Real-Time Detection of Restlessness Caused by Rapid Eye Movement Sleep Behaviour Disorder

Bachelor Graduation Thesis

Delft University of Technology

TU Delft

Momo Medical

Real-Time Detection of Restlessness Caused by Rapid Eye Movement Sleep Behaviour Disorder

by

Student Name	Student Number
J.R. Post	5221498
K. Kandiyoor	5290996
P. Kremers	5344417

TU Delft Supervisor:	Prof.dr. Paddy French
Momo Medical Supervisor:	Ir. Thomas Bakker
Project Duration:	April, 2023 - June, 2023
Faculty:	Faculty of Electrical Engineering, Mathematics & Computer Science, Delft
Cover:	Storyset Illustrations

TUDelft

Abstract

The objective of this project was to develop a real-time restlessness detection algorithm for Rapid Eye Movement Sleep Behavior Disorder (RBD). This project was completed in collaboration with Momo Medical, a fast-growing start-up, and developer of the BedSense device. Our project aimed to design a system capable of detecting RBD episodes and bringing patients to a lighter sleep stage. To accomplish this, proprietary data and existing data from an RBD patient were collected through the BedSense device. Additionally, data was extracted using a hardware system (developed by another subgroup) from a non-RBD test subject. The collected data was utilized to design and implement a restlessness detection algorithm. Given the limited data collected for this project, a classical detection algorithm was developed instead of machine learning techniques. The software system demonstrated impressive performance, achieving a balanced accuracy of 95.94%, an average sensitivity of 100%, and an average specificity of 91.88%.

The results of this project were integrated with another project seeking to develop a sensor system that brings an RBD patient to a lighter sleep stage. This project built a sock system with embedded sensors and a vibration module to achieve this. The proof of concept of the final integrated system demonstrated a non-invasive method of detecting and preventing episodes caused by RBD.

Preface

Rapid Eye Movement Sleep Behavior Disorder (RBD) is a neurological parasomnia disorder caused by the absence of muscle paralysis during REM sleep, leading to individuals acting out their dreams and potentially causing harm to themselves and others. The goal of this project was to design a non-invasive and user-friendly system capable of detecting episodes caused by RBD and bringing patients to a lighter sleep stage.

In order to achieve this goal, two subgroups were formed; one was responsible for developing a hardware system, and another focused on designing a software system. This thesis focuses on the design process, methodology, and results of the software subgroup, and highlights the collaborative integration of both hardware and software components to realize the final system.

The goal of the software subgroup was: to develop a real-time restlessness detection algorithm caused by RBD. This project was completed in collaboration with Momo Medical, a fast-growing start-up, and developer of the BedSense - a device placed under mattresses that tracks the sleep characteristics of residents in nursing homes.

We would like to express our sincerest gratitude to the Momo Medical team: Thomas Bakker, Jeroen Kant, Danny Eldering, and Karen van der Werff. We are particularly grateful for Thomas Bakker's input and guidance in helping us realize a successful project. Furthermore, the quick and thorough technical support provided by Jeroen Kant was of great value. We would also like to extend our thanks to our supervisor, Paddy French, for his mentorship and guidance. Lastly, our sincerest thanks to Bert Verzijl, a patient diagnosed with RBD who graciously agreed to collaborate with us. His invaluable insights have been of great help in the success of this project.

Lastly, a special thanks goes to Kerim Dzhumageldyev, Jan Kruize, and Tijn Schram, the members of the hardware subgroup. It was a pleasure working with you to make this project a success.

Delft, June 2023
Jasper Post, Krishnan Kandiyoor, and Pepijn Kremers

Contents

1 Introduction

In this chapter, important concepts regarding Rapid Eye Movement Sleep Behaviour Disorder that are used in the project will be explained. After that, the goal of the entire project will be defined and the task division of the subgroups will be explained. Lastly, the structure of this thesis will briefly be described.

1.1. Rapid-Eye Movement Sleep Behaviour Disorder

Rapid Eye Movement (REM) Sleep Behavior Disorder (RBD) is a neurological parasomnia disorder that affects muscle atonia during REM sleep [1]. To understand RBD, the sleep stages of humans will briefly be explained.

1.1.1. Sleep Stages

Regular sleep consists of four stages: N1, N2, N3 and REM [2]. The N1, N2, and N3 stages are considered non-rapid eye movement (NREM) sleep, and roughly 75% of sleep is spent in these stages. During a typical 8 hours of sleep, there are roughly 4 to 5 sleep cycles, where a sleep cycle follows the following sleep stage progression: N1, N2, N3, REM. One sleep cycle typically lasts around 90 to 110 minutes, with the majority of the time spent in the N2 stage.

The first sleep stage is 'N1', commonly referred to as 'light sleep'. During this stage, there is some light muscle tone in the skeletal muscles and breathing tends to occur at a regular intervals [2]. Eye movement, heart rate and body temperature all decrease. This sleep stage lasts around 1 to 5 minutes, accounting for approximately 5% of the total sleep time [2]. As the night progresses, an uninterrupted sleeper may spend even less time in this stage [3]. The next stages are N2 and N3, which are deep sleep, and usually, it is quite difficult to wake people up in these stages [2]. They account for 45% and 25% of total sleep respectively [2]. In these stages the body is the most relaxed. When a person is woken up in one of these stages, they often feel groggy and they usually have diminished mental performance for 30 minutes to an hour [2].

The last stage and the focus of this project is the REM stage. This is the stage where dreams occur and brain waves are similar to a wakeful state [2]. However, all skeletal muscles, except for eyes and breathing muscles, are atonic and without movement. In fact, this is the reason this sleep stage is known as rapid eye movement, as the eyes may move much more during this stage compared with the other stages. This stage initially lasts 10 minutes, but it gets longer in subsequent sleep cycles. After this stage, the cycle starts again. Figure 1.1 illustrates an average night of sleep.

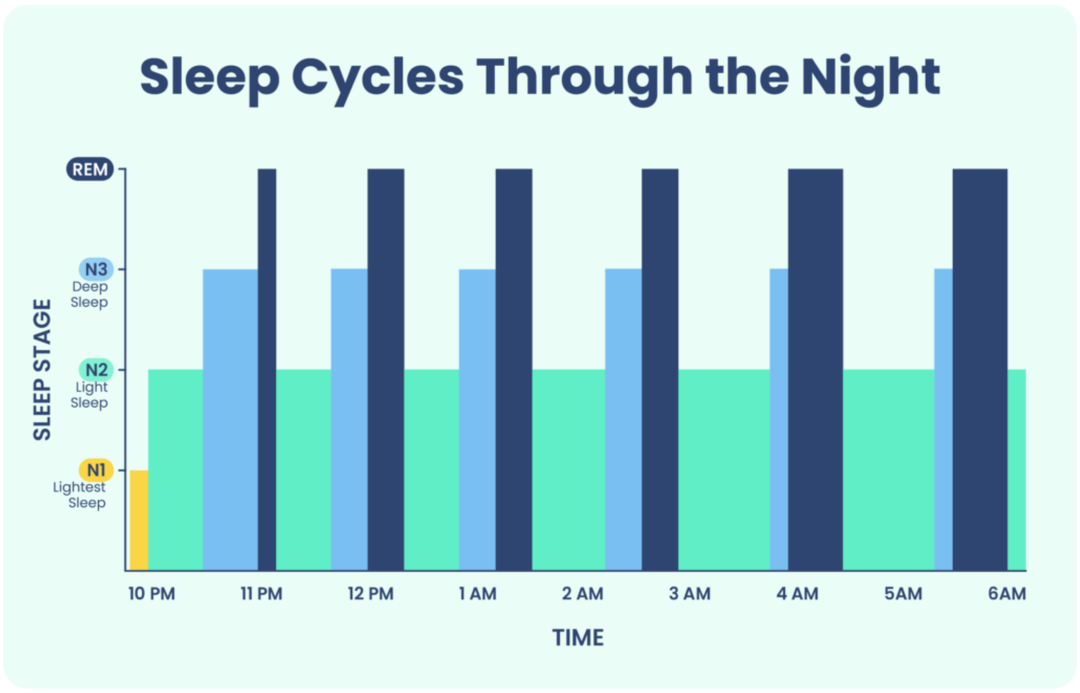

Figure 1.1: An average night sleep with the sleep stages. [3]

1.1.2. REM Sleep Behaviour Disorder

Patients suffering from RBD do not experience any kind of muscle paralysis during the REM sleep stage. This unfortunately leads to patients acting out their dreams, potentially resulting in dangerous situations causing them to hurt themselves or their bedpartners [4]. Injuries that have been documented include lacerations, dislocations, and hair pulling among many others [4]. The frequency of episodes is highly variable, sometimes only occurring once every 2 weeks, but at other times occurring multiple times a night for consecutive nights [4]. On top of that, RBD may be a precursor to neurodegenerative diseases [5], like Parkinson's disease and dementia [1]. Usually, the time between RBD diagnosis and the onset of such diseases is 5 to 15 years [6].

1.1.3. Diagnosis

There are currently three different standards used to diagnose RBD (see Appendix B). Diagnosis is primarily done using polysomnography (PSG), often accompanied by video recordings to visually verify the results. A PSG is a sleep study used to diagnose many illnesses and disorders, and consists of an extensive monitoring system that records electrical activity in the brain, muscles, and retina, as well as respiration and heart rate [7]. The electrical activity is measured using electroencephalogram (EEG), electromyography (EMG), and electrooculogram (EOG) techniques. With the information from these sensors, the sleep stages can be determined, and combined with other data gathered during the study, a diagnosis of RBD can be made.

Traditionally, it was thought that there was a gender disparity regarding the occurrence of RBD [4][5]. However, during a recent group study it was found that this was not the case [8]. Instead, males experience more issues from RBD because they usually have more aggressive dreams. This leads to males seeking out medical help more often than females. Additionally, the same study found that approximately 1% of the general population has RBD [8]. The average age of patients suffering from RBD is typically around 50 to 60 years old [1][9].

1.1.4. Treatment for RBD

There are currently no treatments specific for RBD, but the symptoms can be controlled by medication [10]. Two commonly used medicines to suppress RBD symptoms are clonazepam and melatonin [11], although the effectiveness of these medicines is based upon case series, small clinical trials, and expert consensus [1]. So these medicines may not work for every patient and may introduce unwanted side-effects. Another way to prevent injuries due to nightly movement is to use restraints or employ safety measures such as a sleeping bag [10]. A simple albeit a crude solution, it is the easiest and most cost-effective way to alleviate some of the struggles of RBD.

1.2. State-of-the-Art Analysis

As described, a lot of research has been done on the diagnosis of RBD. For instance, electromyography (EMG) signals can be used for RBD diagnosis. In a study, these EMG signals were fed into a machine learning algorithm with a random forest classifier, achieving an accuracy of 92% [12]. EMG is also an effective measure for diagnosing RBD without a machine learning algorithm [13].

Less extensive research has been done on the real-time detection of RBD episodes. Research has mainly been conducted on detection at a later stage when the patient has already left their bed [14]. This research used a pressure sensor to detect whether a patient had left the bed. When the pressure sensor detected a sudden drop in pressure, meaning that the patient left the bed, a voice recording plays. This was a voice recording of a family member that told the patient to go back to bed. In addition to the pressure sensor, one patient used a bed exit monitor, that works with a tethering cord. When the tethering cord is pulled, the voice recording played too. This research is a step in the right direction but only measured the episode at a later stage. This research showed a decreasing number of episodes after the system has been used for some time.

Furthermore, a study completed on in-bed motion detection and classification [15], showed that it was possible to classify movements in bed with machine learning techniques. This study investigated pressure sensor data and designed a machine-learning algorithm using a Support Vector Machine, Random Forests, and XGBoost techniques to classify movements into one of 9 classes. This approach, however, would detect the movements only when they occur, which is too late for the prevention of RBD episodes.

A proven method to monitor of real-time monitoring RBD patients is to make use of radio-frequency signals [16]. This research was approached by continuously monitoring RBD patients spatially. However, with this approach, when patients show significant movement, timely intervention may be challenging to achieve. Nevertheless, this approach demonstrates a high degree of non-intrusiveness.

1.3. Momo Medical BedSense

This project was completed in close collaboration with Momo Medical, a fast-growing start-up, and developer of the BedSense - a device placed under mattresses that tracks the bed posture of residents in nursing homes. The BedSense consists of piezoelectric and force-sensitive resistors and will be used throughout this project for data collection (see Chapter 3).

1.4. Goal of the Project

As mentioned previously, there are currently no treatments that have been created for RBD patients specifically. The state-of-the-art analysis has demonstrated effective novel methods for diagnosing patients and detecting episodes. However, these methods primarily detect episodes at a later stage, when the patient is already experiencing significant restlessness.

Therefore, the overall goal of this project is to design a complete system that is able to detect upcoming episodes caused by RBD in an early stage and prevent such episodes. The system to be designed must be non-invasive and user-friendly.

In order to realize the above-mentioned goal, two subgroups were formed; one was responsible for developing a hardware system, and another focused on designing a software system (see Figure 1.2).

The goal of the hardware group was to design and build a device capable of gathering biomedical signals related to RBD episodes and bringing the patient to a lighter sleep stage.

This thesis focuses on the design process, method, and results related to the goal of the software subgroup, which was: **To design and implement an algorithm that processes data from the hardware group and the Momo Medical BedSense, detects restlessness leading up to episodes caused by RBD, and subsequently makes a decision on whether or not to bring the patient to a lighter sleep stage.**

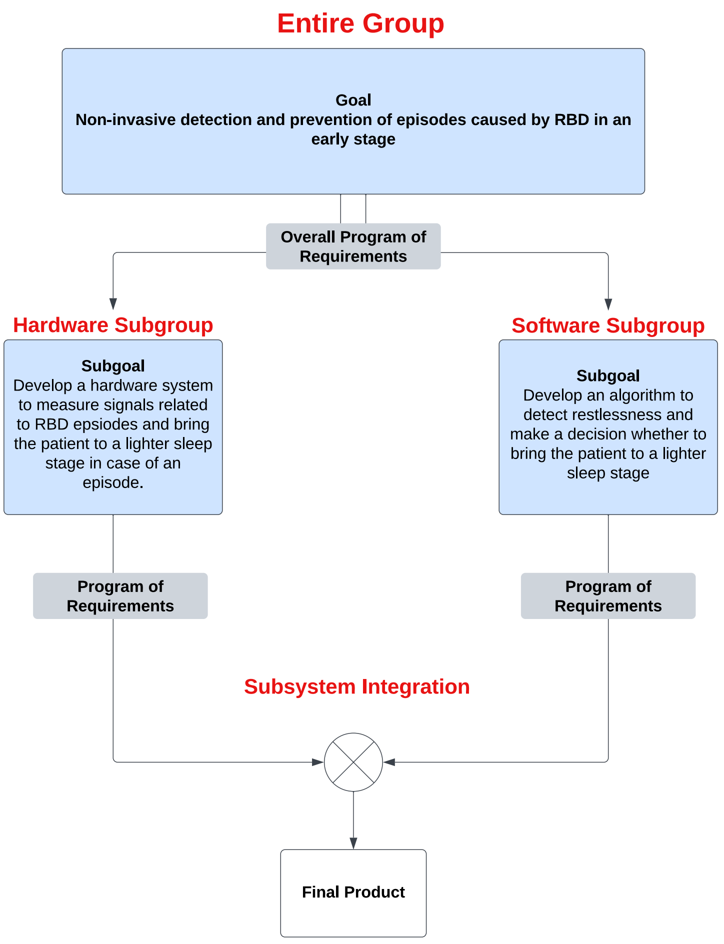
Entire Group
Goal
Non-invasive detection and prevention of episodes caused by RBD in an early stage
Overall Program of Requirements
Hardware Subgroup
Subgoal
Develop a hardware system to measure signals related to RBD epsiodes and bring the patient to a lighter sleep stage in case of an episode.
Software Subgroup
Subgoal
Develop an algorithm to detect restlessness and make a decision whether to bring the patient to a lighter sleep stage
Program of Requirements
Program of Requirements
Subsystem Integration
Final Product

Figure 1.2: Overview of the project and subsystem goals

1.5. Structure of Thesis

The thesis is structured as follows: Chapter 2 describes the program of requirements for the entire system and the software group specifically. In Chapter 3 the data collection process is discussed. In Chapter 4 the data analysis process is described, which leads to the design choices of the algorithm in Chapter 5. This chapter includes the various design choices made using visual and theoretical evidence. The implementation of the algorithm will be discussed in Chapter 6. The results of the algorithm will be discussed in Chapter 7. Finally, Chapter 8 includes a discussion, conclusion, and suggestions on future work.

2 Program of Requirements

2.1. Requirements for the Entire System

As mentioned previously, the goal of the software was to design and implement an algorithm that processes data from the hardware group and the Momo Medical BedSense, detects restlessness leading up to episodes caused by RBD, and subsequently makes a decision on whether or not to bring the patient to a lighter sleep stage.

The requirements created establish measurable criteria that the respective system must meet in order to achieve the project goal. The functional requirements refer to what the system must do, while the non-functional and safety requirements refer to peripheral aspects that need to be addressed. The functional requirements are outlined as follows:

1.1 The system must be applicable to multiple patients, it should not only work for the patient with RBD who is involved in the design of the system.
1.2 The system must work continuously for at least 10 hours, in order to ensure operation throughout the duration of a patient's sleep.
1.3 The system must provide real-time episode detection. This means that data should be collected and processed locally, rather than via a server.
1.4 The time elapsed between an RBD episode onset and bringing a patient to a lighter sleep stage should be less than 15 seconds.

The non-functional requirements are as follows:

2.1 The system should be non-invasive.
2.2 The system should be stand-alone. This means that it should work without any external products or applications.
2.3 The system should be user-friendly. People should be able to use the system without having any prerequisite/technical knowledge.
2.4 The system should be wireless, to allow the patient to move freely in their sleep.

The safety requirements are as follows:

3.1 The system must not cause harm to the patient.

2.2. Requirements for the Software Group

The following requirements have been derived from those of the entire system, and are applicable to the software system. The functional requirements of the software system are as follows:

- A.1 The algorithm must detect at least 80% of all episodes in a given time frame. This means that if a patient has 5 episodes throughout the night, 4 of them should be detected.
- A.2 The algorithm must have a specificity of at least 99%. This means that in an arbitrary time frame, the algorithm can only incorrectly detect an episode 1% of the time.
- A.3 The software system must be capable of communicating through Bluetooth Low Energy with the hardware module.
- A.4 The software system must read and process data, as well as make a decision in real time.
- A.5 The algorithm must make a decision on whether to bring the patient to a lighter sleep stage or not within 10 seconds.

