SonicWalk

Technical Documentation

Roberto Tallarini, UNIMI, 2024

Overview

SonicWalk is a project aimed at extending the use of the Python interface SonicWalk developed by Gabriele (available on GitHub). This interface allows for the acquisition, analysis, and visualization of gyroscopic data produced by motion tracking sensors, with a particular focus on step detection and sonification applied in post-stroke physiotherapy and for patients with Parkinson's disease. Audio feedback, useful for the patient, is produced immediately after each step taken. The application enables the acquisition, recording, and simultaneous visualization of movement data, triggering customized audio samples sequentially (in a circular mode) during the subject's movement.

The primary objective of the expansion was to broaden the available analysis methods to include various types of exercises beyond walking, such as marching in place, swing, and the "double step." Secondly, the aim was to improve usability by first porting the interface from Linux to Windows, and subsequently making SonicWalk a functional graphical application for healthcare personnel, allowing automated management of exercise recordings and their local storage related to the patient.

Through this GUI (created with Python PyQt5 technology), it is possible to:

- Add, search, select, modify, and delete patients.
- Add, select, and remove different audio sources.
- Select various exercises and adjust additional parameters such as sensitivity and mode.

The exercise modes can be divided into three distinct phases: a phase to estimate the patient's average beats per minute of steps, a phase of playback music reproduction at the estimated beats per minute, where the patient tries to follow the rhythm, and a real-time feedback phase.

It is therefore possible to:

- Record exercises, view them in real-time, comment on them, and save them in daily sessions related to the patient.
- Access the local archive to view, analyze, export, and delete saved recordings.
- Clone the data archive (including the JSON files detailing its contents) to another folder and use an additional provided Python script to view the cloned data.
- Completely clear the archive when necessary.

This expansion has transformed SonicWalk into a functional healthcare application that leverages the foundations of communication with sensors, multiprocess management, and gait analysis implemented in the Python SonicWalk interface, while also incorporating significant improvements to the interface itself.

To view the user guide for SonicWalk, click [Here]

Sensors

The Xsens MTw Awinda motion sensors are used to acquire movement data. The setup consists of:

- 2x Xsens MTw Awinda motion trackers
- 1x USB Master stick
- 2x body straps

The two motion trackers are positioned on the Velcro straps wrapped around the subject's ankles. The sensors are equipped with a 3-axis gyroscope and a magnetometer to detect orientation. The data is produced at a maximum sampling rate of 120 Hz and transmitted wirelessly to the master device (USB stick) with a maximum latency of 30 ms.

The packets produced by the motion trackers are sent to the USB Master stick using the proprietary Awinda wireless protocol, with a guaranteed maximum communication range of 10 meters (in an office environment). The orientation information contained in each packet is described through the three Euler angles (Roll, Pitch, Yaw).

Before starting the recording, while the sensors are attached to the subject, an orientation reset is performed. The reference system of the motion trackers is transformed into a new reference system, which is that of the object to which they are attached.

General Operation and Interface Modifications

In the application's GUI, starting the recording corresponds to the initiation of a separate thread from the main thread to establish communication with the sensors. This thread accesses the data produced by the two sensors through a simplified Python interface that relies on the Xsens device API.

MTw Class

The MTw class implements the configuration of the master device, the detection and association of the motion trackers, data acquisition, recording, and proper resource closure. Originally, the class exposed a single public method, **`mtwRecord()`**, as the sole entry point to the application. Now, the **`stopRecording()`** method has been added to allow manual interruption of the recording by the GUI before the preset exercise time expires.

Recording

To initiate the registration procedure, the secondary thread instantiates the mtw.Awinda object, which handles sensor setup. mtw.Awinda establishes connection with the sensors during the initial phase and will throw exceptions in case of errors, which are appropriately handled by the GUI. Subsequently, it starts the recording by calling the **mtwRecord()** method, which returns:

- 2 arrays (one for each signal) of data
- 2 indices indicating the end of significant data in the data buffers of the 2 signals.

Additionally, the following have been introduced:

- 2 arrays of indices related to the 2 signals indicating estimated points of interest where measurements have been taken.
- The average BPM (beats per minute) value.

Parameters Changes

This method has been modified from its original interface to accept multiple input parameters. Originally, the parameters were only:

- plot
- analyze

These two flags specified whether the data should be plotted and/or analyzed. The plotter and analyzer were started as daemon child processes and could read data as it was received from shared memory. Data was placed into shared memory by the MTw object after extracting useful angular information.

However, the plot parameter is now always set to False because plotting is managed by the GUI, and analyze is always set to True. The new parameters allow defining:

- The duration of the recording.
- The path to the music samples.
- The type and mode of exercise.
- The ability to auto-distinguish legs or specify which leg the patient starts with.
- The option to enable or disable audio playback.
- The option to estimate average BPM or not.
- The ability to share reference to a preallocated shared object for use as a data buffer or instantiate an internal shared object.

BPM estimation and hot points

The leg signal analysis is conducted by the Analyzer class. When points of interest, such as pitch or zero crossings in the signal, are identified, several actions occur:

1. The initial index of the window where the point of interest was found is added to a shared index array. This index approximates the position of the point of interest in the signal. Using the window index is preferred over the sample index because each window corresponds to approximately 125 ms, which is a negligible interval for signals of this type. This short duration ensures that no significant variations in signals produced by human legs, even during walking, are missed.

2. The timestamp of the detection is added to a shared array. MTw will be able to examine the two arrays (one for each signal), combine them, sort the timestamps, and calculate the time distance between each of them.

3. Using the Z-score method with a threshold of 3, outliers are removed. This statistical method helps in identifying and removing values that deviate significantly from the average, indicating potentially erroneous data points.

4. Finally, the average BPM is calculated based on the remaining values.

Analyzer Class

The Analyzer class implements various step detection algorithms specific to different exercises and triggers audio samples from a specified sample library. The class is instantiated twice and launched in two separate daemon processes, synchronized at startup via a semaphore in shared memory, each analyzing the signal produced by one leg.

Several shared values and variables are passed to the analysis processes, including buffers, indices, other arrays, the chosen exercise number, sensitivity level, selected leg, etc.

Data buffers are managed externally by the MTW class in a circular mode, as are the indices.

Each Analyzer process cyclically checks every 3 ms if the termination condition is false (when it finds an angle of 1000 degrees in the buffer). If **True**, it takes the last samples of size **winsize** from the buffer in shared memory; if a complete new window is not available, a window of values is composed by queuing new values into a portion of those previous.

Below are the algorithms of the most relevant utility and analysis methods

Most Important Utility Methods:

- The **`zeroCrossingDetector`** method is designed to detect zero crossings within a window of data. This method is crucial for identifying points of interest in signals, such as those used to detect steps during movement.
  The function examines a sequence of data to find points where the signal crosses from positive to negative or from negative to positive. If multiple zero crossings are detected within the data window, they are considered noise and ignored; only a single crossing is considered. Once a zero crossing is identified, the function calculates the derivative (gradient) at the crossing point. If a maximum gradient threshold is specified, the function checks that the calculated gradient does not exceed this threshold. If no threshold is specified or if the gradient is below the threshold, the crossing is considered valid. The function then verifies that the polarity of the zero crossing matches the specified polarity (positive or negative). If all conditions are met, the function confirms that a zero crossing has been detected and returns relevant information such as the gradient of the crossing and its polarity.

- The **`_setNewGradientThreshold`** method is designed to update the maximum gradient threshold based on the current gradient and the previous threshold. This method utilizes Exponential Moving Average (EMA) with a specificated alpha value to dynamically adapt the threshold so that it can respond to changes in the data. Additionally, you can specify a lower threshold limit for the new threshold value to prevent it from becoming too low.

- The **`phaseAnalyzer`** method is designed to analyze the phase of a system based on two successive data windows. This method determines whether the system is in a growth or decay phase by comparing the maximum or minimum values of the windows. The method returns **True** if the specified phase (growth or decay) is detected, otherwise it returns **False**. 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 is less than or equal to the minimum value of the previous window, the method identifies a decay phase. This approach allows for the identification of growth or decay phases between two data windows, reducing the impact of noise on the results because

it is based on the maximum and minimum values of the windows rather than individual samples that may be influenced by noise.

- The **`peakFinder`** method of the Analyzer class is designed to determine if there exists a local positive or negative peak (or trough) within a data window. The method returns **True** if the specified type of peak (or trough) is found in the window, otherwise it returns **False**.

