can further complicate the request for necessary installation permissions to use such systems on protected hospital machines, where approval from the committee is needed to install new solutions.

Given the "simplicity" of 1D signals, the use of simpler and more direct techniques such as temporal analysis has been considered more practical and sufficiently effective for real-time event detection.

3.2 Utility Methods

In the `Analyzer` class, in addition to the private methods `nextWindow()`, `endController()` and `terminate()`, which respectively extract the new window from the shared buffer, check the termination condition, and insert the termination value into the buffer, there are also several analysis methods for the various exercises, as well as a set of utility methods designed to abstract recurring calculations and operations. Utility methods:

- `extractParameters()`: During experimental tests, the need for a system capable of supporting different levels of sensitivity to small movements during analysis and being flexible for configuring detection algorithm parameters was highlighted. Each implemented analysis method was therefore designed to operate on five distinct sensitivity levels. For each level, a study was conducted to map the different values of the analysis algorithm parameters. Each technical parameter is thus wrapped through a single sensitivity value. Note that sensitivity is almost always inversely proportional to system accuracy, as detecting smaller movements can more easily lead to false detections due to noise.
  To ensure modularity and flexibility, and to facilitate the study, modification, and testing of these parameters without having to modify the algorithms, Python code, recompile the executable, or the installable, the parameters are stored in an external *JSON*[19] file named `sensitivity_levels.json`. This JSON file can be updated at any time, even after compilation and installation, and even at *run-time*[20].
  The `extractParameters()` method, based on the chosen sensitivity level, is responsible for retrieving and extracting the values from the JSON file for use during analysis.

- `playSample()`: This method, when invoked, retrieves the current index of the shared circular buffer containing the paths of the musical samples. Using PyGame, the method plays the sound sample corresponding to the path found at the specified index. It then increments the detected movements counter and updates the circular buffer index, pointing it to the next sample to be played.

[19] ***JSON***: (JavaScript Object Notation) A lightweight data interchange format that is easy to read and write, based on text, used to represent data structures such as objects and arrays.
[20] ***Run-Time***: The period during the execution of a program.

- `zeroCrossingDetector()`: This method is fundamental for feature detection in the signal, specifically designed to detect the presence of zero crossings in a data window that is passed as input. Generally, the data represents the current window of 15 samples of orientation received from the sensors. The function analyzes these samples to identify zero crossings by comparing the signs of the values: if it finds two adjacent discordant signs, a zero crossing is detected. If more than one is found, the result is considered noise and the function returns `None`, as it is unrealistic for a signal produced by a human limb to repeatedly cross zero within a short interval (about 0.125 seconds). If only one crossing is identified, the function calculates the corresponding *gradient* using the `Numpy` library. If a maximum threshold for the absolute value of the gradient is defined and the calculated value exceeds this threshold, the function terminates without returning anything. If the threshold is not specified or the gradient is below it, the crossing is considered valid. If a *polarity*[21] has been specified, the function compares the gradient's polarity with the required one. If the gradient is valid, regardless of polarity, the function returns a `ZeroCrossingDetectionResult` object containing the calculated gradient and a boolean flag indicating whether the detected polarity matches the specified one.

  Slope Calculation at a Point
  In the continuous context, the slope of a function $f(x)$ at a specific point is represented by the *first derivative*[22] of the function with respect to the variable x. The slope measures the rate of change of the function and represents the inclination of the *tangent*[23] to the function's curve at that point.

$$\text{Slope} = \left.\frac{df(x)}{dx}\right|_{x=x_0} \tag{3.3}$$

  where x_0 is the point where the slope is calculated.

  In the context of discrete data analysis, such as *time series* or digital signals, the concept of slope is closely related to the gradient. The gradient measures the rate of change of a sequence of discrete data relative to the sampling points and is often calculated using the *finite difference method*. This numerical method allows for approximating derivatives – which are defined only for continuous functions – when working with discrete data.

  Finite differences rely on approximating the derivative of a function at a point using the function values at nearby points. Supporting this methodology, the *Taylor series expansion* plays a crucial role. It allows expressing the values of a function at points near x as a linear combination of the function's value $f(x)$, its derivatives, and the distance h between sampling points.
  Taylor series expansions for a function $f(x)$ around point x are given by the following expressions:

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2}h^2 + \frac{f'''(x)}{6}h^3 + \dots$$

$$f(x-h) = f(x) - f'(x)h + \frac{f''(x)}{2}h^2 - \frac{f'''(x)}{6}h^3 + \dots$$

[21] ***Polarity***: Direction and intensity of a signal in relation to a reference. In this case, it refers to positive and negative signs.
[22] ***First derivative***: Represents the instantaneous rate of change of the function value with respect to the independent variable.
[23] ***Tangent***: A straight line that touches a specific point of a function.

Subtracting the two expansions, we obtain:

$$f(x+h) - f(x-h) = 2f'(x)h + \frac{2f'''(x)}{6}h^3 + \dots$$

From this relation, we can solve to obtain an estimate of the first derivative $f'(x)$:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

This expression shows that the error of the approximation, indicated as $O(h^2)$, is proportional to h^2, which implies that the accuracy of the approximation increases for smaller values of h.

In the specific case of `Numpy`, the gradient of an array of discrete data is computed using second-order central *finite differences*[24] for the interior points of the array. The formula used is:

$$f'(x_i) \approx \frac{f(x_{i+1}) - f(x_{i-1})}{2h} + O(h^2)$$

Where h represents the distance between sampling points and $O(h^2)$ is the error term, which is proportional to h^2. This approach provides an accurate estimate of the gradient at the interior points of the array.

At the edges of the array, where central finite differences cannot be applied, `Numpy` uses unilateral finite differences, which can be first or second order depending on the chosen `edge_order` parameter:

Edge	**Order**	**Formula**
Initial (Left Side)	First	$f'(x_0) \approx \frac{f(x_1)-f(x_0)}{h} + O(h)$
	Second	$f'(x_0) \approx \frac{-3f(x_0)+4f(x_1)-f(x_2)}{2h} + O(h^2)$
Final (Right Side)	First	$f'(x_n) \approx \frac{f(x_n)-f(x_{n-1})}{h} + O(h)$
	Second	$f'(x_n) \approx \frac{f(x_{n-2})-4f(x_{n-1})+3f(x_n)}{2h} + O(h^2)$

Table 3.1: Table of differentiation formulas for initial and final edges of a dataset

The final result is an array of gradients corresponding to each point of the dataset, with the same size as the input array. This method, stopping at the second order, balances computational efficiency with accuracy, making it an effective choice.

- `setNewGradientThreshold()`: This method is used by some implemented analysis algorithms to dynamically update the maximum gradient threshold as soon as a valid gradient is detected. The update is performed using *Exponential Moving Average* (EMA), a technique that allows adapting the threshold in response to changes in data smoothly and reactively. The method is passed the newly calculated gradient and an *alpha* parameter, as well as an additional minimum limit for the threshold.
  EMA is particularly advantageous because it balances the influence of the most recent data with that of past data, allowing a timely response to changes without being overly sensitive to sudden fluctuations. This is useful in scenarios where data may vary rapidly, and a threshold that dynamically adjusts while maintaining some stability is desired.

[24] ***Order***: In the context of Taylor series, represents the number of series terms used to approximate the derivative.

Exponential Moving Average (EMA) is:

$$\text{Thresh}_{\text{new}} = \alpha \cdot \text{Grad}_{\text{new}} + (1 - \alpha) \cdot \text{Thresh}_{\text{prev}} \tag{3.4}$$

where:

- $0 < \alpha < 1$: is the smoothing parameter that controls the influence of the new gradient on the threshold. Higher values give more weight to new data.
- G_{new}: is the value of the newly calculated gradient.
- $\text{Thresh}_{\text{new}}$: is the updated threshold value.
- $\text{Thresh}_{\text{prev}}$: is the threshold value before the update.

To ensure that the threshold does not drop below a defined minimum value, a limit is also applied to the calculated threshold.

$$\text{Thresh}_{\text{final}} = \max(\text{Thresh}_{\text{new}}, \text{Thresh}_{\text{min}})$$

- `phaseAnalyzer()`: This method is designed to determine whether the signal is in a growth or decay phase, based on the last two received data windows. To do this, the method compares the *maximum* and *minimum* values of the windows. Although it is possible to detect growth or decay by comparing samples within the same window, the approach of comparing two consecutive windows is closely related to the operation of the `peakFinder()` method and represents a generalization of it. However, since it is rarely used, it has not been further modified or optimized.

  Specifically:

  - If the maximum value of the current window (`current_window`) is greater than the maximum value of the previous window (`previous_window`), the method identifies a growth phase.
  - If the minimum value of the current window (`current_window`) is less than or equal to the minimum value of the previous window (`previous_window`), the method identifies a decay phase.

  The method returns `True` if the system detects the specified phase (growth or decay), otherwise it returns `False`.

- `peakFinder()`: This method is designed to identify local maxima or minima (positive peaks and minima, notches, and negative maxima) in a signal, which being real-time cannot use traditional functional analysis methods. The analysis is performed by examining three adjacent data windows:

  - `previous_window` (older window),
  - `window` (previous window where the check is performed),
  - `current_window` (current – new window)

  The analysis can only be performed when three "*complete*" windows are available, so it is essential to ensure that the three analyzed windows do not overlap in values.