The non-functional requirements of the software system are as follows

- B.1 The algorithm must be written in Python.
- B.2 All personal and medical data of patients must be processed locally, due to privacy requirements.
- B.3 PEP-8 programming standards must be adhered to when writing Python programs for the software system.

3 Data Collection

Prior to the design, implementation, and testing of the algorithm, an extensive data collection process was undertaken. Data collection served as the foundational step in algorithm development, providing the necessary data to design, validate, and tune the algorithm's performance. Data was collected both internally and through a patient diagnosed with RBD. This data was extracted from the Momo Medical BedSense and the system designed by the hardware subgroup. This chapter describes the data collection process, and how it contributed to the final software system design.

3.1. Momo Medical BedSense

The BedSense consists of six piezoelectric (PE) sensors, four force sensing resistors (FSR), and an accelerometer, used to measure vibrations, force on the bed, and the angle of the bed respectively. The PE sensors are sampled at 120 Hz, and both the FSR and accelerometer are sampled at 10 Hz. The BedSense operates optimally when placed beneath the mattress at chest height. Figure 3.1 depicts the Momo Medical Bedsense.

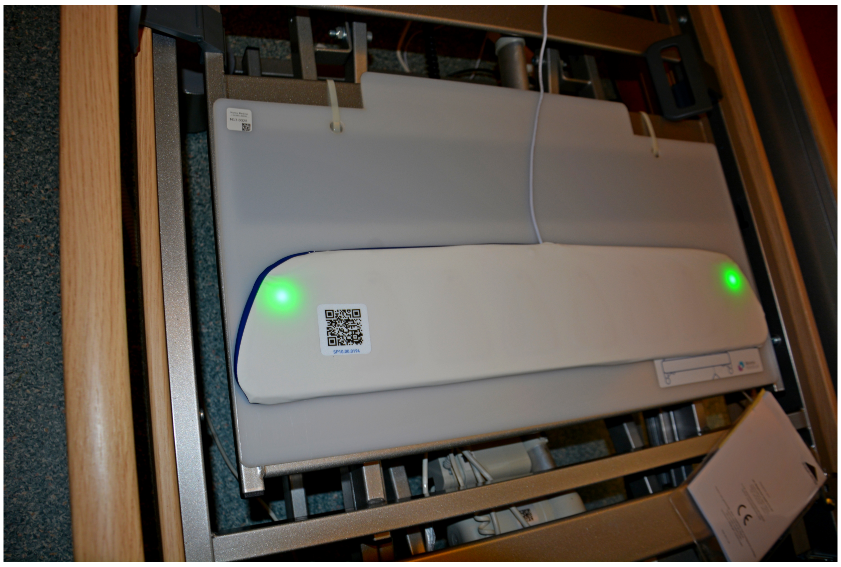

Figure 3.1: Momo Medical BedSense [17]

Piezoelectric sensors measure changes in pressure, acceleration, temperature, strain, or force by converting them to an electrical charge. There are six piezoelectric sensors on the BedSense spread out along the length of the board. In the measurements of these sensors, vibrations, and thus movement coming from the patient can be seen.

On the other hand, the resistance of FSRs change when pressure is applied. There are four FSRs present on the BedSense, and similar to the piezoelectric sensors, the sensors are spread out over the BedSense to record as much information as possible.

An accelerometer is a specialized sensor designed to detect and measure acceleration. In the context of the BedSense, the accelerometer is used to determine the angle of the bed in three axes. The accelerometer explained measures the angle of a bed. Therefore, it was decided that this data was not relevant to the context of this project.

3.2. Hardware Module

The hardware module designed by the hardware subgroup consists of a sock with sensors and a vibration module embedded in it. This sock system has two main uses; to complement the data from the BedSense to design a more robust algorithm (through sensors), and to bring the patient to a lighter sleep stage (through the vibration module). The sensors on the sock consist of two photoplethysmography sensors (PPG), an accelerometer, and an electrodermal activity sensor (EDA).

The PPG sensors measure blood oxygen saturation and blood volume, and the EDA sensor measures skin conductivity. This information can be used to detect stress in a patient. This is relevant within the context of this project, as stress has been proven to be an early indicator of episodes caused by RBD. The accelerometer has the function of measuring physical movement below the waist and complements the data collected from the BedSense. Figure 3.2 shows a prototype of the hardware system [18].

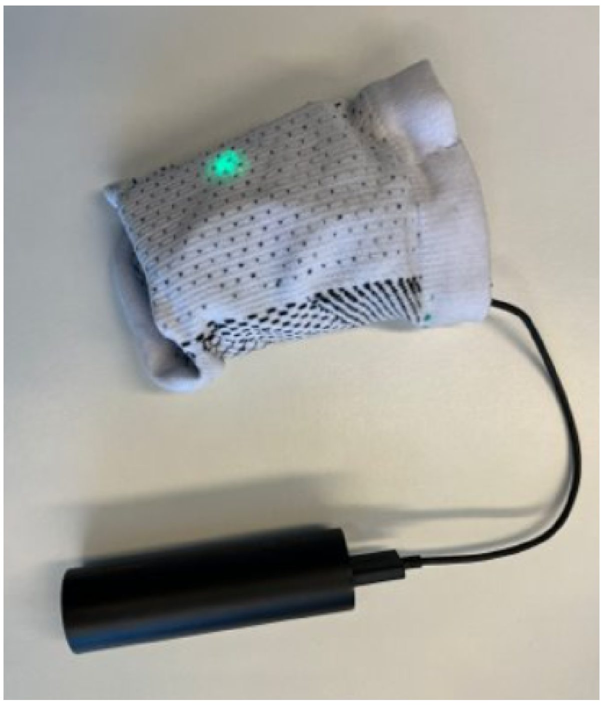

Figure 3.2: Prototype of Sock System

3.3. BedSense Data Collection

3.3.1. Patient Data

Given that the algorithm had to incorporate data from the BedSense, Momo Medical had already gathered data from an RBD patient using the BedSense for this project. The test subject was a male in his late 50s. This was the starting point of the data collection. Based on the visualization of this data, the final algorithm was designed.

The data provided covered the time period between November 2022 to April 2023, but more data was continuously collected throughout the project. The data provided included; PE, FSR, and accelerometer data. The patient also kept track of when episodes happened, their intensity, and if they woke up. Table 3.1 shows part of this data, the complete data can be found in Appendix C.

Table 3.1: Part of the Episode Information Provided by Patient

Date	Timestamp episode	Woke up	Woken up	Comments
21-2-2023 to 22-2-2023	00:22	Yes	No	–
1-3-2023 to 2-3-2023	03:38	Yes	No	–
3-3-2023 to 4-3-2023	01:57	Yes	Yes	Kicking and punching

The BedSense data was provided using CSV files. Each file contained 24-hour data from noon till noon of the next day. Each row contained the sensor data and a timestamp. The time between each row was 0.1 seconds. Since the PE data was sampled at 120 Hz, each row contained 12 data points for each PE sensor. Given that the patient had labeled when their episodes occurred, it was decided that this BedSense data would be used in designing the algorithm. This decision was taken in alignment with requirements A.1 and A.2 that pertain to the software system's performance. Designing the algorithm based on high-quality data increased the likelihood of a better-performing software system.

3.3.2. Proprietary Data

In order to improve the algorithm design, proprietary data from BedSense was collected. This was completed to investigate whether 'standard' movements below the waist could be distinguished from the BedSense data alone. If these movements could be distinguished, the data could be labeled and incorporated into the algorithm, making it easier to detect whether a patient is moving their legs.

An individual without RBD lay on a mattress with a BedSense located underneath it and performed limb movements with varying degrees of severity. For example 'moving left leg up and down slowly' and 'aggressively kicking with right leg'. The results of the PE sensors, when the individual kicked with their left and right legs, are shown in Figure 3.3.

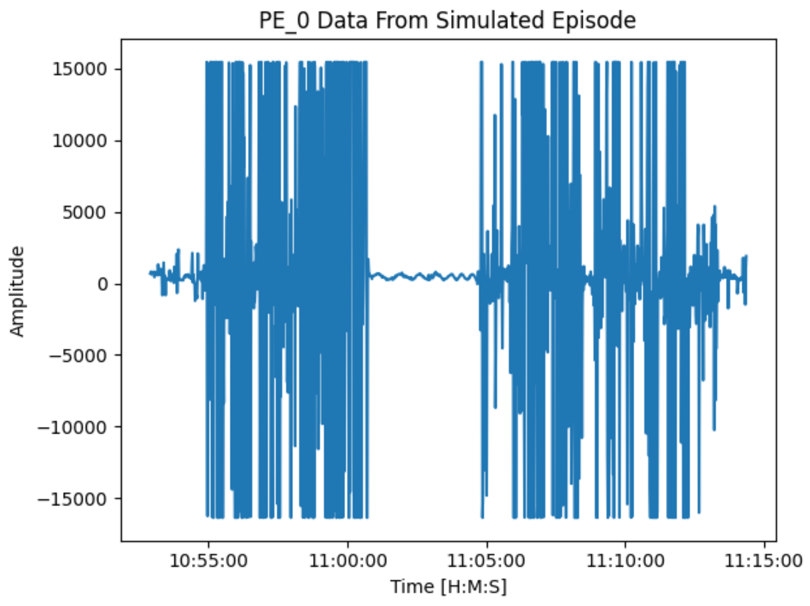

Figure 3.3: PE Sensor Results from Proprietary Data Generation

As can be seen in Figure 3.3, the PE sensors clip after reaching an amplitude of 15000. This is due to the fact that the PE sensors are extremely sensitive, and can pick up even the slightest of vibrations. Between approximately 11:00 and 11:05, the test subject is lying down with (almost) no movement. However, at other times, when kicking or punching started, the sensitivity of the PE sensors resulted in clipping. As a result, it was not possible to clearly distinguish and label kicking from this data. However, it was decided that general movement could still be detected by the PE sensors, so this data was used in the design and implementation of the algorithm. This decision was also taken in line with requirements A.1 and A.2, pertaining to the performance of the software system.

Similarly, the FSR data showed that kicking could not be differentiated through the BedSense alone. However, rolling around could clearly be distinguished from the BedSense data. This was an expected result given that the BedSense is placed around the chest height of a subject. Figure 3.4 shows the result of the FSR sensors from the proprietary data collection when a subject was kicking with their right and left legs.

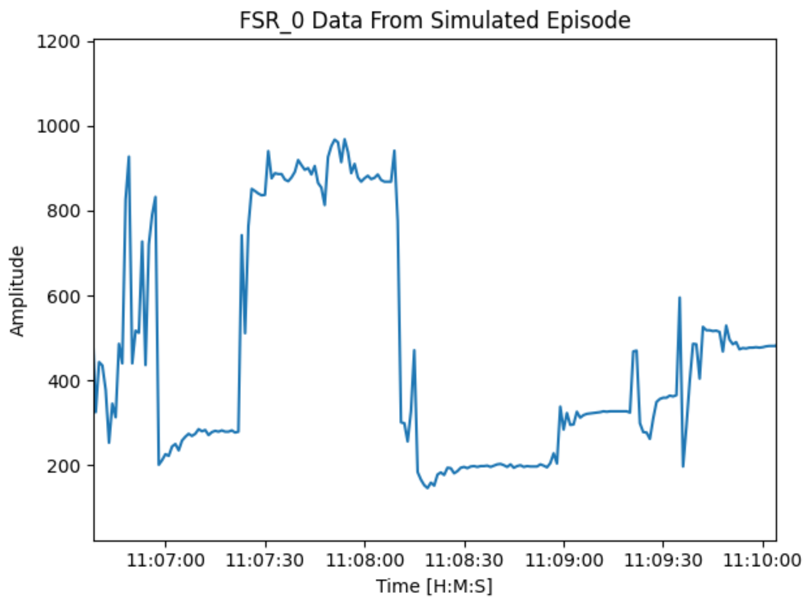

Figure 3.4: FSR Sensor Results from Proprietary Data Generation

The test subject kicked with their left and right legs between 11:08:30 and 11:09:00. These kicks only resulted in (very) slight peaks in the FSR data, meaning that it would not have been useful to label and use this data as there was no clear indication of a kick from the BedSense. The larger spike at approximately 11:08:15 is due to the subject rolling, given that this can easily be picked up by sensors at chest height.

To conclude, the approach of labeling and using data collected internally through the BedSense to distinguish movements below the chest was not taken. This was because these movements could not be distinguished. The results of this attempted approach justified the need for a hardware module that is below chest height, to be able to detect leg movements. Ultimately, this design choice helped fulfill requirements A.1 and A.2, relating to the performance of the algorithm.

3.4. Hardware Module Data

Following collection of data through the BedSense, data was collected through the hardware module (explained in Section 3.2). The intended outcome of this collection was to provide complementary data to the BedSense, that is more relevant to the context of this project.

3.4.1. Patient Data

The intention of collecting RBD patient data through the sock system was to aid in the design of the algorithm. Given that the sock system provides complementary data to the BedSense, collecting this data and designing the algorithm around it would improve the performance of the overall software system. The sock is capable of indicating stress levels through EDA and PPG sensors, given that stress is difficult to simulate, this data had to be collected from an RBD patient. Furthermore, even though motion data can be collected internally (through kicking with a sock on), it is more useful to collect this data from an RBD patient to understand the exact movements made. Finally, if an episode was caught using the sock system, the final software system could be tested on this data.

Unfortunately, due to a complex internet network in the patient's apartment complex, it was not possible to initiate and maintain a stable Wi-Fi connection between the PC and the sock. As a result, no extensive data was collected from the patient with the sock system [18]. For preliminary data collection on an RBD patient, see apendix A.

3.4.2. Proprietary Data

On the other hand, proprietary data through the accelerometers were collected through the sock system. A test subject without RBD lay down on a mattress while wearing the sock system. The test subject made kicking movements with their left and right leg, the results of the accelerometer on the sock sensor can be seen in Figure 3.5. It is important to note that for this experiment, the subject wore the sock on their right foot.

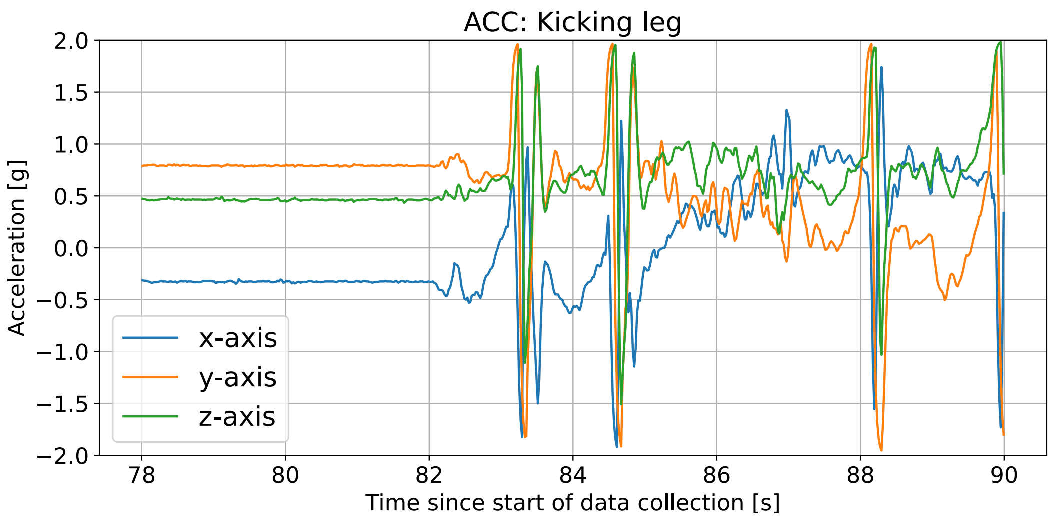

Figure 3.5: Accelerometer Results from Proprietary Data Generation

Figure 3.5 shows that the kicks can clearly be distinguished through the accelerometers on the sock. At approximately 83 seconds, there are clear spikes in all directions (x, y, and z) that show the kicking. These spikes are also present later in time when the test subject was kicking again. It was decided that the accelerometer data from the sock system would be very helpful in designing the algorithm, given that lower body movements could clearly be distinguished. This decision was taken as it was hypothesized that it would improve the performance of the software system. Ultimately, this decision aided in fulfilling requirements A.1 and A.2.

As mentioned earlier, PPG and EDA data was not collected from the test subject, given that stress leading up to an RBD episode would be extremely hard to simulate. However, as mentioned earlier this was not possible. Therefore, the data collected from the sock system only consisted of proprietary data, which helped in designing the motion module of the algorithm (see chapter 5).

4 Data Analysis

After completing the data collection phase, the acquired data was subject to cleaning and processing to ensure its reliability and consistency. Subsequently, the data was visualized using appropriate tools and techniques to gain meaningful insights. This chapter focuses on the analysis of the collected data, specifically targeting the extraction of important information pertaining to sleep restlessness exhibited by RBD patients. The outcomes of this data analysis served as a starting point for exploring multiple avenues in the final algorithm design.

4.1. Data Cleaning

Before useful data can be visualized, the data had to be cleaned. This meant that any irregularities in the data needed to be looked at and possibly replaced or removed. The data also had to be filtered to make sure that there was no power line interference (PLI) or any other noise. Power line interference refers to the noise caused by electromagnetic fields of nearby powerlines and other devices [19]. This noise has a frequency of 50 or 60 Hz, depending on the country. The historical data Momo Medical provided had already been filtered and no PLI was present. The data also did not contain any values such as infinity or NaN. On the other hand, data from the EDA, PPG, and accelerometer needs to be filtered.

4.1.1. Electrodermal Activity Data Filtering

The frequency spectrum of electrodermal activity signals is not clearly defined, however, most researchers agree that it is between 0 - 2Hz [20] [21]. A low-pass filter with a cut-off frequency of 2 Hz was deemed sufficient to filter out all the high-frequency noise, including the PLI. Additionally, the electrodermal activity consists of two distinct components, phasic and tonic [22]. The tonic response is the slow-moving baseline, while the phasic response is the event-related fast-moving component [21]. Due to the fact that the purpose of the EDA signal is to determine stress, it was determined that the use of the phasic component of the EDA signal was sufficient. The phasic component was assumed to be data with a spectrum above 0.05 Hz [23]. A simple highpass filter could be used to retrieve the phasic component, however, advanced and better-performing filtering algorithms have been proposed in a research context, such as cvxEDA [23].

