## Contents

1 Introduction and background  

1 Introduction  

1.1 Motivation  

1.2 Problems caused by sleep disorders  

1.3 Non intrusive sensors  

1.4 TRIO  

1.5 Problem statement  

2 Background  

2.1 Sleep Apnea Syndrome  

2.1.1 Obstructive Sleep Apnea  

2.1.2 Central Sleep Apnea  

2.1.3 Mixed/Complex Sleep Apnea  

2.2 Diagnosis  

2.2.1 AHI  

2.2.2 PSG  

2.2.3 Treatment of OSA  

2.2.4 Diagnosis of Sleep Apnea  

2.3 Treatment of Sleep Apnea  

2.4 Sleep Apnea Syndrome and Related Disorders  

2.5 Other Sleep Related Disorders  

2.6 Future Scope  

2.7 Conclusion  

3 Conclusion  

4 References  

5 Appendices  

6 Acknowledgments
2.3 Sensors .................................................. 18
  2.3.1 Sensor characteristics ................................. 20
  2.3.2 Sensor networks ....................................... 21
  2.3.3 Data Stream Management Systems .................. 22
  2.3.4 Complex Event Processing ........................... 23

3 Related Work .............................................. 25
  3.1 Traditional sensor approaches ......................... 26
    3.1.1 Peak detection ....................................... 26
  3.2 Novel non intrusive sensors approaches ............... 27
    3.2.1 Radar ................................................ 27
    3.2.2 PPG ................................................ 27
    3.2.3 Nintendo QOL ........................................ 28
    3.2.4 Portable ECG ........................................ 29

II Respiration analysis application ......................... 30

4 The application puka ....................................... 31
  4.1 Key functionalities ...................................... 32
    4.1.1 History ............................................... 33
    4.1.2 Program structure .................................... 34
    4.1.3 Runtime requirements ............................... 35
    4.1.4 Preferences .......................................... 37
    4.1.5 Data format .......................................... 37
4.2 Respiration analysis algorithm ............................ 37

5 Modernizing the application ................................. 41

5.1 Installing dependencies ................................. 41

5.2 Identifying decrepit parts of puka .................. 42

5.2.1 Recompiling JMatLink ............................... 43

5.3 Modernization of puka ................................. 46

5.3.1 Approaches ........................................ 47

5.3.2 Evaluation of modernization approaches ....... 48

5.4 Adapter for JMatLink .................................. 49

5.4.1 JMatLink analysis .................................. 50

5.4.2 The matlabcontrol library ......................... 50

5.5 Creating the adapter .................................. 51

5.5.1 Replacing references to JMatLink .......... 52

5.5.2 Interface .......................................... 53

5.5.3 Unit testing ....................................... 54

5.6 Testing the modernized implementation ........... 56

5.6.1 Test data and verification .......................... 56

5.6.2 Simulated data .................................... 56

5.6.3 Real world data .................................... 61

5.7 puka in action ....................................... 62

6 Re-purposing the application for real time analysis 69

6.1 Describe goal and point of realtimification ....... 69
6.1.1 Strategies for real-time solution / Initial approach 70

6.2 Implementation .............................. 71

6.2.1 puka reduced ............................... 73

6.2.2 Respiration Analyser ....................... 73

6.2.3 Historical data ............................... 74

6.2.4 Data serving ................................. 74

6.3 Modifying puka for online usage ............ 76

6.3.1 Changes to puka ............................. 76

6.3.2 Window sizes ............................... 77

6.4 TODOS and NOTES ............................ 78

III Results 80

7 Evaluation 81

7.1 Metrics ........................................ 81

7.2 Experiments ..................................... 81

7.3 Results and Adjustments to puka ............ 82

7.3.1 Results ....................................... 82

7.3.2 Decimate ...................................... 82

7.3.3 Mark pause peak and trough location ...... 83

7.3.4 Peak and trough classification ............. 84

7.4 Result/Analysis online ......................... 84

8 Discussion 85
9 Future Work

9.0.1 Signal processing ................. 88
9.0.2 beyond puka .................. 88
9.1 Other approaches to puka .......... 88
9.1.1 Recompiling puka .............. 88
TODO

Cato Danielsen

July 22, 2016
Preface
Preface

Abstract

Acknowledgements
Part I

Introduction and background
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 [22]. Other literature has for a long time pointed out the risk of mental health issues related to sleep apnea [24], such as depression.