  The method returns True if the detected feature matches the specified one (positive peak, negative peak, positive minimum, or negative minimum), otherwise it returns False.

- **Searching for a local maximum:**
  - The method compares the maximum value of the `window` with the maximum values of the adjacent windows (`previous_window` and `current_window`).
  - If the maximum value of the `window` is higher than the maxima of the two adjacent windows, a local maximum is identified.
  - Subsequently, the method checks if this local maximum is positive or negative.
- **Searching for a local minimum:**
  - The method compares the minimum value of the `window` with the minimum values of the adjacent windows (`previous_window` and `current_window`).
  - If the minimum value of the `window` is lower than the minima of the two adjacent windows, a local minimum is identified.
  - Subsequently, the method checks if this local minimum is positive or negative.

The algorithm checks for features in the central window, which is chronologically the penultimate one. This means it is always delayed by one window compared to the current signal. However, since each complete window is composed in about 125 ms, the detection delay is generally small and not perceivable in the application context. In fact, no analysis algorithm used relies solely on peak detection to determine when to emit a sound. Peak detection is mainly used to support subsequent detections. In practice, the system does not require emitting sounds exactly at the peak point but can do so with some temporal flexibility.

The window-based comparison approach, rather than directly comparing three adjacent samples, was used because it significantly reduces the influence of noise. Adjacent samples, in fact, experience small random fluctuations that could lead to incorrect and/or repeated identifications of local maxima and minima. However, the growth or decay phase of an entire window is generally preserved, as Xsens MTw Awinda sensors provide quite clean and robust signals with small fluctuations, allowing window comparison.

To further improve the method, a light *Gaussian filter* can be applied to the windows to smooth the data and further reduce noise, thus improving detection accuracy. However, it is important to avoid overly aggressive Gaussian filters or merging windows, as this could otherwise introduce steps between them and detect changes that are not actually present.

- `runAnalysis()`: When the `Analyzer` processes have been created, synchronized, and can start executing, the internal `runAnalysis()` method is responsible for calling the analysis algorithms – which will be described in detail below – corresponding to the exercise and inputs specified in the GUI. `runAnalysis()` operates iteratively within each `Analyzer`, cyclically calling the analysis algorithms at regular intervals of 3 ms. In each cycle, the method extracts a new data window from the corresponding leg buffer and analyzes it.

3.3 Walking

In walking, both legs perform the same type of movement, which implies that the same detection algorithm is applied to both analysis processes. The signals produced by the Motion Trackers show a generally sinusoidal and therefore periodic trend, although they may exhibit unpredictable variations related to execution precision, influenced by the patient's condition. In this context, each step represents a cycle measured from the initial heel contact with the ground to the subsequent contact of the same heel.

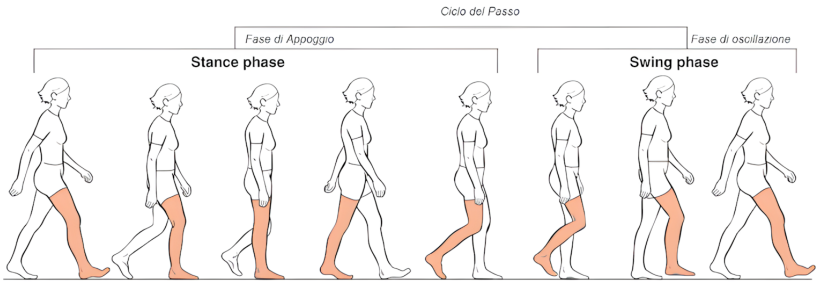

Figure 3.8: Step cycle. Source: `www.formativezone.it`

As observed from the step cycle (fig. 3.8), every time the heel touches the ground (positive angle), it is followed by a shift of the load onto the leg, bringing the tibia to a position perpendicular to the ground, and then continuing the movement with a negative angle. This perpendicularity corresponds to an angle of 0°. Note that the setup of the sensors in the MTw class always occurs with the patient standing upright, so the sensors are calibrated to detect an angle of 0° when both legs are perpendicular to the ground. This crossing of perpendicularity is thus represented as a negative zero crossing in the resulting signal.

Figure 3.9 shows five complete step cycles in the signal produced by one leg; the interval between two consecutive red features (negative zero crossings) represents one step.

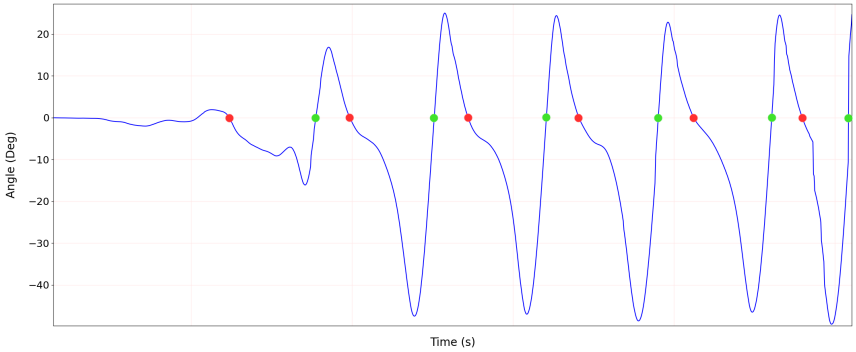

Figure 3.9: Signal from one leg produced by 5 steps.

The step analysis algorithm is tasked with performing the following detections:

- A negative gradient zero crossing marks the end of the swing phase and the beginning of the support phase.
- A subsequent positive gradient zero crossing indicates that the foot has returned to the swing phase.
- A further negative gradient zero crossing completes the step, which is counted and accompanied by the playback of an audio sample.

To play samples in conjunction with the heel strike (temporally anticipated compared to the perpendicularity of the tibia to the ground), it is possible to spatially anticipate the detection by translating all new signal windows by, for example, 5 degrees lower.

$$f_{\text{translated}}(x) = f(x) - 5 \tag{3.5}$$

This strategy is possible because valid positive peaks of a step ideally always exceed 5 degrees, allowing the translation without losing the zero crossings. Since the zero crossings of interest are negative, this spatial anticipation results in a temporal anticipation (if they were positive, there would be a delay) that allows the system to sound in correspondence with the heel strike. In practice, it is as if the system is activated at 5° rather than at 0°.

In image 3.10, the signal before (gray) and after (blue) the translation is represented, highlighting the zero crossing points. The points that would be detected without translation are shown in gray, while those detected with spatial anticipation are indicated in pink. As can be seen, this displacement allows for an early temporal detection.

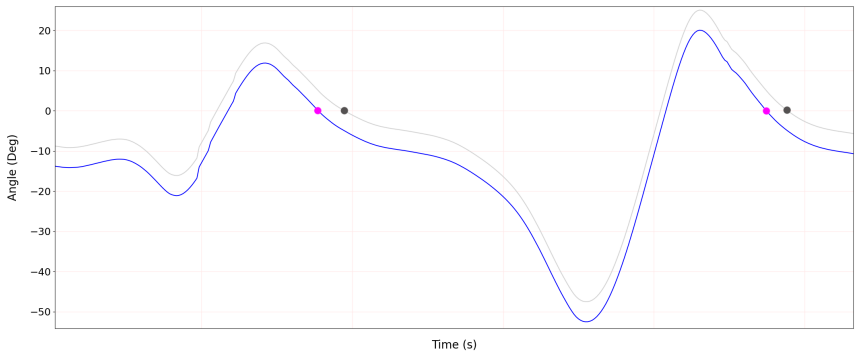

Figure 3.10: Spatial translation of -5 degrees with resulting temporal anticipations on negative zero crossings.

Zero Crossing Validation

The function `stepDetector()` analyzes the content of the last received data window. Using the `zeroCrossingDetector()` method described on page 23, the function looks for zero crossings with a negative gradient within the data window and returns `True` if one is found.

- In such a small window, approximately 125 ms, only a single zero crossing is considered valid by `zeroCrossingDetector()`, as frequent polarity changes can be considered noise.
- During the period from the initial contact to the load response (corresponding to 0-10% of the step cycle), due to noise, a gyroscopic signal may cross zero with a negative gradient more than once. For this reason, to avoid false detections, a timeout is used during which all detected zero crossings are ignored. This timeout, although parameterized, is set for each sensitivity level to 100 ms from the last valid negative zero crossing, corresponding to 7% of the frequency of a fast walk, about 1.5 steps per second for a single leg.

  Figure 3.11 shows an example of step rejection: the first cycle, highlighted with a red segment, represents a step that will be rejected because its period is below the temporal threshold indicated by the green segment. The cycle marked by the blue line, however, will be considered a valid step.