4.1.2. Photoplethysmogram Data Filtering

The photoplethysmogram (PPG) is used to determine if a patient is stressed. This is done by determining the heart rate from the PPG data, and the heart rate is related to stress. To accurately capture all possible heart rates and remove all other noise, the PPG data was filtered with a 4th-order Butterworth band-pass filter with cutoff frequencies of 0.4 Hz and 4 Hz [24]. This frequency spectrum covers a heart rate of 24 - 240 bpm, which is relatively wide considering the average heart rate is between 60 - 100 bpm.

4.1.3. Accelerometer Data Filtering

The accelerometer on the sock is not used to determine stress but provides complementary movement data alongside the BedSense. The only noise that needs to be removed is power line interference, which can be done using a notch filter at 50/60 Hz.

4.2. Visual Data Analysis

The algorithm needs to incorporate movement-based data from the BedSense and accelerometer from the sock, and stress-based data from the EDA and PPG data. In order to devise an approach for the algorithm, indicators that differentiate baseline and episode data need to be found. This step is not required for the EDA and PPG data, since their indicators have already been determined. However, the data of the accelerometer and BedSense need to be analyzed for indicators. It was decided that the most appropriate method to detect indicators of episodes was through visualization. Therefore the data was visualized in a number of ways.

First of all, the time-amplitude relation of the sensors was plotted. Figure 4.1 shows the time-amplitude relation of the average of the 6 piezoelectric sensors of the BedSense. The red vertical line is the timestamp provided by the patient as being the start of the episode. A few minutes before the red line, the amplitude of the data points abruptly increases and stays high for a few minutes. This shows that the restlessness of the patient increased in that timeframe, thus suggesting that there was an episode caused by RBD. However, the start timestamp of the episode seemed to be slightly off, since the restlessness was already present before the red line. This was the case for other episodes too, so an estimation (green line) was added to properly convey the actual start of an episode.

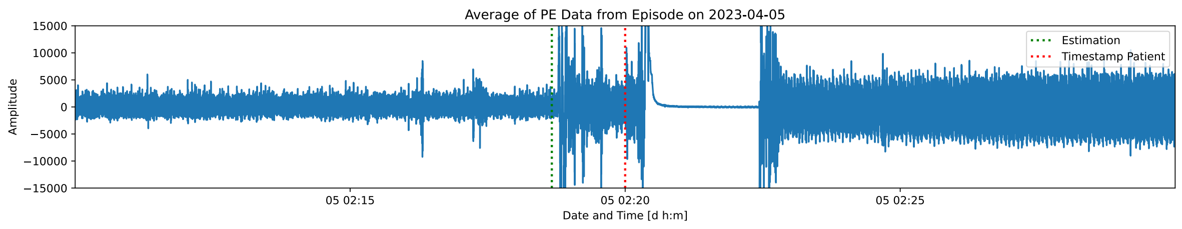

Figure 4.1: Time-amplitude relation of the 6 piezoelectric sensors averaged.

Figure 4.1 shows that the episode lasted less than 5 minutes, and also that there were periods when the patient was not restless. An episode was thus concluded to not be a continuous period of time of pure restlessness, but bursts of restlessness over a period of a couple minutes. Furthermore, at the end of the episode, the amplitude of the PE sensors approaches 0 and stays there for a few minutes. This means there were no vibrations, so it is possible the patient woke up and decided to get out of bed. The moment the patient got back into bed is also clearly visible.

Due to the fact that other movements apart from the episode were also visible, only looking at the amplitude of the PE sensors was not enough to determine if there was an episode. Therefore the time-amplitude relation of the force sensing resistors was also analysed. Figure 4.2 shows this relation of the average of the FSR. The episode is more difficult to distinguish using the FSR data. There are some minor amplitude differences at the beginning of the data, but it mostly stays constant. It is, however, visible when the patient gets in and out of bed.

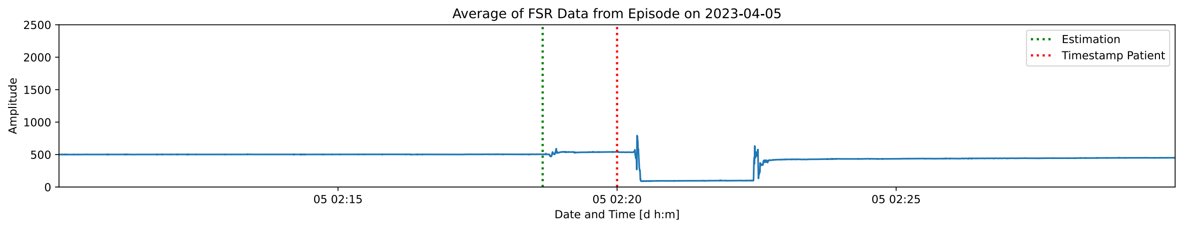

Figure 4.2: Time-amplitude relation of the 4 force sensing resistors averaged.

The averages of the sensors show some interesting things, but looking at the individual sensors uncovers some interesting phenomena. Figure 4.3 and 4.4 show the data of PE sensors 0 and 5 respectively, which are located on opposite sides of the BedSense. The amplitude of the data points of PE sensor 5 is much higher before and during the episode compared to PE sensor 0. It may be due to the fact that the patient was lying on the side where PE sensor 5 is located. After the patient got back into bed, the amplitude is roughly the same, so the patient would lie in the middle of the bed.

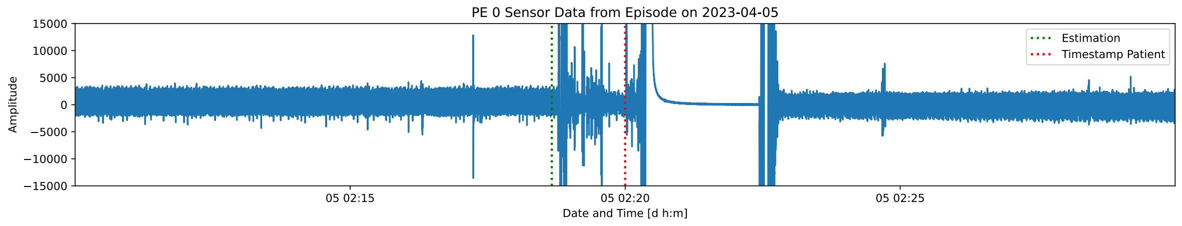

Figure 4.3: Time-amplitude relation of piezoelectric sensor 0.

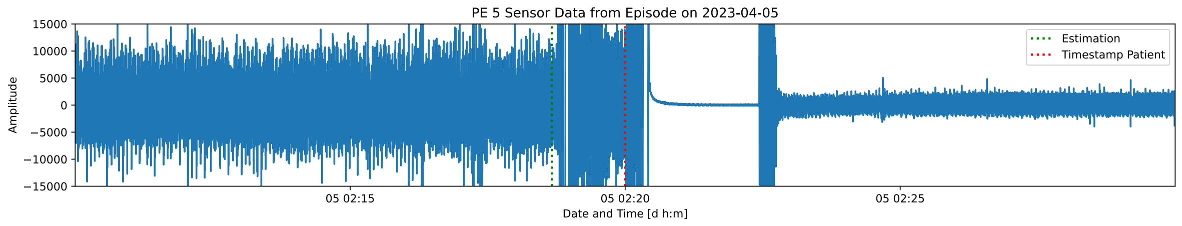

Figure 4.4: Time-amplitude relation of piezoelectric sensor 5.

Figure 4.5 and 4.6 show the data of FSR 0 and 3 respectfully, where FSR 0 is located on the same side as PE sensor 0 and FSR 3 on the same side as PE 5. From this data, it is also clear that the patient was lying on one side of the bed before and during the episode. So, from this, it can be concluded that the algorithm needs to look at all sensors, not just a few.

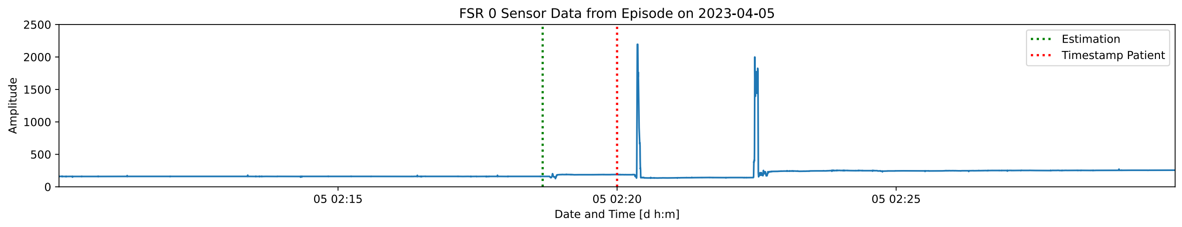

Figure 4.5: Time-amplitude relation of force sensing resistor 0.

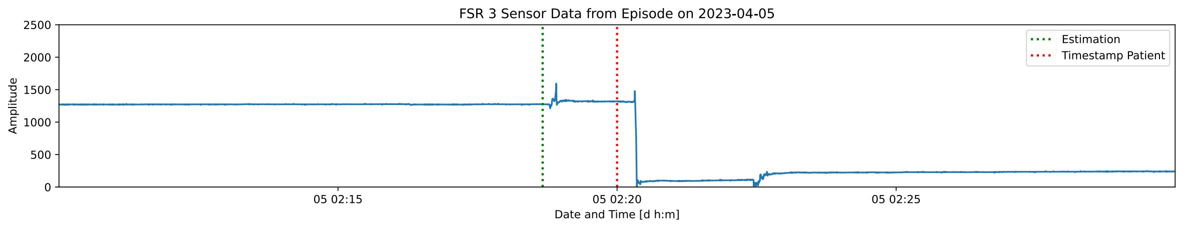

Figure 4.6: Time-amplitude relation of force sensing resistor 3.

The restlessness during an episode created a lot of vibrations. Therefore, the next step taken was to plot the time-frequency spectra of the PE data of the episodes. Figure 4.7 shows a time-frequency plot of the average of PE sensors of the same episode as the above graphs, however, the yellow line now depicts the estimated start of the episode. Figure 4.7 shows that the amplitude of the frequency components increases if there is an episode. Unfortunately, this is also the case when other movements occur, such as getting in and out of bed.

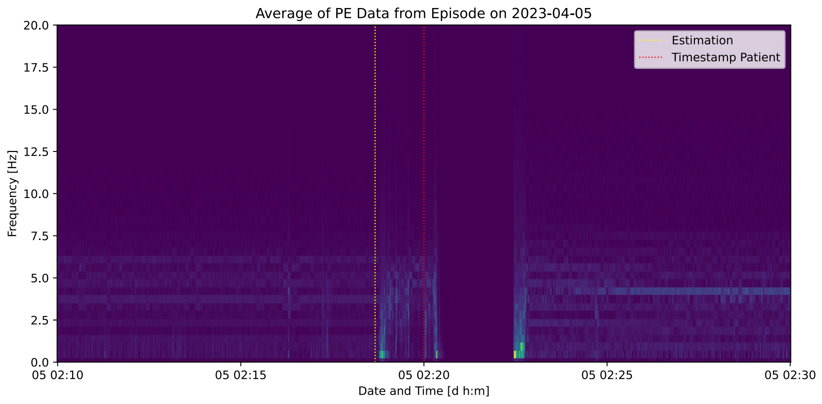

Figure 4.7: Time-frequency relation of the average PE data.

Additionally, individual frequency components were not explored further as a way to determine if an RBD episode occurs, since not every episode is the same. Therefore, the sum of the amplitude of the frequency components was calculated and plotted, see Figure 4.8. The episode is clearly visible in the graph between 02:18 and 02:21. Before the episode, the amplitude is very low, which indicates that there is no restlessness. After the episode, the moments the patient got out and in bed are again visible.

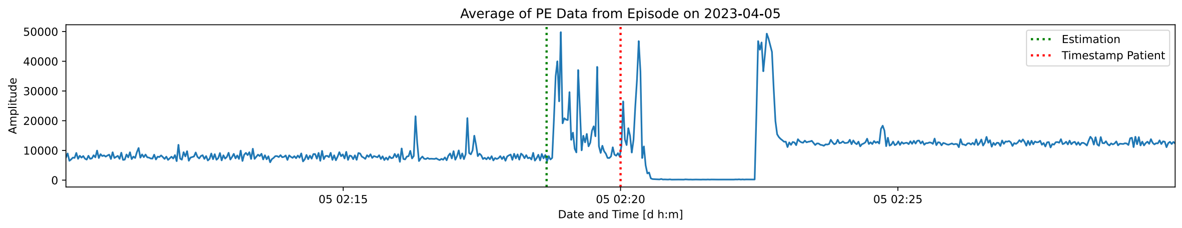

Figure 4.8: Sum of the frequency bands of the average PE data.

5 Algorithm Design

This chapter describes the choices made during the design of the algorithm. Each of these choices needed to be aligned with the Program of Requirements, and they were justified while keeping the main goal of the algorithm in mind, which was detecting a period of abnormal, excessive, and potentially harmful movement during sleep. To reach this goal multiple approaches were considered. These will be discussed in this chapter, as well as the final approach.

5.1. Approaches

In the state-of-the-art analysis (Section 1.2), it was mentioned that machine learning has been proven to be an effective approach to in-bed movement detection and classification, i.e. to enable abnormal movement detection. Machine learning algorithms are capable of detecting patterns that humans could never see. However, a downside of machine learning algorithms is that they need a lot of diverse data, or else there is a significant risk of the algorithm becoming biased. The available data at the start of the project consisted of a single patient with 15 episodes in a time period of half a year. This is too little and non-diverse. A machine learning algorithm would be very biased and not usable by other patients. Requirement 1.1 states that the system, and thus the algorithm should be versatile; usable for multiple patients. Therefore, it was decided to not use machine learning.

Thus, it was decided that the algorithm would have to be a classical algorithm. This typically consists of step-by-step instructions processing an input, resulting in the desired respective output. To design a classical algorithm, first, available data needs to be analysed, which can be seen in Chapter 4. The design choices can be made, motivated by logic and human observations.

In the data analysis, the frequency of the PE sensor data was analysed. It was hypothesised that a higher frequency of vibrations could indicate more movement. This approach was not chosen. There was no clear presence of large amplitudes of specific frequency components during episodes, as the PE sensors had been filtered for any frequencies above 13 Hz. It was concluded that this does not leave enough information in the frequency domain to distinguish certain movements.

It was further hypothesized that setting thresholds on the amplitude of the PE signals could have been another approach. However, given that the PE signal clips quite quickly, this approach also does not provide enough information for detection.

As is visible in Figure 4.8, the total amplitude of the frequency spectrum of a period of measurements is an indicator of an episode. This relates to the energy in the original signal, as mentioned below in section 5.2. As the goal is the detection of a period of excessive movements, it was deduced that it is logical to use the energy of a period of the FSR and PE signals as an indicator. The increased movement would likely cause increased activity in the energy of the FSR and PE signals.

5.2. Algorithm Design

After the data visualization, a strategy for the algorithm was devised. It was clear that multiple sensors would have to be used in tandem to accurately determine the occurrence of RBD episodes. The algorithm consisted of several components, each focusing on a single sensor or a group of sensors. The following subsections will explain the design and operation of the various algorithm components.

5.2.1. Movement Detection using PE & FSR Data

In Chapter 4, it was determined that the sum of the amplitude of the frequency components of the PE sensors was a good indicator of when an episode occurs. During the data analysis stage, this was simply calculated as the sum of frequency components. This was sufficient for visualizing the data, but for a real-time algorithm, it is quite slow. The square of the amplitude of frequency components represents the power within that frequency band, and considering that power is directly related to energy with:

$$P = \frac{E}{t} \tag{5.1}$$

where P, E and t are power, energy and time respectively. If t is kept constant, by using a sliding window of constant width, for example, energy instead of power can be used to determine if an episode occurs. According to Parseval's theorem [25], shown in Equation 5.2 in which $x[n]$ is defined as a discrete-time signal and $X[k]$ is its equivalent in the frequency domain, the total energy of a signal is conserved when going from the time domain to the frequency domain and vice versa. This removed the need for the Fast Fourier transform and made the energy calculation much quicker and less computationally intensive. As a result, the algorithmic implementation without the Fast Fourier transform helps achieve requirement A.5, relating to the time frame in which the algorithm must make a decision.

$$E = \sum_{n=0}^{N-1} |x[n]|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X[k]|^2 \tag{5.2}$$

5.2.2. Movement Detection using Accelerometer Data

The accelerometer in the sock complemented the BedSense with regard to motion detection. To determine the amount of movement, the absolute values of the three axes, in the x, y, and z directions were summed and the sample standard deviation of the sum was calculated. A high standard deviation indicates there is a lot of movement. The formula for the standard deviation is

$$\text{s} = \sqrt{\frac{\sum\limits_{i=1}^{N} (x_i - \bar{x})^2}{N-1}} \tag{5.3}$$

where N is the number of samples, x_i the i-th sample and $\bar{x}$ the mean of the samples.

5.2.3. Out of Bed Detection using FSR Data

As was determined during the data visualization, the energy of the vibrations was not enough to accurately determine if an episode occurs. Other movements, such as getting in and out of bed, also produce enough energy to make it seem like an episode. It takes the average person roughly 60 to 90 minutes to reach the REM sleep stage [26], so the algorithm only needs to start detecting episodes an hour after a person got into bed. This would eliminate the false positives that occur when a person gets into bed. The FSR data can be used to detect if a patient is in bed or not.

Figure 5.1 shows the design of the out-of-bed detection component of the algorithm. If the FSR signal is higher than a certain baseline, there is a patient in bed. The baseline is not equal to 0, since there is always a mattress on top of the BedSense, which produces a positive non-zero force. This baseline value may therefore also be different for differing sleeping conditions. If a patient is detected, a timer is slowly decremented. Once the timer reaches 0 or lower and a patient is still in bed, the output is 1, else the output is 0.