According to the literature the estimated prevalence of sleep apnea is 2%-4% of the middle aged adult population in USA[71]. 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 [18]. 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 an 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. We will look into some examples of these solutions in chapter 3. This can be done by either 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.

1.2 Problems caused by sleep disorders

TODO: Why are sleep disorders important to detect...

1.3 Non intrusive sensors

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 an 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 wearig 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 also detect whether a patient is asleep or not, we need systems to analyse these signal to derive useful information.

For this thesis we will attempt to adapt the software and algorithms found in existing work using sensors for deriving respiratory information from physical sensors. The data generated from the sensors will be used to generate

In order to evaluate the real time capabilities of such and
adaptation we create a test framework that can give us insights to the limitations and potential challenges.

The first step is to identify and analyse existing software to find a solution that have the functionality we need.
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[36]. 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
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.

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\[53, 10\]. 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\[52\].

Hypertension, also known as high blood pressure, is an often reported co-morbidity of OSA\[13\]. 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\[56, 47\].

OSA has also been linked as a risk factor for cardiovascular diseases, stroke, abnormal glucose metabolism, insulin resistance, and diabetes mellitus \[52\], \[62\]. Cerebrovascular diseases and OSA have been pointed out to have a bi-directional relationship\[18\], and as a result of the hypertension and reduced cerebral blood flow the risk for cerebrovascular 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"\[15\].
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 brain's control of the respiratory effort, described as a loss of ventilatory control[68]. 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 CO² levels. Hypercapnic and nonhypercapnic. Hypercapnic is defined as elevated CO² 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")[9].

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[67].

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[9]. 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.
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"[17]. 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 followed 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:

<table>
<thead>
<tr>
<th>Events</th>
<th>Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-4</td>
<td>Normal</td>
</tr>
<tr>
<td>5-14</td>
<td>Mild</td>
</tr>
<tr>
<td>15-29</td>
<td>Moderate</td>
</tr>
<tr>
<td>30 or more</td>
<td>Severe</td>
</tr>
</tbody>
</table>

Table 2.1: AHI severity scale

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

\[
AHI = \frac{(Hypopneas + apneas) \times 60}{TotalSleepTime(minutes)}
\]

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"[20] 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[44]:

---

13
With electrodes:

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEG</td>
<td>Electroencephalogram monitors the electrical activity in the brain.</td>
</tr>
<tr>
<td>EOG</td>
<td>Electrooculogram measures eye movement.</td>
</tr>
<tr>
<td>EMG</td>
<td>Chin Electromyogram monitors level of muscle tone around the chin area.</td>
</tr>
<tr>
<td>ECG</td>
<td>Electrocardiogram monitors the heart rhythm recorded from the movement of electrodes</td>
</tr>
<tr>
<td>Respiration</td>
<td></td>
</tr>
</tbody>
</table>

Other sensors:

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Audio</td>
<td>Upper Airway Sound Recording</td>
</tr>
<tr>
<td>Thermistor or Inductive Respiratory Plethysmograph (RIP)</td>
<td>Respiratory effort and flow</td>
</tr>
<tr>
<td>Limb EMG</td>
<td>Limb Movement and Body Position</td>
</tr>
</tbody>
</table>

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 you 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 a 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 [21].

![Figure 2.2: Sleep stages][60]

The EOG is useful for identifying and studying the REM sleep stages. It uses electrodes positioned near the corner of each eye 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 mentalis, submentalis muscle, and/or messater region [58]. 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
electrodes placed on the patients body. Today the standard ECG consists of 12 leads in order to monitor all three dimensions of the heart \[63\]. Typically there are six limb leads placed on arms and legs and six precordial leads placed across the chest. Each lead has a specific angle from which it observes the heart in order to

\[TODO: \text{reread book for better explanation.}\]

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.

QRS is a pattern seen in an ECG that indicates the pulses in a heart beat and their duration. Each part of a QRS complex shows the activity

\[TODO: \text{QRS -very brief!}\]

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 uses elastic bands 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\[4, 29\].

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. TODO: finish

![ECG signal](image)

(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[40]. TODO: finish

Oxygen saturation is a useful parameter for detecting OSA, as the SaO² (blood oxygen saturation) drops after the onset of an apneic/hypopneic event. According to Division of Sleep Medicine at Harvard Medical School[54], 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[18].