    - Finding a Local Peak:
      The method compares the maximum value of the current window (`window`) with the maximum values of the previous window (`previous_window`) and the next window (`current_window`). If the maximum value of `window` is greater than the maximum values of both the other windows, a local peak is identified. Subsequently, it verifies whether this peak is above or below zero.

    - Finding a Local Minimum:
      The method compares the minimum value of the current window (`window`) with the minimum values of the previous window (`previous_window`) and the next window (`current_window`). If the minimum value of `window` is less than the minimum values of both the other windows, a local trough is identified. Subsequently, it verifies whether this trough is above or below zero.

  This approach allows for identifying significant local peaks or troughs by comparing the maximums or minimums of three data windows, thereby reducing the influence of noise. Directly comparing three individual samples instead of windows could lead to incorrect identifications of peaks and troughs due to erroneous fluctuations caused by noise.

  To further enhance the method, applying a light Gaussian filter to the windows can smooth the data further and reduce noise, thereby improving peak detection accuracy. However, applying a too aggressive Gaussian filter, as it is applied to each signal window separately, could introduce artifacts and falsely detect additional peaks.

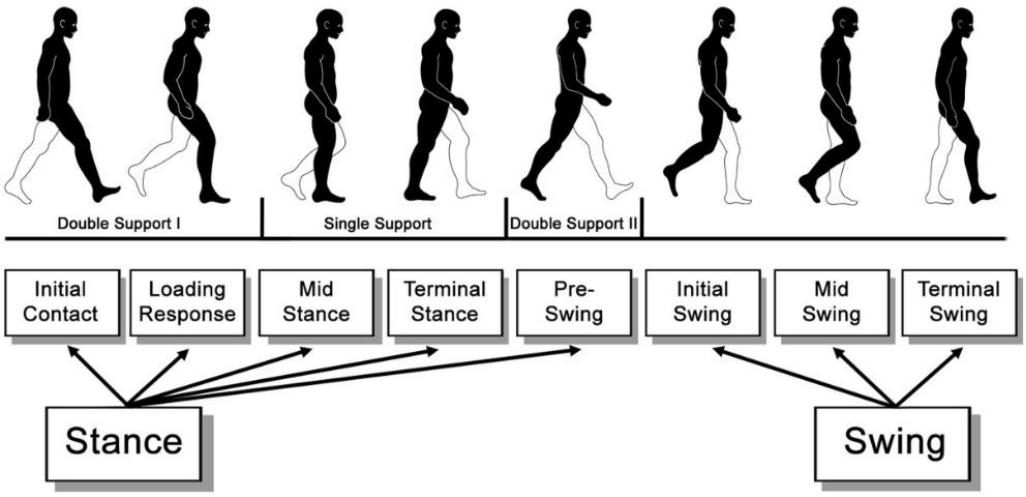
  One drawback of this algorithm is that it always looks for a potential peak in the middle window. However, the middle window chronologically represents the second-to-last arrived, meaning that with each new window, the algorithm is one window behind in peak detection. Nevertheless, each window completes in about 125 ms, making the delay in detection negligible and imperceptible.

  To use this method effectively, it's essential to ensure that the three passed windows do not overlap in values. Essentially, the method can only be called when there are three "complete" windows available.

Analysis Methods:

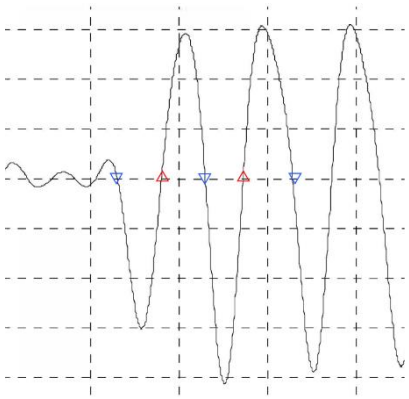
Walk *"__detectStep()"*

The gyroscope signal generated by the motion trackers placed on the ankles exhibits a sinusoidal behavior. The gait cycle is measured from the initial contact of a heel to the subsequent contact of the same heel with the ground.



A step can be characterized by two zero crossings with alternating polarities, detectable successively, each time the shin returns perpendicular to the ground.

The figure depicts a signal with two labeled cycles where the area between two downward-facing triangles represents a step.

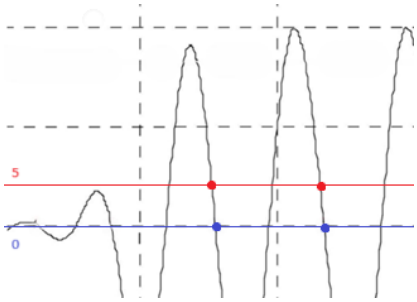


A step is detected as follows:

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

The audio sample is played concurrently with the validation of a step (end of the step), specifically during the second zero crossing (negative zero crossing). Negative zero crossings occur shortly after the heel contacts the ground, when the leg is perpendicular to the ground again.

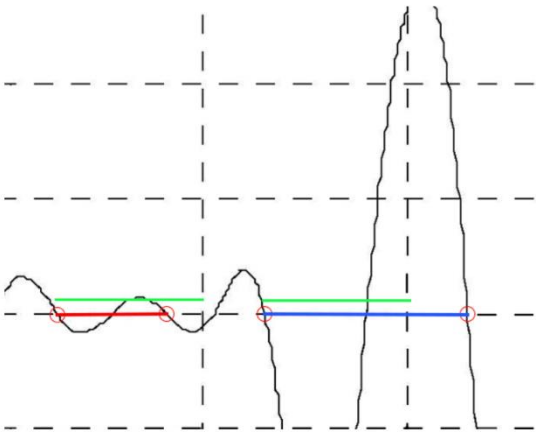
To play the samples concurrently with the heel strike (anticipated before the shin becomes perpendicular to the ground), you can anticipate the detection by shifting the entire signal down by 5 degrees. This ensures that the zero crossing is detected earlier and the system plays the sound correctly. This strategy is feasible because the valid positive peaks of a step ideally exceed (by several degrees) the 5-degree shift, allowing for translation without losing the zero crossings.



The function **`stepDetector()`** analyzes the content of a small buffer containing the last 15 readings from the motion tracker. It utilizes the **`zeroCrossingDetector()`** method to search for any zero crossings with the specified polarity within this window and returns True if found.

- In each window, at most one zero crossing is considered valid by **`zeroCrossingDetector()`** because in such a small window (approximately 125 ms), polarity changes more frequent than once can be considered noise.

- A gyroscope signal can cross zero with a negative gradient more than once during the period from initial contact to load response, which corresponds to 0-10% of the step cycle. A timeout method is used to prevent false detections. The timeout is set to 100 ms, corresponding to 7% of the average fast walk of 1.5 steps per second (per single leg). During the timeout period, all detected zero crossings are rejected, and the timeout is reset after the first valid zero crossing with a negative gradient is detected.

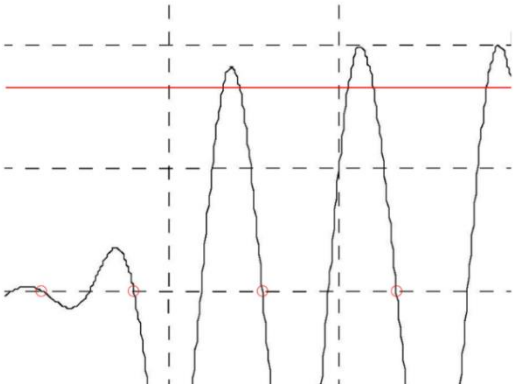


The waveform in the figure shows an example of step rejection. The first cycle marked with the red segment represents a step that will be rejected due to its shorter period compared to the threshold (indicated by the green segment). The cycle marked by the blue line will be considered valid.

To validate steps, an adaptive peak threshold is used to avoid false positives caused by instantaneous movements and small device fluctuations. A step is considered valid only if the positive peak detected before the negative gradient zero crossing exceeds the threshold.

In this case, real-time detection of the peak isn't necessary. Instead, the highest value recorded since the last valid zero crossing is stored in memory. When another zero crossing occurs, the peak between the two crossings is identified as the highest value, which can then be stored in an array and reset for the next detection.

