Contents

i

Stream based respiration analysis

Cato Danielsen

August 15, 2016

Preface

Preface

Abstract

The prevalence of sleep disorders has been a wildly debated topic the last century. As these types of disorder do not have unambiguous symptoms, it can be very difficult to diagnose.

This thesis proposes a stream based respiration analysis to decompose a respiratory sensor signal into low level events which can be used by a separate system to detect symptoms. The solution uses adaptive windows to read information from a data stream and the analysis is based on the PhysioNet application puka. The application puka relies on an decrepit library for communication between Java and MATLAB which has to be modernized.

The system has been tested using synthesised data as well as real physiological signals found in PhysioBank to validate the respiration analysis. The results from the stream based solution is compared with the original version as well as compared to the ground truth of the synthesised signals.

The algorithms used in puka shows great promise for detecting certain types of respiratory events, such as peaks and troughs, but we will have to make changes to the underlying algorithm to make it viable for real world signals which contain different types of noise.

Part I

Introduction and background

Chapter 1

Introduction

A good nights sleep is important in order to stay physically and mentally healthy. Research has shown that the lack of proper sleep can be linked to many health issues.

According to the National Institute of Health (USA) sleep apnea, if left untreated, can lead to different health risks. Among these are increased risk of high blood pressure, heart attack, stroke, obesity, diabetes, heart failure, increased chance of irregular heartbeats and increased chance of having work-related or driving accidents [24]. Other literature has for a long time pointed out the risk of mental health issues related to sleep apnea [27], such as depression.

According to the literature the estimated prevalence of sleep apnea is 2%-4% of the middle aged adult population in USA[75]. One thing we find as a broad consensus in the literature, is that a lot of sleep apnea patients go undiagnosed, as much as 80% to 90%, depending on the criteria for diagnosis.

The clinical term sleep apnea was introduced in 1973 by after the first international symposium on "Hypersomnia with Periodic Breathing" in 1972 [16]. The terms sleep apnea syndrome and obstructive sleep apnea was coined in 1976. Over the last 40 years we have seen an increase in interest and concern over the effects of sleep disorders and it has been discovered to be a more common medical problem than previously assumed.

1.1 Motivation

The most common way and the gold standard of detecting sleep disorders is with a *polysomnography* (PSG) that requires a patient to sleep with monitoring equipment in a sleep lab. A PSG can also be referred to as a sleep study and it monitors a variety of parameters in order to diagnose sleep disorders. These parameters are described further in Section 2.1.

An important question is: if we already have a accurate and precise way of detecting and diagnosing sleep disorders, why are so many occurrences of sleep related disorders undiagnosed? According to the literature there are several key factors as to why these cases go undiagnosed and untreated. Some of the symptoms associated with sleep disorders, such as excessive daytime sleepiness, daytime irritability, difficulty of concentration and waking with headaches, can be ambiguous and it is difficult for a doctor to identify a sleep disorder based only on symptoms observable in a consultation. The symptoms can be vague and ambiguous and the threshold for recommending a costly, overnight procedure without having clear indications that it is a sleep disorder causing the symptoms can be difficult to justify for the clinical staff. As it is not always clear whether symptoms are caused by sleep disorders other more easily diagnosed alternatives are explored first. The overnight PSG requires technology, personnel, dedication and experience.

This is a recognized problem and attempts have been made to create pre-screening tools in order to detect sleep disorders. This can be done by using mobile devices with their built in sensors such as smart phones, or using custom made home usage device such as home PSGs or other sensors that monitor parameters that can indicate sleep disorders, such as respiration rate, blood oxygen levels, heart rate, body movement or other relevant metrics.

Also, a patient with a sleep disorder will not yield the same result for each PSG recording, as a patients sleep pattern can change from night to night. It would be even more costly to have a patient spend multiple nights in a sleep laboratory for several tests in order to determine the exact extent of the sleep disorder. This

also brings us into the problem of sleep quality during the sleep study. A PSG requires multiple electrodes connected to a patient which can cause the patient to not be able to fall asleep or give a false or imprecise impression on the sleeping pattern of the patient.

Even if a PSG is accurate (the current gold standard for sleep related measurements), the threshold for doctors to order a PSG is relatively high due to the cost and effort required to do a complete PSG. This makes the need for non intrusive pre-screening tools in order to clinically diagnose the cases of sleep disorders. If a patient can with minimal effort take a test without the use of intrusive sensors and in their own home, closer to a normal nights sleep it might be easier to justify a more thorough examination.

Figure 1.1: Equipment used in a PSG

1.2   Problems caused by sleep disorders

Sleep disorders cause large expenses for the society and the individual affected. Kapur et.al.[30] estimates out that due to the

prevalence and the serious adverse consequences of sleep apnea, the disorder can cause as much as $3.4 billion in addition medical costs in the U.S.

In addition to the financial impact, other studies suggest that sleep disorder can have an impact on not only the physiological state of a person, but also have on psychosocial aspects of a persons life, such as school performance, family relation, work problems, and other social impairments[27]. The physiological effects are vary in severity and type. We will look closer at these effects in Chapter 2.

1.3 Non intrusive senors

In order to create a system that can detect sleep disorders without the need for overnight stay at a sleep laboratory or the presence of clinical personnel, we will look into the use of non intrusive sensors.

By sensor we are talking about a device or multiple devices coupled together, able to detect bio markers, such as respiration stops, in order to indicate sleep disorders. Sensor technology will be described in Section 2.3.

The quality of being non intrusive is that the patient is not hampered of put in physical discomfort by the sensor, as they would have with a sensor that require electrodes or a mask or other probes that might cause discomfort. Whether a sensor is intrusive or not is not well defined, but varies based on different parameters. If we have a sensor that requires the user to sleep with a elastic band around their chest, this might not be seen as an intrusive sensor for a healthy person as they have no problem attaching and wearing the sensor. But for a person with limited mobility, the act of attaching the sensor might prove to be a considerable inconvenience.

1.4   TRIO

This thesis is aimed to become a part of the ongoing project TRIO. The project is a collaboration between the Distributed MultiMedia Systems (DMMS) and Nano Electronics (NANO) research groups, both a part of the Institute of Informatics (IFI) at University of Oslo (UIO), The Intervention Centre at Oslo University Hospital (OUS), and Novelda AS. The project description states that the main goal is to develop systems based on non invasive sensors that can be used in a home environment to identify parameters indicating the need of medical intervention.

One such parameters is respiration. Respiration signals can be used to indicate acute health related problems, but can also combined with knowledge about the wakeful state of a patient help diagnose sleep disorders.

1.5   Problem statement

If we can obtain a *respiratory signal* from a *non invasive sensor* and detect *low level events* that describe useful changes in the signal, we can then pass these on to a separate system, such as TRIO, to do analysis in event space. By analysing the respiration signal outside the TRIO system, we free up resources, as the signal processing required for a data stream with raw data from a sensor is significantly larger than from a stream with events.

For this thesis we attempt to identify and adapt software and algorithms found in existing work for deriving respiratory information from physical sensors. From the data generated from the sensors we detect the low level events that is in turn transmitted to a analysis system.

1.6   Approach

The first step is to identify and analyse existing software to find a solution that have the functionality we need.

- **Find** and **analyse** existing software

- Identify useful **parameters** detected by the software

- **Modify** if necessary

- **Adapt** (convert results into **events**) for *real-time* use and **evaluate** viability

1.7   Structure

To understand the symptoms we are trying to detect, we take a look at the physiological background for sleep disorders in Chapter 2. We also take a look at the types of diagnostic tools are used for determining these types of illness.

Chapter 3 takes a brief look at what solutions exist that can be used by this thesis to analyse respiratory signals. We analyse the application and algorithm we find in preparation for the modification described in Chapter 4 and Chapter 5. We make changes to modernize and adapt the found solutions in Chapter 4 while Chapter 5 introduces the new contributions.

The evaluation in Chapter 6, where we test and validate the adaptation and new contributions. The results from these experiments are discussed in Chapter 8.

Chapter 2

Background

The system described in our problem statement will make use of *sensors* in order to detect *sleep disorders*. The *sensors* captures physical phenomena and converts it into *signals*, that we in turn *process* into *events* for TRIO. This chapter explain some of the underlying concepts for such a system.

2.1 Sleep Apnea Syndrome

Sleep Apnea Syndrome (SAS) is sleep disorder characterized by the disruption of airflow during sleep. SAS is often divided into one of three sub diagnosis, Obstructive Sleep Apnea (OSA), Central Sleep Apnea (CSA), and Mixed Sleep Apnea (MSA), also know as Complex Sleep Apnea.

All diagnosis have in common either total stop or a reduction of respiration with a subsequent decrease in blood oxygen levels. The cause of these respiration reductions is what defines the type of SAS. An apnea event is the name for a complete stop of respiration for at least 10 seconds, while a hypopnea event is defined as an at least 10 seconds reduction in ventilation of at least 50% of normal airflow during sleep[39]. When the blood oxygen level is reduced the body is aroused from sleep in order to resume normal breathing. The arousal from normal sleep reduces the sleep

quality.

2.1.1 Obstructive Sleep Apnea

Pathogenesis

OSA is also known as Obstructive Sleep Apnea/Hypopnea Syndrome (OSAHS), due to the occurrence of both apneic and hypopneic events. In OSA the upper airway (UA) passage is either completely or partially blocked. There are multiple structural or anatomic factors that have been discovered to cause UA blockage, and these blockages occur in the pharynx. The pharynx is the area where the nasal and oral cavity meet and it has both the digestive, speech and respiratory functions in human anatomy. The pharynx area consists of muscles and soft tissue and it is necessary to be able to collapse and close the UA for digestive and speech purposes while awake. The negative pressure created by the inspiration process can cause the soft tissue region to collapse, causing blockage.

Normal Breathing Pathway   Blocked Breathing Pathway

Figure 2.1: Obstructive Sleep Apnea

There are also genetic factors as some have smaller airways that also can contribute to the lack of airflow. Nasal obstruction can lead to mouth breathing, which predisposes to abnormal airway dynamics that favors not only pharyngeal collapse but also what is called backward displacement of the tongue. The soft tissue of the tongue can cause UA blockage.

In addition to the soft tissue risk factors, the bone structure of the jaw region can be positioned in such a way that the tongue is predisposed to be pulled back into the pharynx during sleep during sleep stages with decreased muscle tone.

The factors that can increase the risk of UA blockage makes OSA difficult to predict and diagnose.

Epidemiology

Patients with anatomical vulnerability are considered to be more susceptible to developing OSA[57, 9]. These vulnerabilities can be enlarged tonsils, recessed mandible, small upper airway, impaired retrolingual airway among others. Each of these case is not a clear indication of OSA, but can be a contributing factor. Other factors that increase vulnerability for OSA include age, obesity, menopause, sleep hygiene, and certain health behaviors such as cigarette smoking and alcohol use[56].

Hypertension, also known as high blood pressure, is an often reported co-morbidity of OSA[11]. During the lowered blood oxygen levels experienced during an apnea or hypopnea event results in increased activity in the autonomic nervous system in order to increase the oxygen level. The literature suggests that as much as 50% of OSA patients suffers hypertension even during wakefulness[62, 51].

OSA has also been linked as a risk factor for cardiovascular diseases, stroke, abnormal glucose metabolism, insulin resistance, and diabetes mellitus [56, 67]. Cerebrovascular diseases and OSA have been pointed out to have a bi-directional relationship[16], and as a result of the hypertension and reduced cerebral blood flow the risk for carebrovascular diseases such as stroke is increased.

As Fusetti points out, *the common association of OSAS with hypertension and obesity in general population makes it difficult to separate their respective independent role in the long-term cardiovascular and metabolic consequences associated with OSAS*[13].

2.1.2 Central Sleep Apnea

Pathogenesis

While obstructive apnea is caused by blockage of the airways, a central apnea is the complete stop of respiratory effort as a consequence of imbalance within the brains control of the respiratory effort, described as a loss of ventilatory control[72]. While instability in the upper airway leads to obstructive sleep apnea, the imbalance of ventilatory control can lead to both obstructive and central sleep apnea.

Epidemiology

CSA can manifest in two broad categories according to the wakefulness CO_2 levels. Hypercapnic and nonhypercapnic. Hypercapnic is defined as elevated CO_2 levels in the blood. Patients often exhibit some degree of daytime hypercapnea and this condition is often worsen during sleep. Two patterns are often used to classify hypercapnic: impaired central drive ("won't breathe") and impaired respiratory motor control ("can't breathe")[8].

Impaired central drive can be caused by physiological factors that diminish ventilatory function, but has also been linked to genetic factors without anatomic pathology. Opioid-based medication have for a long time been pointed out to have a respiratory depressant effect[71].

Impaired respiratory motor control can experience CSA due to abnormalities in the signaling of the respiratory system. It can be caused by a wide range of neuromuscular disorders that causes some stage of the signaling process to not be able work properly.

Cheyne–Stokes breathing is a nonhypercapnic breathing pattern that is most commonly observed in patients with congestive heart failure and left ventricular systolic dysfunction[8]. During Cheyne–Stokes the patient increases the breathing rate gradually in a crescendo/decrescendo pattern broken up by apneic events. Arousal typically occurs mid-cycle at the peak of ventilatory effort

11

rather than at the cessation of apnea.

2.1.3 Mixed/Complex Sleep Apnea

Pathogenesis

As defined by Guilleminault, Tilkian and Dement in 1976, *mixed apnea is defined by cessation of airflow and an absence of respiratory effort early in the episode, followed by resumption of unsuccessful respiratory effort in the latter part of the episode* [15]. This diagnosis is a combination of central and obstructive sleep apnea. In some cases when the respiration effort stops as a result of CSA, the pharynx region is collapsed due to the lack of pressure, so when the body is aroused into resuming breathing efforts it is still completely or partially blocked.

Epidemiology

These episodes of central apneas followd by airway collapse and obstructive apneas and hypopneas are considered to be multifactorial. Obesity and/or snoring has been linked as a contributing factor for developing mixed apnea in CSA patients as the increased risk of high passive airways which leads to higher susceptibility for airway collapse[7]. The same article also points out mixed apnea in in patients that are administered chronic doses of opioid medications.

As this diagnosis is a combination of Central and Obstructive sleep apnea, many of the same health effects can be found.

2.2 Diagnosis

Hypopneic and apneic events are common symptoms of sleep apnea, and in order to diagnose the different conditions. Respiratory

Disturbance Index (RDI) is often used in sleep studies, but it includes other disturbances other than hypopneic and apneic events. This calls for a more specialized scale to diagnose sleep apnea.

2.2.1 AHI

Apnea–Hypopnea Index (AHI) is a commonly used index for the severity of sleep disturbances during the course of the total sleep time of a patient. The AHI usually refers to the number of events per hour of sleep. The number of events can be used to measure a severity score, where:

0-4	Normal
5-14	Mild
15-29	Moderate
30 or more	Severe

Table 2.1: AHI severity scale

In order to calculate the AHI we use the number of apneic and hypopneic events per hour

$$AHI = (Hypopneas + apneas) * 60/TotalSleepTime(minutes)$$

The AHI combined with daytime symptoms, such as EDS, dry mouth or headaches when waking up, is the basis of diagnosis for sleep apnea.

The first indication that often warrants the sleep study is the daytime symptoms, but according to the literature there are patients without any associated clinical symptoms (asymptomatic apnea). The literature suggests that the effect of these asymptomatic patients still suffer altered heart rate during daytime without symptoms or co-morbidities[3].

As the name implies, AHI counts both apnea and hypopnea events and is very useful for OSA detection, since a patient suffering from OSA can exhibit both apnea and hypopnea events.

There are several different non intrusive ways of indicating a diagnosis of sleep disorders. Questioners such as the Berlin Questioner, STOP BANG and Epworth Sleepiness Scale (ESS) are used in order to screen for and discover the usual symptoms of sleep disorders. One example of a study using the Berlin Questioner (BQ) and Epworth Sleepiness Scale (ESS) is *A Norwegian population-based study on risk and prevalence of obstructive sleep apnea*[19] where it was used to make an estimate on the prevalence of OSA in the Norwegian population. These questioners help researchers to estimate the prevalence of OSA, but for a clinical diagnosis a physical examination such as a sleep study is needed.

2.2.2 PSG

In order to detect sleep disorders in patients, we need to monitor certain physiological parameters of the patient in order to classify the type of As mentioned in Section 1.1 the gold standard for sleep disorder diagnosis is the polysomography (PSG) or sleep study.

The function of PSG is monitoring of a patient during sleep using an array of medical equipment that is simultaneously recorded. The types of parameters depend on the type of PSG used. As there are at least `number of sleep disorders` types of sleep disorders diagnosed by sleep studies, variations on what types of signals recorded is classified by different types of PSG. According to AAST (American Association of Sleep Technologists) the standard PSG has the following parameters[47]:

With electrodes:	
EEG	Electroencephalogram monitors the electrical activity in the brain.
EOG	Electrooculogram measures eye movement.
EMG	Chin Electromyogram monitors level of muscle tone around the chin area.
ECG	Electrocardiogram monitors the heart rhythm
Respiration	recorded from the movement of electrodes
Other sensors:	
Audio	Upper Airway Sound Recording
Thermistor or Inductive Respiratory Plethysmograph (RIP)	Respiratory effort and flow
Limb EMG	Limb Movement and Body Position

The EEG documents wakefulness, arousals and sleep stages during the sleep study, which is important in order to know whether symptoms occur while the patient is sleeping and at which sleep stage it occurs. Sleep stages are often classified into five separate stages; 1, 2, 3, 4 and REM (rapid eye movement), or into REM and nonREM stages.

- In stage 1, muscle activity slows down, the eyes move slowly and the subject drift in and out of sleep.

- In stage 2 the brain waves becomes slower and the eye movement halts.

- In stage 3 the brain waves becomes very slow with occasional smaller, faster waves.

- In stage 4 the brain almost exclusively produces the same slow brain waves as in stage 3.

Stage 3 and 4 are referred to as delta sleep, which is the namesake of the extremely slow brain waves (delta waves) found in these stages. During delta sleep there is no muscle activity or eye movement. During REM sleep breathing becomes more rapid and irregular, eyes move rapidly and limb muscles are temporarily paralyzed. The brainwaves during REM sleep increase to an

activity level which is comparable to an non sleeping person. In order to detect REM sleep, other parameters such as EOG and EMG combined with EEG are usually used. Novel solutions have been proposed in order to be able to monitor all sleep stages with the use of only EEG [20].

Stages						
Waking	REM Sleep	NREM Sleep				
			Light Sleep		Deep Sleep	
Stage 0	Stage R		Stage 1	Stage 2	Stage 3	Stage 4
Eyes open, responsive to external stimuli, can hold intelligible conversation	Brain waves similar to waking. Most vivid dreams happen in this stage. Body does not move.		Transition between waking and sleep. If awakened, person will claim was never asleep.	Main body of light sleep. Memory consolidation. Synaptic pruning.	Slow waves on EEG readings.	Slow waves on EEG readings.
16 to 18 hours per day	90 to 120 min/night	4 to 7 hours per night				

Figure 2.2: Sleep stages[65]

The EOG is useful for identifying and studying the REM sleep stages. It uses electrodes positioned near the corner of each eyes to measure the existing resting electrical potential between the cornea and Bruch's membrane in order to determine the position of the eyes.

For sleep studies EMG is used in the metalis, submentalis muscle, and/or messater region[64]. The EMG records the muscle tone and is used as a criterion for staging REM sleep. EMG can also be used on other muscle groups to determine sleep disorders, such as monitoring leg muscles in order to detect restless leg syndrome.

Each time a heart beats it is triggered by an electrical impulse. The ECG (also called EKG) records these impulses as they travel through the heart. The electrical activity is recorded using

16

electrodes placed on the patients body. A modern standard ECG consists of 12 leads in order to monitor all three dimensions of the heart [68]. Typically there are six limb leads placed on arms and legs and six precordial leads placed across the chest. The precordial leads has a specific angle from which it observes the electrical impulses generated by special cells within the heart.

The limb leads monitor what is called the frontal plane, while the precordial leads monitor the horizontal plane. Each node records the average current flow at any given moment. Each heartbeat is described as an RR interval, also known as a cardiac cycle. Based on which electrode records activity the RR interval can be further segmented into smaller and identifiable intervals of the cardiac cycle and used in diagnosis and evaluation of the heart and breathing of a patient.

There are multiple ways to record the respiration rate during a PSG. Nasal and oral airflow are often recorded either with nasal *thermistors* or thermocouple, which uses changes in temperature to measure the airflow with prongs or probes placed in or near the mouth or nose.

Another way of recording is to measure the physical movement the body during respiration using respiratory inductance plethysmography (*RIP*), which can use elastic strain gauges, impedance electrodes or air cuffs to detect movement. In the case of strain gauges, they are placed around the torso and abdomen to record the movement of the body as a patient inhales and exhales. Based on the inflation and deflation of the chest and abdomen area, the respiration rate can be derived. Both of these methods are used as ground truth in assessing the respiratory rate in sleep studies[4, 33].

When none of these respiratory signals are recorded, other techniques can be deployed. One such technique is to use the ECG signals to derive the respiration rate. ECG, or electrocardiography, measures the electrical signals generated by the heart. There are different ways of obtaining the respiration rate from an ECG signal and also from the ECG electrodes themselves. One method calculates the respiration rate based on beat to beat variation RR intervals (Figure 2.3a). This technique is based on respiratory sinus

arrhythmia (RSA) which is a natural variation in the heart rate.

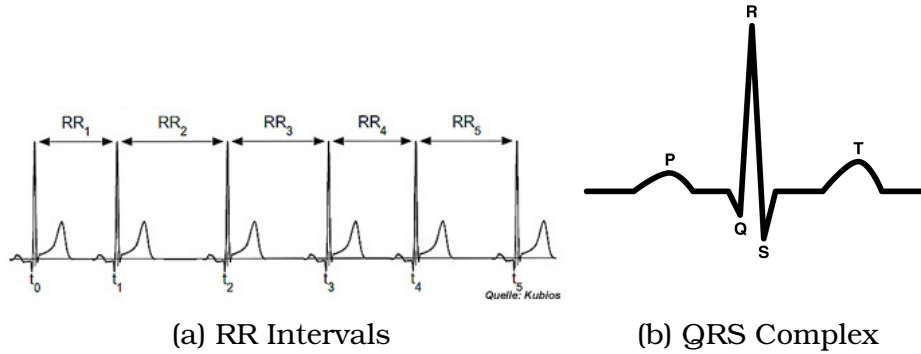(a) RR Intervals   (b) QRS Complex

Figure 2.3: ECG signal illustrated

Another technique is ECG Derived Respiration (EDR). When a patient breaths the ECG electrodes on the chest surface move relative to the heart due to the lungs filling and emptying. The transthoracic impedance varies as a result of the expansion and contraction of the lungs and from the mean cardiac electric axis show variations that correlate with respiration[43]. EDR can be used to derive respiration rate and detecting apneas.

Oxygen saturation is a useful parameter for detecting OSA, as the SaO^2 (blood oxygen saturation) drops after the onset of an apneic/hypopneaic event. According to Division of Sleep Medicine at Harvard Medical School[60], the SaO2 is usually around 96% - 97% at sea level. A dip to 90% is generally considered mild, while dips to between 80% to 89% are classified as moderate and saturation below 80% are severe.