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[57]. 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"[19]

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"[23]. 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 receiving 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 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.

TODO: describe transducer?

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 contact 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"[41].
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 deployed in.

The accuracy of a sensor's 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[23]. 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.

Figure 2.4: Precision and accuracy[23]

McGrath and Scanaill[37] 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 an 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.

20
"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"[49]. 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 database management system (DBMS) is concerned with persistent storage of data, and is often used in conjunction 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. 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[61].
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.
Chapter 3

Related Work

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

There are many different sensors on the market with different capabilities. Some have proprietary software which delivers physiological data such as heart rate, respiration per minute, temperature etc. These solutions can not give us insight into sleep disorders as the data delivered is too generalized. The same techniques that are used by such proprietary systems can be used to detect other respiratory events. The solution we create should be platform agnostic as long as the sensors provide signals with the same characteristics as the signals used by the base system we use.

The PSG was developed in the sixties, and has since been the gold standard for all sleep studies, but as mentioned earlier, the threshold for conducting a PSG recording is quite high based on the price, amount of work and time consumption involved with it. As a result of this a many occurrences of OSA go undiagnosed.
3.1 Traditional sensor approaches

In Section 2.3 we described two different methods of classifying sensors. Either as contact and non contact or intrusive and non intrusive. Even though there are inconveniences to the usage of the traditional sleep studies, there are lessons to be learned from the signal processing.

3.1.1 Peak detection

Peak detection is a common problem in different fields and there have been many approaches to this problem within the field of computer science. As Dumpala, Reddy and Sarna points out in 1982 "Detection of peaks is one of the most commonly encountered problems in the processing of digitized biological signals"[8].

There are many algorithms found in the literature for peak detection, such as Korten and Haddad implementation from 1989[28], or R.J. Marshall article from 1985[34]. Korten and Haddad focus on plethysmography signals while the article by Marshall describe a general purpose algorithm for different types of physiological signals. These algorithms generate the basis for more modern implementations such as Todd and Andrews’ article from 1998[64].

Elgandi, Norton, Brearly, Abbott and Schuurmans describe three different approaches to peak detection[11]. The algorithms (which are stated in the article as "commonly used in real-time analysis of PPG signals") described in this article are used on tested on PPG signals, but can be used for other signal types. The three algorithms described are

1. Local Minima and Maxima,
2. First Derivative with Adaptive Thresholds, and
3. Slope Sum function with An Adaptive Threshold.

The algorithms are of interest if we are to create a respiration analysis software from the ground up or modify existing solutions with more novel signal analysis.
3.2 Novel non intrusive sensors approaches

With the decreased cost of sensors and the increase in computational power, automated and non intrusive sensor systems has been created to detect and help diagnose previously difficult to detect symptoms and illnesses.

We will take a brief look at some sensor systems that are available.

3.2.1 Radar

Venkatesh, Anderson, Rivera and Buehrer[66] has developed a system using a Ultra Wideband (UWB) pulse radar to detect respiration and heart rate. The work is based on similar work using Doppler radars. The UWB radar has a much stronger material penetration capability. The article [66] lists possible usecases such as detecting people trapped under debris or snow, through-the-wall health monitoring in hostage rescue scenarios, vital-signs monitoring for athletes performance, or similar to the TRIO system, as a non invasive method of assessing vital signs of a patient. The team presents a mathematical framework for analysis for analysing the potential of measuring the chest cavity in order to derive respiration and heart rate. The article shows promising results for respiration monitoring even through walls.

3.2.2 PPG

Photoplethysmogram (PPG) is a frequently used non intrusive monitoring technique using optically obtained volumetric measurement of an organ. In sleep studies this measurement is often obtained by using a pulse oximeter which indirectly monitors the oxygen saturation by detecting changes in the light absorption in the skin.
PPG can then be used to monitor heart rate and cardiac cycle, but also to monitor the respiration.

TODO: eksempel

### 3.2.3 Nintendo QOL

In late October, 2014, Nintendo president Satoru Iwata announced that the company would engage in improving people's quality of life (QOL) through entertainment. The first stated theme of this project is health, and in the presentation the concrete example used is "visualizing sleep and fatigue". Implementation details are sparse, but the Figure 3.1 shows a similar conceptual idea as described in Subsection 3.2.1.