The initial threshold is set at 2.0 degrees to avoid detections during periods of inactivity, and it is gradually updated during walking. During operation, this threshold is updated by calculating the minimum positive peak of the last 10 steps. Negative zero crossings preceded by a peak below this threshold are ignored. As shown in the figure, the first negative zero crossing is preceded by a peak below the threshold (indicated by the red line) and will be rejected, while the following two steps will be correctly detected because both have peaks above the validation threshold.



To allow the threshold to gradually adapt to the subject's walking pace, steps with a positive peak up to 3.0 degrees below the threshold are still considered valid. This mechanism allows the threshold to decrease slowly over time, at a maximum rate of 3 degrees at a time, while it increases much more rapidly.

This approach ensures that when the threshold is low, both small and large steps can be detected. However, as the threshold increases significantly, it becomes less likely to detect small steps immediately because the high threshold prevents false positives. On the other hand, maintaining the threshold too low would increase the risk of false detections, which is why the threshold is dynamic rather than static.

Utilizing gyroscopes for step detection offers several advantages, primarily portability and non-invasiveness, coupled with reliable performance in detecting steps at lower speeds. In a study involving 30 healthy subjects, tests were conducted for 1-minute intervals at various speeds on flat terrain, with subjects walking back and forth within 2-5 meters from the device. Subjects were allowed to freely vary their walking speeds, and the average speed was calculated in steps per minute. The results were categorized into three speed ranges: slow, medium, and fast.

The algorithm demonstrated performance exceeding 93%. However, the main errors arose from steps with positive peaks below the threshold, often due to very short steps or rotations in place. Increasing the walking speed led to more frequent rotations in place, negatively affecting algorithm performance. This was particularly evident when the leg movement angle did not change polarity, such as during stationary oscillations or walking in place without forward movement.

Table 1: Test results

Speeds	Actual steps	Detected steps	Accuracy
≤ 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%
≥ 80 steps/min	92	90	97.8%
	95	89	93.6%
	95	91	95.7%
	92	90	97.8%
	90	87	96.6%

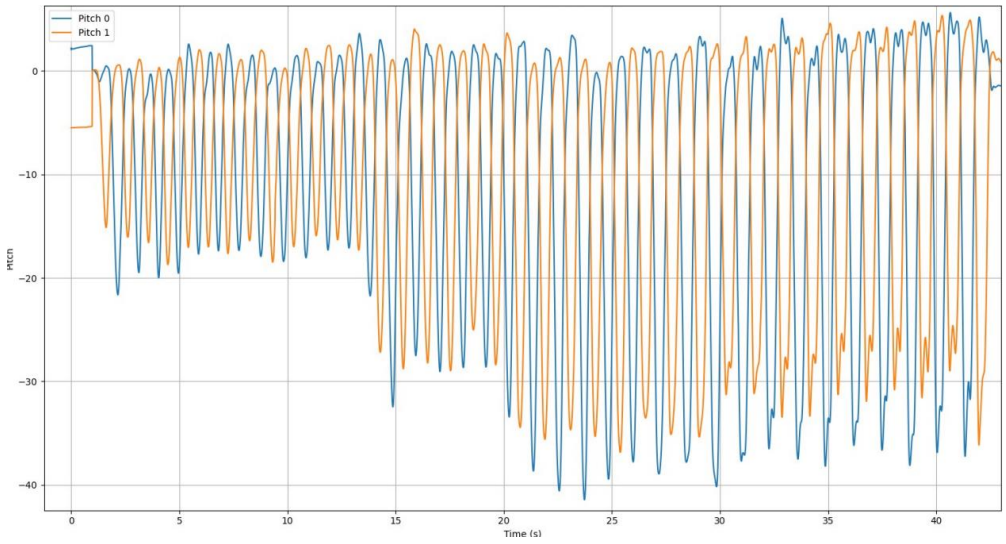
In this analysis method, the following values were parameterized in a JSON format to adjust the levels of sensitivity and precision of the algorithm, where sensitivity and precision are inversely proportional:

- Displacement (signal translation)
- Valid range below the threshold
- Minimum threshold value
- Wait time between 2 detections

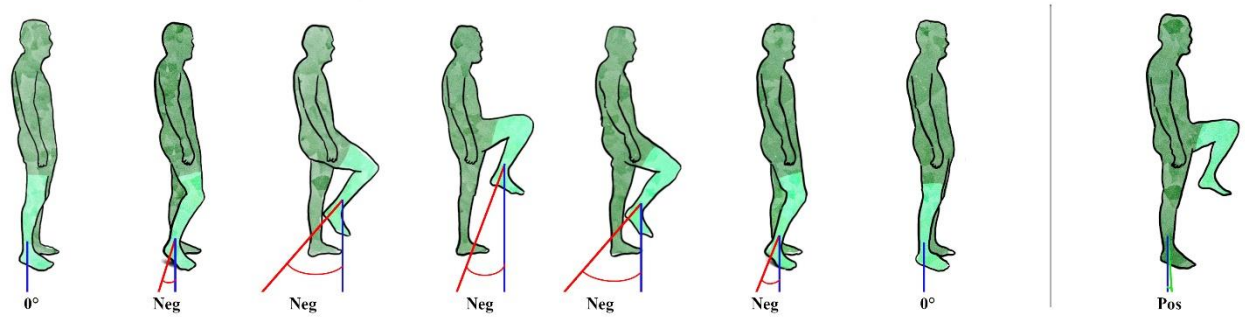
March in place *"__detectMarch"*

The algorithm for detecting stationary marching is quite flexible and relies on detecting steps during walking; however, characterizing this type of signal allows for a simplified logic in validating detections.

Stationary marching produces a sinusoidal signal that primarily extends into the negative quadrant. As shown in the graph below, during exercise performed by a healthy individual at varying speeds, positive peaks are low while negative troughs are much more pronounced.



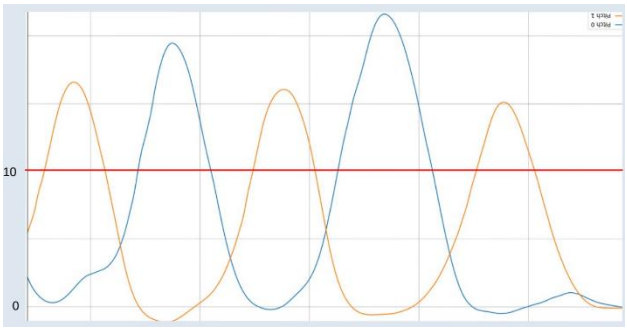
In knee lift during stationary marching, the ankle moves more compared to regular walking, producing deep negative troughs. When the leg returns to touch the ground, a positive zero crossing occurs instead of a negative one as in walking.



To validate steps by detecting zero crossings and peaks similar to walking, it was found that it is better to work with high peaks far from zero rather than low peaks that can be mistaken for noise.

To align the signal with the walking algorithm, simply invert the polarity of each sample of the signal relative to the time axis, thus obtaining high positive peaks and negative zero crossings of interest.

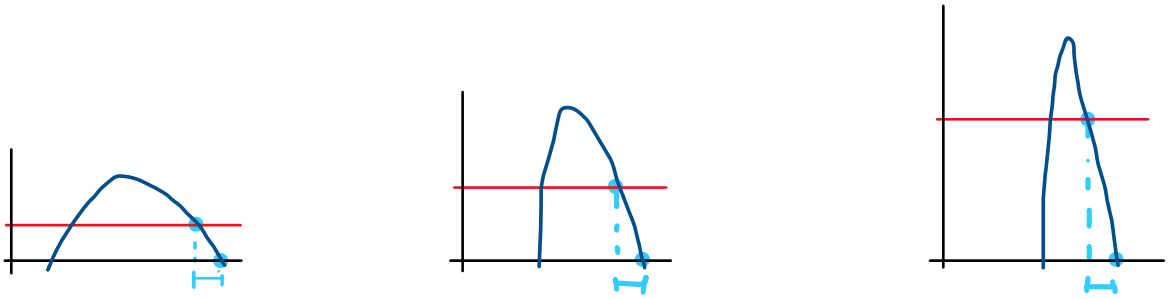
Using a variant of the walking algorithm, negative zero crossings preceded by valid peaks are always sought. However, since the peaks are very high, they are easily distinguishable from noise and impulsive movements through a fixed threshold set by default to 10 degrees (instead of using a variable one), which excludes all noise and includes all peaks. A peak greater than 10 degrees is considered valid, and the step is validated at the subsequent negative zero crossing, triggering the system sound.