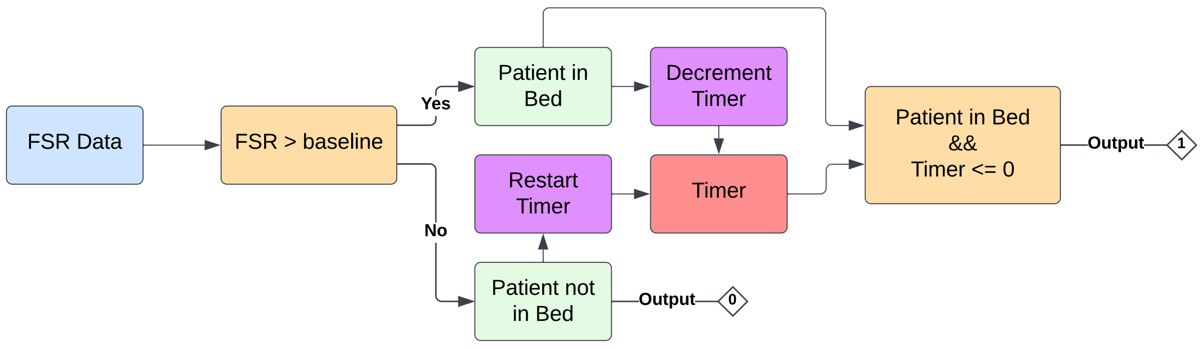

Figure 5.1: Out of Bed Flowchart

5.2.4. Stress Detection using PPG Data

The PPG data is used to determine the heart rate which is related to the stress response of the patient. An ideal PPG signal is shown in Figure 5.2. The heart rate is equal to the difference between consecutive systolic peaks [27]. After filtering (described in Section 4.1.2), the systolic peaks of the PPG data need to be found. This required a peak finding algorithm. Given the complexity of implementing such an algorithm, an already tested and implemented algorithm was used (see Section 6.2). Once the systolic peaks have been determined, the heart rate is simply the average difference between consecutive peaks:

$$\text{bpm} = \frac{60 \cdot f_s}{(N-1)} \sum_{i=0}^{N-1} (x_{i+1} - x_i) \tag{5.4}$$

where f_s is the sample rate, N the number of systolic peaks and x_i the sample index of the i-th systolic peak.

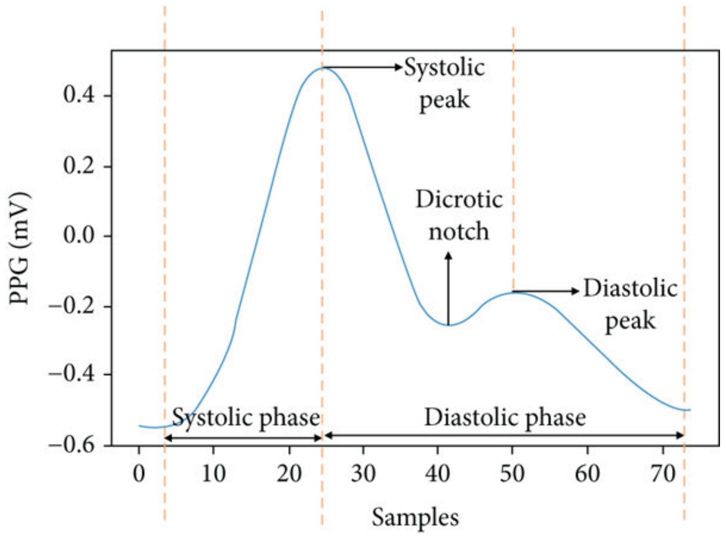

Figure 5.2: Ideal photoplethysmogram signal [28]

5.2.5. Stress Detection using EDA Data

The EDA sensor measures the conductivity of the skin which is related to stress response of the patient. After filtering (described in section 4.1), only the phasic component of the EDA is left. Figure 5.3 shows a typical phasic signal and some of the features that can be extracted from it. The average amplitude and average rise time will be used to determine if a person is stressed. This requires an algorithm that can determine onsets and peaks. An already implemented algorithm will be used (see Section 6.2). The average rise time and average amplitude are then:

$$\bar{T}_{rise} = \frac{1}{N}\frac{1}{f_s}\sum_{i=1}^{N}(p_i - o_i)$$
$$\bar{A} = \frac{1}{N}\sum_{i=1}^{N}(y[p_i] - y[o_i]) \tag{5.5}$$

where N is the number of peaks, f_s the sample rate, p_i and o_i the indexes of the i-th peak and onset respectively, and y the phasic EDA signal.

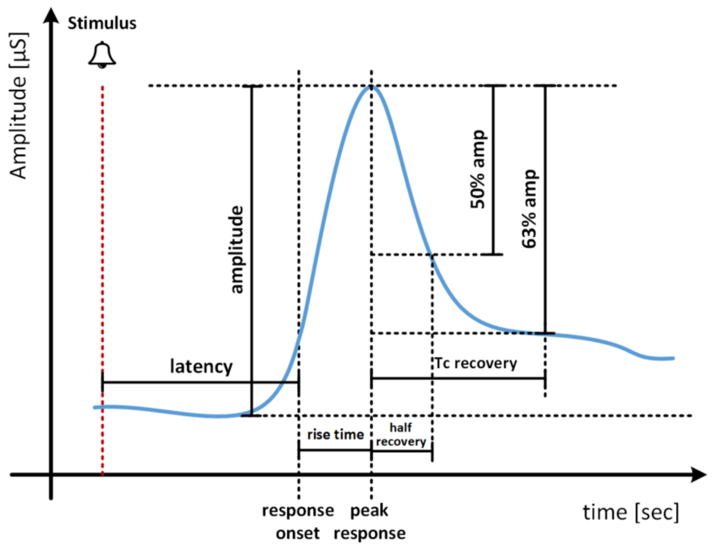

Figure 5.3: Typical skin conductance response [29]

5.2.6. Decision Making Process

The decision making process will use all the indicators described in the previous sections. At the start, it will be split up into two components, motion activity and stress response. The motion activity component incorporates the indicators from PE, FSR and accelerometer in the sock, and determines if there is a lot of motion. The stress response component uses the data from the PPG and EDA sensors to determine the stress level of the patient. If there is both high motion activity and a high stress response, an episode has been detected. If not, there is no episode. The entire process can be seen in Figure 5.4.

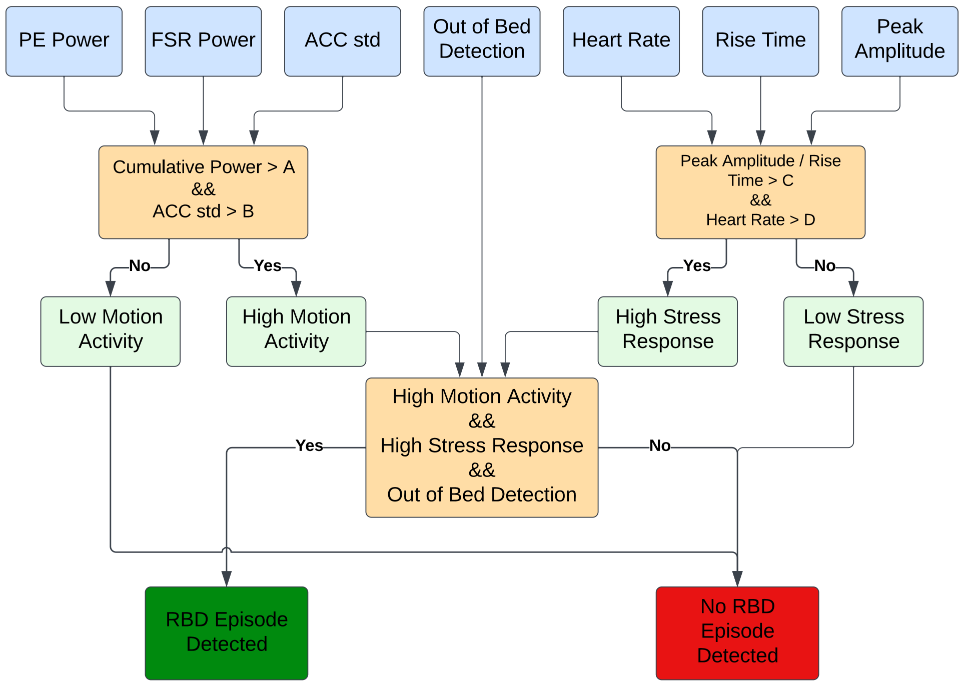

Figure 5.4: Flowchart of the decision making process

5.3. Total System

Figure 5.5 shows the complete block diagram of the algorithm design, and in Chapter 6 the implementation of the system will be discussed. As can be seen, all the data on which the decision is based is gathered together. From this, the stress response, as well as the motion activity, can be analysed, and a well-informed decision is made.

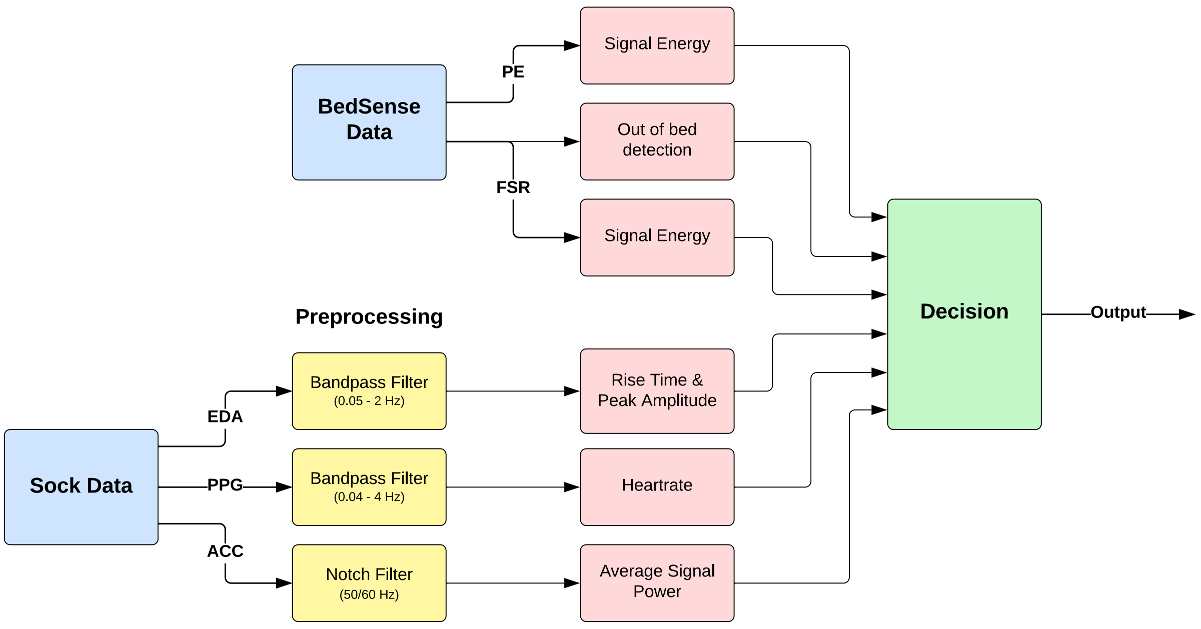

Figure 5.5: Block diagram of the algorithm design

6 Algorithm Implementation

The complete software system consisted of two modules combined into one algorithm, this can be seen in Figure 6.1. The motion module processed the data from the BedSense and the accelerometer in the sock. The stress module processed the PPG and EDA data from the sock. This design choice was made given that the data from the sock was only available at a later stage of this project. The division of the algorithm also made it possible to assess the individual performance of both modules, in terms of how well they can detect RBD episodes. The sock system has been designed to be complementary to the BedSense, so when the two modules are combined the results should improve. The final implementation of the algorithm is a Python class called: `DetectionRealTime`. The `DetectionRealTime` class is the main class that contains all the functions needed in the algorithm. These are incorporated as methods of the class. Each instance of the `DetectionRealTime` class also holds class variables that may be accessed by any method of the class. This chapter will further elaborate on how the algorithm was implemented. The source code can be found in Appendix D.

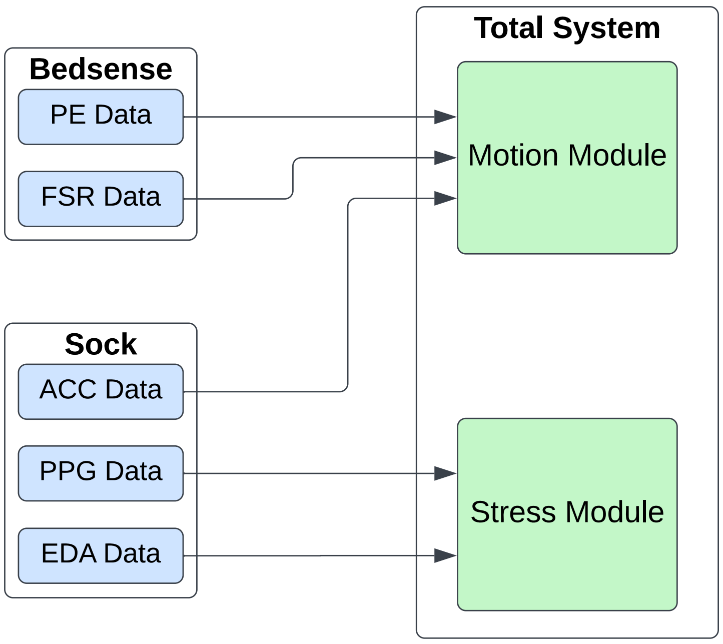

Figure 6.1: Block diagram showing the total system integration

6.1. Motion Module

The motion module consists of three methods: `signal_energy`, `filter_acc`, and `process_acc`. The method `signal_energy` processes all FSR and PE signals, and stores their power in separate class variables. The calculation is done using the formula discussed in Section 5.2.1. The accelerometer data is initially filtered using the `filter_acc` method and is subsequently processed using the `process_acc` method. The filtering is done using the SciPy [30] package. The processing is done as discussed in Section 5.2.2.

6.2. Stress Module

The stress module uses the EDA and PPG data to predict if an RBD episode occurs. The module consists of two methods: `process_eda` and `process_ppg` which process the EDA and PPG data respectively. The Python packages Neurokit2 [31] and HeartPy [27] were used to efficiently extract RBD episode indicators. Neurokit2 can automatically filter and analyze the EDA data. It filters the data using a 4th order Butterworth filter with a cutoff frequency of 3 Hz. The cutoff frequency is slightly higher than discussed in Section 4.1.1, but during testing, the performance was not negatively impacted. To extract the phasic component, the cleaned signal is passed through a median value smoothing filter to remove areas of rapid change. The phasic signal is obtained by subtracting the smoothed signal from the original signal. This approach performed better than a high-pass filter or the cvxEDA algorithm described in Section 4.1.1. Once the phasic component is extracted, Neurokit2 can analyze the signal and determine various features, which can then be used to determine the average rise time and average amplitude as discussed in Section 5.2.5.

The method `process_ppg` uses the HeartPy package to filter the PPG signal and extract the heart rate. The signal is filtered using a 4th order Butterworth bandpass filter with cutoff frequencies 0.4 and 4 Hz as discussed in Section 4.1.2. After filtering the cleaned signal is analysed and the peaks are extracted, and from these peaks, the heart rate is determined.

6.3. Total System Integration

Finally, the motion module and stress module were combined. The code can be seen in Appendix D.1. As can be seen in Figure 5.5 all the information from both the BedSense and the Sock have been gathered for the decision. The decision also takes into account if the patient has recently been out of bed or not. The method `decision` then determines if there is an episode or not. Figure 5.4 shows a flowchart of this process. In order to make the algorithm quick, extensive use of the Numpy [32] and Pandas [33] packages for efficient data processing were used.

For the proof of concept, the system accesses the BedSense data using API requests to the Momo Medical database. This is incorporated into the method `importdata_Bedsense`. For a finished product, it would be ideal to retrieve this data straight from the BedSense using Bluetooth, because when using API requests there is a delay of about 5 seconds, which is not desirable as it increases the response time of the algorithm. The data from the sock system is sent to a laptop with Bluetooth and then imported to the algorithm by the method `importdata_Sock`. The algorithm processes about 5 seconds worth of data every second. The 5 seconds of data is necessary to get an accurate heart rate from the PPG signal.

7 Testing and Results

This chapter will go over the results of the system, or to be more specific part of the system. Due to technical issues, no measurements were made using the sock system. Therefore there are no results of the stress module and the total system. There are months worth of patient data recorded by a BedSense, which enabled the testing of the motion module excluding the accelerometer data. Therefore, results presented for the 'motion module' throughout this chapter exclude the accelerometer.

7.1. Testing and Verification

Originally, the intention was to first test both the stress and motion modules individually. Following this, the entire system (integration of both modules) was to be verified and tested. However, due to unforeseen circumstances mentioned above, only the algorithm that uses data from the BedSense was tested. For this purpose, a testing environment/pipeline was created, see Appendix D.3 for the code.

7.1.1. BedSense Testing Algorithm

To test at least part of the algorithm, an implementation of the motion module using only the FSR and PE signals was made. See the code in Appendix D.2. The main method of the `Detection` class is `algorithm (self, df: list[pd.DataFrame])`. The input of this method is a list with pandas DataFrame objects, which consists of sensor data from the Bedsense. The output of the algorithm is a list containing either a zero or one, which are generated every second. A zero represents no episode, whereas a one represents an episode. The motion module itself, without the accelerometer data, should provide a good estimate of if there is an episode or not.

7.1.2. Testing Pipeline

For testing the motion module, data from the BedSense was used. The data was collected from an RBD patient 3. This data was also labeled with timestamps indicating when episodes start and end. In order to effectively test the algorithm, it was run on entire nights of data with episodes present. Following this, the output of the algorithm was compared to the optimal outcome. See Appendix D.3 for the respective code.

To define the 'optimal outcome' the goal of the project had to be considered. As mentioned in Section 1.4, the goal of the project is: 'to detect restlessness leading up to episodes caused by RBD, and subsequently make a decision on whether or not to bring the patient to a lighter sleep stage'. RBD episodes need to be prevented, so the desired output cannot be defined as positive during episodes and negative otherwise. For this reason, the definition of the metric sensitivity was altered.

Sensitivity (or true positive rate), indicates how well the system correctly identifies positive instances. It gives a ratio of the proportion of actual positives that are correctly classified as positives. See Equation

7.1

$$\text{Sensitivity} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{7.1}$$

For the scope of this project, it was decided that sensitivity per night will be defined as the number of episodes, during which the first 10 seconds the algorithm outputs a positive, divided by the total amount of episodes. See Equation 7.2.

$$\text{Sensitivity} = \frac{\text{Nr episodes detected within 10 seconds}}{\text{Nr total episodes}} \tag{7.2}$$

7.2. Results

For the results of tests that were run on the motion module, different metrics will be considered. Sensitivity, specificity, and balanced accuracy. See the respective code in Appendix D.5 Sensitivity has already been discussed, and defined above.