2.2.3 Treatment of OSA

In order to effectively treat OSA, physicians have to consider the severity of the disease, co-morbidities and the patients preferences. A non surgical option is lifestyle changes, such as weight loss, avoidance of alcohol and nicotine, position therapy and treatment of co-morbid conditions. Continuous Positive Airway Pressure (CPAP) or therapy is described as a first-line therapy for moderate to severe OSA[16].

CPAP consist of a air pump, tube, and a mask, which provides pressurized air into the patients throat via the mask. The pressurized air helps avoid negative pressure from the inspiration collapsing the airway.

APAP devices (Autotitrating PAP) detect snoring, airway resistance or impedance in order to only administer positive airway pressure. It also uses diagnostic algorithms in order to adjust the amount of pressure, but are far more complex than a standard CPAP and require calibration by a sleep technician. They do though have the advantage of adapting the pressure to sleep stage and sleep position, reducing the risk of discomfort due to too high pressure during sleep stages with more relaxed muscle tone.

Surgical treatments for OSA is centred around reducing the risk of collapse and removing potential obstructions. Surgical techniques can be to remove some of the soft tissue in the pharynx region, reposition the soft tissue by skeletal mobilization, or by-passing the pharynx region[63]. There is no standard procedure found to eliminate OSA.

Another approach that can be utilized is pharmacological treatment, but the literature suggest that such treatment has not been successful. A review by Hedner, Grote and Zou from 2008 concludes: *Currently, no widely accepted pharmacological treatment alternatives are available for OSA*[18]

2.3 Sensors

The name sensor has according to Webster's New World College Dictionary its roots in classical Latin *sentire*, which means to sense. *A sensor is a device which responds to stimuli, or an input quality, by generating processable outputs*[26]. This is how Kalantar-zadeh defines sensors. He also points out that the outputs of a sensor are always functionally linked to input stimuli of the sensor.

The term sensors refers often to two aspects, i.e. the sensor that quantitatively measures an input quality and the component that converts it to a readable signal for the device or person re-

ceiving the recordings. The part of a sensor that is responsible of taking the input signal of the sensory apparatus and converting it is referred to as the transducer. A *transducer* converts one type of energy to another and is sometimes used interchangeably with sensors.

An example of a simple sensor is litmus paper, which usually is used for determining whether a solution is basic or acidic. The litmus paper is exposed to the the solution and reacts to the stimuli by changing colour, allowing an observer to read the results.

The output from sensors is a representation of the measured property and this can be described in different ways depending on the property measured. Over time the output can be used to create a sequence of data points called a *time series*.

There are different ways a sensor can be constructed in order to record some quality of the real world. *Contact* and *non contact* sensors are two broad categories can be used to describe sensors. Sensors that are described as *non invasive* do not necessarily have to be *non contact* sensors, but rather refer to the level of disturbance or discomfort the sensor cause for the monitored patient. A *non contact* sensor can be *invasive* if the operation of the sensor generates noise, while a *contwact* sensor might be very light and not noticeable by the wearer, and hence be considered a *non invasive* sensor.

In general, a *non invasive* sensor can be defined as that it will not interrupt a patients normal sleep. As this criteria is subjective, it makes the grouping of sensors difficult to pin down.

Signal processing is an umbrella term for operations applied to the signal. J. Moura defines processing as *operations of representing, filtering, coding, transmitting, estimating, detecting, inferring, discovering, recognizing, synthesizing, recording, or reproducing signals* [44].

2.3.1 Sensor characteristics

Ideally a sensor should be able to measure a desired quality (input) of the physical world without any other input being registered. This is referred to as *sensitivity* towards the desired input and an *insensitivity* towards other potential inputs. It is important that a sensor does not affect the input or the environment it is deploy in.

The *accuracy* of a sensors recording is the correctness of the output compared with the actual value of the quality it measures. Deviation from the actual value of the quality can be due to rounding error, inaccurate sensor, calibration error, too low resolution etc. The example Kalantar-zadeh uses is a temperature sensor measuring a real temperature of 20.0°C. If the sensor measures 20.1°C it is more accurate than if it had measured 21.0°C[26]. This is not to be confused with *precision*, which is the capacity to get the same result from repeated measurements of the same quality under the same conditions. The difference between precision and accuracy is illustrated in Figure 2.4.

Low precision - low accuracy   High precision - low accuracy   High precision - high accuracy
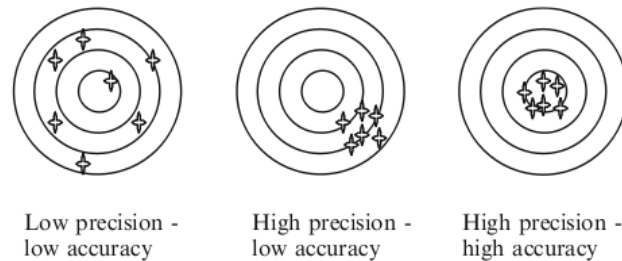
Figure 2.4: Precision and accuracy[26]

McGrath and Scanaill[40] describe "$v1.0$ sensors" as simple measurement of quantity, such as a mechanical thermometer. For the second generation of sensors we add computational power and communication which allows the sensor to process the data it records and transmit it to other devices. An example of this can be a acidity sensor, which is connected to an actuator which controls a valve in order to restore the Ph level to a preset value based on the sensors readings. At this stage the cost of production is still so high that it is not commonplace and highly specialized.

"Sensors $v3.0$" is described as when private consumers adopt the use of sensors. At this point sensors that previously were too expensive for consumers can be found in smart devices and in affordable home-use devices. In addition to the computational power introduced in "$v2.0$", the connectivity to the Internet opens up for new avenues for communication and pervasive sharing of data in real time. The data recorded by smart devices can be used for location tracking, health applications, consumer habits, and other areas.

"$v4.0$" is the stage we are currently stepping into. The capabilities of sensor systems have been increased due to increased computing power, smaller sizes, increased connectivity and more affordable prices.

2.3.2   Sensor networks

As defined by Phoha, LaPorta and Griffin sensors and sensor networks can be described with the following characteristics: *they monitor changes in the operational environment and collaborate to actuate distributed tasks in dynamic and uncertain environments*[53]. Each sensor has a task, a measurement of the physical world to perform and converts it into a signal. There are two primary approaches to how to process the data recorded: either distributed or centralized.

A human body can be compared to a centralized sensor network. We have different sensing devices such as eyes, touch, smell and hearing among others. The signals from these sensors are processed and coordinated by the central nervous system and the combined information provided from the different sensors gives us information about the world and gives us the ability to detect events around us based on the combined data recorded from the surrounding environment.

A distributed sensor network uses the sensor-nodes themselves to do processing. As the name implies, the sensors do not relay all the information gathered to one centralised storage/processing unit. Each sensor works autonomously but collaboration

can be achieved by letting each node share and request information from the network as a whole.

2.3.3 Data Stream Management Systems

Data Stream Management Systems (DSMS) are used in order to process the information gathered continuously by sensors or sensor networks. A data stream is a continuous (possibly infinite) sequence of data tuples. Traditional database management system (DBMS) is concerned with persistent storage of data, and is often used in conjuncture with DSMS. Instead of sporadic writes and frequent reads, as found in most traditional DBMS, DSMS have to filter out relevant events as data arrives. Access to the data is done as it arrives, thus the system has to continuously read and write data to memory.

A DSMS can not make use of a traditional query language, but instead uses what can be described as a *Continuous Query Language* (CQL). It can also be referred to as StreamSQL, as it shares the declarative nature of of SQL-like language. There is no standard language, but several prototypes has been created. A common trait is that all queries has to be one-pass queries, due to the stream-centric nature of a DSMS. An *event* is a *match* to a Continuous Query (CQ) on transient data. Results of a CQ is then passed on to *sinks* who consume the resulting matches, while the data in the stream can be passed forward to a different system, discarded or stored in a persistent database system.

It is important to note that a source does not have to be a physical sensor, but can just as easily be another DSMS or similar system running different queries. This way we can multiplex and demultiplex any given data stream.

As a data stream can be potentially infinite, the DSMS cannot do aggregation or analysis of data when it has gathered a "complete" set. Many DSMS uses a windowing technique to look at portions of the data as it arrives. These windows can be time or tick-based. Tick-based windows waits for N number of entries to arrive, while time-based windows aggregate on certain intervals.

Figure 2.5: A simplified Data Stream Management System

Aggregations can be averages, sum, count for time-based windows etc.

Each arriving tuple has to be marked with a time stamp. There are different strategies, all with different pros and cons. The main issue when dealing with time in a distributed systems is synchronization. If the sender attaches the time stamp we need mechanism in order to make sure their timing mechanisms are synchronized precisely. This approach, when the time stamp is injected by the data sources is called **explicit** time stamp, while **implicit** introduces the time stamp when the data arrives at the DSMS. This introduces an extra workload on the system, especially if we have multiple inputs. Depending on the domain the application is created for we also have to consider what is more important. The time when the data was created or the time the data arrived at the DSMS.

2.3.4 Complex Event Processing

While a DSMS detects changes in state, an isolated event that signifies things that happen in a stream of data, *Complex Event Processing* (CEP) combines data from multiple sources to infer events

or patterns for complicated situations. TRIO makes use of a CEP called *Esper*, developed by *EsperTech*[21].
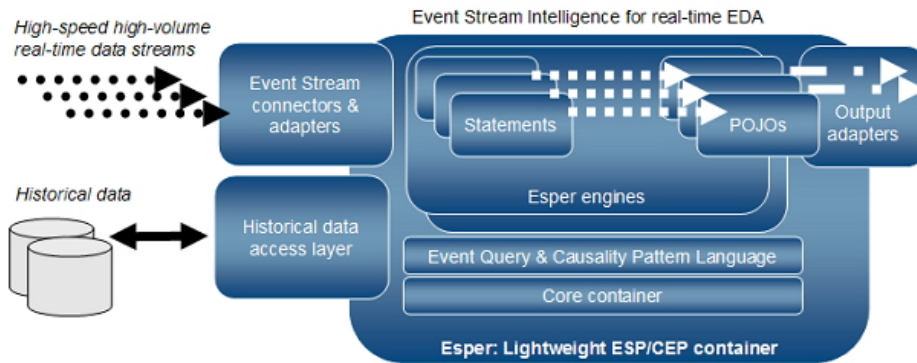
Figure 2.6: Esper components[22]

In order to make sense of data recorded by multiple sensors or a sensor network, they can be grouped together into what is called a *logical sensor*. By multiplexing signals from multiple sources, be it sensors or external sources, a logical sensor can learn new information and detect complex events based on multiple inputs.

An trivial example of a complex event can be a system utilizing a temperature sensor and a smoke detector. The logical sensor created from these two physical sensors can use both signals to detect a fire by combining, and decrease the chance of a false positive from a kitchen appliance or other device that generates heat.

Part II

Respiration analysis application

Chapter 3

Existing applications/Basis for analysis

Because of the estimated high number of undiagnosed cases of OSA and the high cost of sleep studies, there has been a conducted much research into non intrusive methods of detecting and diagnosis of sleep disorders.

There are many existing solutions on the market with different capabilities. Some of these solutions come with software which delivers high level physiological data such as heart rate, respiration per minute, temperature etc. Similar techniques to what is used by existing systems can be used to detect other low level respiratory events, such as onset of pauses in respiration and peak and trough detection.

BioRadio[23] has created software for their sensors that promises real-time visualization of data-streams, which suggests that there is potential for real-time analysis as well. But as many with other projects, the software is propitiatory, which turns out to be the most common problem when trying to find an existing solution. We need to, based on the goal defined in the problem statement (Section 1.5), find existing solutions that allows us to define the low level events we want to detect.

To create a new respiration analysis system from scratch would allow us to have full control over the definition of the events,

but will require considerably more work than using an existing solution. The quality assurance of the results will also require a much deeper understanding of the analysis, rather than working with existing and testes software. In order to reduce the work load of such an undertaking libraries such as *The BioSig Project*, which is an open source library for biomedical signal processing[46]. This library can help us considerably, but will still require more effort and a deep understanding of signal processing than by relying on an existing solution.

The application we use for the respiration analysis is called *puka*. The decision to use this particular solution is based on the fact that even though a lot of other more recent and novel approaches exist, none of their implementations can be found. Since puka is not only implemented, but also open source we are able to tailor it to our needs and make modifications where we see fit. The source code for the application can be found on PhysioNets websites[54].

PhysioNet Resource is a public service funded by funded by the National Institute of Biomedical Imaging and Bioengineering (NIBIB) and the National Institute of General Medical Sciences (NIGMS) at the National Institutes of Health. The service PhysioNet can be divided in three parts:

1. **PhysioBank**: a collection of digital recordings of physiologic signals, time series, and related data

2. **PhysioToolkit**: a library of software for physiologic signal processing and analysis2

3. **PhysioNetWorks**: a virtual laboratory for collaboration

3.1 Key functionalities

To generate events for the Esper engine in TRIO (see Subsection 2.3.3) we need a system that can analyse signals from respiratory sensors. Based on time series generated by sensors such as RIP or

thermistor based respiration monitoring we must be able to derive events that are significant to the detection of sleep disorders.

In the analysis system of the signal gathered from sensors we look for two main functionalities:

1. detect stops in respiration (effort)

2. detect these in as close to real time as possible

The first one is found in puka, assuming we can make the application run on modern systems. For the second functionality we have to make modifications to the existing application as the original application was created to analyse pre recorded signals.

The signals analysed are discrete-time signals or time series which can be represented as waveforms. This representation makes it easy to illustrate the signal and visually detect events such as inspiration and expiration start and stop. The design of puka is such that it takes a time series as input, finds respiration peaks and troughs and then calculates the pauses between each breath. These types of events can in turn be used for a real time analysis in order to fulfil the second quality.

Both the peak detection and the pause detection has different thresholds, the peak detection using one for determining what constitutes a peak or trough, while the pause detection has one for classifying what is a pause in a signal. The threshold for the pause detection is hard coded in the script, while the peak detection has this variable as a parameter, allowing the user to adjust it during the execution of the analysis.

One advantage of using an existing implementation is that it has been created by programmers with a through domain knowledge. Not only does an implementation of a respiration analysis require knowledge of signal processing, but also a great understanding the underlying algorithms. By basing the system on an existing implementation we can more easily get started on creating a system that can integrate with TRIO as a whole. Since the application is open source we can also make changes as we see fit if necessary.

Figure 3.1: The main flow of puka (generalized)

3.2 Terminology

To avoid confusion about the terms used in puka and similar terms used in both Chapter 4 and Chapter 5, we briefly will go through key terminology used by puka.

When puka uses the term *record* it is referring to a time series persistently stored either in a *file* or *database*. When using ASCII files the record is read into memory using MATLABs *load*.

The section of the *record* puka analyses is called a *clip* and is defined by a *onset time* and *end time*, two variables that dictate the start and end indexes in the *record*.

3.3 History

Puka was written by Joset A. Etzel, Erica L. Johnsen, Julie A. Dickerson and Ralph Adolphs in 2004 to analyse pre recorded data collected from equipment and software from BIOPAC Systems, INC. BIOPAC is a company founded in 1985 that makes physiological measurement tools. The authors of the software found that puka was able to analyse other physiological signals as well. The latest implementation (2004) contains ECG and respiration analysis tools. The respiration signal it uses for the analysis are gathered from strain gauge sensors that measure the circumference of the chest and/or abdomen as it expands and contracts during respiration. The strain gauge respiration data are time series that show the conductivity of the strain gauges signal which reflect inhalation and exhalation as the chest and/or abdomen. The respiratory analysis was designed to use signals collected with a TSD201 Respiratory Effort Transducer, a single strain gauge recorder.

Puka uses MATLAB to calculate descriptive statistics such as heart rate variability, peak-trough respiration sinus arrhythmia and respiratory variables from ECG and the strain gauge respiration data[55].

The same analysis can also be applied to other signals that share the same characteristics as strain gauge respiration data, such as thermistor sensors and RIP (described in Subsection 2.2.2). These types of signals fluctuate around a base value, and give us a respiratory waveform when plotted against time. The rise and fall in amplitude is representing different physical attributes, such as conductivity in the case of RIP, or temperature in the case of respiratory thermistor.

3.4 Program structure

The main control and structural code of puka is written in Java. This part of the code is responsible for I/O operation, interactions with the user and data persistence. Interactions with MATLAB are synchronous operations, originally via the library JMatLink, initi-

31

ated by the user using a GUI. The GUI has been created using the Netbeans module Form, but since the intent is finally to strip away the GUI, no modification or upgrades are necessary for this project, and any description of the GUI code is therefore skipped.

The application is initialized via the *frmMain* main class which instantiates the GUI and prompts the user as to what data input to use. Data can either be read from a database or from a file that adheres to the format allowed, described in Section 3.7. A simple program must be written to change the raw text output of the PhysioNet data to the format specified by puka. Physionet has a program library which contain programs that allow us to easily convert the signal files into a more suitable text format.

        TODO: illustration of the application, diagram?

The calls to the MATLAB engine are done via the JMatLink library. The MATLAB scripts that are executed is located in the folder *matlabscripts*, found in the puka source code. The instance of the JMatLink proxy which all classes communicate with MAT-LAB through is instantiated in the *frmLoadData*-class and used throughout the lifetime of the application.

3.5   Runtime requirements

In this thesis a 64-bit Windows 7 machine is used to compile and run all software. As stated in the user manual[55], operating systems other than Windows XP and 2000 has not been tested. The manual also list six main external dependencies that have to be installed in order to run puka.

Dependency	Comment
Java	Minimum v1.4 according to documentation
MATLAB	R13 was released in 2002
Cygwin	No version specified
WFDB	Installed within Cygwin
JMatLink	Latest version (V1.3.0) released in 2005
~~MySQL~~	Ignored since puka supports file storage

Table 3.1: External dependencies for puka

On the machine used for this thesis the Java code has been compiled with *Java JDK 1.8.0*, with the exception of the attempt to compile JMatLink when *Java 1.4* was used (see Section 4.2). *MATLABR2012b* has been used to run all scripts. This is version of MATLAB available to students at UiO, making it a natural choice for this thesis.

Puka depends on parts of the WaveForm DataBase (WFDB) Software Package. The WFDB Software Package is a curated list of specialized software for usage with PhysioBank data. The bulk of the necessary software is found in the WFDB library which is an API for access to PhysioBank.

The WFDB library is available both for command line usage and as a library for MATLAB. puka makes use of a small subset of the package (Table 6.6), but other components can be useful for reading, retrieving and manipulating the recordings found in Physiobank. The package requires Cygwin and certain libraries within the environment. Cygwin replicates significant parts of the POSIX system call API for a Windows environment, which WFDB package applications depend on.

ecgpuwave.exe and *convertecg.exe* are separate from the WFDB library, but can be compiled using the compilers *gfortran* and *gcc* respectively. Both of these programs are used in the ECG analysis.

Cygwin allows us to utilize the *gcc* and *gfortran* compilers in a Windows environment, which are necessary to compile the support applications *convertECG* and *ecgpuwave* respectively. *rdann* is used to read the file format used by Physiobank. It can read both local files or download the files from Physiobank web service containing signals.

ann2rr, *ihr*, *ECGPUWave* and *convertECG* are all used by the ECG analysis, and therefore not described in much detail.

ann2rr reads a WFDB record and an accompanying annotation file and returns the the RR interval in number of samples, and a vector of sample numbers representing the onset of these RR

intervals. *ihr* reads an annotation file (specified by the annotator and record arguments) and produces an instantaneous heart rate signal.

The standalone tool *ECGPUWave* analyses an ECG signal and detects the QRS complexes and locating the beginning, peak, and end of the different stages of the QRS complex. Another standalone tool is *convertECG* converts ASCII text files into the binary WFDB data format. The QRS complex is a segment of an RR interval, described in Subsection 2.2.2.

rdann	move annotation created by ecgpuwave to external file
ann2rr	create an RR interval series
ihr	create a instantaneous heart rate series
ECGPUWave	marks ECG waveforms
convertECG	converts ecg.txt into wfdb .dat format

Table 3.2: WFDB programs used by puka

The JMatLink library is used for communicating with the MATLAB engine from Java runtime. JMatLink was created by Stefan Müller in 1999 [35] to allow users to interact with MATLAB via a web server, running a Java program. The last iteration of the library ($v1.3.0$) was released in 2005 and the source code can still be found on Sourceforge[37].

3.6 Preferences

In the startup process puka looks for the *preferences.txt* file in the working directory which is the directory in which the program was launched. The preferences consist of the absolute path to helper programs such as the WFDB applications and *convertECG*. In addition to keeping a track of helper application the preferences also keeps track of certain meta-data about signal clips to be analysed by puka.

The preferences window contains five tabs with different values (Figure 3.2). These preferences has to be set for each system.

Paths
WFDB tools
Installation directory (eccgpuwave.exe and puka.jar)
WFDB data file directory (download and signals)
ConvertECG.exe directory
ECG
Signal Frequency (hz) (even though under ecg spec, used in resp)
Signal unit (mV)
Signal Gain (adu/mv)
ADC resolution (bit)
Zero-level (adu)
Length of Record H:M:S
Data columns
Column for ECG and respiratory signal
Onset trigger
Clips
Clip name and length (num samples)
Database
List of database connections

Figure 3.2: Preferences stored in preferences.txt

3.7 Data format

The program can either read data from a database or from a raw text file. Each line in a text file represents a sample and if we have multiple channels they are separated by a white space character. The column number of the signal used by ECG and respiratory analysis is indicated in pukas preferences file which can be edited in the GUI or directly in the text file.

3.8 Respiration analysis algorithm

The algorithm that is used in puka for respiration analysis is implemented in MATLAB. The scripts that contain the algorithm are found in the *matlabscripts* folder, and is split up into several *m*-file containing the logical components of the algorithm based on the steps in the algorithm.

The project site describes puka in the following terms: *puka incorporates a new method of identifying the breaths and pauses in strain gauge belt recordings. This technique locates the points of maximum inspiration and expiration for each breath as well as post-inspiratory and post-expiratory pauses*[55].

The manual does not contain any description of the algorithms used, but the peak detection is well documented by Todd and Andrews [69]. None of the other parts of the analysis is documented, and we therefore need to analyse the different components in order to describe them.

The algorithms used in the respiration analysis identifies critical parts of a recording according to the documentation and description. The critical parts are peak, trough, post-inspiratory (*PI*) and post-expiratory (*PE*) pause. These four parts are useful events for detecting sleep apnea, and will have to be converted into events for the TRIO system. PI and PE pauses are the length of the section in a time series after the signal has flattened, as shown in Figure 3.3.