The threshold is implemented as a signal shift rather than a simple threshold, allowing for the anticipation of zero crossings. The threshold is parameterized in a JSON and can vary based on the selected sensitivity.

Under standard conditions, slower exercises result in lower peak heights. Increasing system sensitivity reduces displacement and the spatial anticipation.. However, with lower peaks and gentler gradients, less spatial anticipation is needed to achieve the same temporal anticipation. Conversely, during fast execution, higher peaks, reduced sensitivity to small movements, increased gradients, and greater spatial anticipation are required to achieve the same temporal anticipation.

Moreover, we can better distinguish the timing of a sound related to slower actions compared to faster actions. Therefore, any excessive anticipation in an exercise with very high peaks and thus very fast movements will be negligible.



Automatic Leg Detection “detectLeg()”

Synchronization of Analysis Processes

The goal was to develop a method to detect leg movements and differentiate the algorithm to use for exercises where the legs perform different movements. To achieve this, a shared class called **`ProcessWaiting`** was implemented to synchronize the leg processes at startup. This class ensures that both leg processes are started and synchronized with a delay of less than 0.0001 seconds.

The **`ProcessWaiting`** class initializes two locks to ensure thread safety in the context of multiprocessing. Additionally, it sets up two events that will be triggered by the two analysis processes. Both processes interact with the `start` method, which operates as follows:

- **Calling the Start Method:** When one of the processes calls start, it checks the state of the first event.
- **Activation of the First Event:** If the first event is not active (meaning it hasn't been triggered yet), it activates this event.
- **Activation of the Second Event:** If the first event is already active (meaning the first process has already started and called start), then the second event is activated.
- **Resetting Signals and Starting Analysis:** Once both events have been activated, indicating that both processes have called start, the analysis processes can begin, and the events are reset for future synchronization.

Auto-detection with shared variable between processes

If the analysis begins with leg auto-detection, both processes initiate the **`detectLeg`**`()` method. This method utilizes another shared class, **`LegDetected`**, which uses locks to ensure consistency and prevent deadlocks.

The underlying idea of the method is that typically, starting from a standstill, the leg that moves forward first is detected as the leading leg. This convention allows distinguishing the initial movement signal of that leg.

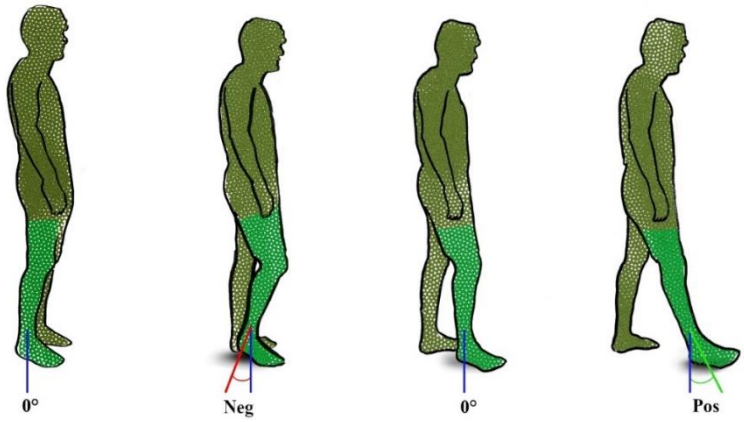
During synchronized analysis, the first process to detect this movement communicates to the shared object that it has detected the leading leg and subsequently selects the analysis algorithm for that leg. The other process, upon noticing the variable change in the shared object, understands that the leg has been detected and uses the algorithm corresponding to the other leg.

This method makes the sensors independent of the leg, but recognition requires detecting slight variations in the signal during the initial movement, making the method less robust, especially in cases of patients with motor deficits who struggle to replicate natural joint movements.

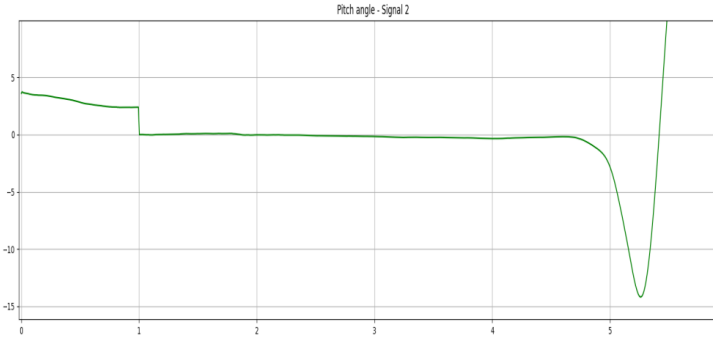
For this reason, the method is present in the interface but hidden from the application, where manual selection of the sensor for each leg and specifying the starting leg is preferred.

Algorithm

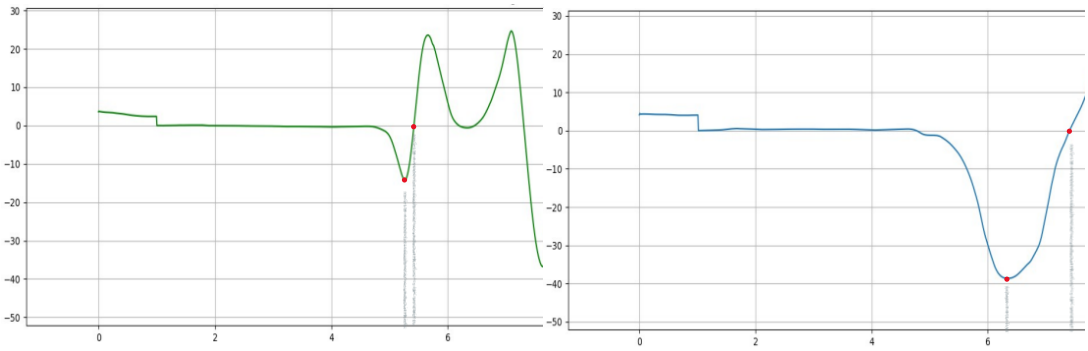
Starting from a standstill, the first movement of the leg brought forward is the bending of the knee. This fast movement involves the following phases in rapid succession: a negative zero crossing, followed by a significant negative minimum, and finally a positive zero crossing.



The concept is that the first process to detect a significant negative minimum followed by a positive zero crossing is dealing with the leading leg and can communicate this information to the other process through a shared variable

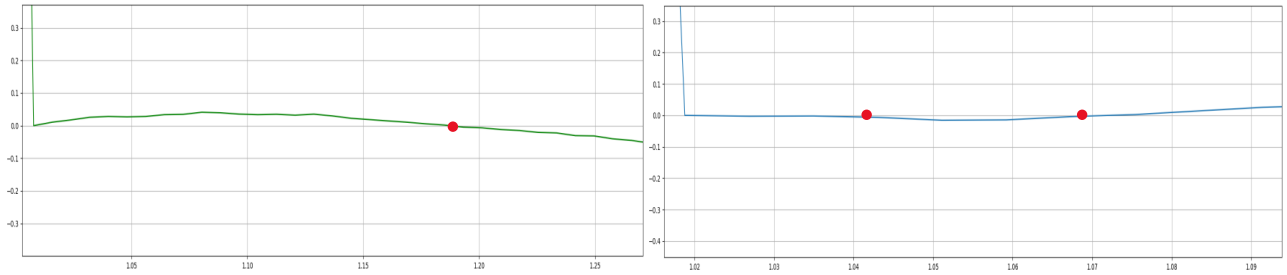


The leg behind may follow a similar pattern (negative zero crossing, minimum, positive zero crossing), but with a markedly different timing. The leading leg quickly bends the knee to bring it forward and place the foot down. The trailing leg, on the other hand, generates analogous signals but with much slower variations, not tied to knee bending (which cannot bend in reverse), but rather to the actual movement of the leg, which is slower and broader.