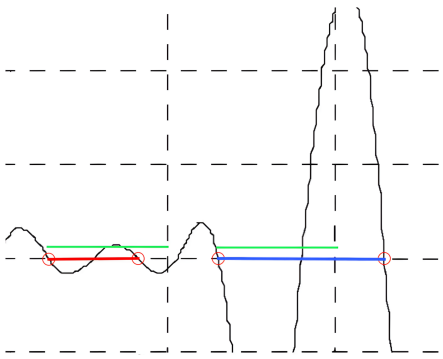

Figure 3.11: Example of temporal step rejection (red). Source: [4]

- To validate the steps, an adaptive peak threshold is used that can adapt to signal variations over time. A step is considered valid only if the highest value (peak) recorded before the negative gradient zero crossing exceeds this threshold. Consequently, when the threshold is low, both small and large steps can be detected. However, with a higher threshold, detecting small steps becomes more difficult, advantageously reducing the risk of false positives but decreasing the system's sensitivity.

  Peak detection does not occur in real-time; it is determined by analyzing the history of recorded values from the last negative zero crossing up to the current detection.
  When a valid peak, that is, above the threshold, is identified retrospectively, it is stored in an array that keeps the history of the last 10 detected peaks, initially populated with values such as 5°. The following negative zero crossing and therefore the associated step are also considered valid, allowing the system to play the audio sample and/or record the timestamp. Subsequently, the dynamic threshold is updated to the lowest peak value present in the history, allowing it to increase gradually.

  If a peak is below the threshold but within a range of, for example, 3°, it is still considered valid and contributes to updating the threshold. This allows the threshold to also decrease slowly over time, down to a minimum of 5°, ensuring adequate flexibility in detecting steps of varying intensity.

  The mechanism is designed such that the threshold decreases by at most 3° at a time, while it can increase more quickly in response to higher peaks. The initial threshold is set to 5°, with a validity range of 3°, thus providing a 2° margin to avoid noise detection during stasis.

  If a peak is below the validity range of the threshold, it is discarded, the zero crossing is not considered, and the step is not validated.

  In figure 3.12, the first negative gradient zero crossing is preceded by a peak below the threshold (indicated by the red line) and therefore will be discarded. In contrast, the next two steps will be correctly detected, as both have a peak above the validation threshold.

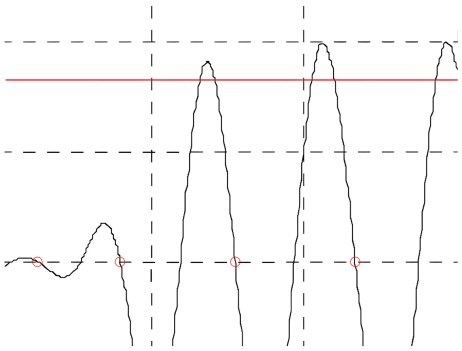

Figure 3.12: Validation threshold of the walking algorithm with rejection of the first peak. Source: [4]

3.3.1 Test Results

"*In a study of 30 healthy subjects, the tests, lasting 1 minute, were performed at different speeds on flat ground, with the subjects walking back and forth within 2-5 meters of the device. The subjects could freely vary their walking speed, and the average speed was calculated in steps per minute. The results were divided into three speed categories: slow, medium, and fast. The algorithm showed performance above 93%*" [4].

Speeds	Actual steps	Detected steps	Accuracy
$\leq$ 60 steps/min	47	45	95.7%
	48	48	100%
	54	54	100%
	58	57	98.2%
	58	58	100%
60-80 steps/min	65	65	100%
	75	74	98.6%
	71	69	97.1%
	68	65	95.5%
	75	73	97.3%
$\geq$ 80 steps/min	92	90	97.8%
	95	89	93.6%
	95	91	95.7%
	92	90	97.8%
	90	87	96.6%

Figure 3.13: Table of the gait algorithm results. Source: [4]

"*However, the main errors arise from steps with positive peaks below the threshold, often due to very short steps or stationary rotations. Increasing the speed leads to more frequent stationary rotations, negatively affecting the algorithm's performance. This occurs especially when the angle of the leg's movement does not change polarity*" [4].

The parameters displacement (signal translation), minimum threshold value, validity range below the threshold, and waiting time between two detections were subsequently parameterized in the JSON file to address the issues mentioned above.

3.4 Marching in Place

The algorithm for detecting marching in place is flexible and is inspired by the one used for step detection in walking. However, the nature of the signal produced during marching in place allows for simplifying the zero crossing validation logic.

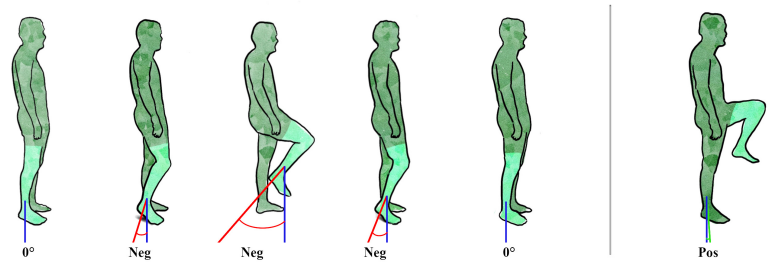

Figure 3.14: Marching in place cycle. Illustrated by: Nicola Montagnese

During marching in place, as in walking, the legs perform similar movements, generating two analogous sinusoidal signals. When one leg is perpendicular to the ground (standing straight), the angle is 0°. Raising the knee causes the ankle to retract, producing a zero crossing and forming a negative angle. When the knee reaches perpendicularity to the body, the ankle is retracted to the maximum, generating a notch with a negative local minimum. Subsequently, as the knee lowers, the angle of the ankle progressively returns to zero, reaching it when the foot is fully on the ground. At this moment, the body's load is completely transferred to that foot, as the other leg starts its movement. The load transfer and the lifting of the other leg cause a slight forward shift, resulting in a zero crossing and the formation of a small positive peak. When the other leg finally returns to the ground, the angle of the first leg returns to 0°, and the cycle repeats.

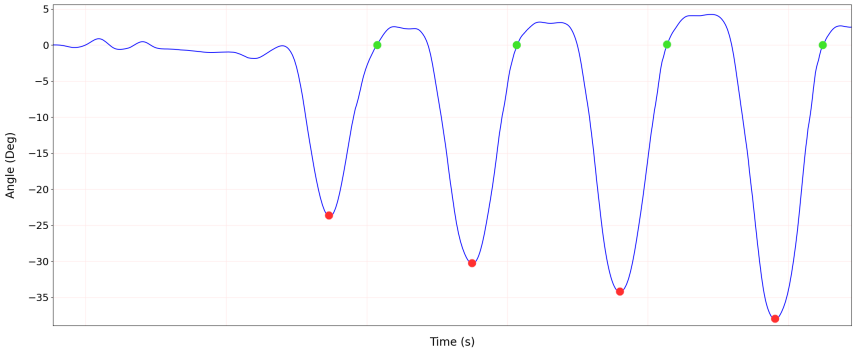

Figure 3.15: Marching in place with variable amplitude movements and highlighted features.

The signals produced during marching in place mainly extend into the negative quadrant, with the minima and positive zero crossings representing the features of interest. To utilize part of the step detection algorithm used in walking, it was sufficient to reverse the polarity of each signal window. In this way, the features of interest align with those analyzed by the walking algorithm: the notches become peaks, and the positive zero crossings transform into negatives.

Compared to walking, however, raising the knee during marching in place causes a wider movement of the ankle, generating higher peaks for small movements.
This characteristic keeps the distance between valid peaks and noise peaks wide even in small movements, making it always possible to separate them using a fixed threshold in the middle (set, for example, at 10 degrees). This way, all actual peaks are validated as they are always above the threshold, while the noise is always below and is ignored.

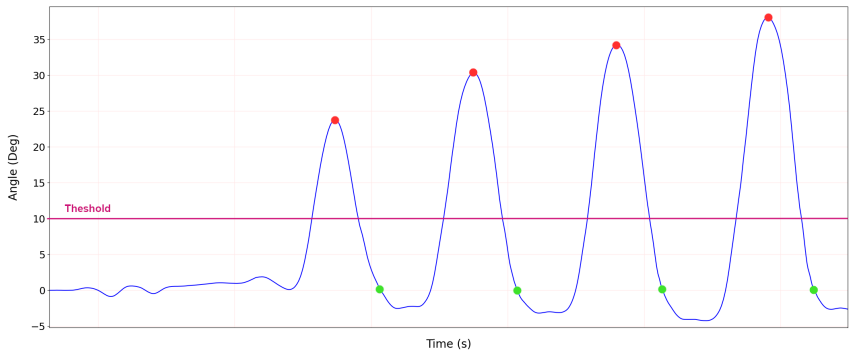

Figure 3.16: Marching in place with displacement after inversion and highlighted features

Since the zero crossing of interest occurs when the foot is fully on the ground and the other leg has already started the movement, but the goal is to detect the step as soon as the foot touches the ground, a temporal anticipation is necessary. As in walking, this anticipation is obtained by shifting the signal downward.

When there are lower peaks, generated by almost negligible knee flexions, they are typically associated with a more severe condition of the patient, who performs the exercise with greater insecurity, less balance, and more slowly. In this situation, a lower peak validity threshold is required. Since the execution time is longer and the peaks are lower, the gradients become less steep, which means that to maintain the same temporal anticipation, the displacement must also be reduced proportionally.
On the other hand, when the safety in performing the exercise increases and the knees are bent more, the threshold can be raised. This reflects an improvement in performance and a higher speed of the patient, who, with the increase in peak height, makes it necessary to also raise the displacement to maintain the same temporal anticipation.