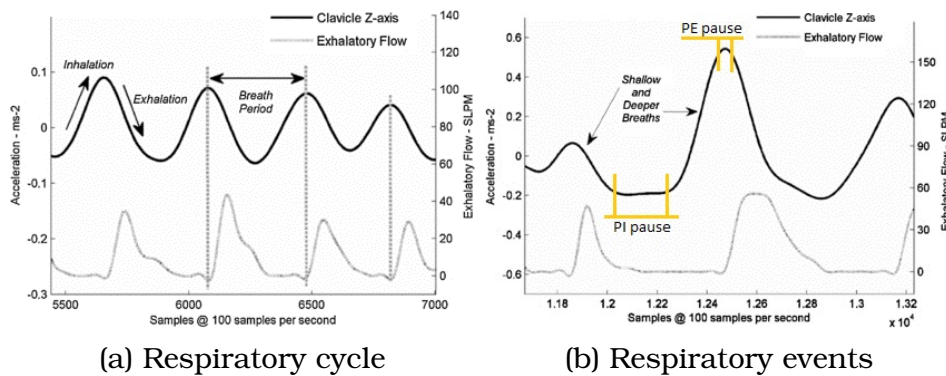(a) Respiratory cycle   (b) Respiratory events

Figure 3.3: Charecteristics of respiration

The program uses an algorithm which the puka manual splits up into five steps

1. Load and prepare a given signal,

2. identify breath through peak and trough detection,

3. check validity of the peaks and troughs,

4. mark pauses at each peak and trough and finally

5. statistical computation based on the results from the analysis.

Data loading

The application reads a *record* which is the signal file into memory and stores it within the MATLAB engine as the *data1* variable. This is then split up using the *onsetTime* and *endTime* variables to create what puka calls a *clip*.

The *onset time* and *end time* are passed to the peak detection algorithm (Item 2) and are used to make sure that the found peaks and troughs are within the clip defined in the application. According to the puka manual, certain signals contain an *onset time*, and if not puka will use MATLAB to detect the first trigger point and use this as the stimulus onset time.

This is done by rounding the entire signal into a binary time-series. All values within the series that are below 0.5 are considered *false*, while the rest are *true*. The *onset time* is set to the midpoint in the first *false* range.

Peak Detection

Firstly, the algorithm makes a pass over the whole clip, marking peaks and troughs using a *peak detection algorithm* based on Todd and Andrews' peak detection algorithm published in 1999[69]. When we use the term peak in this section, we refer to both peak and troughs as both are in principle the same. *A peak elements is any element that dominates both a preceding element and subsequent element. Correspondingly, a trough element is any element that is dominated both by a preceding element and by a subsequent element*[69]. By dominating the surrounding elements the authors refer to the amplitude being greater in the element than the surrounding elements. Element does not necessarily mean a single point in a time-series, but can also refer to a range of points.

The suggested algorithm goes through a given signal Q. Each index in Q contains the amplitude of a sample in the time-series. Within the loop a variable d indicates the direction of the signal on the *y-axis*, that is the trend in the amplitude of the signal. δ is used as the threshold for defining whether a change in trend signifies a peak or is a minor deviation.

The user is able to input the *factor* variable shown in Listing 3.1 when running the peak detection algorithm to adjust the sensitivity.

Listing 3.1: Calculation δ threshold in pukas implementation

```
% factor is the number to use when making the threshold;
   default is 0.1
th = abs(prctile(Qd,75) - prctile(Qd,25)) * factor;
```

The variable a records the index of a *maximal* element since the last *trough* while b contains the index of the *minimal* element since the last *peak*. A variable S records all indices of the maximal

38

elements since the last peak *if* the signal is rising, minimal in the case of troughs.

Using detecting a peak as an example and letting i as the current index during the loop, a is only updated when $Q[i]$ is higher than $Q[a]$. When $Q[i] == Q[a]$, we add $Q[i]$ to S. When $Q[i]$ is significantly smaller than the last maxima ($Q[a]$), the direction (d) is changed, S is stored and the algorithm looks for troughs by using $Q[b]$ now that d is changes. The significance of a change is determined by the *threshold*, which is a global value which is added to $Q[i]$ when comparing whether the value is significantly larger or smaller than the last maximum or minimum. The implementation of the algorithm that is used in puka can be found in Section 8.4.4.

The direction d is in the puka implementation initially set to *unknown* and in the first iteration of the algorithm kept as such. This is because i, a and b are all initialised to the same value. For every iteration i is incremented, so when we reach the second iteration $Q[i]$ is different from $Q[a]$ and $Q[b]$ and the algorithm can now look for differences between the two. d is kept as *unknown* until we reach a $Q[a] >= Q[i] + \delta$ which indicates *downward* trend, or $Q[i] >= Q[b] + \delta$ giving us a *upwards* trend.

The function returns two arrays, one containing the index of all peaks and the other containing troughs.

Validity of Peaks and Troughs

After the peak detection is completed, puka classifies the peaks with three classifications:

1. valid,

2. invalid and

3. questionable.

The arrays returned from this function are of the same length as the result from the peak detection with classifiers for each entry.

The classification is based on analysing each side of a peak within a certain window size. The size of the *classification window* is hard coded into the script and defines the length (number of samples) in each direction of a given peak or trough the script will look to validate the peak or trough.

The classification window is *only* used for this script when evaluating the validity of a given peak or trough. As stated in the source code (*classifyPeaks.m*): "try 1 second windows around each peak/trough, centered on found peak 1000 Hz signal decimated by 5, so now 200 Hz; 200 data pt window either side". More on this implementation detail in Subsection 5.3.1.

The application classifies the found peaks and troughs using the *classifyPeaks* script. The classification is based on the total negative and positive difference of the amplitudes of all neighbouring indexes on both side of a peak. It calculates the difference between each point within the *classification window* and finds all indices where the difference is either negative or positive and sorts them in based on whether they are before or after the peak or trough.

Listing 3.2: Classification of window surrounding a peak

```
diffWB4 = diff(windowB4); % difference between all
    adjacent pts in the window
diffWAf = diff(windowAf);
[indNegB4] = find(diffWB4 < 0); % neg diff = curve going
    down
[indPosB4] = find(diffWB4 > 0); % pos diff = curve going up
[indNegAf] = find(diffWAf < 0); % neg diff = curve going
    down
[indPosAf] = find(diffWAf > 0); % pos diff = curve going up
```

If the *classification window* start or end of window higher than the peak (or lower in the case of troughs), the point is labelled as *invalid*. The difference calculated in Listing 3.2 is used to check whether (in the case of peaks) the difference is negative in front of the peak and positive after by summarizing the difference.

Pause Detection

After the classification the user is prompted to evaluate the marked peaks and make the final call on which peaks to accept and which to discard based on the plots of the signals. When the peaks and troughs are identified and validated, the algorithm calculates the pause, if any, surrounding the peak or trough.

Check for same-height indexes around each peak and trough found in the validated points. This is done checking both direction on the y-axis from the current peak or trough location, and based on the threshold the algorithm looks at the total difference between values until it reaches a slope higher than the threshold. The index when the threshold is reached is the end or beginning of a pause.

Statistical Computation

After the four steps of the respiratory analysis, puka conducts a statistical computation consisting of: number of breaths, shortest breath, longest breath, average breath length, standard deviation of breath length. For *PI* and *PE* pause calculations the system calculates the average *PI* and *PE* pause by adding the length of all respiration pauses and dividing by the total number of pauses. In the process of calculating the average puka also reports the longest and shortest respiration pause found in the clip.

Chapter 4

Modernizing

The best case scenario is if we can launch the application to verify and test it and then begin the adaptation for integrating it with TRIO. In this chapter we look at the process of making the standard version of puka able to run on modern systems. In order to do this we firstly map the dependencies that require updating and consider different solutions for making updating or circumventing the dependency.

The first natural step in this process is to attempt to execute the system as described in the accompanying installation manual, but when doing so we get an error message (Listing 4.1) which can be used to identify why the application won't work out of the box. When the hindrance is identified we can take steps to mend it.

4.1   Identifying decrepit parts of puka

Since the respiration analysis is implemented in MATALB, puka has to have some way of communicating with the MATLAB engine from the Java runtime environment. It is, as described in Chapter 3, written to use JMatLink, a third party library, in order to achieve this. The JMatLink library is distributed as a *dll* (Dynamic-link library), but the source code is freely available as well. The library is implemented in C and Java.

The JMatLink-manual has instructions for installation on Windows 98 and Windows 2000 only. The installation described in the manual is to copy the *dll* into the Windows *System32*-folder, but running puka after this operation results in the error message shown in Listing 4.1.

Since we are running on a Windows 7 operating system we have to explore the option of loading the third party library as a system library. According to the official documentation [41] there is an official tool, *regsvr32.exe*, in the Windows OS for registering libraries.

Listing 4.1: Trying to launch puka after adding JMatLink

```
ERROR: Could not load the JMatLink library
    This error occures, if the path to
    matlab's <matlab>\bin directory is
    not set properly.
    Or if JMatLink.dll is not found.
Exception in thread "main" java.lang.UnsatisfiedLinkError:
    C:\Windows\System32\JMatLink.dll: %1 is not a valid
    Win32 application
```

Both the 32 bit and 64 bit versions of *regsvr32.exe* result in the same error message (Figure 4.1), suggesting the *dll* is incompatible with our OS.
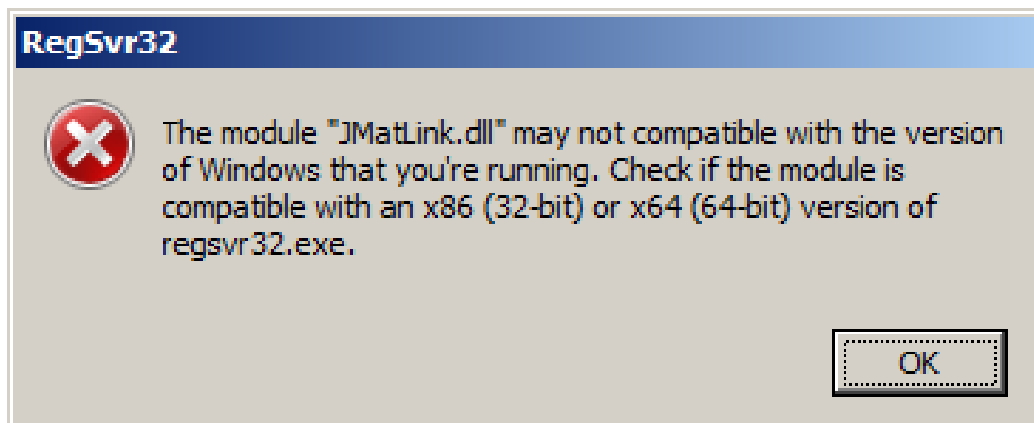

Figure 4.1: Resulting error message from regsvr32

By running the *jmatlink.dll* through *Dependency Walker*[42], we are able to map the dependencies of the library, and it seems

43

that some of the 32-bit Windows native libraries JMatLink is dependent upon are only found as 64-bit versions on our version of Windows, and others are not found at all.

^	Module	File Time Stamp	Link Time Stamp	File Size	Attr.	Link C
⬛❓	API-MS-WIN-APPMODEL-RUNTIME-L1-1-0.DLL	Error opening file. The system cannot find the file specified (2).				
⬛❓	API-MS-WIN-CORE-WINRT-ERROR-L1-1-0.DLL	Error opening file. The system cannot find the file specified (2).				
⬛❓	API-MS-WIN-CORE-WINRT-L1-1-0.DLL	Error opening file. The system cannot find the file specified (2).				
⬛❓	API-MS-WIN-CORE-WINRT-ROBUFFER-L1-1-0.DLL	Error opening file. The system cannot find the file specified (2).				
⬛❓	API-MS-WIN-CORE-WINRT-STRING-L1-1-0.DLL	Error opening file. The system cannot find the file specified (2).				
⬛❓	API-MS-WIN-SHCORE-SCALING-L1-1-1.DLL	Error opening file. The system cannot find the file specified (2).				
⬛❓	DCOMP.DLL	Error opening file. The system cannot find the file specified (2).				

Error: At least one module has an unresolved import due to a missing export function in an implicitly dependent module.
Error: Modules with different CPU types were found.
Warning: At least one delay-load dependency module was not found.
Warning: At least one module has an unresolved import due to a missing export function in a delay-load dependent module.

Figure 4.2: Missing Windows libraries and error message from Dependency Walker

This leads us to conclude that the precompiled *dll* file is not compatible with the operating system we have available.

4.2   Recompiling JMatLink

The latest version of the library was released in 2005 (*v1.3.0*), but according to the change log it has not seen much development since then. Since the library is, at the time of writing, over 15 years old it is difficult to make it run on a modern systems. It can be described as legacy software, in the sense that it can not be easily installed and executed on a modern system. We therefore need to make modifications to be able to test puka for use in TRIO. The source code for both puka and JMatLink is publicly available meaning one or both can be altered in order to make puka compatible with modern system and modern versions of MATLAB.

The *jmatlink.dll* file that is found pre-compiled by the author cannot be used so we attempt to compile a new version. The source code of the library is accompanied by a build file for Ant, a Java-based build tool [12].

The *build.xml* file used to compile the project contains hard coded values that has to be changed in order to compile the library

44

locally. These includes the path to *Java Development Kit* (JDK), *Borland C++ Compiler*[10] and MATLAB compiler support libraries (Listing 4.2). These components need to be installed on the host machine in order to attempt a recompilation of the library.

Listing 4.2: Hardcoded paths in build script

```
-Ic:\j2sdk1.4.2_06\include
-Ic:\j2sdk1.4.2_06\include\Win32
-Ic:\bcc\INCLUDE
-IC:\MATLAB6p5\extern\include
-IC:\MATLAB6p5\simulink\include
```

The installation paths has to be updated to match the host system on which we are building on. As evident from the same parameters, we need to update the arguments passed to the compiler based on the system we are using. We also have to make sure the targets within the different parameters actually exist.

Listing 4.3: From JMatLink build file

```xml
<target name="compile" depends="env">
<!-- compile object file -->
   <exec executable="bcc32" dir="${build.src}/jmatlink/" >
      <arg line="-Ic:\j2sdk1.4.2_06\include
         -Ic:\j2sdk1.4.2_06\include\Win32 -c -3 -a8 -w- -b
         -g30 -Ic:\bcc\INCLUDE -oJMatLink.obj
         -IC:\MATLAB6p5\extern\include
         -IC:\MATLAB6p5\simulink\include -O1 -DNDEBUG
         JMatLink.c"/>
   </exec>
<!-- link object file to DLL -->
   <exec executable="bcc32" dir="${build.src}/jmatlink/" >
      <arg line="-DLL -eJMatLink.dll -tWD
         -Lc:\bcc\lib\32bit -Lc:\bcc\lib
         -LC:\MATLAB6p5\extern\lib\win32\borland\bc50
         libmx.lib libmat.lib libeng.lib JMatLink.obj" />
   </exec>
  <move file="${build.src}/jmatlink/JMatLink.dll"
     todir="${build.dir}" />
  <delete file="${build.src}/jmatlink/JMatLink.obj" />
  <delete file="${build.src}/jmatlink/JMatLink.tds" />
</target>
```

When trying to build JMatLink with Java 1.4 JDK, Borland 5.x using ant 1.8.2 we get the self describing errors in Listing 4.4. This is not surprising considering the *include.cpp* file is not found in the indicated folder in our installation of MATLAB. There is no support for the Borland compiler in the version of MATLAB we have available (2012b). It is also more relevant to find a solution that can support modern versions of MATLAB instead of relying on an older and specific version.

Listing 4.4: Errors when attempting to compile

```
[exec] Error E2194: Could not find file
   'Files\MATLAB\R2012b\extern\include.cpp'
[exec] Error E2194: Could not find file
   'Files\MATLAB\R2012b\simulink\include.cpp'
```

The MATLAB documentation[38], states that the compiler support needed to build the JMatLink library is not present in MATLAB 2012b (the version available to UiO students). In the original build-file we can see it includes a references to MATLAB6.5 specifically, which did have support for the $Borland5.x$ compiler, but has not been present in subsequent releases.

Without any documentation as to what the missing components do, the compilation of the library is difficult to complete. Based on the error messages from the compiler, we can deduce what components found in *MATLAB 6.5*, the *bc50.cpp*, will have to be included in the project. A dependency on an obsolete version of MATLAB is not desirable, so other approaches to running puka might decouple dependencies to the MATLAB implementation.

According to MathWorks, MATLAB had support for the Borland compiler up until Release 2007b. The versions of MATLAB that have support for the Borland compiler are 32 bit versions, so the necessary libraries for compiling for MATLAB 2012b 64-bit does not exist. So even if the JMatLink source code is available, the hurdles of compiling and linking the library are far greater than writing a wrapper to reproduce the functionality of the library.

The process of compiling the original version of JMatLink is extra convoluted because of the lack of documentation and the fact that the dependencies are deprecated and no longer maintained.

46

The necessity for a system library and specific MATLAB versions to run puka is not ideal. A more portable solution will make it more useful. Therefore the focus will be shifted to a strategy to modernize the software.

4.3   Modernization of puka

If we are to make a version where we won't have to recompile the JMatLink library for each target, it might be of interest to look at different approaches than the use of JMatLink. We take a look at relevant options that can be considered for such a process.

Seacord, Plakosh and Lewis[61] describe four reasons for changing software:

1. **Perfective** - improvements to the software. Adding new functionality, enhance performance, improve usability.

2. **Corrective** - repairing defects in the software.

3. **Adaptive** - changes made to changes in the environment, such as changes to operating systems, language compiler or tools, database management system etc.

4. **Preventive** - these changes are mode to improve the future maintainability of the software.

The changes we make to the software is primarily adaptive, since the main issue is making the software run with a newer version of MATLAB and Windows. During the modernization we attempt to include preventive changes as well in order to be able to adapt to future changes in the environment by making the dependencies loosely coupled by avoiding interdependencies where possible.

4.3.1 Approaches

Virtual machine

By running an old version of MATLAB and an operating system such as Windows 98, 2000 or Windows XP 32-bit, we should be able to run JMatLab library without having to recompiling the library. These are the platform and software puka was designed to run with, so this will allow us to start testing the application quickly.

Update JMatLink or write a new library

As the source code for JMatLink is hosted on *sourceforge.net* [37], and the source code for puka is available on PhysioNet[54] we can potentially update JMatLink to make it compatible with modern systems and software.

Writing a MATLAB wrapper

By removing the Java code, we strip away the need for the decrepit library. We can utilize the MATLAB scripts containing the algorithms by calling them from a new *controller* scrip, removing the need for calls from Java to the MATLAB engine.

Create Adapter for JMatLink

We can modernize the calls to the MATLAB engine by utilizing a modern Java - MATLAB interface and wrapping the calls that are intended for JMatLink in the new interface. This allows us to avoid much changes to the original source code.

Instead of using JMatLink we look at the potential of using MatLabControl[28] as an adapter between the JMatLink interface and Java code to intercept and reroute calls to MATLAB to the new

interface, that has none of the tightly coupled dependencies on specific system libraries and also supports calls to a 64 bit version of MATLAB. The adapter is based on pukas usage of the methods in JMatLink.

4.3.2 Evaluation of modernization approaches

Each approach has its pros and cons that has to be weighted up against the intention for the planned changes to puka. Ideally we want to make a version of the application that can

- run on modern operating system,

- no version restriction on MATLAB,

- demanding as few dependencies as possible and

- allowing changes and optimizations to be easily implemented and tested.

Based on these criteria we can out of the gate exclude running the application on a virtual machine, as this solution is only for running the unaltered version of puka. This is also not in line with the overall goal of modernizing and utilizing puka in new ways such as real time analysis. Practical consideration for this approach will also be how get a hold of licenses for discontinued software such as Windows 98 and older versions of MATLAB. This solution is only considered as a last resort or for experimenting with the original source code and library if necessary.

Updating or writing a brand new library for Java to MATLAB interaction will require a lot more work than other solutions. As described in Section 4.2, the library depends on decrepit systems, and a modernization of the library will require deep understanding of MATLAB internals, which is not within the scope of this thesis.

By creating a new control layer in MATLAB we get to remove parts of the software that is redundant to the respiration analysis. For a real time implementation of the respiration analysis

this approach has to be considered, as the control layer has to be redesigned. The steps that are manual in the original version of puka will be altered into automatic versions or ignore the steps altogether. This option will be discussed further in Chapter 8, as the process of rewriting the control code is easier to verify when translating within the same language.

As it stands the control code is written in Java, which makes the option of writing an adapter for JMatLink a strong candidate. The flow of the respiration analysis is easier to comprehend and replicate when using the same programming language, and it is easier to verify that the same procedure is executed.

The first iteration of the modernized puka makes use of an adapter for the calls to JMatLink, and relays them to an existing and more modern library for the Java to MATLAB communication.

4.4   Adapter for JMatLink

The adapter has to capture calls that are intended for JMatLink, and execute the same operation as the library would have. Based on the JavaDocs for the library we can map what types and methods we need for the adapter.

There are a few existing solutions for executing MATLAB scripts from a running Java application. There is support for calling Java classes from MATLAB, but no official way for the other way around. The solutions range from corporate [6] to hobby projects[31].

A promising project is *matlabcontrol*[28], as it seems to be frequently cited on MathWorks forums, it had an active development and has been forked and continued after the closing of the *google code* service where the project was initially hosted.

4.4.1 JMatLink analysis

We need to assess what types are being set and used Java, in order to make sure the conversion of these types is done properly. We know the return type of all the methods used in JMatLink[36] and will have to make sure the conversion between system is correct.

JMatLink method	return type and description
engGetArray	double[][], Used for both 1 and 2 dim array
engGetScalar	double

Table 4.1: JMatLink methods used by puka

For our implementation of the adapter we have to prioritize the methods that have been used in the implementation of puka. The software adheres to a strict naming convention which allows us to find all instances of the JMatLink class and what methods are called via text searches in the source code.

These searches show that all *get* calls to MATLAB returns a scalar, single dimensional array, or two dimensional array of Java primitive double. There are also methods for setting these types, converting from Java primitives to MATLAB types.

The most used call to the library is the *engEvalString* method, which takes an expression in the form of a string and executes this in the MATLAB-engine. This allows the Java application to change the working directory, load data, call functions, and all other command line operations as a user is normally able to execute in a MATLAB shell.

Before we implement the adapter we create unit tests for all calls puka does through the MATLAB interface (Listing 4.5). We control that values passed between each system is appropriately converted and retain the correct value. We must also make sure that operations and conversions return the expected result and type.

4.4.2 The matlabcontrol library

matlabcontrol was originally created as a *Remote Method Invocation* (RMI) wrapper around an existing Java to MATLAB library made by Kamin Whitehouse at University of Virginia[74]. MATLAB have had the ability to make use of Java code since version 5.3 (R11) with the Java MATLAB Interface(JMI). Whitehouse sought to provide techniques[73] to call MATLAB commands from Java with a program written in 2001 using undocumented parts of the JMI library. The work on this Java class has been continued by Joshua Kaplan and the project *matlabcontrol*[28].

There are a few projects that have kept the *matlabcontrol* project available even though the Google Code-service has been shut down. The source code has been hosted on github and been made available through the Maven project management software and comprehension tool[1] allowing us to easily set up puka and a version of *matlabcontrol* together.

The two versions that we found to be interesting are:

- **matlabcontrol**[52] (fork on Github) and

- **MatConsoleCtl**[70]

Both projects are based on the code hosted on Google code, and now hosted on GitHub. They have both also been published as Maven artefacts. The project called *matlabcontrol* contains the pre packaged jar file of the original *matlabcontrol 4.1.0*, while *MatConsoleCtl* has seen changes made to it since it was forked from the original repository.