(DOUBLE STEP; Leading leg - green; Trailing leg - blue)

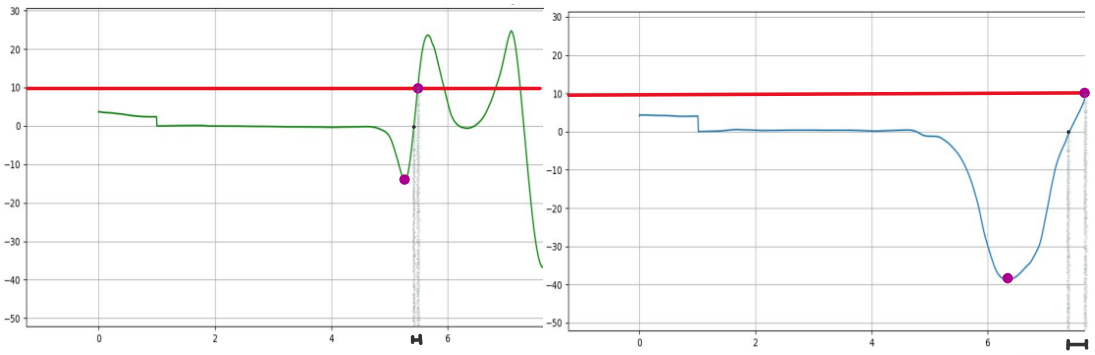
However, due to noise, unexpected zero crossings could occur.



The algorithm addresses this scenario by shifting the signals and applying a maximum threshold for the negative minimum caused by knee bending; in this way:

- A Gaussian filter is applied to the signal window to reduce noise. This is particularly crucial initially when both leg signals are close to zero. The Gaussian filter smooths the signal window, reducing oscillations that could cross zero and lead to incorrect detections.

- Both signals are shifted downward. This reduces the likelihood of detecting invalid zero crossings at the beginning because signal areas close to zero, which could erroneously cross zero, are all shifted directly below zero. Moreover, the translation delays the detection of the positive zero crossing because the signal must traverse a greater distance to reach zero. However, this delay is advantageous because the front leg's movement is faster than the rear leg's movement, so the same spatial delay significantly delays the detection of the positive zero crossing in the rear leg temporally. This decreases the probability of error and of detecting the rear leg's zero crossing first.

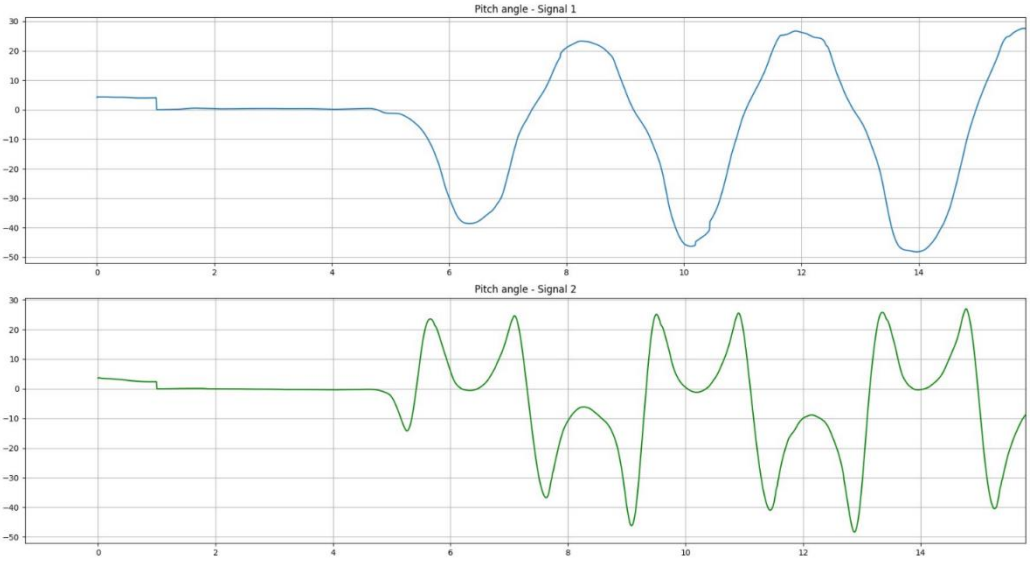


- A positive zero crossing is considered valid only if the signal has recorded a negative minimum lower by at least 0.1 degrees compared to the displacement. In natural leg movements, there is always a slight knee bend that generates a significant negative peak.

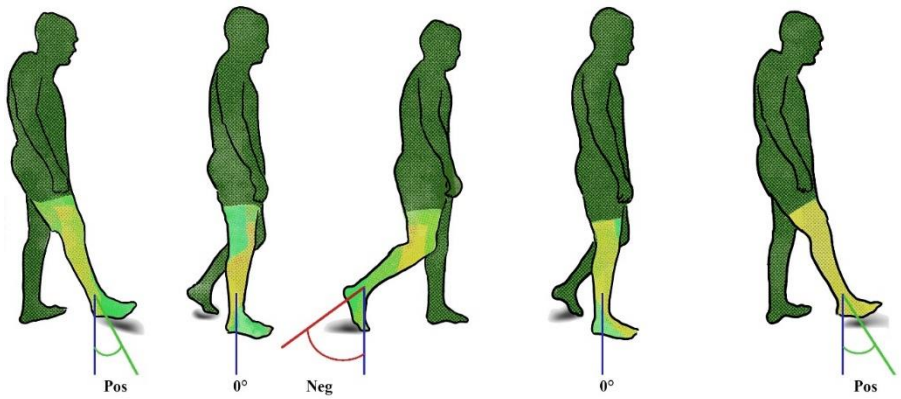
  This criterion further reduces the risk of considering irrelevant local negative minima of the wrong leg as valid, due to noise oscillations. It is unlikely that noise, already smoothed out, would generate such a deep minimum while simultaneously being the first to produce a positive zero crossing in that leg's signal.

Double Step

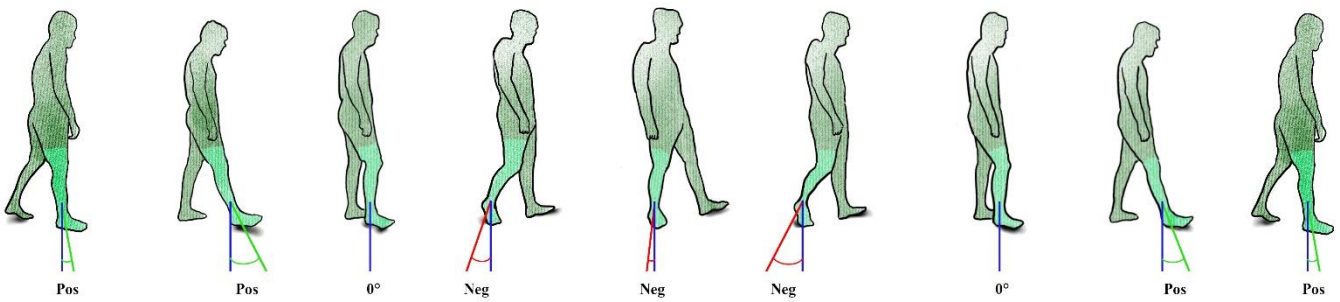
The double step is an exercise where the legs play different roles and require two parallel analysis algorithms. The movement is characterized as follows:



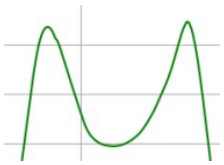
- One leg swings steadily, creating a sinusoidal pattern.



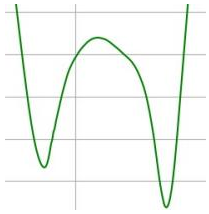
- The other leg takes a step forward followed by a step backward, alternating this movement. During the maximum extension forward or backward, the weight shifts onto this leg. This signal forms a double positive peak followed by a double negative trough, and so on.



- The heel strike is the maximum positive extension angle and is represented by the first positive peak. When the foot is fully placed on the ground, weight is shifted onto it, causing the signal to descend towards zero (or cross zero). Subsequently, the foot moves backward and gradually lifts off the ground, with the last moment of contact being just the heel touching the ground, returning to maximum extension again (second peak).
- After the lift-off, a step backward is taken, crossing zero as the foot moves into the negative quadrant.



- The first negative trough represents the touch of the toe to the ground, approaching zero represents the shifting of the weight, and the second descent represents the subsequent negative trough where the foot touches again only with the toe.
- The foot is brought forward again, the signal returns to the positive quadrant, and the cycle repeats.