Due to this direct proportionality, the threshold is not necessary and can be incorporated into the displacement, determined by the chosen sensitivity level. For example, by shifting the signal down by 10°, not only is the desired temporal anticipation achieved, but all noise peaks below 10° (which fall below zero) are also automatically excluded, keeping only the positive peaks above the threshold.

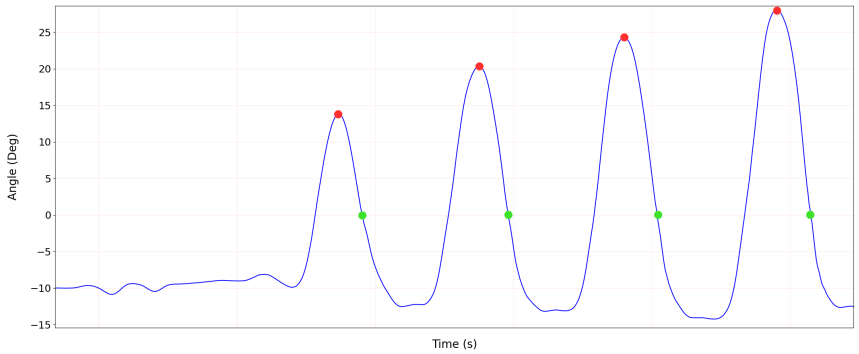

Figure 3.17: Marching in place with displacement after inversion and highlighted features

3.5 Leg Detector

Some exercises require that the legs move differently from each other, generating different signals that need to be analyzed with distinct algorithms. For this reason, it is essential to be able to distinguish between the legs.

A first solution to this problem, chosen as the best solution for the SonicWalk software, is based on the sorting of sensor identifiers: each Xsens MTw Awinda sensor has a unique ID, and the IDs are reordered in alphanumeric order. The sensor corresponding to the first ID is always associated by the interface with the right leg, while the second ID corresponds to the left leg. This sorting system allows for clear identification of which sensor is associated with which leg, manually determining from the GUI which is the starting one, and thus establishing the analysis algorithms to be used for each of them.

A second and more complex solution, which has not been integrated into the current user interface, has been developed and implemented in the method `detectLeg()` which, although functional in limited contexts, requires further refinements. This solution has the potential to automatically distinguish the legs at the beginning of the movement, thus allowing the automatic assignment of the analysis algorithm to each leg and making the sensor order irrelevant.

The principle behind recognition is that, in any exercise, when starting from a position with feet together, the leg that moves first is the one that is brought forward. This convention allows for the identification of the initial movement signal of the advancing leg and distinguishing it from the other one.
To enable auto-detection, it is first necessary to synchronize the analysis processes at the time of their creation to ensure that the process associated with the first movement is detected first. For this purpose, the shared class `ProcessWaiting`, described in detail on page 12, is used, followed by the shared class `LegDetected` (page 12), which allows the first process that detects the forward leg pattern to communicate this information to the other process to then select the corresponding analysis algorithm, also allowing the other to apply the appropriate algorithm for its own leg.

Currently, the automatic recognition algorithm requires the detection of small variations in the signals during the initial movement, making it, although functional, not very robust, especially in the case of patients with severe motor deficits who have difficulty reproducing the natural movement of the joints.

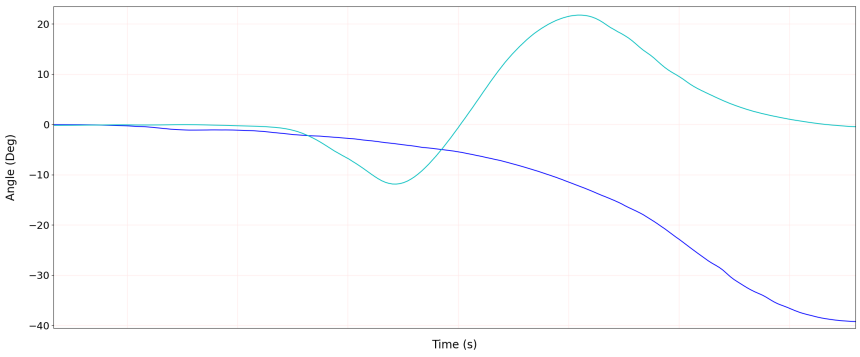

Figure 3.18: Initial Movement Signals of Both Legs - Exercise: Swing

In the graph 3.18, the starting signals of the Swing exercise generated by both legs are visible. The blue signal represents the leg that initiates the movement of the exercise with the first step forward, while the blue signal represents the other leg, which remains behind while weight is distributed forward, thus generating negative angles. As we can see, the blue signal is characterized by a positive peak, as the moving leg produces positive angles. However, it is preceded by a notch that can vary in magnitude. This dip, located in the negative quadrant, corresponds to the knee flexion required to take the first step forward from a static position with feet together. The magnitude of this flexion depends on the motor characteristics of the subject and the execution speed.

The algorithm is designed to analyze both signals in real-time, correctly distinguishing them before the positive peak in the blue signal occurs. This avoids missing any detection during the execution of the analysis algorithms.

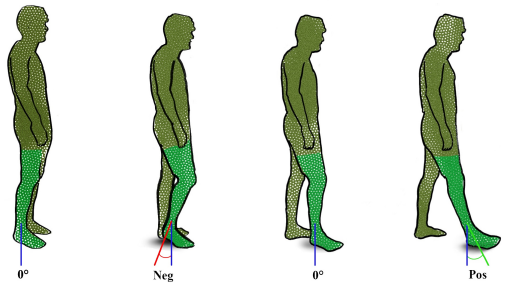

Figure 3.19: Diagram of the Initial Step with Feet Together Start. Illustrated by: Nicola Montagnese

The knee flexion of the initial leg is a much faster movement compared to the others and involves the following phases in rapid succession: a negative zero crossing, followed by a negative minimum, and finally a positive zero crossing. The leg not involved in the initial movement may follow a similar or identical sequence of events, usually with a significantly different timing.

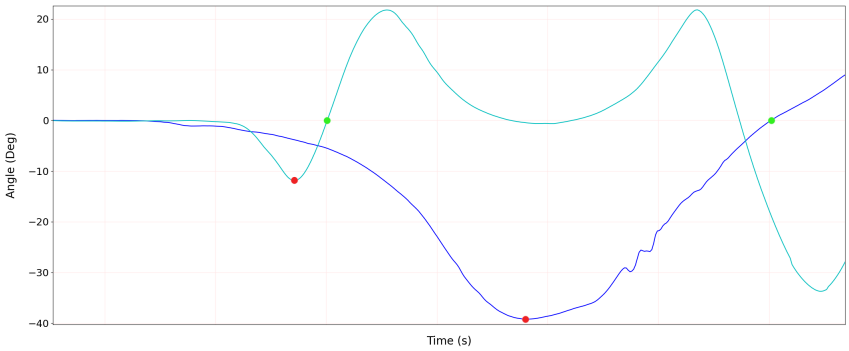

Figure 3.20: Initial Movement Signals of Both Legs with Highlighted Hotspots - Exercise: Swing

In image 3.20, the signals highlight how the starting leg quickly flexes the knee to bring it forward and place the foot. The rear leg, which does not undergo knee flexion, still generates a notch, but much deeper and with slower variations caused by the gradual shifting of the body's weight forward, leading to a positive zero crossing much later than that of the starting leg. The first process to detect a negative minimum followed by a positive zero crossing is therefore assumed to be the one handling the starting leg.

Due to noise and small movements, however, oscillations during the static phase may generate unexpected zero crossings, leading to potential incorrect detections. In image 3.21, we see an example of how signals have multiple zero crossings when they are close to zero continuously.

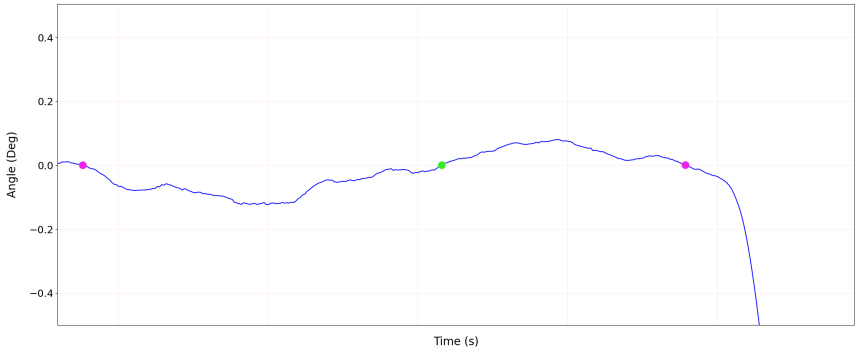

Figure 3.21: Noise in the Initial Static Signal of One Leg with Highlighted Zero Crossings - Exercise: Anterior-Posterior Load Shift in Tandem Position

We observe that, before the occurrence of significant zero crossings (in the figure the last zero crossing represented), there have already been a negative zero crossing and a positive one, separated by a small dip, caused by noise. Without further checks, this could lead to incorrect identification of the signal profile, mistakenly making it appear as if it belongs to the leg that advances first, as all three points of interest have been detected.