Since MatConsoleCtl contains the source code and has been maintained since the last release in 2013, this seems like a good candidate for this thesis. The changes since 4.1.0 seem to be mostly minor bug-fixes such as error handling and a demo project for tutorial purposes.

In the following sections we use *matlabcontrol* as the name for the library for *MatConsoleCtl* as this is the name of the original

52

library. According to the author, the name was changed to ensure that we don't infringe the original project's license, but the package name is still *matlabcontrol*. So to avoid confusion when comparing the source code of the adapter, we stick to the package name.

4.5 Implementation

We have identified what parts of the application that does the respiration analysis and focuses the changes to the source code on this part.

Once the application is adapted for modern system we use the existing GUI in puka for testing, but ultimately we only make use of the respiration analysis part of the application. To create the adapter for *JMatLink* we configure the project up as a Maven project to handle build and dependencies. We create a package called *JMatLinkAdapter* which we then import into the puka source code.

4.5.1 Adapter methods

The calls to MATLAB can be summarized as *get*, *set* and *exec* calls. The *exec* calls are instructions for the MATLAB engine such as running scripts or changing working directory, while the *get* and *set* calls move data back and forth between runtime environments.

The *matlabcontrol* library has built in methods for type conversion between MATLAB and Java for primitive types, but there are certain differences to be aware of. As stated in the documentation for the library[29], it is not possible to send Java primitives directly to MATLAB. All variables are treated as arrays in MATLAB, even scalar variables. This means the programmer has to keep track of the types used in the MATLAB scripts and decide what method to use when retrieving the variable. When getting scalar variables we need to cast the first (and only) member of the array to a *Java double primitive*. We also make sure that MATLAB considers it a scalar by using the built in function *isscalar()* after setting

a scalar.

For conversion of single dimensional arrays *matlabcontrol* automatically converts between Java and MATLAB arrays. Most Java primitives have a corresponding MATLAB array type, with two exceptions. These two are `char[]` and `long[]` and will cause a MATLAB to throw an exception if used in MATLAB version R2009b or higher.

When converting multi dimensional arrays, we need to make use of the *MatlabTypeConverter* class, which can be found in *extensions*. This class converts between Java array to the type MatlabNumericArray. If the conversion is not done, the resulting array in MATLAB will be a cell array. A cell array is data type with indexed data containers cells that can contain any data type. These containers are called *cells*. The scripts used in puka expect arrays of the type *double*, and will throw an exception if they receive cell arrays due to the inability to do double precision mathematical operations on cell arrays. By throwing the exception to the callee we make the code more testable.

4.5.2 Replacing references to JMatLink

The source code changes to be made to puka in order to swap out the original *JMatLink* library can be found by a text search through the source code for instances of the *JMatLink*. The instances of the *JMatLink*-class are renamed *JMatLinkAdapter* to make it clear that we are no longer using the actual JMatLink library.

The instances of *JMatLink* have been given the same name throughout the entire project, meaning we only replace the initialization of the class, since we keep the method names as described in the JavaDoc. The *JMatLinkAdapter* implements an interface based on the JavaDoc to make sure we maintain the same return types and arguments as expected by the existing code.

```
// Old implementation:
engMatLab = new JMatLink(); //initiate connection
try {
  System.loadLibrary("JMatlink"); // load system library
```

```
  engMatLab.setDebug( true );
  int intC = engMatLab.engOpen(); //open connection to
     MATLAB
} catch (Exception e) { e.printStackTrace(); }
//=========================================================
// Replaced with:
engMatLab = new JMatLinkAdapter();
```

We change the instantiations of *JMatLink* in the source code
to the the adapter class (Listing 4.5.2). Calls to the loading of the
JMatLink library can be removed, as it is no longer a system li-
brary we are dependent upon. The creation of a new instance of
the library is done in *frmLoadData*. The library is also loaded in
frmConvert.java, but this class is only instantiated by itself, and is
probably separate from puka and can safely be ignored.

Calls to JMatLink are now intercepted by the JMatLinkAdapter
class which replicates the expected behaviour by using matlabcon-
trol.

Even though the API for both libraries is similar there are
certain differences we needed to take into account when writing
the adapter. The major concern is to make sure we convert types
and arrays correctly. As the analysis of the puka source code in
Section 4.4, we have the following Java types to convert between
Java and MATLAB, and back again:

Java Type	MATLAB Type
double	double scalar
double[]	Numeric array
double[][]	Numeric array

All of the Java types passed to MATLAB can and should be
represented as a *numeric array*. This allows the MATLAB scripts to
do calculations without the risk of receiving a non numeric type
which will be incompatible with the scripts that implement the
peak detection.

4.5.3 Interface

Based on the javadoc and the source code, we can create a list of methods that the application makes use of. Ideally we want to create an adapter that replicates the entire functionality of the JMatLink library. But the prioritized methods are the ones in use by puka. These methods are shown in Listing 4.5

The following methods are the complete list of methods found in the JMatLink library, but only the ones used by puka will be described further.

Listing 4.5: Methods used in puka from JMatLink

```
//matlab session:
engOpen() : void
engClose() : void
setDebug(boolean debugB) : void
kill() : void

//matlab commands:
engEvalString(String evalS) : void

engGetScalar(String arrayS) : double
engGetVariable(String arrayS) : double
engGetArray(String arrayS) : double[][]

engPutVariable(String arrayS, double[][] valuesDD) : void

engPutArray(String arrayS, double valueD) : void
engPutArray(String arrayS, double[] valuesD) : void
engPutArray(String arrayS, double[][] valuesDD) : void
```

4.5.4 Unit testing

When writing the adapter we first write the unit tests to validate the communication between Java and MATLAB. *Matlabcontrol* has been fairly well documented, but we implement unit tests to validate that the results are as expected. The type conversion between the two systems (Java and MATLAB) has to be correct, and the

return types has to match the expected types found in puka.

To test that arrays are imported correctly we perform matrix addition and multiplication to verify that both Java and MATLAB gives us the same result.

Java uses zero-indexing for the arrays. The first element in a given array is given position zero, as opposed to MATLAB whose index starts with *one*. The conversion of indexing is handled by the *matlabcontrol* library, so no consideration has to be made to this potential problem, but it can be important to make note note of the difference. To prove that this is handles correctly, we write a unit test to verify that values at the first and last position of an array are the same.

JUnit is used for testing the methods in JMatLinkAdapter. The tests has to cover each method used by puka (Listing 4.5). They also have to verify that results are as expected in order to be certain that the type conversion works as expected.

Some of these tests has to rely on other parts of the interface to be able to automatically assert the result. In addition we add print-statements to both the MATLAB window and the standard output. The only test for which we can not find a JUnit assert solution for is the debug print from matlabcontrol.

In order to be able to test for exceptions we let all methods throw exceptions. This leads to changes in the puka source code. For example for *engGetScalar* we need to add *throws MatlabInvocationException* statements to the methods that makes use of the method and surround the callee with a try/catch statement.

Chapter 5

Re-purposing for real time analysis

The algorithm used for respiration analysis in puka was originally implemented to be run on a pre recorded signal and with user input to verify and adjust the parameters during the analysis. We take a look into what changes to be done in order to automate the analysis and also provide a capability to analyse data streams instead of pre recorded signals.

The intention is to create low level events that TRIO can use to derive useful information about the sleep quality and detect deviations from normal sleep in real-time. The achieve this goal we do the following:

1. Make puka work without user input and

2. detect and report low level events in real-time.

By performing the signal analysis in real-time, TRIO will be able to detect sleep disorder symptoms and report it to other actors that can actuate based on this information. puka has algorithms for finding peaks and troughs based upon the surrounding signal, so we can not classify a peak the instant it occurs due to the definition of a peak. A peak is the highest point in a given section of a signal, surrounded by increasing and decreasing amplitude. What

we want to achieve is an event detection as close to real-time as possible.

5.1   Terminology

As we have already looked at the what terms puka uses to describe signals in Section 3.2, we need to adapt and change terms as we look at creating a real time implementation based on the puka application.

The use of the term *record* makes little sense when creating an real-time system. As the source of data for the analysis changes from pre-recorded persistent signals to volatile ever changing signals, we need a more descriptive name. A *data stream* is a more suitable name for the type of data source used in the real-time version. A data stream is defined by USDCs Federal Standard 1037C as *A sequence of digitally encoded signals used to represent information in transmission*[66]. Another way of describing it is data sequence being transmitted with a sequence of time intervals.

While puka uses the term *clip* to indicate an area of a *record*, we use the term *window* as it is a more descriptive and standardized term in the literature for real-time applications such as Data Stream Management Systems. There are different window types that will be discussed in Section 5.2, but the important part for now is that this is a selection of a a set of sequential tuples, which is stored in memory for analysis.

When we refer to the *online* version of the reduced version of puka, we are describing the part of the system that reads a *data stream* and analyses it as data arrives. When using the *offline* we are referring to a part of the application which analyses a local file, using the reduced version of the respiration analysis without any user interaction.

The individual components of the system are described in Section 5.3. *puka* refers to the original implementation found on PhysionNet, but with the modernization implemented as described in Chapter 4.

5.2 Design

Looking at the design of the algorithms, we have identified the parts that require changes in order to be able to supply TRIO with useful events close to real-time without any intervention from a human user. When executing the original implementation puka the user is prompted to validate the *onset time*, *peaks and troughs* and the *pauses* detected by the system. Each of these interactions has to either be removed or redesigned in order to be able to analyse the respiration as the data sequence arrives.

We change the flow of the controlling Java application from a file centric to a stream centric paradigm, allowing it to analyse data as it arrives. The record will in our real-time implementation be a *data stream* from a sensor simulator and we analyse *windows* within this data stream.

5.2.1 Automation

The basis for the real-time version is a reduced version of the modernized implementation of puka without the user interaction to adjust threshold and validity of peaks and troughs. Ideally these interactions can be automated for a more accurate analysis. These interactions are the *peak detection* and *classification* which both use a threshold to determine what constitutes a peak or through.

Noise and the *threshold*

For real world signals we expect them to have variations in the amplitude. A sensor can subject to calibration errors and misplacement. Worn sensor also are exposed to movement unless the subject wearing the sensor is completely still, which is highly unlikely. Sensors also add noise to signals as no sensor is 100% insensitive to other inputs. In addition to recording the desired input a sensor will also record unwanted environmental input. All these factors contribute to noise in the signal.

To be able to distinguish peaks and troughs caused by noise puka uses a *threshold* in the *peak detection algorithm*, which defines the threshold for defining a peaks and trough as such, discarding peaks and trough which are not distinct based on the value of the threshold.

When a user is prompted to modify this value, a plot of the time series is shown, which helps the user determine the required value of the threshold. As we are trying to remove these manual interactions between the analysis and the user, we look at two main approaches to automate this evaluation.

1. Signal processing, clean the noise from the signal,

2. Adaptive threshold.

There are many techniques for removing or reducing noise in physiological signals. Signal smoothing, or filtering is used in many scenarios and can be done in both hardware and software. In our case we want a software solution as we want the system to be hardware agnostic. We do not want to be dependent on a given sensors filtering capabilities.

Given a smoother signal, we can assume the existing noise tolerance within the peak detection algorithm given the default value to the threshold is good enough to be able to distinguish true respiration events form false events. Even if it is not, it is favourable to have a signal with a minimal amount of noise.

Commonly used signal smoothing algorithms are *moving average algorithm* and *Savitzky-Golay algorithm*. The main criteria that we have for a smoothing filter is that it is fast enough to be able to smooth the signal and analyse it before the next window is ready for analysis.

The *moving average filter* takes an array of raw (noisy) data $[y1, y2, \ldots, yN]$ and converts it to a new array of smoothed data. Each point in the smoothed data is the average of an odd number of consecutive $2n + 1$ ($n = 1, 2, 3, ..$) points of the raw data.

A better procedure than simply averaging points is to perform a least squares fit of a small set of consecutive data points

to a polynomial and take the calculated central point of the fitted polynomial curve as the new smoothed data point. The filter was describe by Savitzkay and Golay in 1964 [58] and is widely used in many scientific fields.

Savitzky and Golay were interested in smoothing noisy data obtained from chemical spectrum analyzers, and they demonstrated that least squares smoothing reduces noise while maintaining the shape and height of waveform peaks [59]. This seems to be a good fit for our purpose, and the implementation of the filter already exist in MATLAB. This allows us to test the filter on both synthesised data in **??**

Many of these algorithms are already implemented and available in MATLAB. Which algorithm is best suited for our intentions is difficult to predict, and most likely, further steps within the realm of signal processing will have to be taken to smooth and flatten a real world signal to make it as readable as the synthesised. For this thesis we focus on the real time and data stream aspect and discuss the need for more advanced signal processing based on the results found in Chapter 6.

Another approach to automate the validation of peaks can be to adjust the *threshold* based on metrics detected in the signal. Examples of metrics that can be used to calculate the threshold can be Signal-to-Noise ratio or average signal strength.

By keeping track of the average amplitude of a given signal, we can assume that peaks and trough that have significantly smaller amplitude than this average are noise. The definition of *significantly smaller* has to be adjusted based on the qualities of the sensor and signal and require analysis and initial setup based on the signal time-series generated by the sensor in operation.

This approach introduces problems of its own though: sudden changes in amplitude can result in false positives or false negatives until the average reflects the change.

The different approaches have to be tested with noisy signal and have functionality and performance compared to determine the better solution. When testing with ideal synthesised signals without any noise added, this parameter can safely be ignored.

Both approaches require extra computation as we need to evaluate or process the signal, but the adaptive threshold can be calculated while the analysis is carried out, reducing the need to iterate through the signal window multiple times for each pass of the peak detection algorithm.

Onset time

The reason for the onset time is not stated anywhere in the puka documentation or within the source code. The installation manual refers to *onset trigger* channels present in recordings which are used to find the initial onset time. Since the simulated data or data streams not will not include this channel this parameter is ignored. puka instead calculates the onset using the *findOnset* script described in Section 3.8. The original version of puka updates both the *onsetTime* and *endTime* variables, shifting position of the clip. This shift in position can be a problem if the *recording* is not long enough for the entirety of the clip size.

Since the only documentation for the algorithms beyond the peak detection, we have to make some guesses as to the intention of the onset time. One such assumption can be that it is done to align the *clip* with a zero crossing, not using time of the analysis on un-eventful parts of the signal. Since we want the implementation to analyse all data-points we ignore this parameter, setting it to have a default value of 1 as MATLAB arrays are one-indexed.

5.2.2 Real-time

For our purposes we create a program that feeds data into the stripped down version of puka which only contains the respiration analysis. We need to create a Java controller class which will control the execution of the puka respiration analysis. This controller handles communication with data stream sources and initiates the analysis. This controller also handles the historical data which might be needed to extend a given window to be able to detect peaks and troughs that are too close to the onset of the window to detect otherwise.

Window

The following section explains the main design discussion related to windows, i.e their semantic and the need for historical data.

Since the application receives data stream as input we need a window model in order to define the *window* in order to continuously analyse sequences of the data stream and to avoid the need to receive the entire data stream (which can per definition be infinite) to perform the respiration analysis. As discussed in Subsection 2.3.3, windows in Data Stream Management Systems are either time based or tick based. For our purpose, both these paradigms mean the same, as the ticks or data points are sent at a constant rate. This is not accounting for data lost in transmission or latency. But in principle we use a timed window to read from the data stream as the sample rate is constant. If we have a sample rate of 1000 samples per second, a length defined window of 1000 ticks is practically the same as a 1 second window.

A tick or count based system is easier to implement, and is therefore the preferred technique in the initial implementation.

Another parameter when designing the window model is how to update the window. There are different approaches to this, and we need to choose one that allows us to avoid detecting the same events multiple times but also make sure that the entirety of the data stream is analysed.
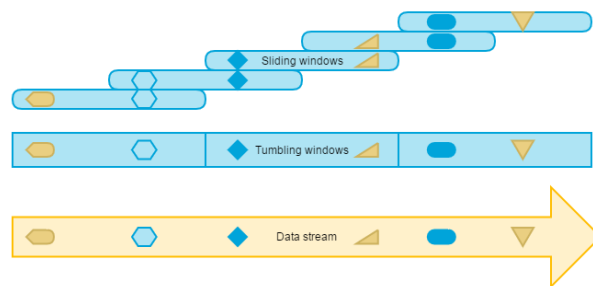
Figure 5.1: Sliding and Tumbling Windows

Sliding windows overlap as shown in Figure 5.2. Each window slides into the next one and for our purpose will require us to

evaluate whether the results of subsequent windows have discovered the same events and discard duplicate results.

A simpler implementation is *tumbling windows*, where there is no overlap, and therefore no need for duplication checks. A potential problem with such a solution is that events in the intersections of windows are lost. Due to time constraint and complexity added by a *sliding window* solution, the *tumbling window* solution is implemented first and analysed in Section 6.5.
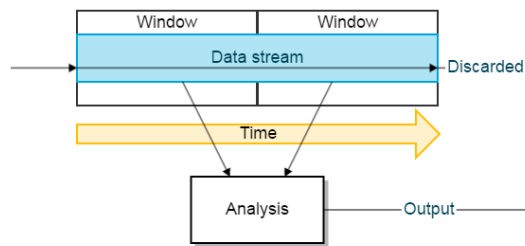

Figure 5.2: Simple window implementation

History

When the running the analysis with windows, there is a concern that peaks and troughs in the intersection of windows will not be detected, due to the design of the algorithm. If a peak or trough is close to the very end or beginning of a signal, there is not enough information to determine that it is dominated by either preceding or subsequent signals.

In order to detect the events in the border regions of a windows when using tumbling windows, we need a scheme to preserve a segment from the previous window and add it to the next window in order to avoid discarding data with undetected events.

By temporary storing the remaining signal in the current window after the last found event we can add the remainder to the new window. A trade off has to be taken into account for the implementation. We can choose to either have a set window size for the analysis or create a solution with a *adaptive* window size.

As it is very important for the system that the analysis is completed as close to real-time as possible it is tempting to use a fixed size, as this gives us a more predictable analysis execution time.

A different approach is to append the window to a *history window*. The history window consist of the respiration signal which is left over from the last analysis execution. By left over, we refer to the data after the last detected peak or trough. The increased window size has to be reflected in the appropriate variables in the analysis.

Parallelism

To avoid the network reading blocking for the analysis and vice versa, both of these aspects are implemented in *threads*. A thread is a lightweight process that shares the memory space with other threads created by the same process. This allows for concurrent execution and the shared memory space allows for inter process communication. This is used to ensure that all the data sent to the application is read even while the analysis is executing.

5.3   Implementation

Both the algorithm for detecting peaks and troughs and the algorithm for calculating pauses in respiration can be modified in order to improve the real-time capability of such a system, but as stated in the previous section, the first iteration focuses on the surrounding Java controller implementation. There are 4 main parts of the implementation of new *puka* package:

1. isolate and extract the respiration analysis, remove GUI and user interaction from the analysis application,

2. create application for serving data,

3. initiate analysis from controller that reads sensor data in real time,

4. read and evaluate the results

These parts are then tested and validated using the original puka as a reference and the metrics described in Section 6.1.
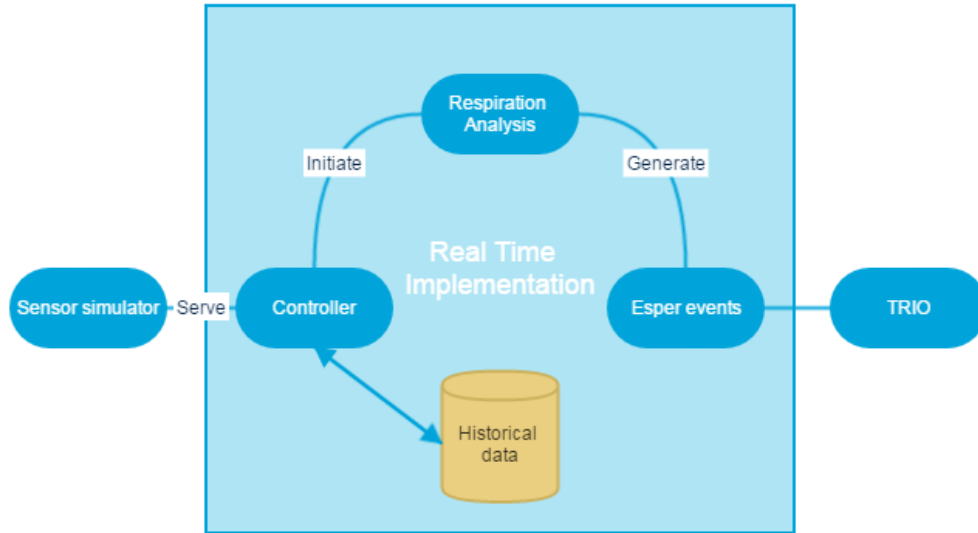

Figure 5.3: General flow of the real-time system

5.3.1 puka reduced

This application controls the user-application interaction via a text shell. This shell allows a user to initiate the respiration analysis either from a local file or request data from the serving service described in Subsection 5.3.2. This application is the basis for the testing framework and is designed to easily accept new types of signal or analysis. The application stores results from all executed analysis and controls the evaluation of these results.

The first iteration of the real-time implementation does not make any significant changes to the MATLAB implementation of the respiration analysis, but instead focuses on the Java control structure around the script. The part we want to isolate and extract is the respiration analysis found in puka and automate it. The resulting class structure will be as shown in Figure 5.4.
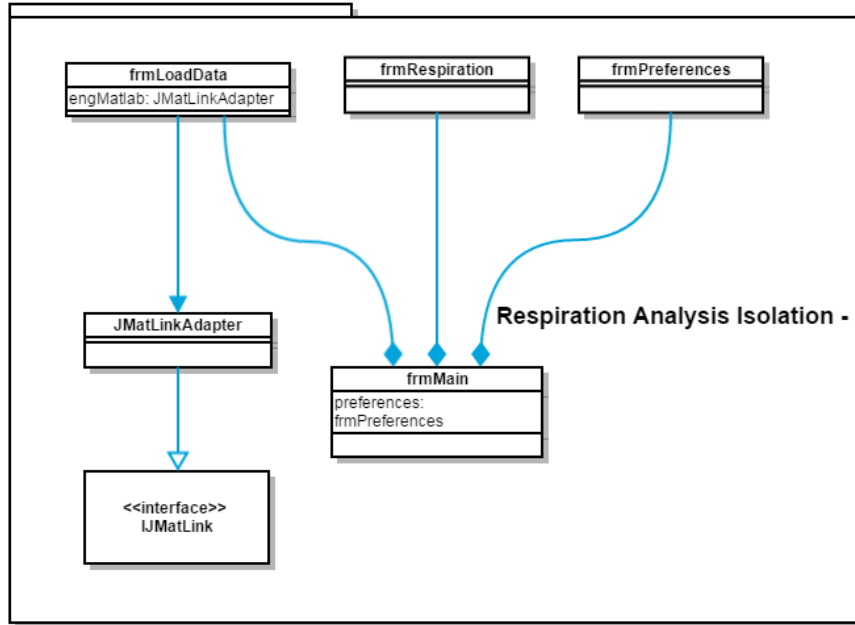
Figure 5.4: UML of the respiration analysis aspect of puka

The option to run puka on a local signal file is intended to mimic the original puka, and to test the now isolated respiration analysis. This option is initiated from a text shell and without the GUI and the user input during the analysis. When analysing data from the data serving service the application uses an adapted version, attempting to analyse the signals in real-time. Both options must be able to store the results in order to evaluate the results.