__detectDoubleStep()

The method **__detectDoubleStep()** is called in both analysis processes when performing the double step exercise. This method internally determines which algorithm to use for each leg based on the **detectLeg()** method or user-specified parameters.

stepLeg()

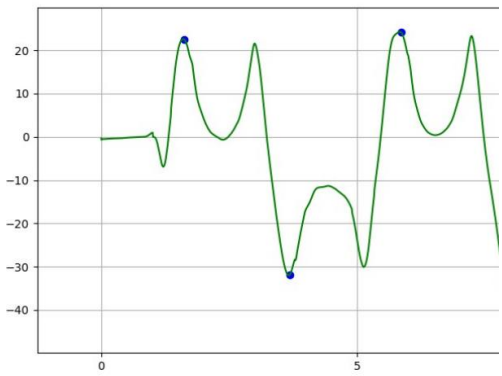
The algorithm implemented in the **stepLeg()** method handles the analysis of the signal from the leg performing forward and backward steps during the double step exercise..

This algorithm relies on the real-time peak detection method **peakFinder()** to detect peaks and troughs in the signal. Therefore, it checks that the incoming data windows are "complete," which is a necessary condition for **peakFinder()**. If the data windows are not complete, the method waits for additional data windows to arrive and correctly queues the data, gradually composing three complete windows.

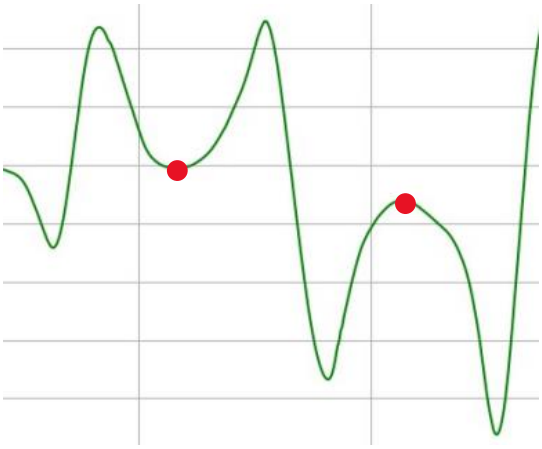
Peaks are considered valid only if they exceed a dynamic threshold, which can vary based on the minimum peak detected in the last 10 peaks. This threshold is similar to the one used in walking, with a validity range that also allows lower values to enable it to decrease. Additionally, for a peak to be considered valid, it must be at least 0.1 seconds apart from the previous one.

Only the magnitudes of positive peaks and negative minima contribute to the dynamic threshold; negative peaks and positive minima do not contribute.

Initially, the goal was to trigger the application sound at the moment of the first heel (or toe) contact with the ground (at each "first" peak or trough).



The point of maximum load transfer onto the foot is the dip between two positive peaks or negative troughs (positive trough or negative peak).



Due to the signal's configuration, we are certain that all peaks in the rising phase in the negative quadrant will be negative. However, in the descending phase in the positive quadrant, the minimum might dip slightly below zero. For this reason, solely for detecting such minima, an upward shift of the signal is applied.

To achieve better synchronization of the sound with the other leg, it was observed that it is more effective to trigger the application sound when the foot is fully grounded, somewhere between the heel strike and the complete load transfer dip.

The time required to fully ground the foot varies depending on the execution speed of the exercise, making it impossible to use a fixed delay for the sound after peak detection.

If the dip and peak heights were consistent, you could shift the signal to align all intermediate points between peaks and troughs along the x-axis. However, this isn't feasible because neither the peak heights nor the weight loaded on the patient's foot are predictable, making the depth of the dip relative to the peak uncertain. A single shift of the signal may therefore not capture all points of interest. Moreover, the dip varies greatly, making it challenging to accurately update a dynamic displacement.

Fast Movements: In fast movements, the time between foot contact and weight load is very short. The perceivable time difference between full foot contact and full weight load is almost negligible. Triggering the sound during the peak load is effective because the time from peak to trough is minimal, and the peak load occurs immediately after full foot contact.

Slow Movements: In slow movements, the time between the peak (foot contact) and trough (weight load) is longer. Triggering the application at the peak load can result in a delay in sound compared to full foot contact. After full foot contact, the weight shift may take longer to complete, causing a misalignment between the sound and the actual foot contact time.

Normally, full foot contact, especially in slow movements, occurs within 0.3 seconds (0.4 seconds in backward steps where the toe touches first). For fast movements, the time for full foot contact decreases drastically.

ALGORITHM

After detecting the first positive peak, the algorithm waits for 0.3 seconds before triggering the sound. If the movement is fast, the point of the next positive minimum will be detected before this time elapses, and the system will immediately sound without further delay.

This approach ensures that for fast movements, the sound activates at the positive minimum, whereas for slow movements, it activates at the expiration of the timer representing the maximum foot contact time.

(The image on the left represents a fast movement, while the one on the right represents a slow movement. The yellow arrow indicates where the sound occurs.)

Afterward, it looks for a new positive peak, then a negative minimum. When the negative minimum is detected, it waits for 0.4 seconds before sounding. If a negative peak is found before the time elapses, the system immediately plays the sound.

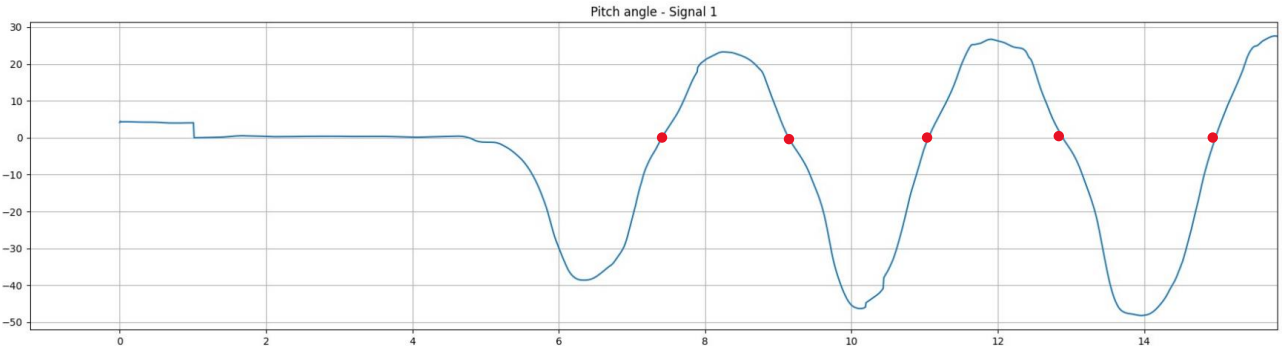
Subsequently, it searches for a new negative minimum, then a positive peak, and the cycle continues.

otherLeg()

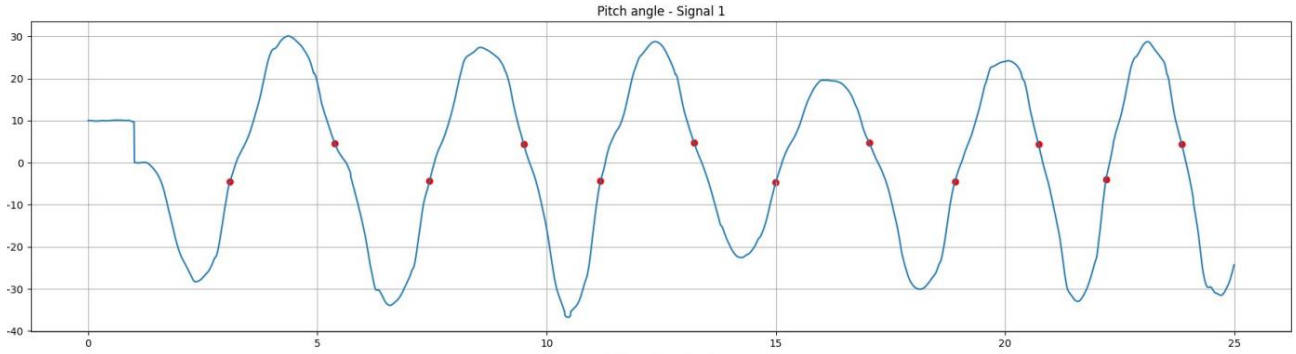
The **`otherLeg()`** method handles the analysis of the signal from the leg performing stationary oscillations during the double step exercise..

In this scenario, the signal follows a sinusoidal pattern and naturally intersects zero, allowing for the detection of zero crossings. When the body moves forward, the leg remains behind, generating a negative angle. When the body moves backward, a zero crossing occurs and the angle becomes positive. This cycle repeats with each oscillation.