![Visualizing Sleep and Fatigue By Five “Non” Sensing Automatic Measurements](image)

Figure 3.1: Non contact sensor from Nintendo

"Inside the QOL Sensor is a non-contact radio frequency sensor, which measures such things as the movements of your body, breathing and heartbeat, all without physically touching your body." is the promise of the presentation and shows the wide interest from more than niche companies in non invasive sensor technology.
3.2.4 Portable ECG

TODO: eksempel
Part II

Respiration analysis application
The application we use for the respiration analysis is called \textit{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\cite{50}.

PhysioNet Resource is a public service 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 analysis
3. **PhysioNetWorks**: a virtual laboratory for collaboration
4.1 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, based on a threshold, 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.

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.
4.1.1 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\[^{51}\].

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.

### 4.1.2 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, initiated 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 \texttt{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 Subsection 4.1.5. 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 \texttt{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.

### 4.1.3 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 [51], 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.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>Minimum v1.4 according to documentation</td>
</tr>
<tr>
<td>MATLAB</td>
<td>R13 was released in 2002</td>
</tr>
<tr>
<td>Cygwin</td>
<td>No version specified</td>
</tr>
<tr>
<td>WFDB</td>
<td>Installed within Cygwin</td>
</tr>
<tr>
<td>JMatLink</td>
<td>Latest version (V1.3.0) released in 2005</td>
</tr>
<tr>
<td>MySQL</td>
<td>Ignored since puka supports file storage</td>
</tr>
</tbody>
</table>

Table 4.1: 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 Subsection 5.2.1]. *MATLAB R2012b* 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 4.2], 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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdann</td>
<td>move annotation created by ecgpuwave to external file</td>
</tr>
<tr>
<td>ann2rr</td>
<td>create an RR interval series</td>
</tr>
<tr>
<td>ihr</td>
<td>create a instantaneous heart rate series</td>
</tr>
<tr>
<td>ECGPUWave</td>
<td>marks ECG waveforms</td>
</tr>
<tr>
<td>convertECG</td>
<td>converts ecg.txt into wfdb .dat format</td>
</tr>
</tbody>
</table>

Table 4.2: 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 [31] 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[33].

4.1.4 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 4.2]. These preferences has to be set for each system.

4.1.5 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.

4.2 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.
### 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 4.2: Preferences stored in preferences.txt
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"[51].

The algorithms used in the respiration analysis identifies critical parts of a recording. The critical parts are peak, trough, post-inspiratory (PI) and post-expiratory (PE) pause. These four parts are useful metrics for detecting sleep apnea, and will have to be converted into events for the TRIO system.

The program uses a four step algorithm to analyse the respiration signal:

1. Identify peaks and troughs,
2. check validity of peaks and troughs,
3. mark pauses at each peak and trough and finally
4. centre peaks and troughs.

When this has been done puka does the following statistical computations: 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 of the clip as well as reporting the longest and shortest. The statistical computations gives us an indication of what kind of events puka is able to detect.

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 1998[64]. When we use the term peak in this section, we refer to both peak and troughs as both are in principle the same. Both indicate a local extreme and a point of inverse growth or the halt of growth. The function returns two arrays, one containing the index of all peaks and the other containing troughs.

After the first pass the peaks are classified. puka operates with three classifications of peaks:
1. valid,
2. invalid and
3. questionable.

The arrays returned from this function is 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 window size is hard coded into the script and will be discussed as we make changes to the algorithm in chapter 6.

To initialize the analysis the scripts makes sure the first and last peak is far enough from the beginning and end respectively to have room for the classification window. The application classifies the peaks based on the negative and positive difference on each side of a peak. If the start or end of window higher than peak (inverted for trough) the peak is classified as invalid.

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. This is done checking each direction from the current peak location, and based on the threshold the algorithm looks for neighbour sections with a slope higher than the threshold. The pause is indicated with two indexes for each peak/trough, indicating start and stop of each pause.

The last step uses the pause start and stop and centres the peak or trough in the middle of each pause. All the new peaks and troughs are saved in arrays with the final result and plotted and also used for the statistical computation done after the analysis of a given clip.
Chapter 5

Modernizing the application

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 usable 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 (??) 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.

5.1 Installing dependencies

TODO? Describe successful installations?
5.2 Identifying decrepit parts of puka

To execute puka it has to have some way of communicating with MATLAB from the executing Java code. It is, as described in chapter 4, written to use JMatLink 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 5.1.