Specificity is the ability of the system to correctly identify true negative instances. See equation 7.3.

$$\text{Specificity} = \frac{\text{True Negatives}}{\text{True Negatives} + \text{False Positives}} \tag{7.3}$$

Balanced accuracy is a metric that takes into account both sensitivity and specificity to evaluate the performance of an algorithm on imbalanced data. It provides a balanced assessment of the algorithm's ability to correctly estimate both positive and negative instances, irrespective of the output distribution. In the case of RBD episode detection, the output is not well distributed when looking at positive and negative outputs. This uneven distribution is caused by episodes only taking a couple of minutes throughout the night. See equation 7.4 for the balanced accuracy formula.

$$\text{Balanced Accuracy} = \frac{\text{Sensitivity} + \text{Specificity}}{2} \tag{7.4}$$

7.2.1. Anomalies

It was expected that a large number of false negatives would be present in the testing results. This is because the motion module only takes movement into account. With just the BedSense data, it is very hard to distinguish between different movements as well as the severity of the movement (see Chapter 3). It is designed for nursing homes; to give an estimate on low/medium/high activity over a longer period. Therefore, the motion module might see a small movement during sleep as an episode, causing a lot of false negatives.

Given that the available data is labeled by the patient themselves, it cannot be guaranteed that the labeling is flawless. For example, certain episodes might have been missed or the algorithm could have been set off by their bed partner. This may also have caused some anomalies.

The patient has recorded video footage synchronous with the measurements taken from the BedSense. Some of the footage was analysed to investigate the cause of certain false positives in the results. The conclusion was that most false positives were caused by minor movements. These minor movements do not justify the decision to bring the patient to a lighter sleep stage, and this shows the limited effectiveness of the motion module on its own.

7.2.2. BedSense Results

These are the results of the motion module without the accelerometer of the sock. As expected (due to the false positives) the specificity is lower than requirement A.2.

Table 7.1: Test Results of BedSense Algorithm

Date	Balanced Accuracy [%]	Sensitivity [%]	Specificity [%]
22-02-2023	96.65	100	93.30
02-03-2023	95.06	100	90.11
04-03-2023	96.68	100	93.35
05-04-2023	96.79	100	93.57
11-04-2023	96.92	100	93.84
16-04-2023	95.45	100	90.90
26-04-2023	95.71	100	91.43
28-04-2023	95.17	100	90.35
05-05-2023	96.34	100	92.68
07-05-2023	96.23	100	92.46
13-05-2023	94.21	100	88.42
19-05-2023	94.22	100	88.44
20-05-2023	97.86	100	95.71
24-05-2023	95.85	100	91.69
Average	95.94%	100%	91.88%

These results mean that all episodes have been detected within 10 seconds. This also conveys that in 100 minutes, the motion module gives a false positive for a total of approximately 8 minutes. This does not fulfill requirement A.2. However, these results reflect an algorithm in which accelerometer data, and maybe, more importantly, the stress module data are not present. Stress is a big indicator of RBD episodes, as the dreams that are enacted during an RBD episode are mostly nightmares. The information on whether or not the patient is stressed will enable the algorithm to filter out a lot of false positives caused by small movements.

7.2.3. BedSense Non-RBD Test Results

To put the results of the patient into context, the motion module was tested on a large dataset collected from non-RBD patients. See Appendix D.6 for the corresponding code. The sensitivity is now not an insightful metric, as desirably the algorithm should not go off at all. Specificity will therefore be considered. Momo Medical provided 5 nights worth of anonymous BedSense data from 7 random residents of nursing homes. The resulting average specificity that was found was 97.04%. As expected this was higher than the specificity found when running tests on the RBD patient, as RBD patients move more during their sleep. There could be multiple reasons why the algorithm went off for non-RBD patients. First of all, there is no information on the nursing home residents from which this data came, so they themselves might have medical sleep issues. Furthermore, the motion module does not consider stress levels, it only looks at movement. Therefore, it can be set off by a lot of movements, for example: the residents laying awake in bed at night, the residents turning or the residents waking up from a nightmare.

8 Conclusion and Future Work

8.1. Discussion

As seen in Figure 6.1, the sock system and BedSense feed data to the algorithm which runs on a laptop. The algorithm then takes and sends a decision on whether or not to activate the vibration module on the sock. A more fitting implementation of the final design would have been to flash the algorithm on the BedSense or the sock system. However, due to time constraints, it was decided that this was the most appropriate implementation of the system, that sufficed for a proof of concept.

Table 7.1 shows the results of the motion module of the final algorithm. As can be seen, the average balanced accuracy was 95.94%, which fulfills requirement A.1. Furthermore, the average specificity was 91.88%. This meant that requirement A.2 was not fulfilled, since it was listed that the specificity should be 99%. However, it is important to note that these results only reflect the motion module of the algorithm. Due to technical difficulties, it was not possible to record an RBD episode with the patient wearing the sock and sleeping on the BedSense. Therefore, no data was collected from the patient through the sock system. If this was the case, threshold values could have been determined for the stress module, and the entire software system could have been fully implemented and tested.

Furthermore, in the design of this algorithm, data was only collected (through the BedSense) from one patient with RBD. However, ideally, the data would have been collected from multiple patients, aligning with requirement 1.1. Due to time constraints and a lack of contact with other RBD patients, this was not possible. However, to contextualize the results of the algorithm based on one patient, the motion module was tested on non-RBD patients. The results showed that the algorithm performed with a 97.05% specificity. This increase in specificity can be attributed to the fact that non-RBD patients make fewer erratic movements in their sleep.

Lastly, the performance metrics of the algorithm could have been substantially improved if machine learning were to be used. However, due to the limited and non-diverse data available at the start of the project, using machine learning algorithms posed a risk of bias. Since the requirement was for a versatile algorithm applicable to multiple patients, the decision was made to exclude machine learning from the approach.

8.2. Conclusion

The goal of this project was; to design and implement an algorithm that processes data from the sock system and the MomoMedical BedSense, detects restlessness leading up to episodes caused by RBD, and subsequently makes a decision on whether or not to bring the patient to a lighter sleep stage. In order to achieve this goal, various requirements were set in collaboration with the subgroup responsible for the sock system.

The requirements (and therefore the goal) were achieved by first collecting relevant data from various sources, such as the BedSense and the sock system developed by the hardware subgroup. The BedSense data provided useful information on the movement of a patient, whereas the sock system allowed for the extraction of biosignals such as PPG and EDA data, as well as complementary motion data. Following this, the data was pre-processed and visually analysed. As a result, indicators of upcoming RBD episodes were extracted. Finally, the algorithm was designed and implemented with; movement, out-of-bed, and stress detection components.

The average balanced accuracy of the software system was 95.94%, the average sensitivity was 100%, and the average specificity was 91.88%. These results mean that requirement A.1 was met, however, A.2 was not met. This is due to the fact that data was not collected from the sock system during an RBD episode, and thus the stress module has not been fully implemented yet. An attempt was made to collect data from the RBD patient; however, due to the complex WiFi network of the patient's apartment complex, it was not possible to collect data. Therefore, more data should be collected with the sock to potentially further develop it for use on patients.

Furthermore, requirement A.3 was not met, given that the hardware subgroup concluded that Low Energy Bluetooth did not have enough bandwidth for the system. Instead, WiFi communication was used for communication between the software system and the sock. The software system communicates in real-time with the BedSense through API requests, fulfilling requirement A.4. Requirement A.5 was also met, as the algorithm makes a decision every second and the WiFi connection with the sock has low latency.

Nevertheless, the software system demonstrated its ability to detect RBD episodes and make a decision on whether to bring a patient to a lighter sleep stage.

8.3. Future Work

The result of the designed algorithm seems promising. The algorithm can always accurately detect episodes. A finished product however would need to have improvements in both its design and implementation.

Enhancing Efficiency and Locality

The first aspect that would need to be improved is the fact that the algorithm is run on a laptop. This means that a laptop needs to be on for the whole night. Furthermore, the algorithm currently gets the BedSense data from Momo Medical servers using an API request. This is quite slow and results in a delay of roughly 5 seconds. It is possible to send the data from the BedSense directly to the laptop, but that would require a modified BedSense. It would be better if the algorithm could be run locally on either the BedSense, on the system the hardware group designed, or both.

Optimizing Algorithm Execution and Security

Running the algorithm on any of these devices would require it to be rewritten in a language that can be run on a microcontroller. This could be Cython or C++ among other programming languages. While not difficult per se, it would require some time. However, it would solve the issues mentioned above. A laptop would no longer be needed and the delay is greatly reduced, since the data no longer needs to be retrieved from a server, but can instead be received from the BedSense directly. Additionally, this would make the whole system more secure and private, since the data would not need to leave the device.

Improving Algorithm Generalization and Customizability

Another aspect that would need to be improved is that the algorithm is based on data from a single test subject. This has likely introduced some bias and therefore the algorithm may not work as well for other patients, since the threshold values are based on one patient. In order to improve this, data of multiple patients in different situations should be collected and the algorithm should be tested on it. Due to the different sleeping arrangements of patients, it seems likely that the algorithm will not work well in other situations. A system should be implemented that either automatically adjusts the threshold values or lets users customize the threshold values themselves (similar to a patient in the loop system). This should be done in such a way that it is user-friendly.

Machine Learning Algorithm

Ultimately, it may be concluded that a classic algorithm based on data from the BedSense does not work well enough to solve the issue. If that is the case, a machine learning algorithm should be tested. Machine learning algorithms can detect patterns in data that humans could never see. Therefore, these algorithms may provide higher accuracies than the algorithm described in this thesis. However, this would require a complete overhaul of many systems designed in this thesis and more diverse data.

References

[1] C. H. Schenck, B. Högl, and A. Videnovic, *Rapid-Eye-Movement Sleep Behavior Disorder*, 1st ed. Cham: Springer, 2019.

[2] A. K. Patel, V. Reddy, K. R. Shumway, and J. F. Araujo, *Physiology, Sleep Stages*. StatPearls Publishing, 2022.

[3] E. Suni and N. Vyas, *Stages of sleep: What happens in a sleep cycle*, May 2023. [Online]. Available: `https://www.sleepfoundation.org/stages-of-sleep`.

[4] L. Ferini-Strambi and M. Zucconi, "Rem sleep behavior disorder," *Clinical Neurophysiology*, vol. 111, S136–S140, 2000, Sleep and Epilepsy Supplement, ISSN: 1388-2457.

[5] J. F. Gagnon, M. A. Bédard, M. L. Fantini, *et al.*, "Rem sleep behavior disorder and rem sleep without atonia in parkinson's disease," *Neurology*, vol. 59, no. 4, pp. 585–589, 2002, ISSN: 0028-3878.

[6] I. Arnulf, "Rem sleep behavior disorder: Motor manifestations and pathophysiology," *Movement Disorders*, vol. 27, no. 6, pp. 677–689, 2012.

[7] J. V. Rundo and R. Downey, "Chapter 25 - polysomnography," in *Clinical Neurophysiology: Basis and Technical Aspects*, ser. Handbook of Clinical Neurology, K. H. Levin and P. Chauvel, Eds., vol. 160, Elsevier, 2019, pp. 381–392.

[8] E. K. St Louis and B. F. Boeve, "Rem sleep behavior disorder: Diagnosis, clinical implications, and future directions," *Mayo Clinic Proceedings*, vol. 92, no. 11, pp. 1723–1736, 2017.

[9] B. F. Boeve, "Rem sleep behavior disorder," *Annals of the New York Academy of Sciences*, vol. 1184, no. 1, pp. 15–54, 2010.

[10] R. N. Aurora, R. S. Zak, R. K. Maganti, *et al.*, "Best practice guide for the treatment of rem sleep behavior disorder (rbd)," *Journal of Clinical Sleep Medicine*, vol. 06, no. 01, pp. 85–95, 2010.

[11] A. Roguski, D. Rayment, A. L. Whone, M. W. Jones, and M. Rolinski, "A neurologist's guide to rem sleep behavior disorder," *Frontiers in Neurology*, vol. 11, Jul. 2020.

[12] N. Cooray *et al.*, "Enabling automated rem sleep behaviour disorder detection," *The 40th International Conference of the IEEE, EMBC*, Jul. 2018.

[13] A. B. Neikrug and S. Ancoli-Israel, "Diagnostic tools for rem sleep behavior disorder," *Sleep Medicine Reviews*, vol. 16, no. 5, pp. 415–429, Oct. 2012.

[14] M. J. Howell, P. A. Arneson, and C. H. Schenck, "A novel therapy for rem sleep behavior disorder (rbd)," *Journal of Clinical Sleep Medicine*, vol. 7, no. 6, pp. 639–644, Dec. 2011.

[15] M. Alaziz, Z. Jia, R. Howard, X. Lin, and Y. Zhang, "In-bed body motion detection and classification system," *ACM Transactions on Sensor Networks*, vol. 16, no. 2, Jan. 2020.

[16] X. Yang *et al.*, "Monitoring of patients suffering from rem sleep behavior disorder," *IEEE Journal of Electromagnetics, RF and Microwaves in Medicine and Biology*, vol. 2, no. 2, pp. 138–143, Jun. 2018.

[17] Zorggroep Oude en Nieuwe Land. "Slimme sensor verbetert de nachtrust bij bewoners 't Kompas." (2019), [Online]. Available: `https://zorggroep-onl.nl/slimme-sensor-verbetert-de-nachtrust-bij-bewoners-t-kompas` (visited on 06/01/2023).

[18] J. Kruize, T. Schram, and K. Dzhumageldyev, "Non-Invasive Monitoring and Prevention of RBD Episodes," 2023, The thesis of the hardware subgroup.

[19] L. Sörnmo and P. Laguna, "Chapter 3 - eeg signal processing," in *Bioelectrical Signal Processing in Cardiac and Neurological Applications*, ser. Biomedical Engineering, L. Sörnmo and P. Laguna, Eds., Burlington: Academic Press, 2005, pp. 55–179, ISBN: 978-0-12-437552-9.

[20] H. F. Posada-Quintero and K. H. Chon, “Innovations in electrodermal activity data collection and signal processing: A systematic review,” *Sensors*, vol. 20, no. 2, 2020, ISSN: 1424-8220.

[21] A. Greco, G. Valenza, J. Lázaro, *et al.*, “Acute stress state classification based on electrodermal activity modeling,” *IEEE Transactions on Affective Computing*, vol. 14, no. 1, pp. 788–799, 2023.

[22] W. Boucsein, *Electrodermal activity*. Springer Science & Business Media, 2012.

[23] A. Greco, G. Valenza, A. Lanata, E. P. Scilingo, and L. Citi, “Cvxeda: A convex optimization approach to electrodermal activity processing,” *IEEE Transactions on Biomedical Engineering*, vol. 63, no. 4, pp. 797–804, 2016.

[24] A. Temko, “Accurate heart rate monitoring during physical exercises using ppg,” *IEEE Transactions on Biomedical Engineering*, vol. 64, no. 9, pp. 2016–2024, 2017.

[25] J. Proakis and D. Manolakis, *Digital signal processing, principle, algorithms and applications*, 4th edition (Pearson international edition). Pearson prentice hall, 2000.

[26] J. Feriante and J. F. Araujo, *Physiology, REM Sleep*. StatPerals Publishing, 2023.

[27] P. van Gent, H. Farah, N. Nes, and B. Arem, “Analysing noisy driver physiology real-time using off-the-shelf sensors: Heart rate analysis software from the taking the fast lane project.,” Nov. 2018.

[28] T. Athaya and S. Choi, “An efficient fingertip photoplethysmographic signal artifact detection method: A machine learning approach,” *Journal of Sensors*, vol. 2021, pp. 1–18, Oct. 2021.

[29] G. Giannakakis, D. Grigoriadis, K. Giannakaki, O. Simantiraki, A. Roniotis, and M. Tsiknakis, “Review on psychological stress detection using biosignals,” *IEEE Transactions on Affective Computing*, vol. PP, pp. 1–1, Jul. 2019.

[30] P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python,” *Nature Methods*, vol. 17, pp. 261–272, 2020. DOI: `10.1038/s41592-019-0686-2`.

[31] D. Makowski, T. Pham, Z. J. Lau, *et al.*, “NeuroKit2: A python toolbox for neurophysiological signal processing,” *Behavior Research Methods*, vol. 53, no. 4, pp. 1689–1696, Feb. 2021.

[32] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: `10.1038/s41586-020-2649-2`. [Online]. Available: `https://doi.org/10.1038/s41586-020-2649-2`.

[33] T. pandas development team, *Pandas-dev/pandas: Pandas*, version latest, Feb. 2020. DOI: `10.5281/zenodo.3509134`. [Online]. Available: `https://doi.org/10.5281/zenodo.3509134`.

A
Data Collection on RBD Patient

Measurements have been taken with the sock on an RBD patient. This appendix consists of the resulting sock and BedSense measurements. The period from 02:25 to 02:40 will be discussed, because the patient noticed some slight arm movements at 02:29 and leg movements at 02:35. The PE measurements seen in Figure A.1 show vibrations around 02:29, which can be explained as the arm movements. The vibrations after 02:35 could be caused by the leg movements. The FSR data, shown in Figure A.2, indicates a fluctuating pressure around 02:36.

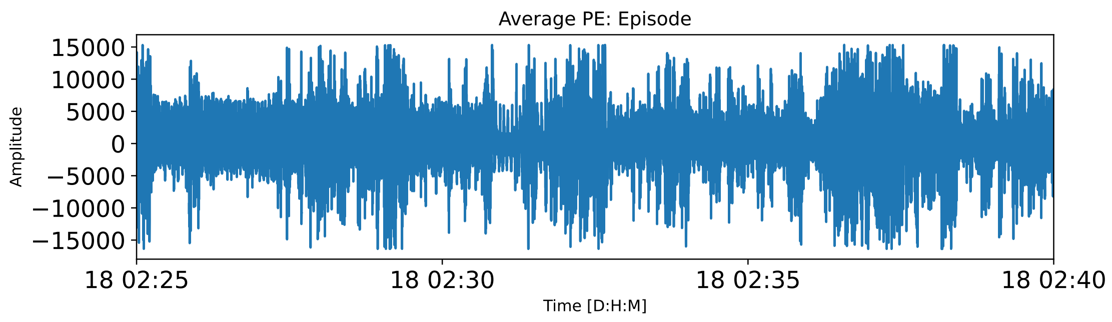

Figure A.1: PE data from the BedSense

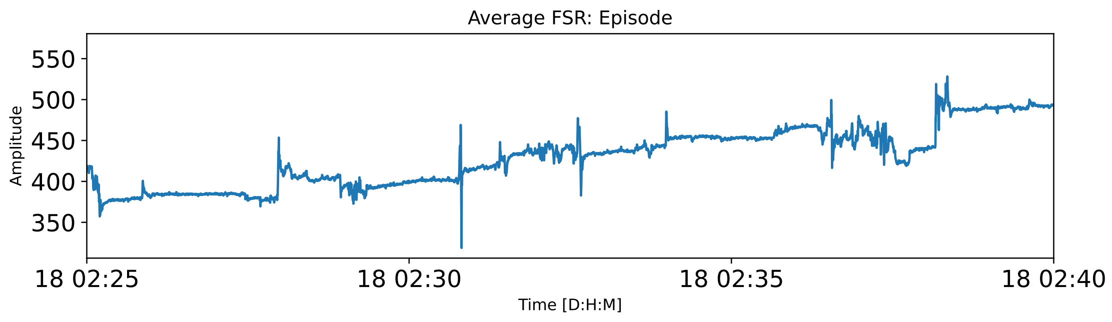

Figure A.2: FSR data from the BedSense

Looking at the accelerometer data from the sock in Figure A.3, both the leg and arm movements can be seen. There is a small peak around 02:29, concurrent with the arm movements. After 02:35 a bigger peak occurs, which shows leg movement. When comparing the accelerometer data with the BedSense data seen in Figures A.1 and A.2, it can be concluded that the sock does indeed provide more information on leg movement.