To avoid false detections during the initial static phase, a 1D Gaussian filter is applied to each window – using the SciPy library – which attenuates oscillations caused by noise.
After applying the Gaussian filter, each analysis window is shifted using a parameterized displacement according to the exercise. This approach reduces the probability of detecting invalid zero crossings at the beginning, as portions of the signal close to zero, which even after smoothing might cross it erroneously, are shifted below zero.

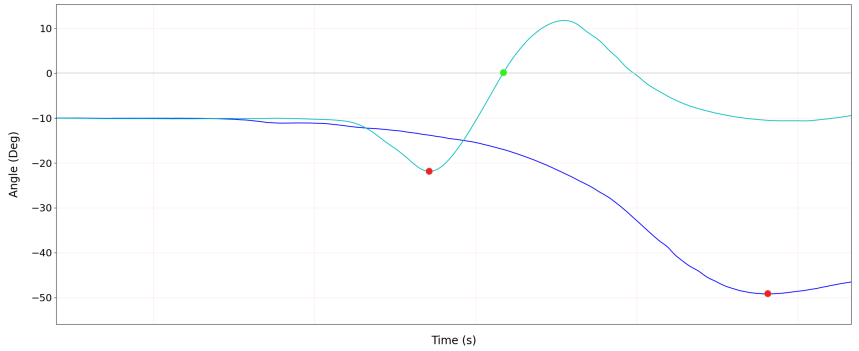

Figure 3.22: Translation of Signals with Highlighted Zero Crossings - Exercise: Swing

The translation also introduces a delay in detecting positive zero crossings. In the context of the swing, this delay is advantageous. The variation of the initial leg is indeed steeper compared to that of the rear leg, so the displacement delays the detection of the positive zero crossing in the rear leg more. This reduces the risk of errors, decreasing the probability of detecting the positive zero crossing of the rear leg first due to possible noise or synchronization errors.

Finally, a positive zero crossing is considered valid only if the signal has recorded a negative minimum slightly lower than the displacement. This criterion ensures that the negative minimum was indeed negative even before translation, maintaining a small margin of error that allows for the exclusion of noisy minima but still considers very small notches, in order to enable the detection of minimal flexions in subjects who might have difficulty performing them naturally.

3.5.1 Gaussian Filter

The Gaussian, or normal distribution, is a probability density function that describes a symmetric distribution around a mean, with a characteristic bell-shaped curve. Mathematically, it is defined as:

$$G(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.6}$$

where:

- x is the independent variable;
- μ is the mean of the distribution, which corresponds to the center of the Gaussian;
- σ is the standard deviation, which controls the width of the Gaussian. Larger values of σ produce a wider and flatter curve, while smaller values produce a narrower and sharper curve.

This function is often used in signal processing to smooth (filter) a signal in order to reduce noise and uniform its trend.
The Gaussian can extend over multiple dimensions; however, when the independent variable x is one-dimensional, as in the case of an array, we obtain a 1D Gaussian that represents a normal distribution along a single dimension.

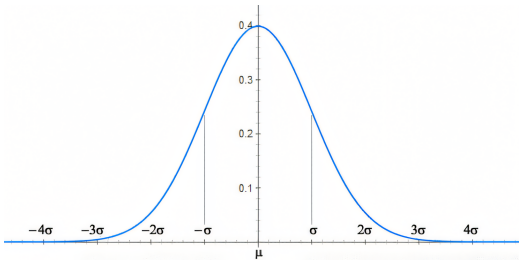

Figure 3.23: Gaussian distribution curve. Source: www.okpedia.it/distribuzione-normale

To apply the Gaussian as a filter to a one-dimensional discrete signal, i.e., a vector of values to be smoothed, `SciPy` constructs a 1D Gaussian kernel. This kernel is a vector (array) of weights, calculated using the Gaussian function evaluated over a range of points in a range around zero determined by the chosen standard deviation σ. The kernel is defined as:

$$K(x_i) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x_i^2}{2\sigma^2}} \tag{3.7}$$

where x_i represents the various points around the center of the kernel (typically centered at $x = 0$) and σ is the standard deviation of the Gaussian.

The sum of all kernel elements is *normalized*[25] to 1 to ensure that applying the filter does not alter the overall amplitude of the original signal.

[25] ***Normalization***: Process of transforming data to bring it within a specific range or to make it comparable on a common scale.

Once the 1D Gaussian kernel is calculated, it is applied to the signal through the discrete convolution operation. Each value in the window is then attenuated based on a linear combination of nearby values, weighted according to the Gaussian kernel. The final result is a smoothed signal, with high-frequency oscillations attenuated.

A larger standard deviation σ produces a wider and flatter kernel, which smooths the signal more intensely as the weights assigned to central values and nearby values are more similar. This makes the smoothing effect more uniform over a wide range of values. Conversely, in a Gaussian with a smaller standard deviation, the central weight is much higher compared to the weights of nearby values. As a result, the filter acts in a more localized manner and the central value is less influenced by surrounding values.

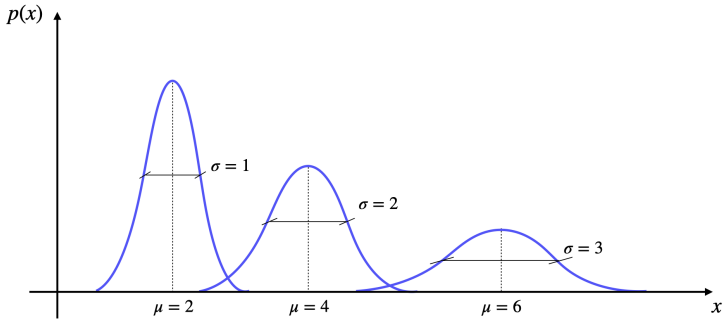

Figure 3.24: Gaussians with varying *sigma*

3.6 Swing

Swing is an exercise in which the legs play different roles. When the execution of *Swing* is initiated, in both analysis processes, the method `detectSwing()` is executed. This method internally determines which algorithm to use for each leg by utilizing the auto-detection provided by the method `detectLeg()` described on page 33, or based on the specifications selected by the user in the GUI.

The movement of this exercise starts with the feet together and the weight distributed on both legs. One leg moves forward, transferring the body's weight onto it, while the other remains fixed. Subsequently, the weight is shifted back onto the fixed leg, allowing the advanced leg to return to the initial position and perform a backward step. The weight transfers to the rear leg and is then brought back to the fixed leg. The rear leg returns to the initial position, repeating the cycle. The fixed leg may slightly bend or lift partially to facilitate the weight transfer.

Below is the graph of the signals produced by the Swing:

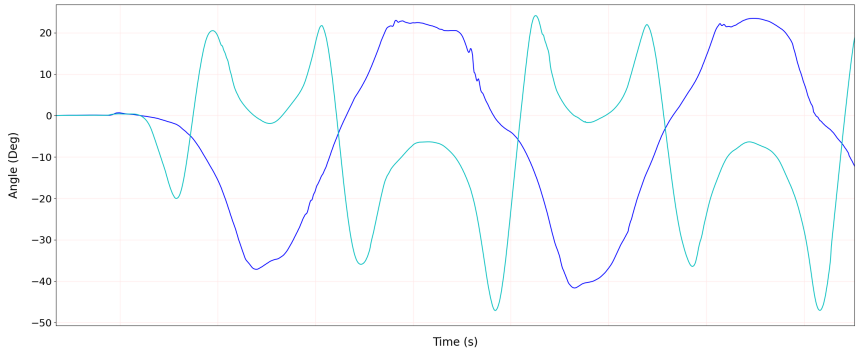

Figure 3.25: Signals from both legs - Exercise: Swing

The signal represented in blue shows a sinusoidal pattern with a rather low gradient (slow variations) produced by the fixed leg, which, when the moving leg advances, creates negative angles, while when it retracts, it creates positive angles.

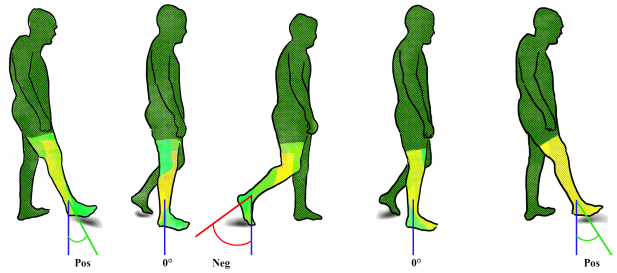

Figure 3.26: Fixed leg, Swing - Illustrated by: Nicola Montagnese

The phase of maximum load on the leg always corresponds to the perpendicular points to the ground, that is, the zero crossings of the signal. Since there is no oscillatory phase detached from the ground for this leg, each zero crossing corresponds to the weight load. Therefore, the system must be able to detect and emit a sound just before each zero crossing.