What is called a *clip* in puka is the same as a *window* in the sense that it is the size of the selection of a given time series we want to analyse. The smaller the windows are the less theoretical latency we get. In Section 6.5 we look into different window sizes and evaluate the limits to these parameters.

The main consideration when choosing window size is the *responsiveness* of the application. The smaller the window, the closer to real-time, but the number of false positives or negatives will necessarily increase as each pass of the algorithm will have

less data to base its analysis on.

There are limits to how small a given window can be and still being able to derive useful information. The signal sample rate plays a crucial role in mandating how small a window can theoretically be. Since we are looking at trends within a signal, we need to compare data points. If the sample rate is very low, for example 5 samples per second, we do not have enough data within a window size of 10 ms to derive any meaningful trends.

The *RespirationAnalysis* class is charge of the analysis and contains the control of the MATLAB code found in puka. We create a minimal and non interactive version of the respiration analysis found in puka and let the *RespirationAnalysis* class execute this code. Here we can easily introduce new implementations by adding new classes with minimal changes to the code.

The main procedures used to analyse the signal is extracted from the main application and reduced to a few calls to the MATLAB scripts with a minimal amount of the control code. These consist mainly of loading data to the right variables and setting other variables. It is at this step we can define the window sizes, based both on the size of the recording and also the onset time. The programmatic flow of the respiration analysis is shown in Section 8.4.4 in Listing 8.5.

The *newPT* function, which detects peaks and troughs in a loaded signal, takes a parameter with the *threshold*. This is also manually set and adjusted by the user. Ideally we implement functionality that keeps track of the average amplitude of the signal in order to dynamically adjust this parameter to fit the signal and avoid errors.

Due to the implementation of the respiration analysis, we need to store data that might contain events from the previous window. We what parts of the preceding window based on the location of the last found peak/trough. The simplest solution is to indiscriminately keep the remaining signal from the last peak or trough, or we can simply check to see if there is an onset point within the remaining time series. As suggested in Section 5.2.2 the signal is prepended to the next window before analysis. If this is

not done the application has no way of detecting peaks and troughs at the very beginning or end of signals.

5.3.2 Data serving

We create an application which reads the data files and serves them to the *puka reduced* application, simulating a sensor. This application functions as a connection oriented data producer, that generates a data stream from the signal files and serves them at a constant rate.

A very simple text based protocol is implemented in order to control the flow of data. The connection phase consist of a simple handshake between the client and the server, where the client sends the server the name of the signal. The server then reads the signal file and stores it in memory to reduce the number of disk IO operations. The size of a given signal file will vary based on the *length* and *sampling rate* of the time series.

Table 5.1 contains the first iteration of the protocol. To keep the protocol extensible and easy to read we reserve a range for different types of communication. $2xx$ is acknowledgements, $3xx$ is modifications and commands, and $4xx$ is reserved for errors.

Function	Parameters	Status code
Request file list	<empty>	REQ
Request file	<file name>,<num>	REQ
Acknowledge OK	<human readable message>	200
Request new rate	<requested rate>	300
Abort	<empty>	400

Table 5.1: Calls from client to Data Feeder service protocol

The software that serves the data is implemented in Java and uses Java NIO socket channels to send data to a connecting application. We also need to make sure that the program is implemented efficiently enough as to be able to send at a realistic rate.

70

Since both the data serving application and the data consuming application is running on the same system, use Java *System.currentTimeMillis*. By using milliseconds we can send data at a rate up to $1000hz$, or 1 per millisecond. To serve the data in a timely fashion we look at two libraries found in the Java language.

The class *java.util.Timer* contains functionality to schedule execution of execution of task in a background thread, either one of execution or repeated executions at regular intervals. According to the documentation[50], the class does not offer real-time guarantees.

ScheduledThreadPoolExecutor inherits from *ExecutorService* class which is found in the concurrent library. *An Executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks*[49]. *ScheduledThreadPoolExecutor* build upon this and pre-allocates n number of threads to execute the task which is being set up, thereby reducing the overhead of creating and starting new threads.

To allow the recipient to determine the boundaries of each data point, we wrap each discrete piece in greater than and less than brackets as such: *<data>*. This is due to the fact that there is no guaranty that each data point will be sent in its entirety. This way the client can reconstruct the data if it broken up and we avoid errors such as the one shown in Listing 5.1.

Listing 5.1: Received data entries with parse error

```
x: '-0.911736'
x: '-0.'
x: '911056'
x: '-0.910373'
```

5.3.3 Reading data and initiating analysis

The receiving application stores the data in a buffer until we have enough data to fill a window, and then applies the algorithms that have been adapted from the ones found in puka. The analysis

method will have to be timed in order to discover how much time we can expect to use on the analysis itself, which in turn puts restrictions on the theoretical window sizes.

Timing is important for the application to work in real time. We have to make sure the analysis does not blocks for the data arriving. We time each module to be able to evaluate how an alteration to either the control structure or the analysis affects the execution time. We also extract both the communication and analysis into separate threads so that neither blocks the execution of the other.

Running in a separate thread is the part of the application which connects to and reads from the server described in Subsection 5.3.2, and checks the integrity of the data. It also keeps track of the current size of the buffer and notifies the analyser when we have enough data for a window. The data is then copied to the the analyser thread and the buffer is cleared and we keep reading data from the simulated sensor while the analysis is conducted in the *RespirationAnalyser* thread.
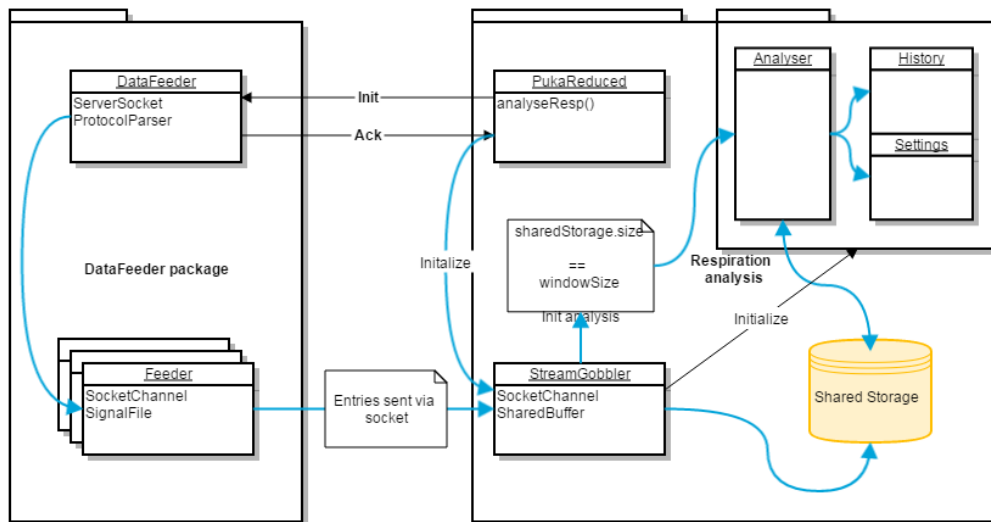
Figure 5.5: Details for the implementation

Part III

Results

Chapter 6

Evaluation

In this chapter we present the evaluation of the original implementation of pukas respiration analysis. We evaluate whether the automated version and the stream reading implementations use of the respiration analysis is comparable to the original and if the goal of creating a close to real time application based on the analysis is feasible.

To achieve this we use metrics that allow us to do a comparison and evaluation of the different version. We want the real-time version of puka to be as accurate and precise as the original implementation on top of being able to process a data stream in a timely fashion.

We have three versions of the respiration analysis to evaluate:

1. Modernized vanilla puka,

2. puka reduced offline, and

3. stream based puka online.

The *modernized* version contains the original puka respiration including the GUI and the manual verification steps. In the reduced and automated *offline puka* we have stripped away the interaction between the user and the application during the analysis

as well as the GUIO in order to be able to create the *online puka* version, which receives data from a stream instead of reading a prerecorded signal.

The modernized implementation of the original puka is used as a reference for the other tests. We compare the automated offline version and the online version with the metrics described in Section 6.1.

First we test the modernized application is demonstrates that it works as described in the documentation. This application will function as the reference for testing the offline puka reduced version.

For the *offline* version we verify that the results from the analysis on different types of signals correspond the reference, while for the real-time we experiment with different clip/window sizes to see how close to real-time the application can detect an event with the existing implementation puka respiration analysis algorithm.

6.1 Metrics

A metric is a unit of measurement which reflects a quality of the measured something, in our case the respiration analysis system. In this section we look at methods which gives us different metrics used to evaluate the results from the real-time implementation compared with a reference result created with the original implementation of puka. By running the same signal through the different implementations we can do a comparative evaluation based on the metrics.

6.1.1 Timing

Since the design of the real-time implementation implies the need for previous windows to be analysed before the subsequent window can be analysed, we need to make sure the execution is able to

execute within the limits set by the window size.

In order to evaluate the real-time implementation we look at the *execution time* of each component. The execution time is dependent on the hardware of the environment the application is running on. The specifications of the test platform are described in Table 6.1.

Determining the *average application response time* can be achieved by creating an application that runs the analysis steps n times, timing the each execution and calculating the average execution time. The size of the signal will also have to be taken into consideration.

OS	Windows 7 Enterprise (64bit) SP 1
CPU	Intel Core i7 2.9GHz
RAM	8GB DDR3
HDD	80GB 7200rpm SATA

Table 6.1: System specification for experiment environment

The components we time are the constituents of the respiration algorithm which are *data loading*, *peak*, *pause-detection*. We want to be able to time each component in order to identify potential bottlenecks and be able to see the distribution of time spent on each component.

We create time stamps between each step using the Java library method *System.currentTimeMillis()*. The metric we get from this is the average execution time in *milliseconds*.

For the real time execution we look at the differences in execution time time for different window sizes. The different signal processing operations described in Section 5.2.1 are also timed using the millisecond timestamps before and after. After having registered the time consumption it can be used to compare different approaches.

6.1.2 Precision and Recall

The metrics *precision and recall* are often used in information retrieval with binary classification of the results.

Our reference analysis gives us the True Positives (TP). Any results from an analysis that are not in the TP set are grouped together as False Positives (FP). If a value that is in the set of TP, but not discovered by our system is called a False Negative (FN). A value that is not a TP in the reference analysis is classified as a True Negative (TN). These are the basic building blocks for calculating metrics such as precision and recall.

These building blocks can be represented as a *confusion matrix* which is a presentation of predicated and actual classification defined by Kohavi and Provost[32]. The matrix contains n x n and gives us the basis for qualitative metrics often used in machine learning and knowledge discovery processes. The 2 x 2 matrix for our purpose can be seen in Table 6.2. In the table *Reference* is defined as verified by the reference analysis, while *Result* are values that are found in the tested analysis implementation.

		Result	
		Yes	No
Reference	Yes	TP	FN
	No	FP	TN

Table 6.2: Confusion Matrix

Based on the building blocks that are TP, FP, FN and TN we can calculate the metrics *precision* and *recall*.

- $Precision = \frac{TP}{TP+FP}$

- $Recall = \frac{TP}{TP+FN}$

This metric is a good starting point for testing our application, as a result is either correct or false based on the reference results. Due to time constraints, we were not able implement some of the explored metrics described in **??**.

The *evaluation* class is a utility class which calculates these values based on what it calls *result* and *reference*. The method for calculating the precision and recall takes two arrays containing indexes of the events and the size of the original signal. It creates a new list of all potential hits, i.e. every index within the size of the original signal and assigns all of them to TN. Then we iterate through the *result* marking each index value as a FN as these have not been verified yet. To verify a index from the result we iterate through the *reference* array. If a given index is marked FN, this is verified, and every TN is changed to a FP as it is only found in the reference array.

As summary of the calculation would be to:

1. set all indexes in a signal sized array to *TN*,

2. go trough result and set registered hits to *FP*, and then

3. go trough reference and change index containing *FP* to *TP*. If the index contains *TN*, set to *FN*.

To create a ground truth we can use information which creates the basis for a synthesised signal or create a reference analysis based on the original puka implementation.

A references analysis is created by storing the *validPeaks* and *validTroughs* from the respiration analysis using the original puka implementation. Based on the *clip* size, we instruct the data feeder application to stop serving after n samples. For this we need to modify the code, adding a stop condition to the data feeder which matches the clip size used in the reference analysis.

For every received window analysed, the application appends the results to a text file for persistent storage and later evaluation. After the *online* application receives the 400 abort code from the server it analyses any reminding data in the window and then stores it as well.

When calculating the precision and recall, the application reads the persistently stored *results* from the analysis and the reference *results*.

We test the application with different *window sizes* to evaluate the performance based on the precision and recall metrics.

6.2   Test data

In order to evaluate the algorithms used for detecting respiration in puka, we use both real world recordings from PhysioNet to derive respiration events from different types of sensors as well as synthesised data. The respiration events derived with puka can then be used as input for the logical sensors found in the TRIO project. Ultimately we want to compare the real world data and the results from the manual analysis found in the apnea annotations in the PhysioNet data with the result from the automated analysis.

6.2.1   Synthetic data

We need different types of signals to validate and evaluate the application. To begin with ideal and perfect signals can be used to verify the functionality of the respiration analysis. Such signals are difficult to find in a large database such as PhyioBank, and it is also time consuming searching for the types of signal we want to test.

To be able to do verification and evaluations we therefore generate synthesised data to be able to recreate specific scenarios to test the application.

A clearly defined *ground truth* is defined when synthesising the signals as we dictate the respiration rhythm and patterns. The formula and manipulation of the signal dictates the placement of peaks and troughs and size and span of pauses in the signal.

By identifying different potential errors or challenges that can occur in real world signals, we can isolate and experiment with the synthesised version to map potential weak points and errors. These will be described in Section 6.4 and Section 6.5.

A normal respiration rate, *eupnea*, varies with age, activity, illness, emotion and pharmaceutical influence[34]. In "Delmar's Comprehensive Medical Assisting", normal respiration rates are defined in Table 6.3.

Newborns	44 RPM
Infants	20-40 RPM
Children (1-7 years)	18-30 RPM
Adults	**12-20 RPM**

Table 6.3: Normal Respiration Rate[34]

In order to evaluate the correctness of the algorithms used in puka it will be reasonable to run some experiments using synthesised data. Not only will we more easily detect errors and deviations when we have generated the data ourself, but we can also create different types of signals in order to verify that the system can analyse the various signal types.

The synthesised signals can be generated based on a sine function. The sine of an angle ω in a right triangle is as the artio of the lengths of the side of the triangle opposite the angle of the hypotenuse[2].

The sine function sin(x) is one of the basic functions encountered in trigonometry (the others being the cosecant, cosine, cotangent, secant, and tangent). Let θ (Figure 6.1) be an angle measured counterclockwise from the x-axis along an arc of the unit circle. Then sintheta is the vertical coordinate of the arc endpoint.[2].

A simple implementation of a sine function in MATLAB gives us a smooth sine curve and a time series that can be
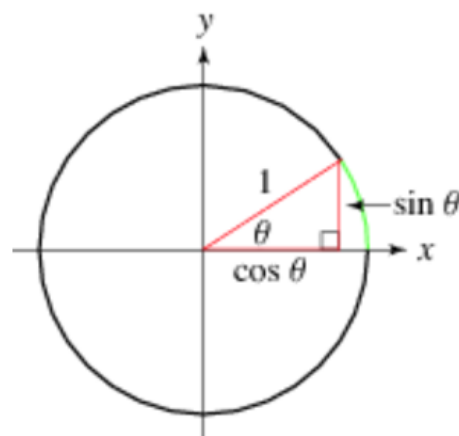
Figure 6.1: Sine definition

80

used as an ideal respiratory signal.

With the basis in the formula that gives us a smooth sine curve we can create a time series that is more akin to actual respiratory signals. Actual respiratory waveforms for signals such as RIP are not smooth, and the during recording several types of noise are introduced. We create a function for adding both random and deliberate noise to a generated signal.
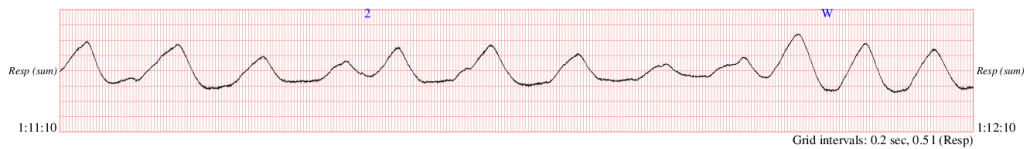

Figure 6.2: Respiration signal from 1 minute of stage 2 sleep [14]. Inspiration for synthesised signal

As described in Evaluation of respiratory inductive plethysmography [5], waveforms from respiratory sensors such as RIP tend to have pauses, especially at the end of expiration.

Listing 6.1: The MATLAB script for generating a sine wave

```
numSamp = 0:2999; % number of samples total in each clip
am = 1;         % amplitude
base = 0;       % offset y axis
offsetX = 0.5;  % offset x axis
respDur = 3000; % duration of each "respiration"

% Simple sine wave frame
sinewave = base + am * sin(2*pi*numSamp/respDur);

%Respiratory like signal frame
respiratiorySignal =
base + cos(am * sin(pi*numSamp/respDur + offsetX) * pi);
```

By manipulating these synthesised signals we can create the different scenarios such as respiration stops, errors in sensors and so on. Some of the potential scenarios we want to control for are listed in Table 6.4. These scenarios are based on observations of
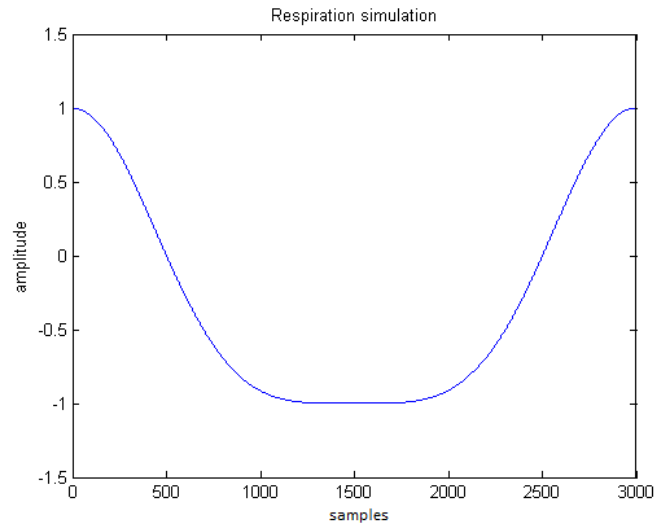
Figure 6.3: A 3000 samples long synthesised signal.

real world signals found in the PhysioBank database, and are but some of potential scenarios.

Using synthesised data allows us to more easily test different scenarios and aspects before moving on to more complex signals. We can also experiment to detect weakness in the puka implementation by adding and removing the different scenario.

white noise	random noise
amplitude drift	changes in amplitude
baseline offset	non-zero centred signal

Table 6.4: The scenarios simulated by synthesised signal

For each scenario we can implement a script that generates that specific signal. By combining these scripts we can compose the signals we need to preform the experiments to test pukas ability to handle the different scenarios creating signals that are similar to the real world examples such as Figure 6.3.

The data has to be converted into a format described in Section 3.7. These are trivial IO operations that require us to manipulate raw text files, delimiting the signal values with newlines, as is the delimiter puka uses when reading data.

White noise may be defined as a sequence of uncorrelated random values. The noise added to the signal is created by adding random deviations from the mean value of the signal, but maintaining the normal distribution. Ideally, based on the sample rate found in the RIP recordings in the CINC data set, we would use 100 samples per second. Due to quirks in the source code we start out using 1000 samples per second and change the scripts to not assume sample frequency. This is described in detail in **??**.

The different implementations can all be found in the MATLAB scripts found in the source code.

6.2.2   Real world data

PhysioNet is a collections of recorded physiologic signals and related open-source software for signal processing and analysis. eThe data available is from different institutions around the world and it contains a variety of digital recordings of physiologic signals and related data for use by the biomedical research community[45].

The data found in PhysioNet is stored as *.dat* files and has to be converted in order to be used in puka. The PhysioNet toolbox offers tools for converting signals into text. The *rdsamp* program is used to read a specified record, either from a local .dat file or from the on-line database. The output is the decimal number on standard output. If a record contains more than one channel it will write output from each channel on the same line separated by tabs. The function also takes parameters allowing us to create ASCII files with delimiter of our own choosing (such as comma to create CSV files), read certain intervals, add time data based on the information found in the header file for the record.

To identify suitable datasets we look for two main criteria: ground truth (AHI and apnea annotations) in order to evaluate the results, and signal types allowing us to use puka to calculate respiration events in a format equivalent to the supported types.

An ideal dataset contains different types of signals to enable us to compare the result on different sources. Annotations of respiratory events would be ideal and annotations for apneic events

facilitates verification of the logical sensor using the events generated by puka.

MIT-BIH Polysomnographic Database contains both a respiratory signal from a nasal thermistor and a respiratory effort signal from inductance plethysmography in some cases both chest and abdomen and others either one of them. Each record includes a header file, a short text file that contains information about the types of signals, calibration constants, the length of the recording. It also contains AHI and sleep stage and apnea annotations which makes it a good candidate for usage in our tests.

The St. Vincent's University Hospital / University College Dublin Sleep Apnea Database also has oro-nasal airflow (thermistor), ribcage movements, abdomen movements (uncalibrated strain gauges). As with the MIT-BIH database we have sleep stage and apnea annotations. In this database the annotation distinguishes between types of apnea, meaning we can control for the different types of apnea. It also contains annotations for other types of respiratory disturbance, meaning it can be a good candidate for future work. The database has both ribcage and abdomen movement recorded with uncalibrated strain gauges

Another good candidate for our purpose is the apnea test database "Data for development and evaluation of ECG-based apnea detectors" which was used in the CINC challenge in 2000. Some of the recordings contains thermistor and or RIP signals in addition to the ECG signal. In the cases where we have all three signals, we be able to compare the results from the different respiration rate estimation techniques allowing for comparative studies of the techniques. These records also contain the necessary AHI in order to determine the accuracy and precision of TRIOs logical sensors. The database also contains annotations for apneic events, which will allow us to see if there is correspondence between the same type of events found by the logical sensor and the data in the database.

All databases mentioned have different sampling rate, baseline and amplitudes, making the use of all of them valuable to identify different aspects of the puka respiration analysis.

6.3 Experiments: the modernized implementation

In *preferences.txt* or via the *Program Preferences* in the GUI we define which column in a raw text file where the signal used in the respiration analysis, as described in Section 3.6. When creating the synthesised data we only have one column, meaning this parameter will be set to 1. As we do not simulate ECG signals the ECG column will be set to -1, as stated in the documentation. The onset trigger will be set to 1 as well. This value will be updated by the execution of the respiration analysis.