Since we are running on a Windows 7 operating system we have to find how to load the dll. According to the official documentation[38] there is an official tool, regsvr32.exe, in the Windows OS for registering libraries.

Listing 5.1: Trying to launch puka after adding JMatLink

```
ERROR: Could not load the JMatLink library
This error occurs, 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 5.1], suggesting the dll is incompatible with our OS.

By running the jmatlink.dll through "Dependency Walker"[39], we are able to map the dependencies of the library. It seems 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.
5.2.1 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 compat-
ible with modern system and modern versions of MATLAB.

The \textit{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 \cite{Ant}. The \textit{build.xml} file used to compile the project contains hard coded values that has to be changed in order to compile the library locally. These includes the path to \textit{Java Development Kit} (JDK), \textit{Borland C++ Compiler} \cite{BCC} and MATLAB compiler support libraries (Listing \ref{lst:hardcoded}). These components need to be installed on the host machine in order to attempt a recompilation of the library.

\begin{verbatim}
-Ic:\j2sdk1.4.2_06\include
-Ic:\j2sdk1.4.2_06\include\Win32
-Ic:\bcc\INCLUDE
-IC:\MATLAB6p5\extern\include
-IC:\MATLAB6p5\simulink\include
\end{verbatim}

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.

\begin{verbatim}
<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
    
\end{verbatim}

Listing 5.3: From JMatLink build file
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 5.4][1]. 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 5.4: 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[35], 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. 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.

5.3 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[55] 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 made 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.

### 5.3.1 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 [33], and the source code for puka is available on PhysioNet[50] 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\[25\] as a wrapper around the JMatLink interface 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.

5.3.2 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 ??, 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 considered in chapter 9, 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.

5.4 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[27].

A promising project is matlabcontrol[25], 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.

5.4.1 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[32] and will have to make sure the conversion between system is correct.

<table>
<thead>
<tr>
<th>JMatLink method</th>
<th>return type and description</th>
</tr>
</thead>
<tbody>
<tr>
<td>engGetArray</td>
<td>double[][]], Used for both 1 and 2 dim array</td>
</tr>
<tr>
<td>engGetScalar</td>
<td>double</td>
</tr>
</tbody>
</table>

Table 5.1: 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.

(TODO: expand about the analysis)

5.4.2 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[70]. 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[69] 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[25].

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[48] (fork on Github) and
- MatConsoleCtl[65]

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.

## 5.5 Creating the adapter

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

Once the application is adapted for modern system we use the existing GUI in puka while 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-project.

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 \textit{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 \textit{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.

\section*{5.5.2 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 \ref{listing:matlab_methods_used}

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.

\begin{verbatim}
//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
\end{verbatim}

\begin{table}[h]
\centering
\begin{tabular}{|c|}
\hline
Listing 5.5: Used methods in puka from JMatLink \\
\hline
\end{tabular}
\end{table}

53
5.5.3 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.

Even though the API 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 5.4, we have the following Java types to convert between Java and MATLAB, and back again:

<table>
<thead>
<tr>
<th>Java Type</th>
<th>MATLAB Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>double scalar</td>
</tr>
<tr>
<td>double[]</td>
<td>Numeric array</td>
</tr>
<tr>
<td>double[][]</td>
<td>Numeric array</td>
</tr>
</tbody>
</table>

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.

Before we implement the adapter we create unit tests for all calls puka does through the MATLAB interface (Listing 5.5). 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.

The interface does 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[26], 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. 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.

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 5.5]. 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.

5.6 Testing the modernized implementation

5.6.1 Test data and verification

In order to evaluate the accuracy and precision of pukas algorithms used for detecting respiration, we use both real world recordings from PhysioNet to derive respiration events from different types of sensors as well as simulated 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.

5.6.2 Simulated data

In the initial testing of puka we use a smaller sample sizes of the recordings representing different challenges as well as using simulated signals in order to reproduce different potential errors that can occur in recordings. The experiments conducted to map potential weak points and errors will be described in Section 7.2.

A normal respiration rate, eupnea, varies with age, activity, illness, emotion and pharmaceutical influence[30]. In "Delmar’s
<table>
<thead>
<tr>
<th>Age Group</th>
<th>Normal Respiration Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newborns</td>
<td>44 RPM</td>
</tr>
<tr>
<td>Infants</td>
<td>20-40 RPM</td>
</tr>
<tr>
<td>Children (1-7 years)</td>
<td>18-30 RPM</td>
</tr>
<tr>
<td>Adults</td>
<td>12-20 RPM</td>
</tr>
</tbody>
</table>

Figure 5.4: Normal Respiration Rate[30]

Comprehensive Medical Assisting”, normal respiration rates are defined in [Figure 5.4].

In order to evaluate the correctness of the algorithms used in puka it will be reasonable to run some experiments using simulated 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 simulated signals can be generated based on a sine function. The sine of an angle \( \omega \) in a right triangle is as the ratio 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 \( \theta \) be an angle measured counterclockwise from the x-axis along an arc of the unit circle. Then \( \sin \theta \) is the vertical coordinate of the arc endpoint, as illustrated in the left figure above[2]."

A simple implementation of a sine function in MATLAB gives us a smooth sine curve and a time series that can be

![Figure 5.5: Sine definition](image-url)
used as an ideal respiratory signal.

Listing 5.6: Test data generation in MATLAB