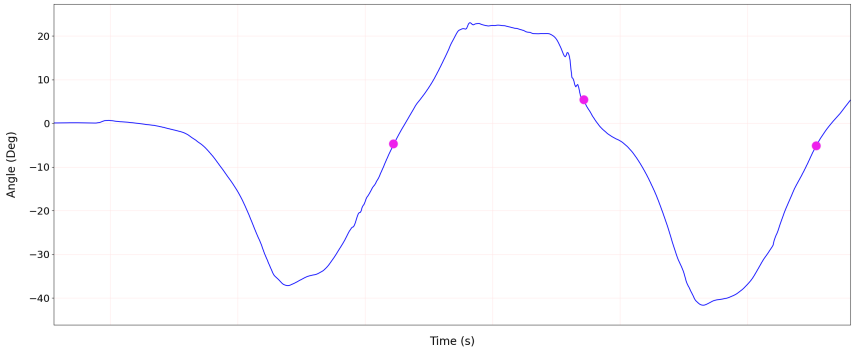

Figure 3.27: Fixed leg with highlighted detections. Exercise: Swing

The blue signal, on the other hand, represents the moving leg, which performs forward and backward steps. During the first forward step, due to knee flexion, a less pronounced negative minimum occurs compared to the others. Shortly after, when the leg reaches its maximum extension, a positive peak is observed. When the weight is transferred to this leg, the angle amplitude decreases, generating a notch, often with a minimum close to zero. When the weight is transferred back and the advanced leg lifts off the ground, it reaches maximum extension again, generating a new positive peak. At this point, it is free to return and the movement continues with a backward step. At the moment of the rear contact, a negative minimum occurs, due to the maximum rear extension. When the weight is transferred to this leg, the angle reduces, generating a negative peak that approaches zero. When the weight is released back onto the fixed leg, the rear leg lifts off the ground and another negative peak is observed due to the maximum extension. The cycle continues with the leg moving forward.

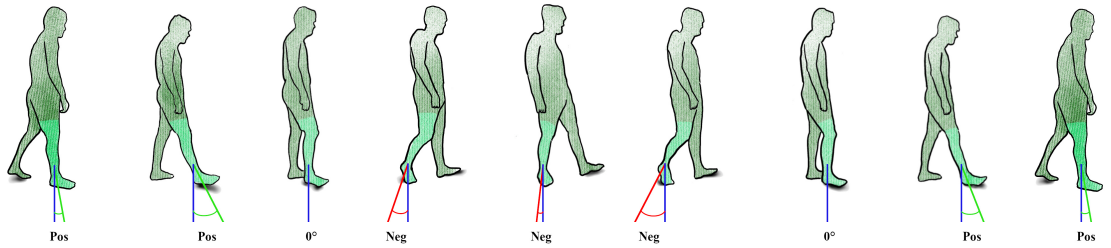

Figure 3.28: Moving leg, Swing - Illustrated by: Nicola Montagnese

Essentially, after the first negative minimum caused by knee flexion at the start, the signal profile shows an alternation of two positive peaks, with a central dip close to zero, followed by two negative minima with a central rise, also close to zero. This pattern repeats cyclically.

The goal is to ensure that the system emits a sound every time the foot's support phase is completed and the weight is therefore shifting onto one leg, that is, when the signal angle tends towards zero, at a point between the initial foot contact and the full load bearing.

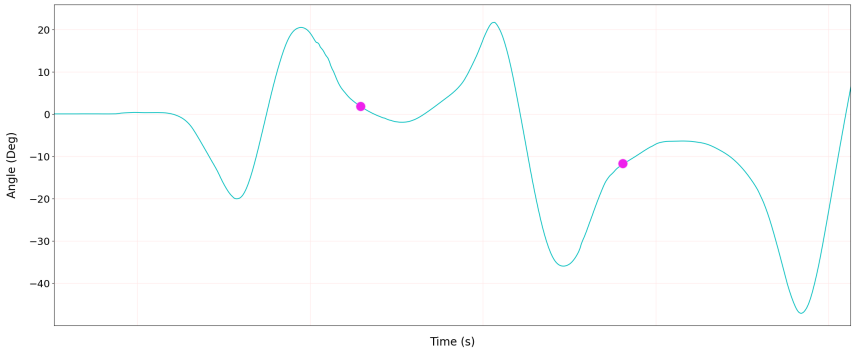

Figure 3.29: Moving leg with highlighted detections. Exercise: Swing

In practice, for the moving leg, we want the system to sound at two specific moments during each cycle:

1. When the moving leg has already reached the first positive peak and the signal is approaching zero, but has not yet reached the minimum. That is, before the positive minimum in the dip between two peaks.

2. When the moving leg has already reached the first negative minimum and the signal is approaching zero, but has not yet reached the peak. That is, before the negative peak in the

rise between two minima.

The minimum between two positive peaks and the peak between two negative minima represent the moments when the weight is maximally ("maximally" and not "completely" because depending on execution accuracy it may not reach zero) transferred to the leg and the movement is about to reverse direction. When working with real-time signal windows, it is very likely that the exact detection of this point occurs after it has already passed; however, the system must sound before reaching the minimum to ensure that the sound is emitted during the loading (support) phase and not during the unloading (lifting) phase.
Furthermore, it is crucial that the rhythm of the sound is synchronized with the detections of the other leg to ensure that the sounds are evenly spaced. To this end, sounding before the signal minimum has proven to be crucial for maintaining regular and consistent spacing.

3.6.1 stepLeg()

The `stepLeg()` method implements the algorithm for analyzing the signal from the moving leg.

The algorithm uses the `peakFinder()` method described on p. 25 to detect all peaks and minima in real-time within the signal windows. A fundamental condition for `peakFinder()` is that the input data windows must be "complete," that is, entirely made up of new data. Therefore, `stepLeg()` checks that the new window does not share any data segment with the previous window. If the windows are not complete, the method waits for more data and correctly queues partial windows.

When a window is complete, it is stored. When a second window is also complete, it is stored. Once three windows are complete, `peakFinder()` is called to compare them and detect any peaks or minima, which are always sought in the penultimate stored window (introducing a fixed time delay of one window to the detections). The first comparison ignores the first window, so any detection within the first 125 ms is ignored. When a new complete window arrives, the oldest one is discarded, as it is no longer needed, and `peakFinder()` is called again to analyze the new triplet.

Peak and Minimum Validation

When a positive peak or a negative minimum is detected, they are considered valid only if they exceed a dynamic threshold. This threshold varies over time and its value is based on the magnitude of the lesser peak or minimum among the last 10 valid detections. The dynamic threshold is intended to be the same as that used during walking and has a validity range below the threshold that allows the threshold to drop as well. Only the magnitudes of valid peaks and minima contribute to the detection history. A 100 ms timeout also prevents consecutive peak detections that are too close together.

As observed in Figures 3.25 and 3.29, due to the signal's shape, we are certain that all peaks during the upward phase in the negative quadrant will be negative. However, during the downward phase in the positive quadrant, the minimum might drop slightly below zero. For this reason, exclusively during the search for positive minima, an upward translation of the signal is applied. Positive minima and negative peaks are considered valid simply when their absolute value is greater than zero, i.e., when they exist. In these cases, the dynamic threshold is not applied, and such values do not contribute to the updating of the threshold itself.

Timing

When a positive peak (or a negative minimum) is detected, corresponding to the first foot contact with the ground, it is desired that the system emits a sound at an intermediate point (corresponding to the complete foot contact), between this and the next positive minimum (or negative peak), corresponding to the weight offloading. However, the time required for the foot to fully contact the ground and the weight to be transferred varies depending on the exercise execution speed, making it impractical to use a fixed delay for the sound after detecting the initial foot contact.

It has been observed that the process of complete foot contact, from the first ground contact, excluding the window delay, is quite rapid and occurs, even for slow movements, within 300 ms (400 ms if the step is performed backward, where the toe contacts first, as toe contact makes balance more stable). The process of weight transfer following complete foot contact is, however, more variable and generally slower.

Assuming the study of the positive quadrant, if the heights of the notches and peaks were constant among themselves, it would be possible to translate the signal so that all intermediate points between the peaks and the minima of the notches intersected the x-axis, detecting them as zero crossing; the same would be possible for the negative quadrant. However, this is not feasible since neither the height of the peaks nor the actual measure of load bearing on the foot performed by the patient is predictable, making the depth of the dip relative to the peak uncertain. A fixed translation of the signal might therefore not capture all points of interest. Additionally, the extreme variability of the dips also makes the application of a dynamic displacement difficult, as it would need to be extremely flexible, losing part of its usefulness.

In fast movements, the time between foot contact with the ground and complete load bearing is very brief. The perceivable time difference between them is almost negligible. It is therefore not necessary to sound at an intermediate point between the initial foot contact and the completion of the load; in this case, complete contact can be perceptually aligned with the point of full load bearing.

In slow movements, the time between the peak and the minimum is greater. Making the application sound during the maximum load bearing detection, due to the window delay, can result in a perceivable delay in sound compared to complete foot contact. After complete foot contact, the weight transfer onto it, moving slowly, may take longer to complete, causing a misalignment between the sound and the actual moment of total contact.