When the application is launched it starts an instance of MATLAB, where we can observe the values being added to the current workspace. The application changes the working directory of the MATLAB instance to the MATLAB-scripts folder set in the *preferences*.

Based on the sampling rate we set the length of a clip. The clip has to be created prior to loading the data and initiation of the respiration analysis. At $1000hz$ we set the "clip length" to $1000 * 10 = 10000$ to make a 10 second clip. The recording we use in this example is 30 seconds of synthesised data, and should therefore give us ample room for the clip size to fit within the recording.
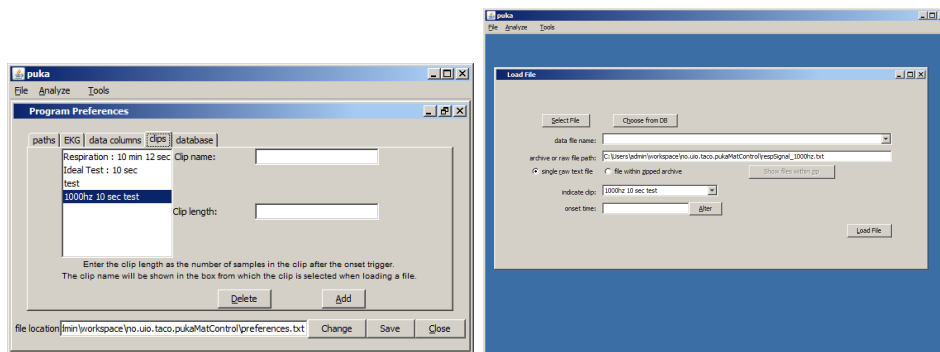
(a) The clips tab in preferences   (b) Load file dialogue

Figure 6.4: Initial steps for respiration analysis

Since we are not using a database for the data, we instead choose the file drop down menu and choose *Load File*. The button

Select File prompts a file selection window allowing us to choose the wanted signal file. When a file has been chose we must also select one of the pre-configured clip sizes. The *Load File* button initiates the actual reading of the file and preparation of the data by copying it to MATLAB as well as reading it into memory.

When loading the data the application looks for an *onset time*. *Onset time* is the first point in the recording where the signal crosses 0 on the Y-axis. The suggested onset time is shown in the input field but this can be changed by the by the user. The waveform is plotted using MATLAB to indicate where in the record the onset time is, and what the signal look like.
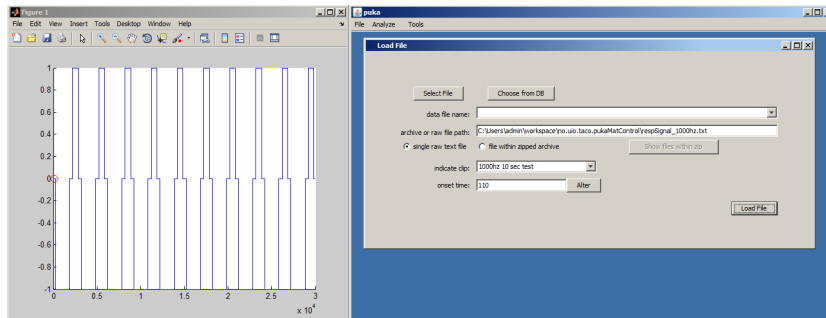
Figure 6.5: Onset time marked both in the input field and on plot

Now that the data is loaded and the clip aligned with the recording we can begin the respiration analysis itself. When it is initiated the user is presented with the window shown in Figure 6.6a. The steps presented in Section 3.8 (identify, validate, mark pause and centre peaks and troughs) are represented by the steps in the GUI. In addition we have a fifth step which is the statistical computations (also described in Section 3.8).

The signal in Figure 6.5 looks different from the one in Figure 6.6b. This is because of the MATLAB script *findOnset.m* uses approximations by rounding the raw signal in order find points close to 0. This was probably done for efficiency, but is not documented why this approach was chosen.

After having run the *calculate pauses*-scrip has completed we have two new arrays that are based on the previous *P* and *T*; *newP* and *newT*. For each peak and trough we have twin tuples marking the beginning and end of the pause around a peak.

86
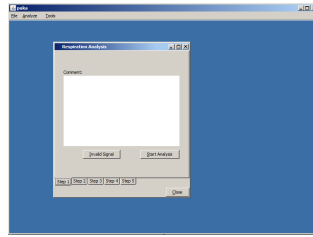
(a) Initial screen for respiration analysis
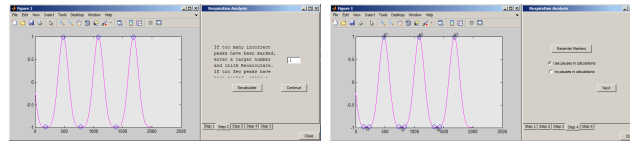
(b) Visual representation of validity

(c) Pauses identified in the troughs

Figure 6.6: The peak detection and validation

puka then visualises the pauses for the user and gives them the option to recentre the points or not use the pause information when calculating the statistical information, as shown in Figure 6.6c.

Finally the statistical computations are display for the user. This demonstrates the application now working on a synthesised recording of 30 seconds with a 10 second clip. The signal was constructed with each respiration lasting 3 seconds from start to finish. For this initial experiment, puka seems to agree (Figure 6.7).

6.3.1 Synthetic scenarios

By testing the scenarios presented in Subsection 6.2.1, we can detect what scenarios puka might not be best suited for, and what steps to take to mitigate the challenges discovered.

All analysis were run using a 45 second long record with a 30 second clip. The record has been synthesised to have a sample rate of 1000 samples per second.
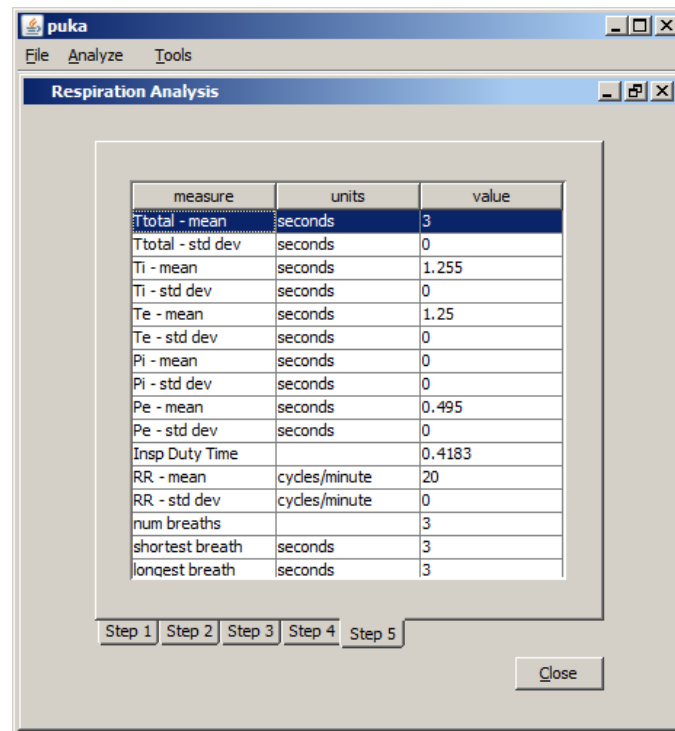
Figure 6.7: Statistical computations from a 10 second clip. Signal with 3 second respiration loops

| Base signal	Issues detected
45 second normal signal	identical result
45 second noisy signal	missed pause, missed peaks
45 second amplitude shift signal	threshold variable tested
45 second shifted baseline signal	failed onset detection
15 second flat signal	failed onset detection

Table 6.5: Summary of scenario testing modernized version

Flat signal

Edge cases, such as a flat or uneventful signal should be tested as well in order to know how the original implementation handles errors.

This is tested by simply generating a signal file with a single value in the entire range. The onset detection does not work on the

flat signal, and when given an explicit onset time of 1 the analysis fails at the peak detection. There is, in other word, no fault handling in puka for handling uneventful signal other than the onset time detection. We assume that is the function of the onset time. This aspect of the implementation that will have to be handled in the real time implementation and is described in **??**.

Noise

When running the analysis on a signal with added white noise in the way described in Subsection 6.2.1, pukas algorithm for validating peaks and troughs comes in to play. As shown in Figure 6.8a, all of the troughs marked as invalid, are in fact valid.
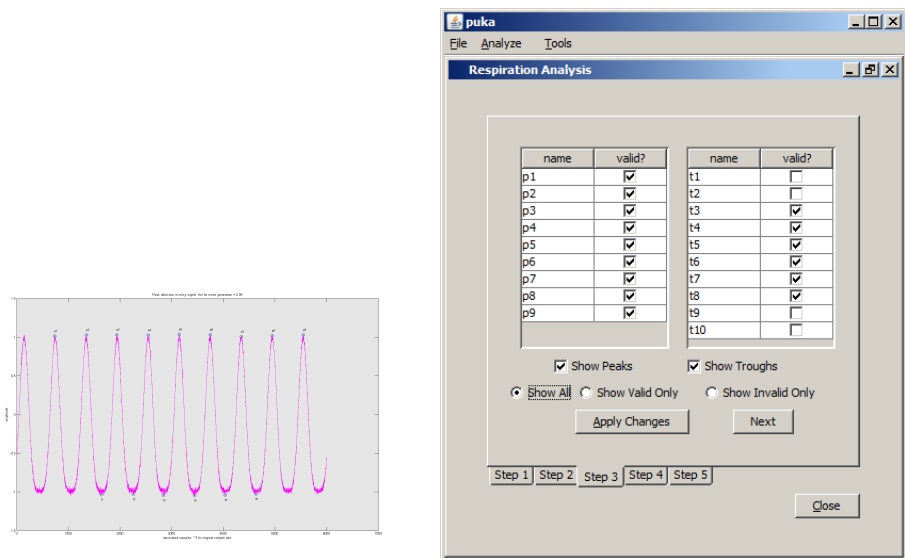
(a) Waveform with valid peaks and trough marked.

(b) List of valid peaks and troughs

Figure 6.8: Valid signals in noisy signal

We can employ the techniques described in Section 5.2.1 to *smooth* the signal when loading it into MATLAB. A different approach to handling a noisy signal is to update the parameters used in the puka peak detection algorithm and pause detection algorithm to make the resilient to noisy signals. This approach requires techniques to continuously monitor the amount and intensity of

the noise levels within the signal. Approaches to this are discussed in **??**.

Baseline shift

If the sensors has not been properly calibrated, the baseline of the resulting signal can end up not be centred around zero.

Based on the analysis of the peak detection algorithm and onset time detection, we see that puka require zero-centred signals in order to conduct the respiration analysis. As long as the signal is at some point able to be rounded to zero, the onset time detection will set this point as onset time. The peak and pause detection does not have the same zero centric dependency as demonstrated in Figure 6.9a. When running the same signal transposed 1 more unit up on the y axis (amplitude), the system is unable to detect a onset-point.

This can be counteracted by calibrating sensor appropriately, but that does not account for movement, interference or other disturbances that might be encountered in a real world scenario.

Amplitude drift

Amplitude shift is when the signal generated from a sensor does not display the same amplitude differences throughout the signal as is the case with the ideal case. In a real world scenario the amplitude might vary due to movement by the monitored subject or variability in the depth of a given breath.

How puka calculates the threshold based on the input from the user shown in Figure 6.6b can be demonstrated using amplitude shifts. Using a signal with three sections within the record that are multiplied with 0.7, 1.5, and 0.4 we can set the threshold to ignore peaks not within the desired range.
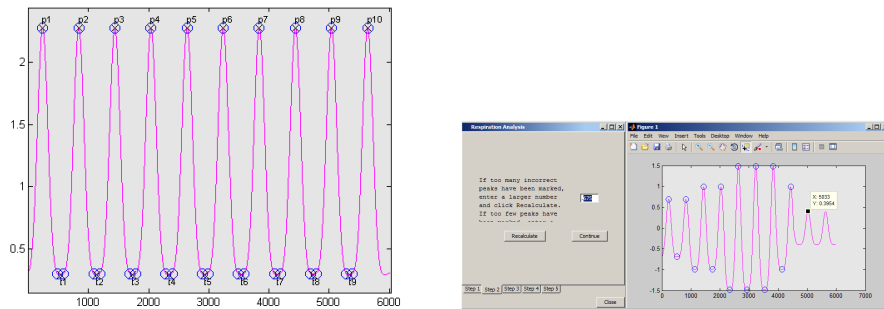
(a) Peaks and troughs detected with baseline shift

(b) Threshold adjusted to ignore peaks and troughs less than 0.4 amplitude

Figure 6.9: Execution of respiration analysis using scenarios on modernized puka

Results WIP

We now know that the white noise added to a signal makes the implemented pause detection in puka fail in detecting pauses even if the peak and trough detection works as long as the amplitude of noise is within the threshold which can be adjusted manually by the user.

Another scenario observed in real world scenarios is amplitude drift. Peaks that are not detected are not prominent. The signals found in PhysioBank does not have a uniform amplitude, nor does a given signal have the same amplitude for all respiration cycles. The maximum and minimum amplitude will naturally vary based on the sensor, calibration, position of the subject and the depth of breath. Again, the threshold for the peak detection defines the minimum amplitude of what constitutes a peak or trough.

When a signal is shifted in either positive or negative direction on the y axis the onset detection is unable to detect a start point in the signal, as it uses the first zero crossing to identify this point. For our purpose, this has little impact as we want to analyse received signals in their entirety and can therefore remove the onset detection from the real time implementation.

The most important finding when running the original implementation of puka on these scenarios is that the peak detection is quite robust as long as the threshold is adjusted to suit the signal. Alternately adjust the signal to match the ideal -1 to 1 amplitude range.

We now also know that the pause detection is very susceptible to random noise.

6.3.2   Real world data

After having run the analysis successfully on an ideal synthesised signal, we look at executing it real world data in order to evaluate if the same restrictions found in synthesised signals is present and if other problems make themselves apparent.

Obvious differences are that very few of the RIP signals found in the PhysioBank databases have the high sample rates which we have assumed in the synthesised signal. The reason for synthesizing the signals with a sample rate of 1000hz is due to comments found in the source code (see Section 8.4.4)

In the MIT-BIH Polysomnographic Database (*slpdb*) we can find multiple signals containing RIP signals with annotations both for *sleep stage* and for *apnea events*. Using a one minute clip from the recording *slp60* we look at what needs to be done before running it through puka.

First of all, we need to remove the header and time stamps from the file, creating a raw text file as described in Section 3.7. We create the *cleaner* application**??**, created for checking the integrity of the signal and also reformatting the file to adhere to the data format required by puka.

The signal have been amplified in order to be analysed by puka. Due to the onset detection not being able to find a valid onset point when all values (amplitude) of the data points in the time series is rounded to zero, we need to boost the signal. This can done by simply multiplying the time series with an arbitrary positive number. We used the following formula to find the factor

to use when amplifying the signal for this execution test: $factor = 1/max(timeSeries)$.

As the respiration analysis executed (Figure 6.10a) on the *slpdb* signal shows, the peak detection works well on real world data as well. The demonstrated run was done without tweaking *onset time*, *threshold*, or *validity* of peaks.
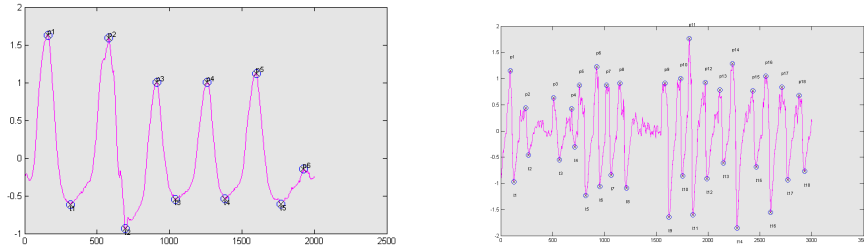
(a) Peaks and pauses detected in *slpdb* signal normal respiration

(b) Peak and pause detection in *ucddb* signal with apneaic events

Figure 6.10: Real world signals analysed with puka

As demonstrated in the experiments where we introduced noise into the synthesised signal, the pause detection of puka does not work well. Using a 2 minute signal from the UCDDB database with two annotated apneaic event we can evaluate whether the pause detection is able to detect stops in respiration. We evaluate both obstructive and central apnea events.

The reason for using this database for testing apneaic events is that it is very easy to find these events in the signal thanks to the separate *respiration event* file with time stamps and data for each apneaic event in the recording.

To be able to detect the peaks and troughs as shown in Figure 6.10b the threshold variable had to be set to 0.7. But even if we were able to detect the relevant peaks and troughs as expected, the pauses around them are not calculated, making it impossible to establish the onset and end of apneaic events.

```
Selected input: record ucddb/ucddb014.rec (Sum),
from [05:28:29.000 02/01/2002] to [05:30:29.000 02/01/2002]

Time     Type  ... Duration ...
```

93

```
05:28:38 HYP-C ...   26    ...
05:29:18 APNEA-O ... 12    ...
```

The expected onset and end of the HYP-C event to be 9 seconds after the beginning of the signal and last 26 seconds, which translates to HYP-C onset start being $9 * 128 = 1152$ and end being $(9 + 26) * 128 = 4480$ in number of samples, while the corresponding *t2* is found at 1330, with no pause detected.

There is no pause detected for the obstructive apnea either. The annotations suggest that the event should be at 49 seconds after the start of the signal, but pause could not be detected by the algorithm in puka. This is not surprising considering the waveform of the signal in Figure 6.10b, as it does not flatten in either case, but rather jumps inconsistently, albeit at a lower amplitude than the average respiration movement

`TODO: Demonstrate` Even with a S-G smoothed signal, the pause detection would still consider the slope as steep enough as to register as

6.4   Experiments: offline puka reduced

Since the experiments in Subsection 6.3.2 showed us certain weaknesses in pukas pause detection algorithm we focus on synthesised signals for the automation and real time versions of puka. If the proof of concept of using puka as a basis for a real time analysis tool we can later go back and adapt and improve the pause detection or introduce signal processing that will allow us to improve signals as they arrive.

In the automated modernized version we have removed the manual verification of the onset time, peak detection threshold, and peak trough validity.

The vanilla version is tested with the scenarios presented in Subsection 6.2.1 as well as signals from PhysioBank. To verify we compare the results from the vanilla puka with the results from the

offline and reduced version we describe in Chapter 4. The result of each execution is in the original version the statistical computations at the end of an execution. The statistical computation can be replicated by removing the GUI related parts of the *CalculateResp* method found in the *frmRespiration* class. These results will then be stored and compared with the results of the vanilla puka execution, which also has to be modified to store the results after the analysis.

Each step of the respiration analysis scripts stores data in numeric arrays within the MATLAB workspace, which we can retrieve and store persistently for comparison. We can achieve this by manually storing the workspace variables in the MATLAB instance, or automatically extract them via the Java controller class and write them to files. The variables containing results that are relevant for the evaluation are the *newP* and *newT*. They contain the results from the pause detection. Each array contains two indexes for each peak and trough, indicating the start and end of a pause.

Note that all indexes stored in these result arrays must me multiplied with 5 to give the correct index in the original signal, since the *newPT* script decimates the signal. This is described in more detail in Section 8.4.4.

Now that we have the result from the analysis in a format we can look at what metrics to use for evaluating them. When having run both the vanilla and the *puka reduced* offline component we compare the results and make sure they match using precision and recall. The results contain the indexes, or sample number in the original time series.

When conducting the analysis on the signal we use the original puka implementation as reference. We run the respiration analysis using the agreed upon clip size. In our case will be 30 seconds, which translates to 30000 samples in a 1000hz signal. After the analysis has completed we store the *newP* and *newT* variables in the MATLAB engine to files that will be read and used as reference when calculating precision and recall. For the automated version of puka we set the *clipLength* in the *Settings* class.

Running the *PukaReduced* application from the *pukaMat-Control* project we have created, we can initiate the automated or reduced version of pukas respiration analysis. The reference and result files are placed in the same directory before running the *evaluation* package, which calculates the precision and recall based on the two executions by merging both the resulting peaks and trough into one signal before evaluating the number of FP, TP, FN and TN values in the resulting sets.

Listing 6.2: Results calculating precision and recall on clean signal with and without respiration pause

```
=== Clean signal, normal respiration ===
Precision: TP(29.0) / (TP(29.0) + FP(0.0)) = 1.0
Recall: TP(29.0) / (TP(29.0) + FN(0.0)) = 1.0
Precision: 1.0
Recall: 1.0

=== Clean signal, stop in respiration
Precision: TP(18.0) / (TP(18.0) + FP(0.0)) = 1.0
Recall: TP(18.0) / (TP(18.0) + FN(0.0)) = 1.0
Precision: 1.0
Recall: 1.0
```

The metric values after executing the analysis using the *pukareduced* version of the analysis is *precision* = 1 and *recall* = 1, indicating identical results. This is expected as it is essentially running the analysis using default parameter values. This verifies that the automated version is equivalent to the original and therefore useful as such for the real-time version.

6.4.1 Flat signal

Edge cases, such as a flat or uneventful signal should be tested as well in order to discover how the original implementation of the respiration analysis handles errors. In order to reduce the latency of discovering events to the transmission of said event, we have to use as small windows as possible resulting in many of them being uneventful.

By uneventful signals, we refer to signals without changes in the time series in both directions significant enough to constitute a peak or a trough. We also experiment using windows containing only one type of event to evaluate the error handling for these edge cases.

Since we are reliant on the onset detection to verify that there is events to detect in a given window, we need to test whether the system is able to handle uneventful signals. We create two signals to check whether the system is able to operate when presented with a flat signal, and also to verify that To handle the scenario described in Section 6.3.1 we need to modify the algorithms used in puka.

The current implementation attempts to evaluate the signal by using the *findOnset* script. When the script fails to detect a good onset, the entire window is stored as history.

Using a 5 second clip of the signal shown in Figure 6.14a with one trough at 1500 samples or 1.5 seconds, the rest of the signal completely flat. The *classifyPeaks* script does not have any error handling for empty arrays thus aborting the analysis and discarding the potentially valid trough.

6.4.2 Timing

The components of the application has to be timed as well. We time the application as described in Subsection 6.1.1 and run the analysis on a 40 second clip giving us the following total execution time

	Execution time
1	1596ms
2	441ms
3	382ms
4	393ms
5	359ms

Table 6.6: Exectuion time for 40 second signal puka offline

97

Based on these results we calculate the average execution time of 634.2ms for five subsequent executions.

6.5   Experiments: online puka reduced

We take a qualitative look at different aspects of the stream based implementation using pukas respiration analysis. When using puka offline we have full control over what signals to use and what parts of these to analyse, and we have in Section 6.3 and Section 6.4 looked at the modernization and automation of the analysis. When working with streams we have no such control and therefore define experiments that reflect the nature of the difference in paradigm.

We design experiments testing:

- ideal signal as a proof of concept demo,

- uneventful window,

- events stretching across windows, and

- execution time.

Using a clean respiration signal for our initial experiment, the real time implementation is able to detect all peaks and troughs, with correct pauses. The *reference* or *ground truth* is created by the formula used to create the signal. Each signal is evenly distributed evenly and is initiated to the minimum value in the signal (-1). This results in the peak being centred in each 3 second frame. At 1000hz there is a peak at every 1500 sample of the signal.