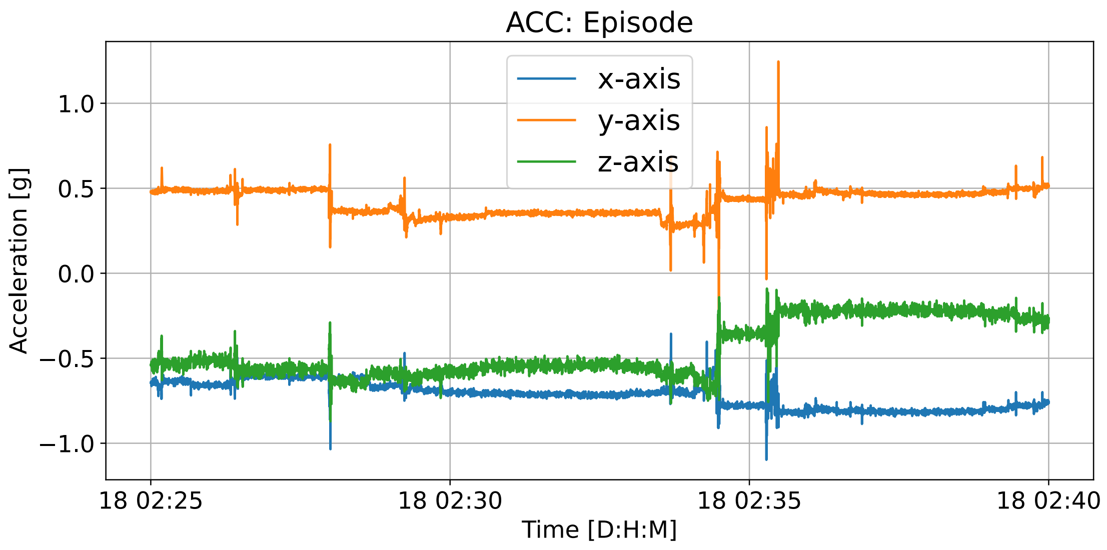

Figure A.3: Data from the accelerometer

In Figure A.4, the raw PPG data is presented. This has been processed and turned into an estimated heart rate, shown in Figure A.5. It shows no abnormalities around the time of the arm movements, but at the same time of the leg movement the heart rate does seem to be higher for a while. After the leg movements the estimated heart rate reaches around 140 bpm, this is likely to be a false outlier.

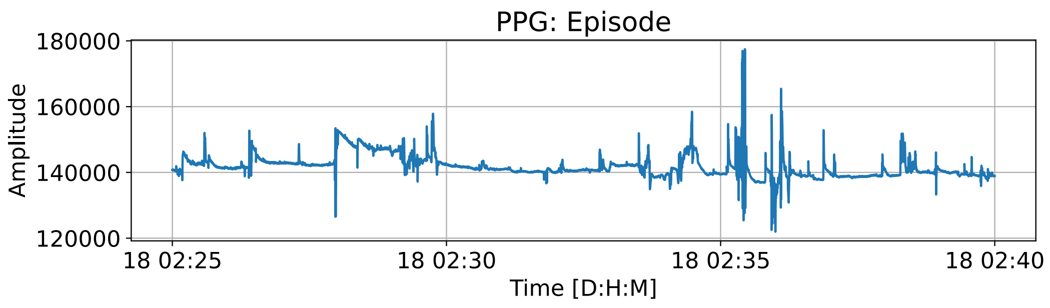

Figure A.4: Data from the PPG sensor

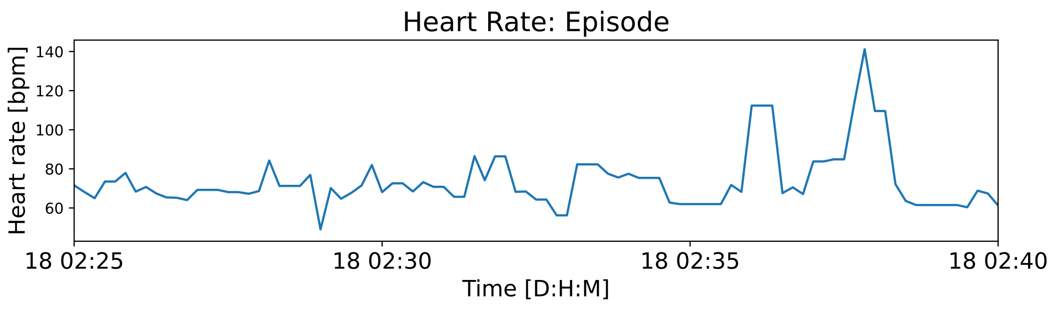

Figure A.5: Extracted heart rate

The EDA data can be seen in Figure A.6. At the same time of the leg movement the value of the raw EDA data drops suddenly, this could be caused by the sensors being moved. But after EDA values gradually start increasing, which could be an indicator of stress.

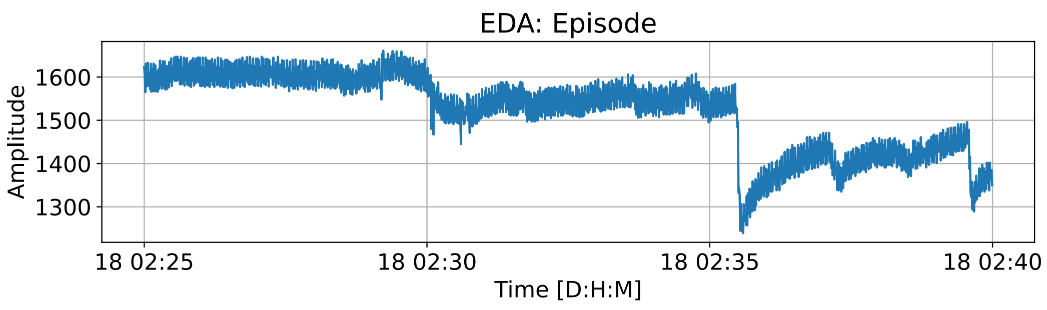

Figure A.6: Data from the EDA sensor

B
RBD Diagnostic Criteria

B.1. International Classification of Sleep Disorders, Third Edition

Table B.1: *International Classification of Sleep Disorders, Third Edition* (ICSD-3). REM Sleep Behavior Disorder Diagnostic Criteria [1].

Criteria A-D must be met	
A.	Repeated episodes of sleep-related vocalization and/or complex motor behaviors
B.	These behaviors are documented by polysomnography to occur during REM sleep or, based on clinical history of dream enactment, are presumed to occur during REM sleep
C.	Polysomnographic recording demonstrates REM sleep without atonia (RWA)
D.	The disturbance is not better explained by another sleep disorder, mental disorder, medication or substance use

B.2. The American Academy of Sleep Medicine Manual for the Scoring of Sleep and Associated Events

Table B.2: The American Academy of Sleep Medicine (AASM) Manual for the Scoring of Sleep and Associated Events. Scoring Polysomnographic Features of REM Sleep Behavior Disorder (RBD) [1].

1. Score in accordance with the following definitions
Sustained muscle activity (tonic activity) in REM sleep: An epoch of REM sleep with at least 50% of the duration of the epoch having a chin EMG amplitude greater than the minimum amplitude demonstrated in NREM sleep
Excessive transient muscle activity (phasic activity) in REM sleep: In a 30 s epoch of REM sleep divided into ten sequential 3 s mini-epochs, at least five (50%) of the mini-epochs contain bursts of transient muscle activity. In RBD, excessive transient muscle activity bursts are 0.1-5.0 s in duration and at least four times as high in amplitude as the background EMG activity
2. The polysomnographic characteristics of RBD are characterized by EITHER or BOTH of the following features:
(a) Sustained muscle activity in REM sleep in the chin EMG (b) Excessive transient muscle activity during REM in the chin or limb EMG

B.3. Diagnostic and Statistical Manual of Mental Disorders

Table B.3: *Diagnostic and Statistical Manual of Mental Disorders, Fifth Edition* (DSM-5). Rapid Eye Movement sleep Behavior Disorder Diagnostic Criteria (Only the Core Features are Reported) [1].

A.	Repeated episodes of arousal during sleep associated with vocalization and/or complex motor behaviors
B.	These behaviors arise during rapid eye movement (REM) sleep and therefore usually occur more than 90 min after sleep onset, are more frequent during the later portions of the sleep period and uncommonly occur during daytime naps
C.	Upon awakening from these periods, the individual is completely awake, alert and not confused or disoriented
D.	Either of the following: 1. REM sleep without atonia in polysomnographic recording 2. A history suggestive of REM sleep behavior disorder and established synucleinopathy diagnosis (e.g. Parkinson's disease, multiple system atrophy)

C
Patients Record of RBD Episodes

Table C.1: Excel sheet with information on episodes kept by patient

Date	Timestamp episode	Woke up	Woken up	Comments
21-02-2023 to 22-02-2023	00:22	Yes	No	–
01-03-2023 to 02-03-2023	03:38	Yes	No	–
03-03-2023 to 04-03-2023	01:57	Yes	Yes	Kicking and punching
04-04-2023 to 05-04-2023	02:24	No	No	–
04-04-2023 to 05-04-2023	03:17	No	No	–
10-04-2023 to 11-04-2023	03:46	No	No	No notification in bedsense
15-04-2023 to 16-04-2023	01:12	No	No	–
25-04-2023 to 26-04-2023	02:53	No	No	–
27-04-2023 to 28-04-2023	01:42	No	No	–
04-05-2023 to 05-05-2023	01:26	No	No	–
04-05-2023 to 05-05-2023	04:48	No	No	–
06-05-2023 to 07-05-2023	01:04	No	No	–
12-05-2023 to 13-05-2023	05:16	No	No	–
18-05-2023 to 19-05-2023	02:34	No	No	–
18-05-2023 to 19-05-2023	02:51	No	No	–
19-05-2023 to 20-05-2023	03:00	No	No	–
23-05-2023 to 24-05-2023	04:05	No	No	–
25-05-2023 to 26-05-2023	01:37	No	No	–

D

Source Code

D.1. algorithmrealtime.py

This is the final algorithm for real time detection.

```python
import heartpy as hp
import neurokit2 as nk2
import numpy as np
import pandas as pd
import time
import pytz
import datetime

from scipy.signal import iirnotch, filtfilt
from software.MomoAPI.tsdb import MomoTSDB

# Class for the entire detection mechanism. Main method is algorithm().
class DetectionRealTime:

    # Initialization method
    def __init__(self, state):
        # Parameters of algorithm
        self.time_length = 1
        self.round = 0.4
        self.countdown = 0  # amount of time length fragments for out of bed cooldown
        self.state = state

        # Constants
        self.fsr_fs = 10
        self.pe_fs = 120
        self.eda_fs = 25
        self.ppg_fs = 50
        self.acc_fs = 50

        # Thresholds
        self.a = 1
        self.b = 1
        self.c = 1
        self.d = 1
        self.e = 1

        self.fsr_power = None
        self.pe_power = None
        self.acc_std = None

        self.heart_rate = None
        self.eda_onset_time = None
        self.eda_onset_amplitude = None

        self.out_of_bed_status = False
        self.tsdb = MomoTSDB(env="develop")

    # Importing the Bedsense data
    def importdata_Bedsense(self) -> list:

        # Initializing datetime objects for API request
        begin = (datetime.datetime.strptime(
            (datetime.datetime.now() - datetime.timedelta(seconds=7)).strftime('%Y-%m-%d %H:%
M:%S'),
            '%Y-%m-%d %H:%M:%S'))
        end = (datetime.datetime.strptime(
            (datetime.datetime.now() - datetime.timedelta(seconds=2)).strftime('%Y-%m-%d %H:%
M:%S'),
            '%Y-%m-%d %H:%M:%S'))

        # API  request
        df = self.tsdb.load_range(begin, end, device_id="11025036", tz=pytz.timezone("Europe/
Amsterdam"),
                                  measurement="sensorOpt")

        df_fsr = df[["time", "fsr_0", "fsr_1", "fsr_2", "fsr_3"]].copy()

        # Initialize a new dataframe
        df_pe_raw = pd.DataFrame()

        # Merge the raw pe data into 6 lists, each corresponding to one sensor
        for i in range(6):
            df_pe_raw[f"pe_{i}"] = df.values[:, i * 12 + 9:(i + 1) * 12 + 9].tolist()

        # Explode the dataframe, such that the data in the lists are put into new rows.
        df_pe_raw = df_pe_raw.explode(["pe_0", "pe_1", "pe_2", "pe_3", "pe_4", "pe_5"],
ignore_index=True)

        return [df_fsr, df_pe_raw]

    #Importing the sock data
    def importdata_Sock(self):

        return

    # Run the live algorithm
    def run(self) -> None:
        last_time = time.time()

        while True:
            if time.time() - last_time >= 1:
                if self.out_of_bed_status:
                    self.countdown = 1200
                elif self.countdown > 0:
                    self.countdown -= 1

                bs_df = self.importdata_Bedsense()
                if self.state == 'total':
                    sock_df = self.importdata_Sock()
                else:
                    sock_df = []
                output = self.algorithm(bs_df, sock_df)
                if output == 1:
                    print("bzzzt")
                else:
                    print("negative")

                last_time = time.time()

    # Algorithm method
    def algorithm(self, bs_df: list[pd.DataFrame], sock_df: list[pd.DataFrame]) -> int:
        fsr_df = bs_df[0]
        pe_df = bs_df[1]

        self.signal_energy(fsr_df, pe_df)
        if self.state == 'total':
            eda_df = sock_df[0]
            ppg_df = sock_df[1]
            acc_df = sock_df[2]
            self.process_acc(acc_df)
            self.process_eda(eda_df)
            self.process_ppg(ppg_df)

        return self.decision()

    # Final decision method
    def decision(self) -> int:
        if self.state == 'bedsense':
            if self.fsr_power > self.a and self.pe_power > self.b:
                return 1
            else:
                return 0

        speed = self.eda_onset_amplitude / self.eda_onset_time

        if self.countdown == 0:
            if self.fsr_power > self.a and self.pe_power > self.b and self.acc_std > self.c:
                if speed > self.d and self.heart_rate > self.e:
                    return 1

        return 0

    # out of bed detection method
    def out_of_bed(self, fsr_data: pd.DataFrame) -> None:
        average = 0

        for j in range(4):
            f = fsr_data[f"fsr_{j}"]
            average += np.average(f)

        self.out_of_bed_status = bool((average/4 < 100))

        return

    # function calculates signal_energy of all sensors
    def signal_energy(self, fsr_df: pd.DataFrame, pe_df: pd.DataFrame) -> None:
        fsr_temp = 0
        pe_temp = 0

        for j in range(4):
            fsr_temp += sum(fsr_df[f"fsr_{j}"][-121:-1] ** 2)
        for j in range(6):
            pe_temp += sum(pe_df[f"pe_{j}"][-121:-1] ** 2)

        self.fsr_power = fsr_temp
        self.pe_power = pe_temp

        return

    # rounding methods
    def rounding(self, value: float) -> int:
        if value > self.round:
            return 1
        else:
            return 0

    # processing methods
    def process_eda(self, eda_df: pd.DataFrame) -> None:

        signals, info = nk2.eda_process(eda_df, self.eda_fs, method_phasic="smoothmedian")

        # Calculating onset time and onset amplitude
        onset_indices = []
        peak_indices = []

        for i, v in enumerate(signals["SCR_Onsets"]):
            if v == 1:
                onset_indices.append(i)

        for i, v in enumerate(signals["SCR_Peaks"]):
            if v == 1:
                peak_indices.append(i)

        # Calculate average onset time
        self.eda_onset_time = np.diff([onset_indices, peak_indices], axis=0) / len(
    onset_indices)

        phasic_data = signals["EDA_Phasic"]
        total = 0

        for i in range(len(onset_indices)):
            peak = peak_indices[i]
            onset = onset_indices[i]

            total += phasic_data[peak] - phasic_data[onset]

        self.eda_onset_amplitude = total / len(onset_indices)

    def process_ppg(self, ppg_df: pd.DataFrame) -> None:
        # Filter the window
        filtered_window = hp.filter_signal(ppg_df, cutoff=[0.4, 4], sample_rate=self.ppg_fs,
    order=4,
                                           filtertype="bandpass")
        # Extract data from the filtered window
        _, measures = hp.process(filtered_window, self.ppg_fs)

        self.heart_rate = measures["bpm"]

    def process_acc(self, acc_df: pd.DataFrame) -> None:
        data = [None, None, None]

        data[0] = self.filter_acc(acc_df["acc_x"])
        data[1] = self.filter_acc(acc_df["acc_y"])
        data[2] = self.filter_acc(acc_df["acc_z"])

        total = np.sum(data, axis=0)

        self.acc_std = np.std(total)

        return

    def filter_acc(self, data: pd.DataFrame) -> np.ndarray:
        b, a = iirnotch(50, 30, self.acc_fs)

        return filtfilt(b, a, data)
```