Algorithm

As illustrated in Figure 3.30, after detecting the first positive peak, the algorithm waits 300 ms before emitting the sound. If the movement is fast, the positive minimum is detected before this interval expires, and the system sounds immediately without further delays. Otherwise, the sound is emitted when the timer expires. In this way, for fast movements, the sound is triggered at the positive minimum, very close to the moment of complete foot contact, while for slow movements, it is triggered at the timer expiration, which represents the maximum contact time.

In both cases, once the system has sounded, the search for the second positive peak begins. After detecting it, the algorithm starts looking for the first negative minimum. When this is found, a 400 ms wait is set before emitting the sound again. If a negative peak is detected before this interval expires, the system sounds immediately, breaking the wait. Subsequently, the algorithm continues with the search for a new negative minimum, and once found, the cycle restarts by looking for a positive peak.

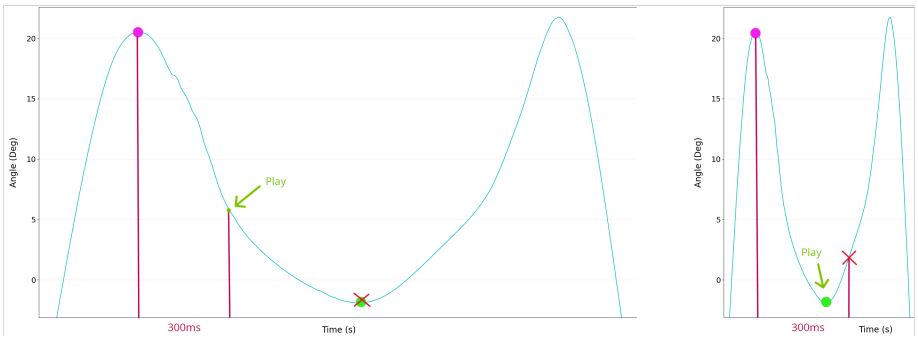

Figure 3.30: On the left, a slow movement; the algorithm sounds at the timer expiration. On the right, a fast movement; the algorithm sounds when detecting the minimum, before the timer expiration.

3.6.2 otherLeg()

The `otherLeg()` method implements the algorithm for analyzing the signal of the fixed leg. This signal, characterized by a sinusoidal pattern, regularly crosses the x-axis, allowing the identification of two zero crossings for each complete oscillation.

As can be observed in Figure 3.26, when the body moves forward, the fixed leg lags behind, creating a negative angle. At this stage, the weight shifts to the advancing leg, allowing the fixed leg to lift. At the moment of recontact, the weight returns to the fixed leg and the body moves backward, causing it to cross zero and making the angle positive. With the weight on the rear leg, the fixed leg can lift again. When it recontacts, the body starts moving forward again, transferring the weight and making the angle negative once more.

Therefore, unlike the walking signal, where only negative zero crossings follow the moment when the foot contacts the ground and positive zero crossings occur during the swing phase of the leg (as illustrated in Figure 3.8), in the case of the fixed leg signal during the `Swing` exercise, both positive and negative zero crossings correspond to the weight unloading after the full contact of the foot and must be detected and reported.

To ensure that the detection of each zero crossing actually corresponds to the moment of initial foot contact and not to the subsequent perpendicularity of the signal relative to the ground, it is necessary to anticipate the detection by spatially translating the signal, similarly to what was done for the gait analysis, but by dividing the algorithm into two alternating phases: the search for positive zero crossings and that for negative zero crossings.

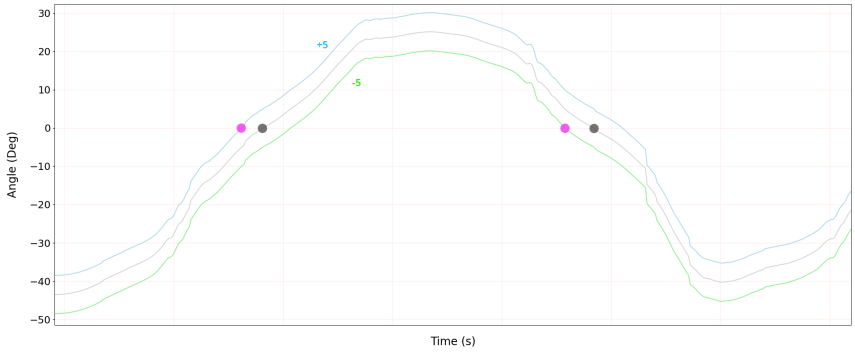

Figure 3.31: In blue the translation to anticipate positive zero crossings, in green that to anticipate negative ones, in gray the original signal.

As we can observe in Figure 3.31, during the phase of detecting negative zero crossings, the signal is translated downward to anticipate detection. Conversely, in the phase of searching for positive zero crossings, the signal is translated upward, as a downward shift would cause a delay rather than an anticipation of detection.

Zero Crossing Validation

In walking, and other exercises, the detection of a zero crossing is compared with the height of the previously recorded peak against a threshold. The algorithm implemented in `otherLeg()` however, although a peak threshold remains an implementable and interesting solution, focuses on evaluating the zero crossings themselves through a dynamic threshold on the gradient and a timeout between consecutive detections.

Timeout

The adoption of a dynamic threshold based on peak height could be useful to avoid validating zero crossings that are too close together due to noise. In this context, a zero crossing is validated only if preceded by a peak high enough not to be considered noise around a previous zero crossing. However, since regular detections of consecutive zero crossings are sufficiently spaced in time (even during exercises performed rapidly), it was sufficient to apply a time-based timeout to prevent multiple detections close to a valid zero crossing. This timeout allows the signal to move away from zero enough to reduce the probability that noisy oscillations are large enough to cross it. Once the signal moves away from this "critical" area, it is unlikely that unwanted zero crossings will occur, as this would require impulsive noise with rather distant outliers that are removed by Xsens signal pre-processing and cleaning algorithms. Nevertheless, if it were to happen, the gradient of the zero crossing caused by an outlier would be so steep that it would not exceed the validation threshold on the gradient.

Gradient Threshold

The gradient threshold is a dynamic threshold designed to avoid detecting sudden zero crossings or those caused by knee bending. When an exercise is not performed correctly, knee flexions can alter the signal: if the leg is at a positive angle and the foot lifts off the ground without keeping the knee extended, a flexion may occur that rapidly reduces angular values, crossing zero. When the leg returns to extend to recontact the foot, the signal crosses zero again and the angle quickly becomes positive. These additional crossings, represented in gray in Figure 3.32, which do not depend on weight unloading on the ground, are undesirable and must be excluded from validation.

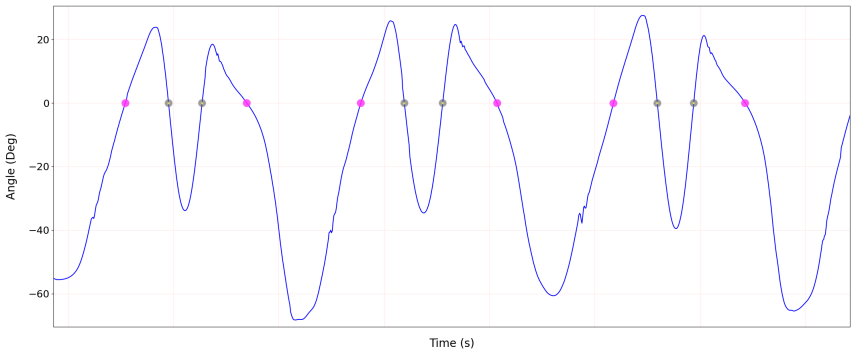

Figure 3.32: Signal pattern of the fixed leg with pronounced knee flexions. In pink the zero crossings (not anticipated for simplicity) to be validated, in gray the additional ones to be excluded.

Since the flexion is very rapid, it causes a change in the signal that is steeper compared to the gradients of the target zero crossings. To exclude the additional zero crossings, the absolute value of the gradients of each crossing (since we consider both positive and negative zero crossings) is compared with a dynamic threshold: values that exceed this threshold are not validated as they are too steep.

A gradient, and thus a zero crossing, is valid when it is below the threshold; however, there is a small validity range above it to allow it to rise over time. The threshold is updated at each valid detection via the `setNewGradientThreshold()` method – described on p. 24 – which implements the Exponential Moving Average (normally with alpha 0.85), capable of dynamically adapting to signal variations and balancing the weight of the old threshold with the new gradients.

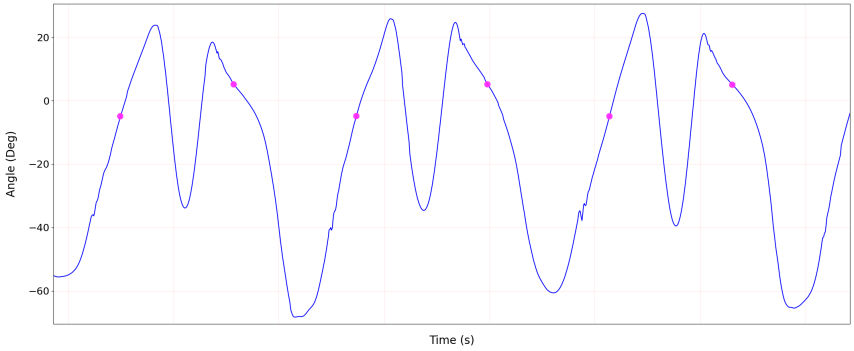

Figure 3.33: Signal pattern of the fixed leg with pronounced knee flexions.
Validation results of the detections.