6.5.1   Event detection comparrison

Running the real time implementation **without** the *history implementation* on *clean synthesised signal* show us that the peak detection works as expected.

Ground truth	Result	P/T
1500	1500 - 1500	P
3000	2755 - 3250	T
4500	4500 - 4500	P
6000	5755 - 6250	T
7500	7500 - 7500	P
9000	8755 - 9250	T
10500	10500 - 10500	P
12000	11755 - 12250	T
13500	13500 - 13500	P
15000	14755 - 15250	T
16500	16500 - 16500	P
18000	17755 - 18250	T
19500	19500 - 19500	P
21000	20755 - 21250	T
22500	22500 - 22500	P
24000	23755 - 24250	T
25500	25500 - 25500	P
27000	26755 - 27250	T

Table 6.7: Analysis on clean signal

The ground truth is defined by the generation of the signal. The signal is based on the synthetic respiration script **??** abd has been manually manipulated to contain the pause in Table 6.8.

Comparing the result of a reference analysis done using the original puka, show us that exactly the same events were detected. In Figure 6.11 is the results from both the reference analysis and the windowed analysis, illustrating a *precision* and *recall* of 1. For every point on the x axis is a the detected events with their placement in the signal on the y axis. All events overlap.

Running the real time implementation **without** the *history implementation* on *synthesised signal* with a 6 second pause surrounding sample *15000* using 10 second *tumbling windows* also works as expected.
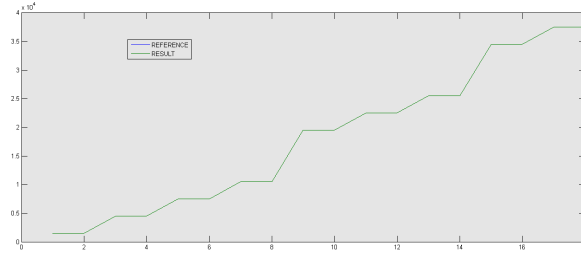
Figure 6.11: Comparison of results from offline and online version using an ideal signal with two six second pauses as a proof of concept

Ground truth	Result	P/T
1500	1500 - 1500	P
3000	2755 - 3250	T
4500	4500 - 4500	P
6000	5755 - 6250	T
7500	7500 - 7500	P
9000	8755 - 9250	T
10500	10500 - 10500	P
15000	11755 - 18250	T
19500	19500 - 19500	P
15000	20755 - 21250	T
16500	22500 - 22500	P
18000	23755 - 24250	T
25500	25500 - 25500	P

Table 6.8: Execution with signal including a six second *pause*

The duration of each pause is defined by the threshold of the pause detection. A pause-tuple is considered correct when they contain the peak or trough index.

Running the real time implementation **without** the *history implementation* on *clean synthesised signal* using 5 second windows.

Ground truth	Result	P/T
1500	1500 - 1500	P
3000	2755 - 3250	T
4500	4500 - 4500	P
6000	5755 - 6250	T
7500	7500 - 7500	P
9000	8755 - 9250	T
10500	10500 - 10500	P
12000	11755 - 12250	T
13500	13500 - 13500	P
15000	14755 - 15250	T
16500	16500 - 16500	P
18000	17755 - 18250	T
19500	19500 - 19500	P
21000	20755 - 21250	T
22500	22500 - 22500	P
24000	23755 - 24250	T
25500	25500 - 25500	P
27000	26755 - 27250	T

Table 6.9: Execution clean signal

TODO: How small can we go?

6.5.2 History

As long as a pause is within a window, the system has no problems detecting the peaks, troughs, and pauses in the signal. We design a signal which has the pause which stretches over a window size (10 second in this example).

We therefore create a signal with a pause that stretches across a window border. When running it without the history:

The areas marked with red circles in Figure 6.12 indicate the position of the missing troughs. We have assumed that the windows will not necessarily align, and have implemented a mechanism for preserving the information after the last detected event

(a) First window　　　　　　　(b) Second window
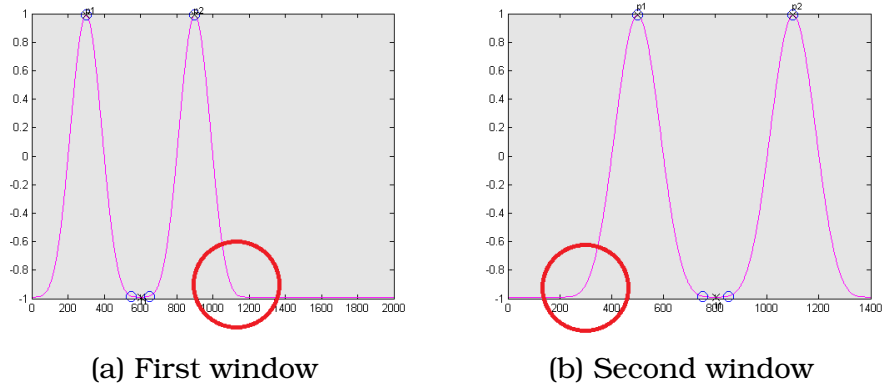
Figure 6.12: Pause in respiration missed due to window size and alignment

and include it in the next analysis window, more thoroughly described in **??**.

When running the same signal with the history implementation activated, we see that the missing troughs from Figure 6.12 are detected in Figure 6.13.
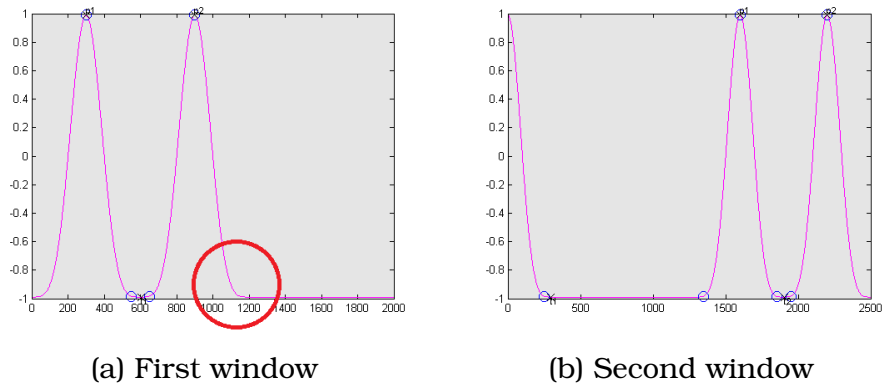

(a) First window　　　　　　　(b) Second window

Figure 6.13: Trough with pause contained within the adaptive window

6.5.3　Flat signals

When running the 20 second signal (Figure 6.14a) using 5 second windows we observe that, as with the result in Subsection 6.4.1,

102

the trough is detected by the peak detection, but the classification of the events causes the analysis to halt. The next window is constructed correctly, but the analysis generates numerous false positives, as demonstrated by the precision calculation: $Precision : TP(0.0)/(TP(0.0) + FP(953.0)) = 0.0$.

```
Start analysis with window: 5000 History: 3480
   Step 1: load data, set start and end time
   ==========================================
```

The same experiment was done with both a peak and trough in the preceding window, which leave us with the same result, detecting a large number of false positives. The same happens when the signal being sent begins with the event less window as seen in Figure 6.14c.

When using a completely flat signal, the use of the onset detection allows us to avoid attempting to analyse these types of signals by simply adding the entire window to the history.
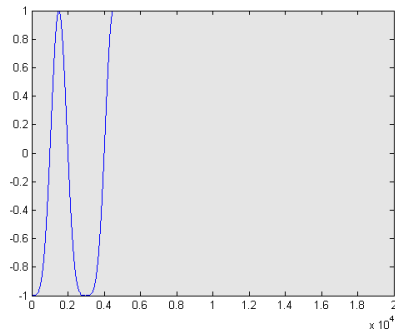
(a) Test signal containing multiple events
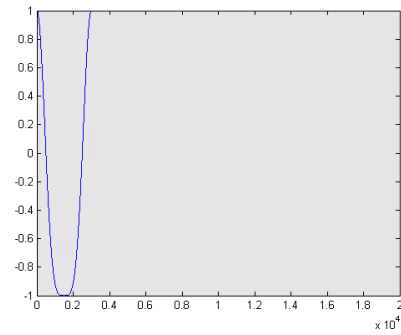
(b) Test signal with one trough in first window
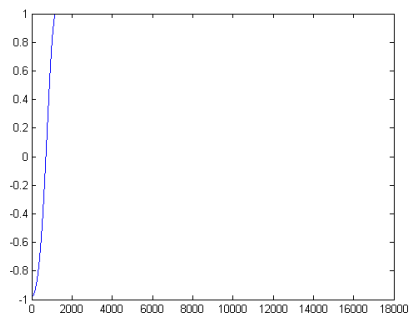
(c) Removing all events, the slope preserved by history
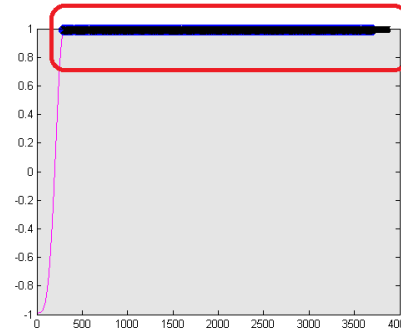
(d) The resulting cluster of *false positives*

Figure 6.14: Signals used for experiments with flat signals. All signals flatten at y-axis: 1

Chapter 7

Discussion

Based on the results from the experiments conducted, we have discovered many aspects of pukas respiration analysis. The documentation for the analysis is sparse, but the experiments clarified many of the challenges of using puka as a foundation for a stream oriented system.

7.1 Modernization

The process of modernizing the puka application has yielded insight into the potential lifespan of libraries such as JMatLink. Even though it is a useful tool, the way it was designed made it ill suited for longevity. The dependencies on system libraries and specific compiler support from the environment made it tightly coupled with the OS and version or MATLAB from the time of the creating. MATLAB as a widely used high level language and has a lot of existing tools and libraries for signal processing, making the exploration of Java - MATLAB bridges hopefully useful for future projects that rely on digital signal processing.

Certain aspects of puka are briefly mentioned and not explained in the documentation which accompanies the source code. Such as the intended function of the *detect onset* function, and why the signal is not automatically processed from the start. This

was discovered to mostly be a convenience for application users, as the application has no error handling for clips with no events. The automatic detection of the onset time can is analogous to removing white-space characters in the beginning of a string.

We do learn that the respiration analysis have problems detecting pauses both in synthesised signals with added white noise and in real world signal. This was not addressed in this thesis due to time constraints, but briefly discussed in Section 8.3.

7.2   Offline version

After having removed the necessity for manual adjustment of the peak detection threshold and prompt for correction of the peak and trough validation, we have a better insight how to eventually create the suggested adaptive threshold (Section 5.2.1), but due to time constraints we were not able to implement such a feature.

The way the analysis handles unexpected signals such as uneventful signals such as signals with just one peak or trough is to abort the execution all together, discarding any information it might have detected. For completely flat signals the onset detection fails, aborting the execution.

The timing done in Section 6.4 shows us that the execution time on a modern system will allow us to use window sizes down to a second.

During the first execution we see a spike in execution time. The first intuition is to blame the *data loading*, but the spike also affects the execution time of the *peak detection* and *pause calculation*. This was observed when repeating the experiment as well, though the cause of the effect has not been found.

```
Step 1 data loading: 516ms
Step 1.5, smoothing: 83ms
Step 2 peak detection: 508ms
Step 3 classify: 198ms
Step 4 pause: 291ms
Step 5 stat calc: 0ms
```

106

```
----------------
Total time: 1596ms
```

```
Step 1 data loading: 125ms
Step 1.5, smoothing: 7ms
Step 2 peak detection: 32ms
Step 3 classify: 80ms
Step 4 pause: 149ms
Step 5:stat calc: 0ms
----------------
Total time: 393ms
```

7.3   Online version

The online implementation is focused on the architectural aspect of the data stream system. It has be able to split a potentially infinite signal into smaller windows without loosing information. When testing the application we device experiments to verify the integrity of data, making sure that the entire signal is processed even if it is split up. The historical data used to create adaptive windows ensures that the entirety of a transmitted signal is analysed.

The 10 second window is able to detect all the existing peaks and troughs. This scenario can be described as a happy path execution, as the pause aligns itself with the window in such a way that the entirety of the pause is within the window. The same is valid for smaller windows, as long as the pause starts and ends within the window.

As demonstrated in Figure 6.12, a trough or peaks that is not detected if the pause spans multiple windows. This is the reason for the desire to implement the *history* functionality described in Section 5.2.2.

When executing the same signal using the *history* implementation, wherein we keep the signal from the last found peak or trough, we are able to detect the events that stretch across windows.

107

As seen in Subsection 6.3.2, pauses in real world data is not always flat, so we can not rely on the *onset detection* to reliably detect windows without events. There needs to be a mechanism for detecting windows without events, which can be added to the adaptive window in order to detect pauses that span across multiple windows. This was not implemented due to time constraints.

Chapter 8

Conclusions and Future Work

In this section we present the result of the thesis (Section 8.1) and try to point out what has to be done to improve the existing solution (Section 8.3) and new and novel approaches that can be taken in the future (Section 8.4.

8.1 Summary

As presented in Section 8.2, this thesis presents a framework for analysing a data stream using the respiration analysis found in *puka*, a tool for analysing respiration signals generated by RIP sensors. We have created a happy path proof of concept, demonstrating that the stream centric approach is able to detect respiration events using synthetic signals.

The performance of the algorithm suggest that we can use much smaller windows than what has been used in our experiments. At the moment, most of the latency between an event being recorded by a sensor and the system detecting it is caused by the window size. By reducing the window size we can reduce the latency between an event in a signal to detection The implementation of the analysis found in puka makes certain assumptions about size of a given window, which will have either be parametrized or otherwise altered to accommodate smaller analysis windows.

The analysis windows are adaptive windows to allow for detecting events that theoretically can stretch across multiple window sizes. The current implementation has been shown to be able to detect peaks and troughs that would otherwise be ignored due to the proximity to either edge of a window, and not being able to detect the subsequent slope.

8.2   Contribution

As we set out to create a respiration analysis that reads a data stream, which can be processed into useful events. The basis for the thesis is the respiration analysis found in puka. We have managed to modify puka to run on a modern system, and then be able to test the functionality. The documentation for the application is rather spars, so the testing of different types of signals has helped us identify the strengths and weaknesses of the application.

After the modernization was complete, we have isolated the respiration analysis. Seeing as puka is an application with a GUI and other functionality than the respiration analysis, we extracted the parts that are relevant for our respiration focused system. We also have removed or automated the respiration analysis that require user input. The isolated version of the respiration analysis has then been compared with the original implementation to verify that the capabilities of the original is maintained.

This reduced version is then used in the stream handling application we have created. This read a data stream from a sensor (or a simulator), which serves a stream of data, which is analysed in *windows* in order to detect events as the data arrives.

8.3   Open problems

During the development and the subsequent experiments certain aspects of the design choices in both puka and our applications were identified as problematic. In this section we take a closer look

on what changes can be done to the existing solution.

8.3.1 Modernization

The *JMatLinkAdapter* described in Chapter 4 has implemented the functionality used by puka and allows the application to function as described. Both libraries *JMatLink* and *MatConsoleCtl* have functionality beyond what is used in puka, much which is not implemented in the adapter due to time constraint and prioritization.

Functionality such as handling multiple MATLAB instances, asynchronous execution of scripts and calculation and exporting of figures and plots are some of the potential functionality which can be implemented according to the documentation in both libraries.

8.3.2 Automation and Real Time implementation

While the *pukaReduced* implementation works well on ideal signal, there are inherent problems discovered in Chapter 6 and described further in **??**.

The *data serving* application has not the entirety of the protocol implemented. The control values for how many times the application serves a given file is hard coded into the application and should be parametrized so that a user can change this as designed in the protocol.

There is no implementation of the generation and transmission of the detected events to the event-detection system.

Even if the separation of concern and modularity has been mentioned as a part of the Subsection 5.3.1, there is potential for isolation reduction of dependencies across the classes. For example, the *RespirationAnalysis* class has code for both on and offline functionality, and should be split into more defined classes for easier maintenance.

The original implementation of puka was not designed for

analysis of arbitrary signals, rather for clips of pre recorded signals where a user already assumed events to be detected. As we reduce the window size in the *online* application, the chances for no activity increases, and this has to be handled by both the *respiration analysis* control code and the MATLAB scripts used by it. We did not have the time to fully analyse the cause of this unexpected behaviour as demonstrated in **??**.

While the peak detection in noisy signal seem to work well as described in Subsection 6.3.2, the pause detection does not find onset and end of apneaic-like events. To be useful with real world sensor data, we have purposed two main approaches:

1. changes to the pause detection algorithm and

2. adding pre processing of received signals.

The *adaptive threshold* described in Section 5.2.1 has not been successfully implemented, but should, alongside a smoothing algorithm be implemented in order to test whether the pause detection can be improved, or if this part of the respiration algorithm will have to be rewritten. The MATLAB implementation of the SG-filter described in Section 5.2.1 was tested without significant effect, but other smoothing algorithms or implementations can prove to be very useful.

8.4   Future Work

In this section we take a brief look at future work that can be done.

8.4.1   Handling the events generated

In order for the events detected by the *online* implementation to be useful, we need to look develop the logical sensor that derive the information based on the events. The logical sensor can combine

the data generated by the *online* application with other data from sensors.

In addition to the events generated by our respiration analysis, we can combine these events with other physiological signals. One such combination might be to combine with oxygen saturation or air flow data to be able to detect symptoms that does not involve reduction in respiration effort.

8.4.2 Other respiration analysis algorithm

The application is designed in such a way, that it should be fairly straight forward for an application developer to use different analysis algorithms. If we were to find one that is more suitable for processing data streams, we can swap out the respiration analysis taken from puka with a new version.

Aspects of such and algorithm we might find useful could be the ability to detect peaks with less data. There are articles describing real time detection [25], but we did not have the time to explore this avenue for this thesis. Another requirement for a new respiration analysis would be noise resistance in both the peak detection as well as the pause detection, as the latter especially is one of the weaknesses found in puka.

8.4.3 Structural changes

As mentioned in Subsection 4.3.2, the dependency on MATLAB can be detrimental to our ambition of creating a portable application as MATLAB requires a paid license. The increases of runtime requirements for the machine conducting the analysis is also a concern.

There are free alternatives to MATLAB such as *GNU Octave* which has been created with MATLAB compatibility in mind. There exist Java to Octave libraries such as *joPAS*[48] and *JavaOctave*[17], which allows us to recreate the MATLAB-scripts in Octave in order to make the entire application independent from non-free licensed software. GNU Octave can, albeit unofficially, be installed on An-

droid devices, increasing the range of hardware the application can be run on.

A different approach is to replicate the entirety of the respiration analysis in Java. By extracting all the functionality to on runtime environment we reduce the hardware requirements and reduce the necessity for developers to understand both programming languages.

8.4.4 Future metrics

One downside to the Precision/Recall method is the binary nature of the metric. We have no indication whether the result of the analysis is far off from the reference or whether it is similar.

The results of the respiration analysis, which is a list of indexes indicating peak and trough start and stop points, can be converted into a time series with the same length as the analysed signal, and each of the results can be encoded at their respective indexes. This allows us to utilize time series comparison techniques and string comparison algorithms.

We can make use of string similarity algorithms, such as *Cosine Similarity* which is also often utilized in information retrieval. It is a measure of the similarity between two vectors, in our case the *reference* and the *result*. The resulting similarity ranges from -1 to 1, meaning exactly opposite and identical respectively.

Apendix

Software

Software delivered:

- **Versioning**
- **v1.4** - greedy history
- **v1.3** - does not use history

The project contains the following packages:

- JMatLinkAdapter: an adapter for usage in puka to replace JMatLink
- DataFeeder: reads a signal file and transmits on socket channel.
- pukaReduced: Shell application for both online and offline analysis.
- PhysionetImporter: Removes time stamp and averages missing values in text files from PhysioBank

- **MATLAB code**
- SignalGenerator: generates synthesised signals. Has to be manually adjusted for the different types of signals desired

To set up the project, import it into Eclipse as a Maven-project. Using Maven, install the dependencies. To start the modernized puka, launch *frmMain* as a normal Java application. Make sure that

- an environment path to MATLAB is set
- the *matlabscripts* folder is copied into the *target* folder
- a *preference.txt* is present in *target* and conatins the correct path to dependencies.

The MATLAB scripts and necessary dependencies are pre-built and located in the *data* folder in the root of the project.

An example paths in the *preferences* file.

```
WFDBDirectory|C:\cygwin64\home\admin\wfdb-10.5.23\app
InstallPath|C:\Users\admin\Documents\puka\puka
WFDBDataPath|c:\temp\
ConvertECGPath|C:\cygwin64\home\admin\wfdb-10.5.23
```

Troubleshooting

The following section contains some of the general troubleshooting done during the development of the puka reduced testing platform.

Decimate

This is very relevant for all the implementation due to the fact that the indexes stored in the results uses the decimated signal as a reference. All of the evaluation software has to take this into account.

The first MATLAB script to be called by the respiration analysis is *newPT.m*. In this script, which is based on *The Identification of Peaks in Physiological Signals*[69], the programmer has assumed a fixed sampling rate, based on the fact that the raw signal (*Qraw*) is down sampled with a factor of 5 using the MATLAB toolbox decimate function.

Listing 8.1: Downsample code in *newPT.m*

```
function [P,T,th,Qd] = newPT(Qraw, factor, onsetTime,
   endTime);
   ...
   Qd = decimate(Qraw, 5); % downsample the signal
   ...
```

A more explicit mention of an assumed sample rate by the

script creator is found in the comments of *classifyPeaks.m*: "go across entire signal, looking at narrow window around each peak try 1 second windows around each peak/trough, centred on found peak 1000 Hz signal decimated by 5, so now 200 Hz; 200 data pt window either side". Here it is stated that the sample rate has been decimated by five, and that the assumed sample rate is 1000 Hz, which is not consistent with how the user is prompted to register the sampling rate for a given record.

This has to at least be considered when creating sample data, but should ideally rewritten into using the parameter in *preferences*, and only down sample if it is needed based on performance. The Java code can pass the registered sampling rate to the MATLAB engine and use this to decimate the signal with a dynamic factor in order to make the window size consistent.

This error was discovered when trying to analyse signal with an early onset time and a peak close to the start of the clip. This results in invalid indexes for the peak classification.

MATLAB script execution freeze

When running puka it sometimes freezes at after entry to *newPT.m*:

Listing 8.2: Last log entry

```
mar 27, 2016 8:03:00 PM
    matlabcontrol.LoggingMatlabProxy eval(String)
FINER: ENTRY [P,T,th,Qd] = newPT(y, .1, onsetTime,
    endTime)
```

Does not return from *engEvalString()*, but when the Java process is stopped, the execution of the MATLAB script finishes. The instance of MATLAB does not respond during this time, so no information about where in the execution it might stop.