D.2. algorithm.py

This is an adaptation of the algorithm, not including the accelerometer, PPG, and EDA signal. Made for testing purposes.

```
import numpy as np
import pandas as pd
from software.detection.classic_algorithm.data_windowing import data_window


# Class for the entire detection mechanism. Main method is algorithm().
class Detection:

    # Initialization method
    def __init__(self) -> None:
        # Parameters of algorithm
        self.time_length = 1
        self.moving_size = 10
        self.round = 0.7
        self.countdown = 2400  # amount of time length fragments for out of bed cooldown
        self.bed = 100

        # Constants
        self.fsr_fs = 10
        self.pe_fs = 120
        self.fsr_n_points = int(self.fsr_fs * self.time_length)
        self.pe_n_points = int(self.pe_fs * self.time_length)

        # To be used
        self.pe_results = []
        self.fsr_results = []
        self.power = []

        self.fsr_window = None
        self.pe_window = None

    # Algorithm method
    def algorithm(self, df: list[pd.DataFrame]) -> list[int]:
        fsr_df = df[0]
        pe_df = df[2]

        # window is iterable with values
        self.fsr_window = self.data_window(fsr_df, 'fsr')
        self.pe_window = self.data_window(pe_df, 'pe')

        # processing pe and fsr
        self.signalpower()

        # Checking the processing outputs
        if len(self.pe_results) != len(self.fsr_results):
            raise ValueError("lengths not the same")

        # Averaging the processing results and rounding them to a 0 or a 1
        final = [self.rounding(i) for i in np.divide(self.power, 2)]

        # out of bed detection applied
        output = self.out_of_bed_filter(final, self.fsr_window)

        return self.moving_average(output)

    # moving average method
    def moving_average(self, output: list) -> list[int]:
        new_output = []
        average = []

        for bit in output:
            if len(average) < self.moving_size:
                average.append(bit)
                new_output.append(0)
            elif len(average) == self.moving_size:
                average = average[1::]
                average.append(bit)
                if average.count(1) >= 2:
                    new_output.append(1)
                else:
                    new_output.append(0)

        return new_output

    # out of bed detection method
    def out_of_bed(self, fsr_data: data_window) -> list[int]:
        output = []

        for frame in fsr_data:
            average = 0

            for j in range(4):
                f = frame[f"fsr_{j}"]
                average += np.average(f)

            output.append(average / 4)
        return [1 if x < self.bed else 0 for x in output]

    # out of bed filter: filters the output to give 0 when recently out of bed
```

```python
    def out_of_bed_filter(self, data: list[int], fsr_window: data_window) -> list[int]:
        n = len(data)

        bedstate = self.out_of_bed(fsr_window)

        indeces = [i for i, element in enumerate(bedstate) if element == 1]
        count = 0

        for i, index in enumerate(indeces):
            if i == 0:
                count += 1

            if (indeces[i-1] + 1) != index:
                count += 1

        print("seconds of out of bed: " + str(len(indeces)))
        print("amount of out of bed: " + str(count))

        for index in indeces:
            for i in range(-self.countdown, self.countdown+1):
                if -1 < index + i < n - 1:
                    data[index + i] = 0

            if n != len(data):
                raise ValueError("length changed!!")

        return data

    #function calculates average signal energy of PE and FSR sensors
    def signal_energy(self) -> None:
        min = 99999

        for value_fsr, value_pe in zip(self.fsr_window, self.pe_window):
            temp = 0
            temp2 = 0

            for j in range(4):
                temp2 += np.average(value_fsr[f"fsr_{j}"])
                temp += sum(np.abs(value_fsr[f"fsr_{j}"]) ** 2)
            self.fsr_results.append(temp)
            temp = 0

            if (temp2/4) < min:
                min = temp2/4

            for j in range(6):
                temp += sum(np.abs(value_pe[f"pe_{j}"]) ** 2)
            self.pe_results.append(temp)

        self.pe_results = np.divide(self.pe_results, max(self.pe_results))
        self.fsr_results = np.divide(self.fsr_results, max(self.fsr_results))
        self.power = [element + self.fsr_results[n] for n, element in enumerate(self.
    pe_results)]

        #self.bed = min + 15
        return


    # Creating objects of class data window
    def data_window(self, data: pd.DataFrame, control: str) -> data_window:
        if control == 'fsr':
            return data_window(data, self.fsr_n_points)
        elif control == 'pe':
            return data_window(data, self.pe_n_points)

    # Rounding function
    def rounding(self, value: float) -> int:
        if value > self.round:
            return 1
        else:
            return 0
```

D.3. test_algorithm_episodes.py

This code was used to test the motion module (without the accelerometer). For clarity only one of the tests has been shown, the other tests are the same but run different data.

```python
import unittest
from software.data_analysis.helper_episodes import episodes
from software.data_analysis.data_import import import_data
from software.TestingScripts.helper_test_functions import comparelists_episode, helper_list
from software.detection.classic_algorithm.Algorithm import Detection
import numpy as np


class TestSuite(unittest.TestCase):
    time_length = 1
    # Runs data of 10 episodes (20-40 min segments)
    strive_percentage = (100, 100, 100)
    detection = Detection()

    # tests the percentage of 1's that are not in the period
    def test_specific_episode_1_1(self):
        episode = episodes[0]
        data = import_data(episode[0], episode[1], "00:15", "06:00")
        detection = Detection()
        output = detection.algorithm(data[1:])
        del detection
        del data
        strive_output = helper_list(TestSuite.time_length, "00:15", "06:00", episode[0],
    episodes)
        if len(output) <= len(strive_output):
            output = np.append(output, np.zeros(len(strive_output) - len(output)))
        else:
            output = output[:len(strive_output)]
        self.assertEqual(len(output), len(strive_output))
        percentage = comparelists_episode(output, strive_output)
        print(percentage)
        self.assertGreater(percentage, TestSuite.strive_percentage)
```

D.4. helper_episodes.py

This is the labelling that matches specific csv files of Bedsense data.

```python
#format: startdate, enddate, starttime, endtime, Bert's Timestamp, Our Estimate, endep
episodes = [("20230221", "20230222", "00:05", "00:27", "00:22:00", "00:22:00", "00:24:30"),
            ("20230301", "20230302", "03:25", "03:43", "03:38:00", "03:38:00", "03:42:00"),
            ("20230303", "20230304", "01:43", "02:03", "01:57:00", "01:57:00", "02:01:00"),
            ("20230404", "20230405", "02:10", "02:30", "02:20:00", "02:18:40", "02:20:30"),
            ("20230404", "20230405", "03:00", "03:23", "03:18:00", "03:16:00", "03:19:00"),
            ("20230410", "20230411", "03:30", "03:50", "03:46:00", "03:41:20", "03:44:40"),
            ("20230415", "20230416", "00:55", "01:17", "01:12:00", "01:12:00", "01:15:00"),
            ("20230425", "20230426", "02:40", "02:58", "02:53:00", "02:53:00", "02:56:00"),
            ("20230427", "20230428", "01:30", "01:48", "01:42:00", "01:42:00", "01:46:00"),
            ("20230504", "20230505", "01:10", "01:31", "01:26:00", "01:24:50", "01:25:30"),
            #("20230504", "20230505", "03:00", "03:21", "03:16:00", "03:16:00", "03:17:00"),
            ("20230504", "20230505", "04:30", "04:53", "04:48:00", "04:47:30", "04:53:00"),
            ("20230506", "20230507", "00:44", "01:09", "01:04:00", "00:59:00", "01:02:00"),
            ("20230512", "20230513", "05:00", "05:20", "05:16:00", "05:15:10", "05:20:00"),
            ("20230518", "20230519", "02:25", "02:40", "02:34:00", "02:34:10", "02:34:10"),
            ("20230518", "20230519", "02:44", "02:56", "02:51:00", "02:51:10", "02:51:30"),
            ("20230519", "20230520", "02:50", "03:05", "03:00:00", "03:00:00", "03:00:40"),
            #("20230522", "20230523", "04:30", "04:44", "04:39:00", "04:39:00", "01:02:00"),
            ("20230523", "20230524", "04:00", "04:10", "04:05:00", "04:05:10", "04:05:35"),
            ("20230525", "20230526", "01:30", "01:42", "01:37:00", "01:36:40", "01:37:45")]
            #("20230531", "20230601", "04:00", "04:18", "04:13:00", "04:13:00", "01:02:00"),
            #("20230601", "20230602", "01:50", "02:03", "01:57:00", "01:57:00", "01:02:00")]

baselines = [("20230220", "20230221", "00:00", "00:30", None, None),
            ("20230220", "20230221", "00:30", "01:00", None, None),
            ("20230220", "20230221", "01:30", "02:00", None, None),
            ("20230220", "20230221", "02:30", "03:00", None, None),
            ("20230220", "20230221", "03:30", "04:00", None, None)]
```

D.5. helper_test_functions.py

These functions are supporting functions for the testing environment.

```python

# Metrics function
def comparelists_episode(predicted: list[int], expected: list[int]) -> tuple[float, float |
    int, float]:
    TP = 0
    FP = 0
    TN = 0
    FN = 0
    episodes = []
    episode = False

    for n, elem in enumerate(predicted):
        if expected[n] == 2:
            episode = False
        if expected[n] == 1 and episode is False:
            print("added")
            episodes.append(0)
            episode = True

        if elem == 1:
            if expected[n] == 1:
                episodes[len(episodes)-1] = 1 #True positive
            elif expected[n] != 2:
                FP += 1
        else:
            if expected[n] == 1:
                FN += 1
            elif expected[n] != 2:
                TN += 1

    # Sensitivity
    # TPR = TP / (TP + FN)
    if len(episodes) > 0:
        TPR = sum(episodes)/len(episodes)
    else:
        TPR = 0

    # Specificity
    TNR = TN / (TN + FP)

    # Balanced accuracy
    Accuracy = ((TPR + TNR) / 2) * 100

    return round(Accuracy, 2), TPR, round(TNR*100, 2)


# Calculates length of output
def len_baseline(time_length: float, starttime: str, endtime: str) -> int:
    hourdif = int(endtime[0:2]) - int(starttime[0:2])
    mindif = int(endtime[3:5]) - int(starttime[3:5])
    secdif = 0

    secdif += (hourdif * 3600) + (mindif * 60)

    length = int(secdif / time_length)

    return length

# creates desired output
def helper_list(time_length: int, starttime: str, endtime: str, startdate: str, episodes:
    list) -> list[int]:
    # timelength is seconds.
    startep = []
    endep = []

    for episode in episodes:
        if episode[0] == startdate:
            startep.append(episode[5])
            endep.append(episode[6])

    hourdif = int(endtime[0:2]) - int(starttime[0:2])
    mindif = int(endtime[3:5]) - int(starttime[3:5])
    secdif = 0

    secdif += (hourdif * 3600) + (mindif * 60)

    totlength = int(secdif / time_length)

    time = []
    hour = int(starttime[0:2])
    minute = int(starttime[3:5])
    second = 0
    logic = 0
    output = []
    count = 0

    for n in range(totlength):
        if (f"{hour:02}" + ':' + f"{minute:02}" + ':' + f"{second:02}") in startep:
            logic = 1
        elif (f"{hour:02}" + ':' + f"{minute:02}" + ':' + f"{second:02}") in endep:
            logic = 0
            count = 0

        if n != 0:
            second += time_length

        if second == 60:
            second = 0
            minute += 1
        elif second > 60:
            second = second % 60
            minute += 1

        if minute == 60:
            minute = 0
            hour += 1
        elif minute > 60:
            minute = minute % 60
            hour += 1

        if hour == 24:
            hour = 0

        time.append((hour, minute, second))

        if count == 10:
            logic = 2 #two means dont care
            count = 0

        if logic == 1:
            count += 1



        output.append(logic)

    return output
```

D.6. NonRBDtest.py

This script has been used to test the algorithm, seen in Appendix D.2, on data from non-RBD patients.

```
from software.detection.classic_algorithm.Algorithm import Detection
from software.TestingScripts.helper_test_functions import comparelists_episode, helper_list
import numpy as np
from datetime import datetime
import os
```

```python
import pandas as pd
import time
from datetime import timedelta

def import_data(start_date: str, end_date: str, patient_nr: int, start_time="22:00", end_time
    ="10:00") -> list[pd.DataFrame, pd.DataFrame, pd.DataFrame, pd.DataFrame]:
    """
    Imports data from a specific day, possibly within a specified hour range.
    Default hour range is 22:00 to 10:00, this may be changed depending on the
    sleep schedule of the patient.

    TODO change start_time and end_time according to the sleep schedule of the
    patient, using the 'in bed' function from MOMO.

    Arguments:
    date : str
        Determines which day of data is imported. Format: ymd
    start_time : str, optional
        Determines the start time of the data that is imported. Default is "22:00".
    end_time : str, optional
         Determines the end time of the data that is imported. Default is "10:00".

    Returns:
    df_angle : pd.Dataframe
        Dataframe with angle and time data
    df_fsr : pd.Dataframe
        Dataframe with fsr and time data
    df_pe_std : pd.Dataframe
        Dataframe with pe_std and time data
    df_pe_raw : pd.Dataframe
        Dataframe with pe_raw and time data
    """

    # print(__file__)
    # print(os.path.join(os.path.dirname(__file__), "../../data.csv"))
    path = os.path.join(os.path.dirname(__file__), f"../../data/nonrbd/{patient_nr}/{
    start_date}_{end_date}.csv")
    # path = f"../../data/Momo_Medical_R&D_Bert_Verzijl_{start_date}_{end_date}.csv"

    #Same day? Or two days?
    start = start_time.replace(':', '')
    end = end_time.replace(':', '')
    if (int(start) < int(end)):
        start_date = str(int(start_date) + 1)



    df = pd.read_csv(path)

    start = round(time.mktime((datetime.strptime(start_date, "%Y%m%d") + timedelta(hours=int(
    start_time[:2]), minutes=int(start_time[3:5]))).timetuple()) * 1000)
    end = round(time.mktime((datetime.strptime(end_date, "%Y%m%d") + timedelta(hours=int(
    end_time[:2]), minutes=int(end_time[3:5]))).timetuple()) * 1000)

    # print(start)
    # print(type(pd.to_datetime(start * 1000000)))
    # print(end)
    # print(pd.to_datetime(end * 1000000))

    # Drop rows when the timestamp is outside the defined time range
    df.drop(df[(df.time < np.float64(start)) | (df.time > np.float64(end))].index, inplace=
    True)
    df.reset_index(drop=True, inplace=True)

    # Create new dataframes with only the angle, fsr and pe_std data
    df_angle = df[["time", "angle_0", "angle_1", "angle_2"]].copy()
    df_fsr = df[["time", "fsr_0", "fsr_1", "fsr_2", "fsr_3"]].copy()
    df_pe_std = df[["time", "pe_std_0", "pe_std_1", "pe_std_2", "pe_std_3", "pe_std_4", "
    pe_std_5"]].copy()

    # Initialize a new dataframe
    df_pe_raw = pd.DataFrame()

    # Merge the raw pe data into 6 lists, each corresponding to one sensor
    for i in range(6):
      df_pe_raw[f"pe_{i}"] =  df.values[:, i * 12 + 9:(i + 1) * 12 + 9].tolist()

    # Explode the dataframe, such that the data in the lists are put into new rows.
    df_pe_raw = df_pe_raw.explode(["pe_0", "pe_1", "pe_2", "pe_3", "pe_4", "pe_5"],
     ignore_index=True)

    # Get first and last timestamp from the dataframe
    t0 = df["time"][0]
    t1 = df["time"].iloc[-1]

    # Create the timestamps for the new upsampled dataframe
    time_col = np.linspace(t0, t1, num=len(df_pe_raw.index)).astype(np.int64)

    # Add the new timestamps to the dataframe
    df_pe_raw["time"] = time_col.tolist()


    return [df_angle, df_fsr, df_pe_std, df_pe_raw]

episodes = (("20230607", "20230608"),
            ("20230608", "20230609"),
            ("20230609", "20230610"),
            ("20230610", "20230611"),
            ("20230611", "20230612"))
save = []

for number in range(7):
    print(f"=====================PATIENT {number}=====================")
    for episode in episodes:
        detection = Detection()


        time_length = 1
        i = 15
        data = import_data(episode[0], episode[1], number, "00:15", "06:00")
        output = detection.algorithm(data[1:])
        strive_output = [0]*len(output)
        del data

        percentage = comparelists_episode(output, strive_output)
        print("specificity " + f"({episode[1]}):" + str(percentage[2]))
        save.append(percentage[2])

    print(f"average specificity ({number}): " + str(sum(save)/len(save)))
    print("\n\n")
```

D.7. data_import.py

This function was used to extract data out of csv files.

```python
import os
import time
import numpy as np
import pandas as pd
from datetime import timedelta

import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.dates as dates

from datetime import datetime

def import_data(start_date: str, end_date: str, start_time="22:00", end_time="10:00") -> list
    [pd.DataFrame, pd.DataFrame, pd.DataFrame, pd.DataFrame]:
    """
    Imports data from a specific day, possibly within a specified hour range.
    Default hour range is 22:00 to 10:00, this may be changed depending on the
    sleep schedule of the patient.

    TODO change start_time and end_time according to the sleep schedule of the
    patient, using the 'in bed' function from MOMO.

    Arguments:
    date : str
        Determines which day of data is imported. Format: ymd
    start_time : str, optional
        Determines the start time of the data that is imported. Default is "22:00".
    end_time : str, optional
         Determines the end time of the data that is imported. Default is "10:00".

    Returns:
    df_angle : pd.Dataframe
        Dataframe with angle and time data
    df_fsr : pd.Dataframe
        Dataframe with fsr and time data
    df_pe_std : pd.Dataframe
        Dataframe with pe_std and time data
    df_pe_raw : pd.Dataframe
        Dataframe with pe_raw and time data
    """

    # print(__file__)
    # print(os.path.join(os.path.dirname(__file__), "../../data.csv"))
    path = os.path.join(os.path.dirname(__file__), f"../../data/Momo_Medical_R&
    D_Bert_Verzijl_{start_date}_{end_date}.csv")
    # path = f"../../data/Momo_Medical_R&D_Bert_Verzijl_{start_date}_{end_date}.csv"

    #Same day? Or two days?
    start = start_time.replace(':', '')
    end = end_time.replace(':', '')
    if (int(start) < int(end)):
        start_date = str(int(start_date) + 1)



    df = pd.read_csv(path)

    start = round(time.mktime((datetime.strptime(start_date, "%Y%m%d") + timedelta(hours=int(
    start_time[:2]), minutes=int(start_time[3:5]))).timetuple()) * 1000)
    end = round(time.mktime((datetime.strptime(end_date, "%Y%m%d") + timedelta(hours=int(
    end_time[:2]), minutes=int(end_time[3:5]))).timetuple()) * 1000)

    # print(start)
    # print(type(pd.to_datetime(start * 1000000)))
    # print(end)
    # print(pd.to_datetime(end * 1000000))

    # Drop rows when the timestamp is outside the defined time range
    df.drop(df[(df.time < np.float64(start)) | (df.time > np.float64(end))].index, inplace=
    True)
    df.reset_index(drop=True, inplace=True)

    # Create new dataframes with only the angle, fsr and pe_std data
    df_angle = df[["time", "angle_0", "angle_1", "angle_2"]].copy()
    df_fsr = df[["time", "fsr_0", "fsr_1", "fsr_2", "fsr_3"]].copy()
    df_pe_std = df[["time", "pe_std_0", "pe_std_1", "pe_std_2", "pe_std_3", "pe_std_4", "
    pe_std_5"]].copy()

    # Initialize a new dataframe
    df_pe_raw = pd.DataFrame()

    # Merge the raw pe data into 6 lists, each corresponding to one sensor
    for i in range(6):
      df_pe_raw[f"pe_{i}"] =  df.values[:, i * 12 + 9:(i + 1) * 12 + 9].tolist()

    # Explode the dataframe, such that the data in the lists are put into new rows.
```

```python
    df_pe_raw = df_pe_raw.explode(["pe_0", "pe_1", "pe_2", "pe_3", "pe_4", "pe_5"],
     ignore_index=True)

    # Get first and last timestamp from the dataframe
    t0 = df["time"][0]
    t1 = df["time"].iloc[-1]

    # Create the timestamps for the new upsampled dataframe
    time_col = np.linspace(t0, t1, num=len(df_pe_raw.index)).astype(np.int64)

    # Add the new timestamps to the dataframe
    df_pe_raw["time"] = time_col.tolist()


    return [df_angle, df_fsr, df_pe_std, df_pe_raw]

data = import_data("20230220", "20230221", "00:20", "00:23")
```