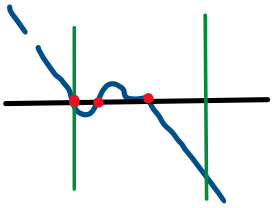
Each time the body moves forward or backward, the foot lifts off the ground, then recontacts and reaches zero. Unlike walking, where not all zero crossings are relevant, here, the system must emit a sound at each zero crossing.



For walking, the goal is to anticipate the detection of zero crossings, similar to the approach used in stationary oscillations. The algorithm is divided into two alternating phases: searching for positive zero crossings and searching for negative zero crossings. During the search for negative zero crossings, the signal is shifted downward to anticipate detection. Conversely, during the search for positive zero crossings, the signal is shifted upward to anticipate detection, as shifting it downward would delay detection.

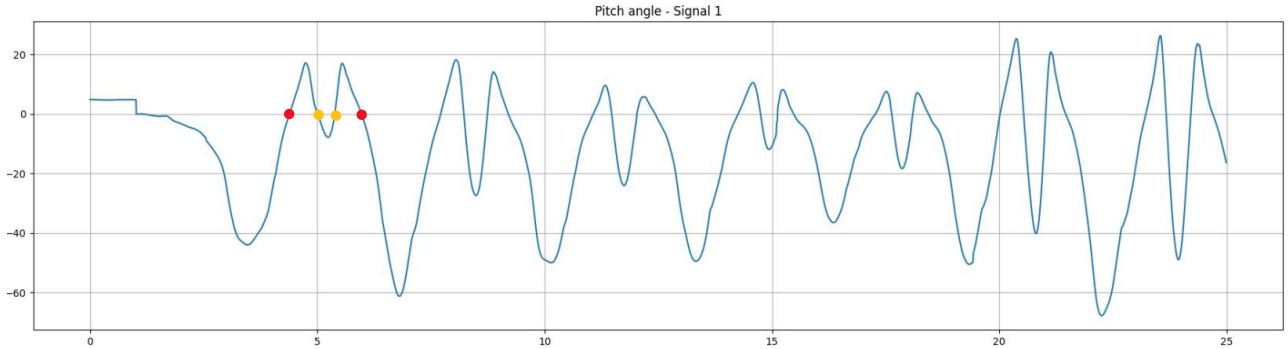


In contrast to walking, the peak heights are not evaluated, but a wait time and a gradient threshold are still used to improve the accuracy of detections. Even after translation, small noises could interfere with correct detections. Since every alternate zero crossing is detected, small oscillations might incorrectly cross zero.



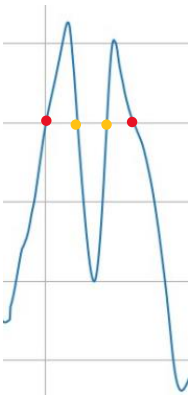
A dynamic threshold on peak heights would have prevented the detection of zero crossings caused by oscillations. However, with a sufficient wait time (represented as the range between the green lines) between successive detections, the same result is achieved. The wait time allows the system to move out of the critical zone near zero after a detection without making further detections. Essentially, a wait time allows ignoring zero crossings that are too close together, which could be due to impulsive noise and oscillations.

Furthermore, if the exercise is not performed correctly, knee bends can occur that alter the signal. Here is an exaggerated example of a signal with a drastic knee bend:



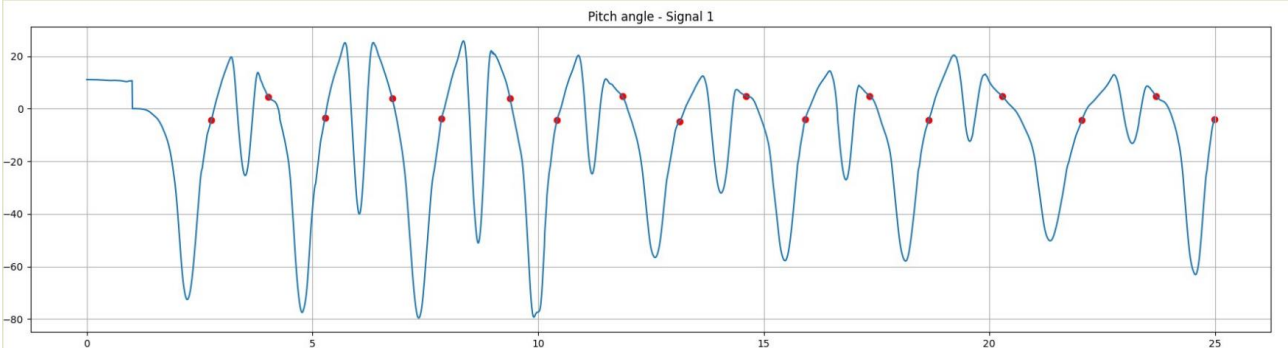
We notice that the signal now has additional dips. Each dip adds two zero crossings to ignore (yellow) between every two zero crossings to detect (red).

GRADIENT THRESHOLD



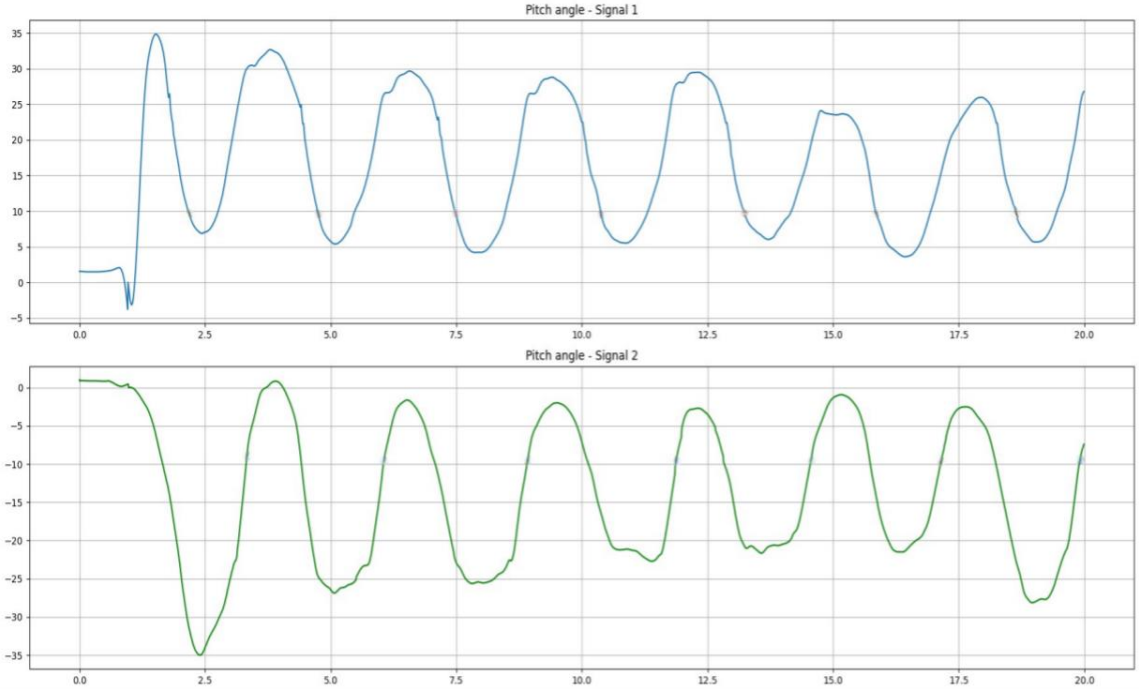
The internal slope of the dips and, consequently, the gradient of the additional zero crossings is higher than that of the zero crossings that should actually be detected. A dynamic threshold on the gradient allows excluding all zero crossings with too steep a slope. The gradient threshold is recalculated using the **_setNewGradientThreshold()** method via Exponential Moving Average (EMA) with alpha 0.85 every time a valid zero crossing is detected.

The threshold has a small range of validity above it to allow it to rise.



Swing

The swing is an exercise in which the legs play different roles, but the executed movement is very similar for both, following a sinusoidal shape. However, the movement of the two legs is specular relative to the x-axis. The leg in front extends, forming angles almost always positive, while the leg behind forms angles almost always negative.