3.7 Anterior-Posterior Load Shifting in Tandem Position

Anterior-posterior load shifting in a tandem position involves leg movements that generate different signals, similar to what happens during the Swing. However, since the signals are very similar, two versions of the same algorithm are executed for the analysis of both legs. The method `detectTandem()` determines which version of `tandemFunction()` to apply to each leg.

The movement starts with feet together and involves positioning the legs in a tandem position, that is, one leg in front of the other in a straight line, which will not swap positions throughout the exercise. The body weight will be repeatedly transferred to the front leg and then to the back leg, and so on. At each weight shift to a leg, the system is expected to beep.

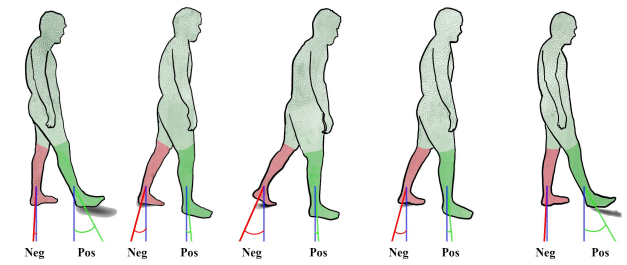

Figure 3.34: Step cycle – Anterior-posterior load shifting in tandem position.
Illustrated by: Nicola Montagnese.

During the anterior-posterior weight shifting on the tandem legs, a sinusoidal oscillatory pattern is generated in both signals. The main difference is that the rear leg will almost always oscillate only in the negative quadrant, while the front leg will predominantly oscillate in the positive quadrant.

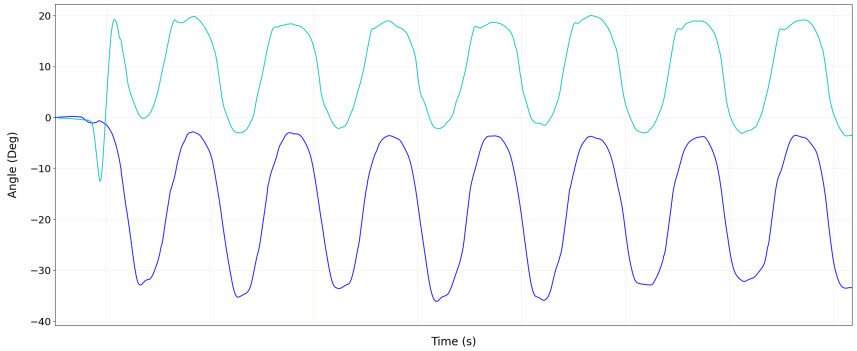

Figure 3.35: Signals from both legs produced by anterior-posterior load shifting in a tandem position.

In both signals, the maximum load point on the corresponding leg is represented by values closest to zero: for the front leg, with angles predominantly in the positive quadrant, approaching zero means decreasing, so the maximum load points are the minima; for the rear leg, which always stays in the negative quadrant, approaching zero means increasing, so the maximum load points are the peaks.
The front leg (blue), when bearing the maximum weight, can cross zero because the body can advance significantly beyond perpendicularity without losing balance. In contrast, the rear leg (dark blue) is very unlikely to reach or exceed zero, as excessive backward movement of the body would compromise balance.

To detect the foot contact, and thus the start of the weight load, it is necessary to identify anticipated points of interest compared to the maximum load point represented by peaks and minima; in other words, for each cycle, it is necessary to detect when both signals approach zero, but before they are maximally close to it. To do this, once the legs are distinguished, both signals are translated to intersect the x-axis: for the front leg, a downward translation is applied, and only the negative zero crossings are detected, thus anticipating the minima, while for the rear leg, an upward translation is applied, and only the positive zero crossings are detected, thus anticipating the maximum peaks.

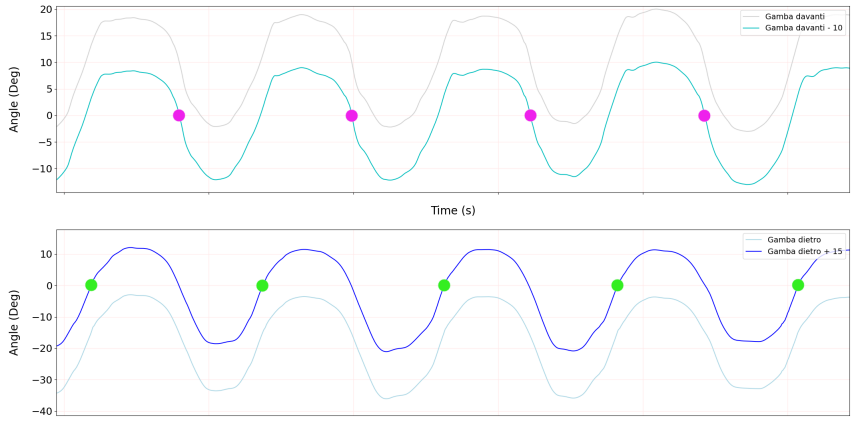

Figure 3.36: Signal translation and zero crossings – Anterior-posterior load shifting in tandem position.

For normal or wide movements, a translation of 5 degrees larger in absolute value is typically applied

to the negative signal compared to the positive one. This is because, while the positive signal often crosses zero slightly natively, the negative one tends to slightly exceed -5 at its maximum peaks and needs a larger translation. We can see from the graphs in Figure 3.36 how this translation difference, besides allowing zero crossings to be detected in both signals, provides fairly uniform time steps between the alternating detections of the two legs, and therefore between the sounds produced.
When movements generate smaller oscillations, the distance from zero becomes more similar for both signals, and the translation factor, determined by the sensitivity level chosen in the GUI, is applied equally to both.

Zero Crossing Validation

We observe that the sinusoidal pattern, for both legs, is similar to the signal produced by the fixed leg during the Swing exercise. Therefore, the issue related to the anterior leg's knee flexion also manifests here, creating, for incorrect execution of the exercise, deep dips that, crossing zero, produce undesirable negative zero crossings that should not be considered valid.

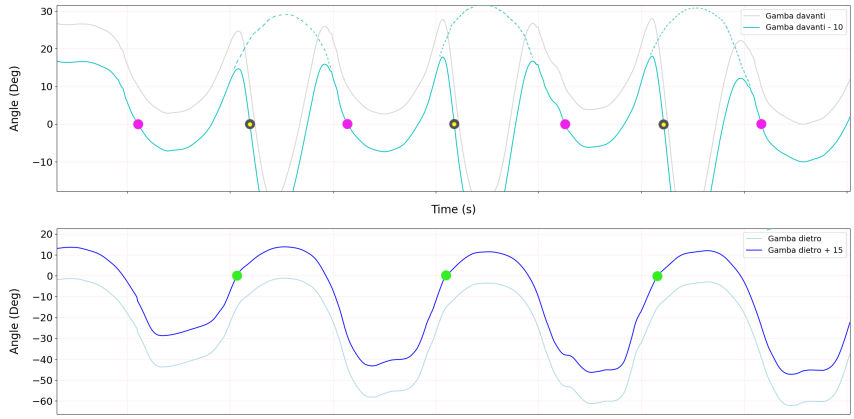

Figure 3.37: Signal translation and additional erroneous zero crossings due to knee flexion – Anterior-posterior load shifting in tandem position.

Given this issue, the same zero crossing validation mechanism used for the fixed leg in the Swing exercise is applied, which includes:

1. Waiting Time: A sufficiently long waiting time is set to avoid detecting zero crossings too close together in the same signal, thus excluding those caused by noisy oscillations.

2. Dynamic Gradient Threshold: A dynamic threshold on the gradient allows distinguishing zero crossings caused by knee flexion from valid ones. Undesired zero crossings are easily distinguishable because they have higher gradients, due to the speed of flexion, that exceed the dynamically calculated threshold on the gradients of valid zero crossings using the method `setNewGradientThreshold()` – described on page 24 – implemented with Exponential Moving Average.

Bibliography

[1] Xsens Technologies B.V. *MTw Awinda User Manual*. Xsens Technologies B.V., Pantheon 6a, P.O. Box 559, 7500 AN Enschede, The Netherlands, 2018. Document MW0502P, Revision L, 3 May 2018.

[2] Pi Productora. Perché python è stato scritto con il GIL?, 3 Ottobre 2020. https://piproductora.com/it/perche-python-e-stato-scritto-con-il-gil/.

[3] A. Bertoni and G. Grossi. Dispensa del corso di elaborazione numerica dei segnali. Università degli Studi di Milano, 2009-2010.

[4] G. Esposito. Gait detection and sonification based on angular rate sensors, 6 Febbraio 2024. Documentation of Step detection and sonification from gyroscope sensor data by SonicWalk.