Listing 8.3: Expected log entry

```
mar 27, 2016 8:12:48 PM
    matlabcontrol.LoggingMatlabProxy eval(String)
FINER: ENTRY [P,T,th,Qd] = newPT(y, .1, onsetTime,
```

```
      endTime)
mar 27, 2016 8:12:48 PM
    matlabcontrol.LoggingMatlabProxy eval(String)
FINER: RETURN
```

After some trail and error we found the problem in one of the
parameters given to the MATLAB engine on start up. The applica-
tion does not work with proxy: *setHidden()* set to `true`. Something
in the implementation creates problems. Without any debug infor-
mation from either MATLAB or Java, we do not know exactly what
causes this bug. But when run with *setHidden* to `false`, the call
to initiate *newPT* returns and the execution continues.

Peak detection sample rate assumption

In the MATLAB scripts containing the respiration analysis, the au-
thors have assumed a given sample rate of 1000 Hz. This is evident
in comments and code in *newPT.m* and *classifyPeaks.m*.

To avoid future confusion, we introduce a sample rate vari-
able set in each script which is in the case of the GUI application
set in *preferences* and in the case of the command line application
in the *Settings* class.

Code snippets

Listing 8.4: Respiration analysis programatical flow

```
// :745 prepare and clear data
frmLoadData.engMatLab.engEvalString("[P,T,th,Qd] =
   newPT(y, .1, onsetTime, endTime)"); //run the matlab
   script

// :536 peak analysis
frmLoadData.engMatLab.engEvalString("[peakLabels,troughLabels]
   = classifyPeaks(Qd,P,T,th);");
```

```
//get the peaks/troughs and labels back into the table
dblP = frmLoadData.engMatLab.engGetArray("P");
dblT = frmLoadData.engMatLab.engGetArray("T");
dblPlabels =
   frmLoadData.engMatLab.engGetArray("peakLabels");
dblTlabels =
   frmLoadData.engMatLab.engGetArray("troughLabels");
FillPeaksTable(0); FillTroughsTable(0); //fill both with
   all peaks/troughs

// :575 Do apply has to be described later...
DoApply(); //call cmdApply first
frmLoadData.engMatLab.engEvalString("[validPeaks,
   validTroughs] = makeValidArrays(P,T,peakLabels,
   troughLabels);");
frmLoadData.engMatLab.engEvalString("[newP] =
   markPeakPauses(Qd, validPeaks, validTroughs, th);");
frmLoadData.engMatLab.engEvalString("[newT] =
   markTroughPauses(Qd, validPeaks, validTroughs, th);");
frmLoadData.engMatLab.engEvalString("plotPauses(Qd,
   validPeaks, validTroughs, th, newP, newT);");

// :481
//control the computation of breathing statistics
ArrayList jcTempList = new ArrayList(); double[][]
   dblTemp; int intTemp = 0; int intC = 0;
double[][] dblTroughs; double[][] dblNewP; double[][]
   dblNewT;

//need the peaks and troughs array regardless of using
   pauses or not
frmLoadData.engMatLab.engEvalString("[peaks,troughs] =
   generatePT(P,T,peakLabels, troughLabels);");
//Ttotal is calculated off of the troughs array – pauses
   don't matter
frmLoadData.engMatLab.engEvalString("[avgTtot,stdTtot] =
   calculateTtotal(troughs);");


//call the matlab scripts to do the calculations either
   with or without pauses
if (rdoUsePauses1.isSelected() == true) { //calculations
   include pauses
```

```java
        frmLoadData.engMatLab.engEvalString("[avgPI,stdPI,avgPE,stdPE]
            = calculatePauses(newP,newT);");
        frmLoadData.engMatLab.engEvalString("[avgTI,stdTI,avgTE,stdTE]
            = calculateInsExp(newP,newT);");
    } else { //pauses ignored; all assumed to be zero
        frmLoadData.engMatLab.engEvalString("avgPI = 0;");
            //set all of these variables to zero
        frmLoadData.engMatLab.engEvalString("stdPI = 0;"); //so
            that CalculateResp() can retrieve
        frmLoadData.engMatLab.engEvalString("avgPE = 0;");
            //the values to show in the table
        frmLoadData.engMatLab.engEvalString("stdPE = 0;");
        frmLoadData.engMatLab.engEvalString("[avgTI,stdTI,avgTE,stdTE]
            = calculateInsExpNoPauses(peaks,troughs);");
    }

    try {
        CalculateResp();
    } catch (MatlabInvocationException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } //shows results in the table and sets in rmData
```

Listing 8.5: Interactions with MATLAB scripts from the Java application

```java
frmLoadData.engMatLab.engEvalString("[P,T,th,Qd] =
    newPT(y, threshold, onsetTime,
 endTime)");
...
frmLoadData.engMatLab.engEvalString("[peakLabels,troughLabels]
    = classifyPeaks(Qd,P,T,th);");
...
frmLoadData.engMatLab.engEvalString("[validPeaks,
    validTroughs] = makeValidArrays(P,T,peakLabels,
    troughLabels);");
frmLoadData.engMatLab.engEvalString("[newP] =
    markPeakPauses(Qd, validPeaks, validTroughs, th);");
frmLoadData.engMatLab.engEvalString("[newT] =
    markTroughPauses(Qd, validPeaks, validTroughs, th);");
frmLoadData.engMatLab.engEvalString("plotPauses(Qd,
    validPeaks, validTroughs,
 th, newP, newT);"); // only used for visual verification
```

```
...
CalculateResp(); // statistical computation
```

Listing 8.6: The implementation of the peak detection algorithm in puka

```
for ind = 1:n
   if d == 1
      if Qd(a) >= Qd(ind) + th
         d = 3;
      elseif Qd(ind) >= Qd(b) + th
         d = 2;
      end;
      if Qd(a) < Qd(ind)
         a = ind;
      elseif Qd(ind) < Qd(b)
         b = ind;
      end;
      S = ind;
   elseif d == 2 % signal rising, trough-to-peak
      if Qd(a) < Qd(ind) % still rising
         S = ind;
         a = ind;
      elseif Qd(a) == Qd(ind)
         S = [S,ind];
      elseif Qd(a) >= Qd(ind) + th
         P = [P,S]; S = ind;
         b = ind; d = 3;
      end;
   elseif d == 3 % signal falling, peak-to-trough
      if Qd(ind) <= Qd(b)
         S = ind;
         b = ind;
      elseif Qd(b) == Qd(ind)
        S = [S,ind];
      elseif Qd(ind) >= Qd(b) + th
         T = [T,S]; S = ind;
         a = ind; d = 2;
      end;
   end;
end;
```

Listing 8.7: Some voodoo going on

```matlab
%This is done checking each direction from the current
   peak location, and based
%%%%%%%%%%%%%%%%%%%%% find the pause to the LEFT of this
   trough %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sth = abs((Qd(prevPeak) - Qd(thisTrough)) * .30); % small
   threshold = 15% of slope height
mth = abs((Qd(prevPeak) - Qd(thisTrough)) * .08); % medium
   threshold
ssth = abs(Qd(prevPeak) - Qd(thisTrough)) * .0005; %
   super-small threshold
m = -1; littles = 0; negatives = 0; stop = 0; % set markers

% loop until end of window is sth HIGHER than the trough
while (Qd(thisTrough + m) - Qd(thisTrough)) < sth
     if (thisTrough + m) == 2
          m = m + 1
          break;
     else
      m = m - 1; % back up one more step
     end;
end;


if thisTrough == 11499
     s = ['m = ', int2str(m), ' before looping back']
end;


while stop == 0 % now loop back towards the trough until a
   stopping condition is met
     if abs(Qd(thisTrough + m) - Qd(thisTrough + m - 1))
        < ssth % slope of last point less than ssth
          littles = littles + 1; % zero or not too steep
             slope
     else
          littles = 0; % other - medium or steep slope
     end;
     if (Qd(thisTrough + m - 1) - Qd(thisTrough + m) < 0)
          negatives = negatives + 1; % this point lower
             than last - should be higher
     else
          negatives = 0;
     end;
```

```matlab
        if thisTrough == 11499
                s = ['m = ', int2str(m), ' looping back']
        end;

        if m == -1
                m = 0;
                stop = 1; % back to trough point - stop
                s = ['m == -1 - LT', int2str(i), '
                    thisTrough=', int2str(thisTrough), '
                    mark=', int2str(thisTrough + m)]
        elseif Qd(thisTrough + m) <= Qd(thisTrough)
                stop = 1; % height is lower or equal to the
                    trough - stop
                s = ['back to start level - LT', int2str(i), '
                    thisTrough=', int2str(thisTrough), '
                    mark=', int2str(thisTrough + m)]
        elseif littles > 30 & ((Qd(thisTrough + m) -
            Qd(thisTrough)) < mth)
                stop = 1; % slope not changing much anymore
                m = m - 15; % move back to middle of the 3
                    littles
                s = ['littles > 30 - LT', int2str(i), '
                    thisTrough=', int2str(thisTrough), '
                    mark=', int2str(thisTrough + m)]
        elseif negatives > 10 & ((Qd(thisTrough + m) -
            Qd(thisTrough)) < mth)
                stop = 1; % went the wrong direction for 10
                    points in a row
                m = m - 10;
                s = ['negatives > 10 - LT', int2str(i), '
                    thisTrough=', int2str(thisTrough), '
                    mark=', int2str(thisTrough + m)]
        else
                m = m + 1; % move one point more towards the
                    trough
        end;
end;
newT = [newT, (thisTrough + m)]; % add in new point
```

Implementation missing on Nice to Have Features

- proper logging

- better separation of concern

- shut-down code. Threads are not properly shut-down user prompted exit

List of Figures

List of Tables

Bibliography

[1] Apache. Apache maven. `https://maven.apache.org/`, 2016.

[2] Eric Weisstein at Wolfram Research. Wolfram mathworld. `http://mathworld.wolfram.com/Sine.html`, 2016.

[3] Jay S. Balachandran, Jessie P. Bakker, Shilpa Rahangdale, Susie Yim-Yeh, Joseph E. Mietus, Ary L. Goldberger, and Atul Malhotra. Effect of mild, asymptomatic obstructive sleep apnea on daytime heart rate variability and impedance cardiography measurements. *The American Journal of Cardiology*, 109(1):140 – 145, 2012.

[4] KarenA. Brown, AhmedA. Aoude, HenriettaL. Galiana, and RobertE. Kearney. Automated respiratory inductive plethysmography to evaluate breathing in infants at risk for postoperative apnea. *Canadian Journal of Anesthesia*, 55(11):739–747, 2008.

[5] Pierre-Yves Carry, Pierre Baconnier, Andre Eberhard, Pierre Cotte, and Gila Benchetrit. Evaluation of respiratory inductive plethysmography : Accuracy for analysis of respiratory waveforms. *Chest*, 111(4):910–915, 1997.

[6] Intrinsyc Technologies Corp. J-integra com. `http://j-integra.intrinsyc.com/support/com/doc/#other_examples/Matlab.htm`, 2014.

[7] Jerome A Dempsey, Sigrid C Veasey, Barbara J Morgan, and Christopher P O'Donnell. Pathophysiology of sleep apnea. *Physiological reviews*, 90(1):47–112, 2010.

[8] Danny J Eckert, Amy S Jordan, Pankaj Merchia, and Atul Malhotra. Central sleep apnea: pathophysiology and treatment. *Chest Journal*, 131(2):595–607, 2007.

[9] Danny J. Eckert and Atul Malhotra. Pathophysiology of adult obstructive sleep apnea. *Proceedings of the American Thoracic Society*, 5(2):144 – 153, 2008.

[10] Inc Embarcadero Technologies. Borland compiler. `http://edn.embarcadero.com/article/20633`, 2013 (accessed February, 2016).

[11] Nieto F, Young TB, Lind BK, and et al. Association of sleep-disordered breathing, sleep apnea, and hypertension in a large community-based study. *JAMA*, 283(14):1829–1836, 2000.

[12] The Apache Software Foundation. The apache ant project. `http://ant.apache.org/`, 2015.

[13] M Fusetti, M VALENTI AB FIORETTI, F MASEDU, M LAURIELLO, and M PAGLIARELLA. Cardiovascular and metabolic comorbidities in patients with obstructive sleep apnoea syndrome. *Acta Otorhinolaryngologica Italica*, 32(5):320, 2012.

[14] Ary L Goldberger, Luis AN Amaral, Leon Glass, Jeffrey M Hausdorff, Plamen Ch Ivanov, Roger G Mark, Joseph E Mietus, George B Moody, Chung-Kang Peng, and H Eugene Stanley. Physiobank, physiotoolkit, and physionet components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220, 2000.

[15] C Guilleminault, A Tilkian, and W C Dement. The sleep apnea syndromes. *Annual Review of Medicine*, 27(1):465–484, 1976. PMID: 180875.

[16] Christian Guilleminault. *Clinical Neurophysiology of Sleep Disorders*. Elsevier, 2005.

[17] Kim Hansen. Javaoctave. `https://sourceforge.net/projects/javaoctave/`, 2013.

[18] Jan Hedner, Ludger Grote, and Ding Zou. Pharmacological treatment of sleep apnea: Current situation and future strategies. *Sleep Medicine Reviews*, 12(1):33 – 47, 2008.

[19] Harald Hrubos-Strom, Aanna Randby, Silje K. Namtvedt, Håvard A. Kristiansen, Gunnar Einvik, Juratesaltyte Benth, Virend K. Somers, Inger H. Nordhus, Michael B. Russel, Toril Dammen, Torbjørn Omland, and Kari J. Kværner. A norwegian population-based study on the risk and prevalence of obstructive sleep apnea the akershus sleep apnea project (asap). *Journal of Sleep Research*, 20(1pt2):162–170, 2011.

[20] SyedAnas Imtiaz and Esther Rodriguez-Villegas. A low computational cost algorithm for rem sleep detection using single channel eeg. *Annals of Biomedical Engineering*, 42(11):2344–2359, 2014.

[21] EsperTech Inc. Esper: Event processing. `http://www.espertech.com/products/esper.php`, 2015 (accessed April 3, 2015).

[22] EsperTech Inc. Esper docs. `http://www.espertech.com/esper/documentation.php`, 2015 (accessed May 31, 2015).

[23] Great Lakes NeuroTechnologies Inc. Bioradio monitor. `https://glneurotech.com/bioradio/wireless-physiological-monitor/`, 2016.

[24] National Health Institute. What is sleep apnea? `http://www.nhlbi.nih.gov/health/health-topics/topics/sleepapnea/`, feb 2015 (accessed February 16, 2015).

[25] Dae-Geun Jang, Sangjun Park, Minsoo Hahn, and Seung-Hun Park. A real-time pulse peak detection algorithm for the photoplethysmogram. *Int. J. Electron. Electr. Engin*, 2:45–49, 2014.

[26] Kourosh Kalantar-zadeh. *Sensors: An Introductory Course*. SpringerLink : Bücher. Springer US, 2013.

[27] Anthony Kales, Alex B. Caldwell, Roger J. Cadieux, Antonio Vela-Bueno, Lynnette G. Ruch, and Susan D. Mayes. Severe obstructive sleep apnea—ii: Associated psychopathology and psychosocial consequences. *Journal of Chronic Diseases*, 38(5):427 – 434, 1985.

[28] Joshua Kaplan. Matlabcontrol. `https://code.google.com/archive/p/matlabcontrol/`, 2013.

[29] Joshua Kaplan. Matlabcontrol javadocs. `http://matlabcontrol.googlecode.com/svn/javadocs/doc/index.html`, 2013.

[30] Vishesh Kapur, David K. Blough, Robert E. Sandblom, Richard Hert, James B. de Maine, Sean D. Sullivan, and Bruce M. Psaty. The medical cost of undiagnosed sleep apnea. *Sleep*, 22:749 – 755, 1999.

[31] Maksim Khadkevich. Jamal. `https://github.com/hutm/JAMAL`, 2013.

[32] Ron Kohavi and Foster Provost. Glossary of terms. *Machine Learning*, 30(2-3):271–274, 1998.

[33] C. Lamberti and J. de Bie. Validation of an ecg-derived respiration monitoring method. In *Computers in Cardiology, 2003*, pages 613–616, Sept 2003.

[34] Wilburta Lindh, Marilyn Pooler, Carol Tamparo, Barbara Dahl, and Julie Morris. *Delmar's comprehensive medical assisting: administrative and clinical competencies*. Cengage Learning, 2013.

[35] Stefan Müller. Efficient integration of real-time hardware and web based services into matlab. `http://jmatlink.sourceforge.net/docs/ESS99.pdf`, 1999.

[36] Stefan Müller. Jmatlink javadocs. `http://jmatlink.sourceforge.net/docs/javadoc/`, 2005.

[37] Stefan Müller. Jmatlink sourcecode. `https://sourceforge.net/projects/jmatlink/files/`, 2005.

[38] Mathworks. System requirements and platform availability. `http://se.mathworks.com/support/sysreq/previous_releases.html`, 2016.

[39] G Mbata and J Chukwuka. Obstructive sleep apnea hypopnea syndrome. *Annals of Medical and Health Sciences Research*, 2:74–77, 2012.

[40] Michael J. McGrath and Cliodhna Ni Scanaill, editors. *Sensor Technologies: Healthcare, Wellness and Environmental Applications*. Apress, Berkeley, CA, 2013.

[41] Microsoft. How to use the regsvr32 tool. `https://support.microsoft.com/en-us/kb/249873`, 2015 (accessed June 23, 2016).

[42] Steve P. Miller. Steve miller applications. `http://www.dependencywalker.com/`, 2006.

[43] George B. Moody. Ecg-derived respiration. `http://www.physionet.org/physiotools/edr/`, jan 2012 (accessed March 12, 2015).

[44] J. Moura. What is signal processing? [president's message]. *IEEE Signal Processing Magazine*, 26(6):6–6, Nov 2009.

[45] National Institute of Health. Physiobank. `http://www.physionet.org/physiobank/`, feb 2015 (accessed April 27, 2015).

[46] Institute of Science, Technology Austria (IST Austria), and Graz University of Technology. The biosig project. `http://biosig.sourceforge.net/`, 6 2016.

[47] American Association of Sleep Technologists. Standard polysomnography. `http://www.aastweb.org/Resources/Guidelines/StandardPSG.pdf`, July 2012 (accessed May 2, 2015).

[48] Group PAS of the University of Deusto. jopas. `http://jopas.sourceforge.net/`, 2005.

[49] Oracle. Java documentation executor service. `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html`, 2016 (accessed June 20, 2016).

[50] Oracle. Java documentation timer. `https://docs.oracle.com/javase/8/docs/api/java/util/Timer.html`, 2016 (accessed June 20, 2016).

[51] Rodrigo P Pedrosa, Luciano F Drager, Carolina C Gonzaga, Marcio G Sousa, Lílian KG de Paula, Aline CS Amaro, Celso Amodeo, Luiz A Bortolotto, Eduardo M Krieger, T Douglas Bradley, et al. Obstructive sleep apnea the most common secondary cause of hypertension associated with resistant hypertension. *Hypertension*, 58(5):811–817, 2011.

[52] Alastair Pharo. Matlabcontrol maven. `https://github.com/socsol/matlabcontrol-maven`, 2014.

[53] Shashi Phoha, Thomas LaPorta, and Christopher Griffin. *Sensor Network Operations*. Wiley-IEEE Press, 2006.

[54] Physionet. Puka source code. `https://physionet.org/physiotools/puka/sourceCode/`, 2004.

[55] Physionet. Puka user manual. `https://physionet.org/physiotools/puka/pukaManual.pdf`, 2004.

[56] Naresh M Punjabi. The epidemiology of adult obstructive sleep apnea. *Proceedings of the American Thoracic Society*, 5(2):136–143, 2008.

[57] Julian P Saboisky, Nancy L Chamberlin, and Atul Malhotra. Potential therapeutic targets in obstructive sleep apnoea. *Expert opinion on therapeutic targets*, 13(7):795–809, 2009.

[58] Abraham Savitzky and Marcel JE Golay. Smoothing and differentiation of data by simplified least squares procedures. *Analytical chemistry*, 36(8):1627–1639, 1964.

[59] R. W. Schafer. What is a savitzky-golay filter? [lecture notes]. *IEEE Signal Processing Magazine*, 28(4):111–117, July 2011.

[60] Harvard Medical School. Diagnosing osa. `http://healthysleep.med.harvard.edu/sleep-apnea/diagnosing-osa/understanding-results`, 2011 (accessed May 13m 2015).

[61] Robert C Seacord, Daniel Plakosh, and Grace A Lewis. *Modernizing legacy systems: software technologies, engineering processes, and business practices*. Addison-Wesley Professional, 2003.

[62] JW Shepard. Hypertension, cardiac arrhythmias, myocardial infarction, and stroke in relation to obstructive sleep apnea. *Clinics in chest medicine*, 13(3):437—458, September 1992.

[63] Aaron E Sher. Upper airway surgery for obstructive sleep apnea. *Sleep medicine reviews*, 6(3):195–212, 2002.

[64] Kevin Shrake, Pam Minkley, Sue Blonshine, Robert A Brown, Michael J Decker, Gregg L Ruppel, and Jack Wanger. Aarc clinical practice guideline. polysomnography. `http://www.rcjournal.com/cpgs/polycpg.html`, feb 1995 (accessed May 16, 2015).

[65] Sleepdex. Sleep stages. `http://www.sleepdex.org/stages.htm`, 2015 (accessed May 31, 2015).

[66] National Telecommunications and USDC Information Administration. Federal standard 1037c: Data stream definition. `http://www.its.bldrdoc.gov/fs-1037/dir-010/_1451.htm`, 2016.

[67] Shinji Teramoto, Hiroshi Yamamoto, Yasuhiro Yamaguchi, Ryoichi Namba, and Yasuyoshi Ouchi. Obstructive sleep apnea causes systemic inflammation and metabolic syndrome. *Chest*, 127(3):1074–1075, 2005.

[68] Malcom S. Thaler. *The Only EKG Book You'll Ever Need*. The Only EKG Book You'll Ever Need. J.B. Lippincott, 8 edition, 1995.

[69] Bryan S Todd and David C Andrews. The identification of peaks in physiological signals. *Computers and biomedical research*, 32(4):322–335, 1999.

[70] Ned Twigg. Matconsolectl. `https://github.com/diffplug/matconsolectl`, 2015.

[71] John V. Weil, Robert E. McCullough, J. S. Kline, and Ingvar E. Sodal. Diminished ventilatory response to hypoxia and hypercapnia after morphine in normal man. *New England Journal of Medicine*, 292(21):1103–1106, 1975. PMID: 1128555.

[72] David P. White. Pathogenesis of obstructive and central sleep apnea. *American Journal of Respiratory and Critical Care Medicine*, 172(11):1363–1370, 2005.

[73] Kamin Whitehouse. The math forum. `http://www.cs.virginia.edu/~whitehouse/matlab/JavaMatlab.html`, 2001.

[74] Kamin Whitehouse. Matlab controll. `http://www.cs.virginia.edu/~whitehouse/matlab/JavaMatlab.html`, 2001.

[75] Terry Young, Mari Palta, Jerome Dempsey, James Skatrud, Steven Weber, and Safwan Badr. The occurrence of sleep-disordered breathing among middle-aged adults. *New England Journal of Medicine*, 328(17), 1993.