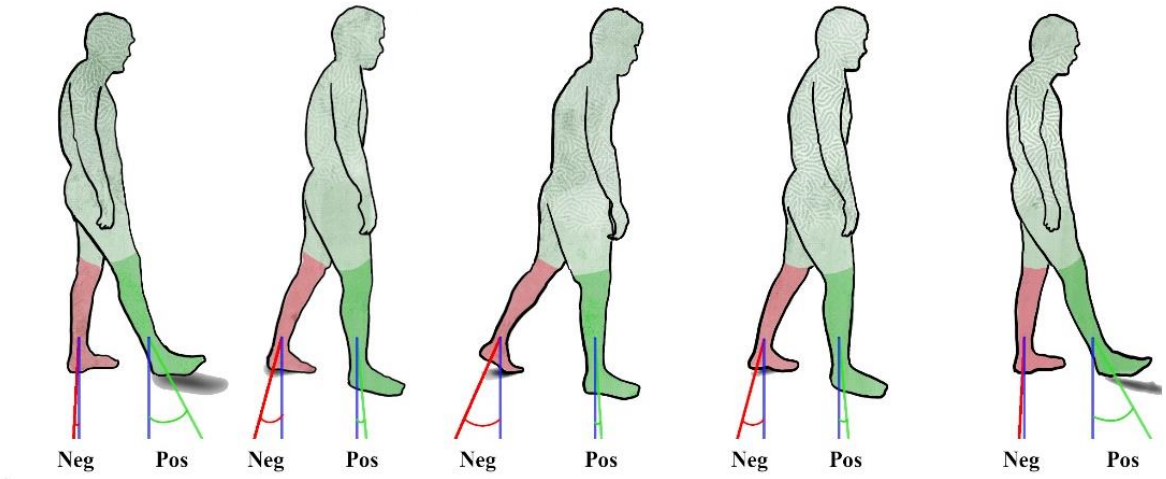
__detectSwing()

The method __detectSwing() is called in both analysis processes when performing the swing exercise. This method internally determines which algorithm to use for each leg based on the detectLeg() method or user-specified criteria.

swingFunction()

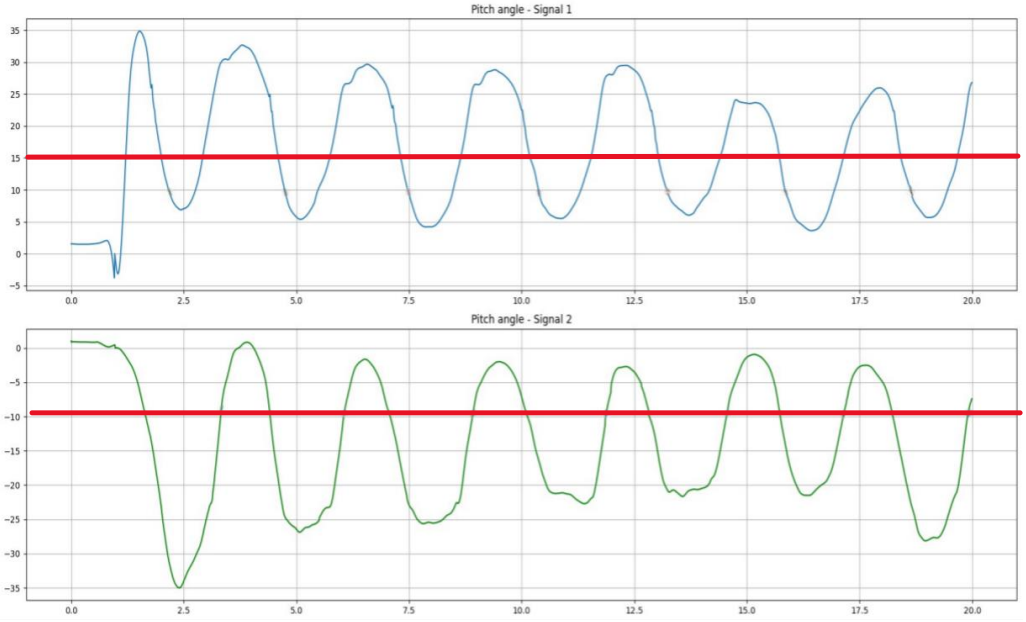
In both signals, we notice a similarity to the static leg configuration in the double step exercise. In the swing exercise, one leg moves away from zero when weight is transferred to the other leg. When the first leg returns to the ground, we approach zero again and aim to detect this moment. When the weight shifts to this leg, we are at the point closest to zero.

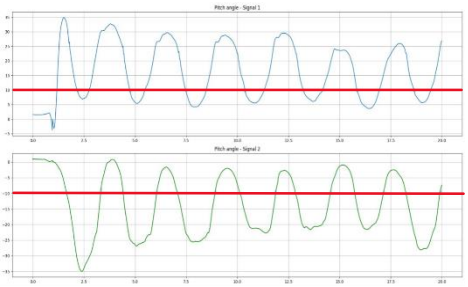
For the front leg, which predominantly forms nearly exclusively positive angles, moving away from zero signifies an increase and moving towards zero signifies a decrease. Conversely, for the other leg, which forms nearly exclusively negative angles, it is the opposite. The minimum angles of the front leg (those closest to zero) rarely drop below 5 degrees. However, the maximum angles of the back leg (those closest to zero) rise to over -5 degrees and approach zero significantly, sometimes slightly surpassing it.



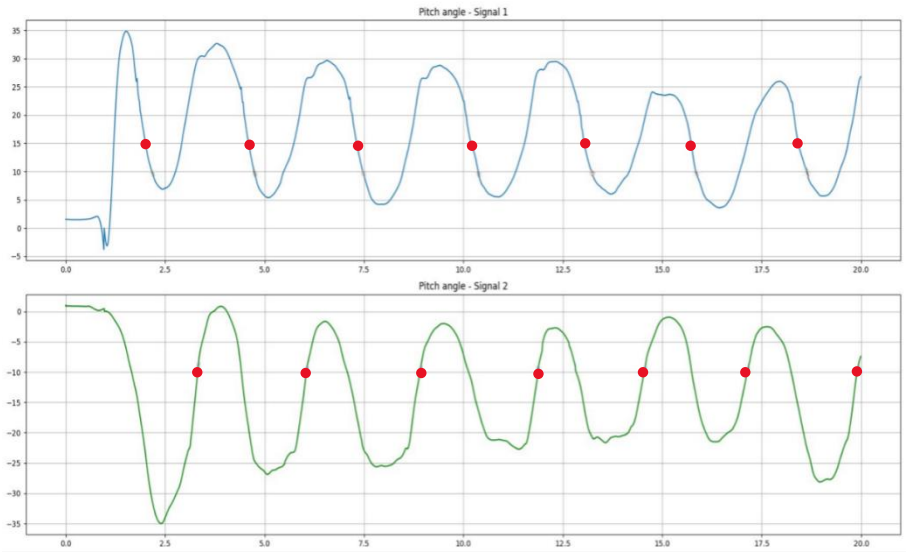
As mentioned, we want to trigger the sound at the relevant points in both signals, specifically when they approach zero (not when they move away). To achieve this, we translate both signals so that they intersect completely with the x-axis, allowing us to detect the zero crossings (negative for the front leg and positive for the back leg).

Since the two original signals have different distances from zero (all peaks of the front leg signal are almost always above 15 degrees, while the troughs of the back leg signal are almost always below -10 degrees), we use two different translations to achieve similar distances between the points originally closest to zero and the intersection zone for both signals. For example, we translate the front leg signal by -15 degrees and the back leg signal by +10 degrees. The translations depends from the velocity and so the hight of the signal.





Example with equal translations (-10 and +10): we notice that in the positive signal we detect closer to zero, while in the negative signal we detect farther away.



(Example of zero crossing detection after translations)

Zero crossing detection is validated similarly to the static leg in double step, using two mechanisms: a waiting time and a dynamic gradient threshold.

1. **Waiting time**: A sufficiently high waiting time is set to avoid detecting zero crossings that are too close to each other, likely caused by noise-induced oscillations. This helps ensure that only true zero crossings, corresponding to relevant movements, are considered.

2. **Dynamic gradient threshold**: Even in swing, knee bends can introduce unwanted zero crossings. The dynamic gradient threshold distinguishes these additional zero crossings from valid ones. Unwanted zero crossings have higher gradients that exceed the dynamically calculated threshold set on valid zero crossing gradients.

These mechanisms ensure accurate zero crossing detection, avoiding false positives due to noise or undesired movements such as knee bends.