D.8. data_main.py

This script runs the visualization of the data.

```python
from data_visualization import visualize_data
from helper_episodes import baselines, episodes

def plot_episodes():
    for i, ep in enumerate(episodes):
        print(f"Processing episode {i}")

        start_date = ep[0]
        end_date = ep[1]
        start_time = ep[2]
        end_time = ep[3]
        ep_time = [ep[4], ep[5]]

        visualize_data(start_date, end_date, ep_time,
                       ("fsr", "time", "average", "std"),
                       ("fsr", "time_freq", "average"),
                       ("fsr", "freq_sum"),
                       ("pe", "time", "average", "std"),
                       ("pe", "time_freq", "average"),
                       ("pe", "freq_sum", "average"),
                       start_time=start_time, end_time=end_time)

        print(f"Finished processing episode {i}")

def plot_baselines():
    for i, ba in enumerate(baselines):
        print(f"Processing baseline {i}")

        start_date = ba[0]
        end_date = ba[1]
        start_time = ba[2]
        end_time = ba[3]
        ep_time = [ba[4], ba[5]]

        visualize_data(start_date, end_date, ep_time,
                       ("fsr", "time", "average", "std"),
                       ("fsr", "time_freq", "average"),
                       ("fsr", "freq_sum"),
                       ("fsr", "power_spec"),
                       ("pe", "time", "average", "std"),
                       ("pe", "time_freq", "average"),
                       ("pe", "freq_sum"),
                       ("pe", "power_spec"),
                       start_time=start_time, end_time=end_time)

        print(f"Finished processing episode {i}")


```

```python

if __name__ == "__main__":
    plot_episodes()
    # plot_baselines()
```

D.9. data_visualization.py

This script contains all functions related to the visualization of data.

```python
import os

import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.dates as dates

from datetime import datetime
from scipy import signal

from data_import import import_data
from helper_episodes import episodes

def visualize_data(start_date: str, end_date: str, ep_time: list[str], *plot_options, **
    time_options) -> None:
    # Check if start_time and/or end_time is/are defined, else use default times
    start_time = time_options["start_time"] if "start_time" in time_options else "22:00"
    end_time = time_options["end_time"] if "end_time" in time_options else "10:00"

    # Import data
    df = import_data(start_date, end_date, start_time=start_time, end_time=end_time)

    angle_df = df[0]
    fsr_df = df[1]
    pe_std_df = df[2]
    pe_raw_df = df[3]

    if ep_time[0] is not None:
        for i, t in enumerate(ep_time):
            # Get day depending on the hour
            day = end_date if int(t[:2]) < 12 else start_date

            # Put the date and time into a single string
            dt = f"{day} {t}"

            ep_time[i] = datetime.strptime(dt, "%Y%m%d %H:%M:%S")

    sensor_data = {"angle": angle_df, "fsr": fsr_df, "pe_std": pe_std_df, "pe": pe_raw_df}

    functions = {"time": plot_time, "time_freq": plot_time_freq, "freq_sum": plot_freq_sum,
                 "power_spec": plot_power_spec_dens, "distribution": plot_distribution}

    if len(plot_options) == 0:
        raise ValueError("No plot options defined.")
    else:
        for sensor, func, *options in plot_options:
            functions[func](sensor_data[sensor], sensor, ep_time, options)

def plot_time(data: pd.DataFrame, sensor: str, ep_time: list[datetime], options) -> None:
    """
    Produce a time plot of a specified sensor and save it as a svg

    Arguments:
    data : pd.DataFrame
        Data to be plotted.
    sensor : str
        The name of the sensor.
    ep_time : list
    options : tuple
        Plot options, used to for example plot the average or
        the standard deviation.
    """

    # Calculate time axis
    time = [datetime.fromtimestamp(int(x) / 1000) for i, x in enumerate(data['time'])]

    # Get date from time axis
    date = str(pd.to_datetime(time[0]))[:10]
    ep = str(ep_time[1])[11:]

    # Create the directory in which the plots will be saved
    dir = make_dir(date, ep, sensor, "time")

    # Create a list of the sensor columns
    sensors = [f"{sensor}_{i}" for i in range(len(data.columns) - 1)]

    for i, label in enumerate(sensors):
        # Initialize the plot and its size
        plt.figure(figsize=(15, 3))
        # Plot the data of the sensor
        plt.plot(time, data[sensors[i]])

        if ep_time[1] != ep_time[0]:
            # Plot the estimated start of the episode
            plt.axvline(x=ep_time[1], color="green", linestyle=":", linewidth="2", label="
    Estimation")
        if ep_time[0] is not None:
            # Plot the timestamp of the patient
            plt.axvline(x=ep_time[0], color="red", linestyle=":", linewidth="2", label="
    Timestamp Patient")

        # Limit the x-axis
        plt.xlim(time[0], time[-1])

        # Limit the y-axis depending on the sensor type
        if sensor == "pe":
            plt.ylim(-15000, 15000)
        elif sensor == "fsr":
            plt.ylim(0, 2500)

        # Label the axis
        plt.xlabel("Date and Time [d h:m]")
        plt.ylabel("Amplitude")
        plt.title(f"{sensor.upper()}_{i} Data from episode on {date} at {ep}")

        # Add a legend if required
        if ep_time[0] is not None:
            plt.legend(loc="upper right")

        # Save the figure
        plt.tight_layout()
        plt.savefig(f"{dir}/{sensor}_{i}.svg")
        plt.close()

    for option in options:
        i += 1

        plt.figure(figsize=(15, 3))

        if option == "average":
            # Calculate the average
            average = data[sensors].mean(axis=1)

            # Plot the average and add a title
            plt.plot(time, average)
            plt.title(f"Average of {sensor.upper()} Data from episode on {date} at {ep}")
        elif option == "std":
            # Calculate the standard deviation
            std = data[sensors].std(axis=1)

            # Plot the standard deviation and add a title
            plt.plot(time, std)
            plt.title(f"Standard Deviation of {sensor.upper()} Data from episode on {date} at {ep}")
        else:
            raise ValueError(f"The given option: {option} is not allowed.")

        if ep_time[1] != ep_time[0]:
            # Plot the estimated start of the episode
            plt.axvline(x=ep_time[1], color="green", linestyle=":", linewidth="2", label="Estimation")
        if ep_time[0] is not None:
            # Plot the timestamp of the patient
            plt.axvline(x=ep_time[0], color="red", linestyle=":", linewidth="2", label="Timestamp Patient")

        # Limit the x-axis
        plt.xlim(time[0], time[-1])

        # Limit the y-axis depending on the sensor type
        if sensor == "pe":
            plt.ylim(-15000, 15000)
        elif sensor == "fsr":
            plt.ylim(0, 2500)

        # Label the axis
        plt.xlabel("Date and Time [d h:m]")
        plt.ylabel("Amplitude")

        # Add a legend if required
        if ep_time[0] is not None:
            plt.legend(loc="upper right")

        # Save the figure
        plt.tight_layout()
        plt.savefig(f"{dir}/{option}.svg")
        plt.close()


def plot_time_freq(data: pd.DataFrame, sensor: str, ep_time: list[datetime], options) -> None:
    """
    Produce a time-frequency plot of a specified sensor and save it as a svg

    Arguments:
    data : pd.DataFrame
        Data to be plotted.
    sensor : str
        The name of the sensor.
    ep_time : list
    options : tuple
        Plot options, used to for example plot the average or
        the standard deviation.
    """

    # Calculate time axis
    time = [datetime.fromtimestamp(int(x) / 1000) for i, x in enumerate(data['time'])]

    # Get date from time axis
    date = str(pd.to_datetime(time[0]))[:10]
    ep = str(ep_time[1])[11:]

    # Determine sample frequency
    fs = 120 if sensor == "pe" else 10

    # Create the directory in which the plots will be saved
    dir = make_dir(date, ep, sensor, "time_freq")

    # Create a list of the sensor columns
    sensors = [f"{sensor}_{i}" for i in range(len(data.columns) - 1)]

    sxx_list = []

    for i, label in enumerate(sensors):
        # Initialize the plot and its size
        plt.figure(figsize=(10,5))

        # Calculate time-frequency
        f, _, Sxx = signal.spectrogram(data[label], fs, mode="magnitude")
        Sxx = Sxx.astype("float64")
        sxx_list.append(Sxx)

        # Calculating correct time axis
        if i == 0:
            t = pd.date_range(time[0], time[-1], len(Sxx[0]))

        # Plot the data
        plt.pcolormesh(t, f, Sxx, cmap="viridis", edgecolors=None, rasterized=True)

        if ep_time[1] != ep_time[0]:
            # Plot the estimated start of the episode
            plt.axvline(x=ep_time[1], color="yellow", linestyle=":", linewidth="1", label="
    Estimation")
        if ep_time[0] is not None:
            # Plot the timestamp of the patient
            plt.axvline(x=ep_time[0], color="red", linestyle=":", linewidth="1", label="
    Timestamp Patient")

        # Limit the y-axis depending on the sensor type
        if sensor == "pe":
            plt.ylim(0, 20)

        # Label the axis
        plt.xlabel("Date and Time [d h:m]")
        plt.ylabel("Frequency [Hz]")
        plt.title(f"{sensor.upper()}_{i} Data from Episode on {date} at {ep}")

        # Add a legend if required
        if ep_time[0] is not None:
            plt.legend(loc="upper right")

        # Save the figure
        plt.tight_layout()
        plt.savefig(f"{dir}/{sensor}_{i}.svg")
        plt.close()

    for option in options:
        # Initialize the plot and its size
        plt.figure(figsize=(10,5))

        if option == "average":
            # Calculate the average
            average = np.sum(np.array(sxx_list), axis=0) / len(sensors)

            # Plot the average and add a title
            plt.pcolormesh(t, f, average, cmap="viridis", edgecolors=None, rasterized=True)
            plt.title(f"Average of {sensor.upper()} Data from Episode on {date} at {ep}")
        else:
            raise ValueError(f"The given option: {option} is not allowed.")

        if ep_time[1] != ep_time[0]:
            # Plot the estimated start of the episode
            plt.axvline(x=ep_time[1], color="yellow", linestyle=":", linewidth="1", label="
    Estimation")
        if ep_time[0] is not None:
            # Plot the timestamp of the patient
            plt.axvline(x=ep_time[0], color="red", linestyle=":", linewidth="1", label="
    Timestamp Patient")

        # Limit the y-axis depending on the sensor type
        if sensor == "pe":
            plt.ylim(0, 20)

        # Label the axis
        plt.xlabel("Date and Time [d h:m]")
        plt.ylabel("Frequency [Hz]")

        # Add a legend if required
        if ep_time[0] is not None:
            plt.legend(loc="upper right")

        # Save the figure
        plt.tight_layout()
        plt.savefig(f"{dir}/{option}.svg")
        plt.close()

def plot_freq_sum(data: pd.DataFrame, sensor: str, ep_time: list[datetime], options) -> None:
    """
    Produce a frequency sum plot of a specified sensor and save it as a svg

    Arguments:
    data : pd.DataFrame
        Data to be plotted.
    sensor : str
        The name of the sensor.
    ep_time : list
    options : tuple
        Plot options, used to for example plot the average or
        the standard deviation.
    """

    # Calculate time axis
    time = [datetime.fromtimestamp(int(x) / 1000) for i, x in enumerate(data['time'])]

    # Get date from time axis
    date = str(pd.to_datetime(time[0]))[:10]
    ep = str(ep_time[1])[11:]

    # Determine sample frequency
    fs = 120 if sensor == "pe" else 10

    # Create the directory in which the plots will be saved
    dir = make_dir(date, ep, sensor, "freq_sum")

    # Create a list of the sensor columns
    sensors = [f"{sensor}_{i}" for i in range(len(data.columns) - 1)]

    sum_list = []

    for i, label in enumerate(sensors):
        # Initialize the plot and its size
        plt.figure(figsize=(15, 3))

        # Calculate time-frequency
        _, __, Sxx = signal.spectrogram(data[label], fs, mode="magnitude")
        Sxx = Sxx.astype("float64")

        # Sum the frequency components
        f_sum = Sxx.sum(axis=0)
        sum_list.append(f_sum)

        # Calculating correct time axis
        if i == 0:
            t = pd.date_range(time[0], time[-1], len(Sxx[0]))

        plt.plot(t, f_sum)

        if ep_time[1] != ep_time[0]:
            # Plot the estimated start of the episode
            plt.axvline(x=ep_time[1], color="green", linestyle=":", linewidth="2", label="
    Estimation")
        if ep_time[0] is not None:
            # Plot the timestamp of the patient
            plt.axvline(x=ep_time[0], color="red", linestyle=":", linewidth="2", label="Timestamp Patient")

        # Limit the x-axis
        plt.xlim(time[0], time[-1])

        # Label the axis
        plt.xlabel("Date and Time [d h:m]")
        plt.ylabel("Amplitude")
        plt.title(f"{sensor.upper()}_{i} Data from Episode on {date} at {ep}")

        # Add a legend if required
        if ep_time[0] is not None:
            plt.legend(loc="upper right")

        # Save the figure
        plt.tight_layout()
        plt.savefig(f"{dir}/{sensor}_{i}.svg")
        plt.close()

    for option in options:
        # Initialize the figure and its size
        plt.figure(figsize=(15, 3))

        if option == "average":
            # Calculate the average
            average = np.sum(np.array(sum_list), axis=0) / len(sensors)

            # Plot the average and add a title
            plt.plot(t, average)
            plt.title(f"Average of {sensor.upper()} Data from Episode on {date} at {ep}")
        else:
            raise ValueError(f"The given option: {option} is not allowed.")

        if ep_time[1] != ep_time[0]:
            # Plot the estimated start of the episode
            plt.axvline(x=ep_time[1], color="green", linestyle=":", linewidth="2", label="Estimation")
        if ep_time[0] is not None:
            # Plot the timestamp of the patient
            plt.axvline(x=ep_time[0], color="red", linestyle=":", linewidth="2", label="Timestamp Patient")

        # Limit the x-axis
        plt.xlim(time[0], time[-1])

        # Label the axis
        plt.xlabel("Date and Time [d h:m]")
        plt.ylabel("Frequency [Hz]")

        # Add a legend if required
        if ep_time[0] is not None:
            plt.legend(loc="upper right")

        # Save the figure
        plt.tight_layout()
        plt.savefig(f"{dir}/{option}.svg")
        plt.close()

# ----------- Currently not used ----------- #

def plot_power_spec_dens(data: pd.DataFrame, sensor: str, ep_time: list[datetime], options) -> None:
    # Calculate time axis
    time = [datetime.fromtimestamp(int(x) / 1000) for i, x in enumerate(data['time'])]

    # Get date from time axis
    date = str(pd.to_datetime(time[0]))[:10]

    # Determine sample frequency
    fs = 120 if sensor == "pe" else 10
```

```python

    # Determine the number of graphs
    n_graphs = len(data.columns) - 1
    n_graphs += len(options)

    cols = 2
    rows = int(np.ceil(n_graphs / cols))

    sensors = [f"{sensor}_{i}" for i in range(len(data.columns) - 1)]

    fig, ax = plt.subplots(rows, cols, figsize=(5 * cols + 2.5, 5 * rows))

    sum_list = []

    for i, label in enumerate(sensors):
        row = i // 2
        col = i % 2

        f, p_dens = signal.welch(data[label], fs)
        p_dens = np.abs(p_dens).astype("float64")

        sum_list.append(p_dens)

        ax[row, col].plot(f, 10 * np.log10(p_dens))
        ax[row, col].set_title(label.upper(), fontsize=20)

    for option in options:
        if col == 1:
            col = 0
            row += 1
        else:
            col = 1

        if option == "average":
            average = np.sum(np.array(sum_list), axis=0) / len(sensors)

            ax[row, col].plot(f, 10 * np.log10(average))
            ax[row, col].set_title(f"Average of {sensor.upper()}", fontsize=20)
        else:
            raise ValueError(f"The given option: {option} is not allowed.")

    if ep_time[0] is None:
        dir = "baseline"
    else:
        dir = "episode"

    fig.supxlabel("Frequency [Hz]")
    fig.supylabel("Power Spectral Density [dBW/Hz]")
    fig.suptitle(f"Power Spectral Density of {sensor.upper()} Data from {date}", fontsize=30)

    plt.tight_layout()
    plt.savefig(f"figures/{dir}/power_spec_{sensor}_{date}_min-{str(ep_time[1])[14:16]}.svg")
    plt.close()

# ----------- Currently not used ----------- #

def plot_distribution(data: pd.DataFrame, sensor: str, ep_time: list[datetime], options) ->
    None:
    sensors = [f"{sensor}_{i}" for i in range(len(data.columns) - 1)]

    for i, label in enumerate(sensors):
        sns.violinplot(data[label])
        plt.xlabel("")
        plt.ylabel("")
        plt.title(f"Distribution of {sensor}_{i}")

        plt.savefig()

    plt.show()


```

```python
def make_dir(date: str, ep_time: str, sensor: str, plot_type: str) -> str:
    """
    Function that makes a directory if it does not exist yet

    Arguments:
    date : str
    ep_time : str
    sensor : str
    plot_type : str

    Returns:
    dir : str
    """
    ep = ep_time.replace(":", "")

    dir = f"figures/{date}/{ep}/{sensor}/{plot_type}"

    if not os.path.exists(dir):
        os.makedirs(dir)

    return dir
```