```matlab
ns = 0:2999;  % number of samples
am = 1;      % amplitude
base = 0;    % offset
dur = 500;   % duration of each "respiration" as
            % number of samples
            % dur:500/Hz (samples per sec) = 5 second
hz = 100;    % samples per second

sinewave = base + am * sin(2*pi*ns/dur);
```

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.

The respiration rhythm has to be taken into account. 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.

![Figure 5.6: Respiration signal from 1 minute of stage 2 sleep](image)

By manipulating these simulated signals we can create 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 Figure 5.7. Using simulated data allows us to more easily test different challenges 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.
Listing 5.7: Respiration simulation

\[
\begin{align*}
ns &= 0:29999; \quad \% \text{number of samples total in each clip} \\
am &= 1; \quad \% \text{amplitude} \\
base &= 0; \quad \% \text{offset y axis} \\
offsetStartP &= 0.5; \quad \% \text{offset x axis} \\
dur &= 3000; \quad \% \text{duration of each "respiration"}
\end{align*}
\]

\[
\begin{align*}
\text{respTest} &= \text{base} + \cos(\text{am} \times \sin(\pi \times \text{ns}/\text{dur} + \\
& \quad \text{offsetStartP}) \times \pi)
\end{align*}
\]

- amplitude fluctuation (threshold variability due to signal strength),
- noise (unexpected fluctuations),
- offset (we know puka relies on a baseline amplitude of 0) and
- baseline shift.

Figure 5.7: Scenarios to replicate with the simulated 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 5.6.

The data has to be converted into a format described in Subsection 4.1.5. 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.

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 Section 7.3.

Since the normal respiration of a human being is not the same as a perfect sinusoid curve, we create a couple of functions to generate more lifelike signals. One such approach is to clip the signal both on inspiration and expiration end (Listing 5.8), to more easily be able to validate detection of peaks and troughs and their beginning and end, in order to detect pauses in respiration.

Listing 5.8: Clipping sine wave

```matlab
% for creating clipped time series
clipped = sinewave;
cutoffTop = 1.0; % am has to be set to > cutoffTop
cutoffBottom = -1.0;

clipped(find(clipped> cutoffTop)) = cutoffTop;
clipped(find(clipped < cutoffBottom)) = cutoffBottom;

figure();
plot(ns/h, clipped);
```

The different implementations can all be found in sineGenerator.m found in the source code.
5.6.3 Real world data

PhysioNet is a collection of recorded physiologic signals and related open-source software for signal processing and analysis. The 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[43].

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 for example create CSV (comma separated value) 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 (.hea) 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.

5.7 puka in action

In order to run puka using the "JMatLinkAdapter" we have to change the instantiations of JMatLink in the source code to the the adapter class [Listing 5.7]. Calls to the loading of the JMatLink library can be removed, as it is no longer a system library 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.

```java
// Old implementation:
egnMatLab = 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();

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

In preferences.txt or via the Program Preferences in the GUI we define which column in a raw text file we will find the signal used in the respiration analysis, as described in Subsection 4.1.4. When creating the simulated 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 $1000 \times 10 = 10000$ to make a 10 second clip. The recording we use in this example is 30 seconds of simulated data, and should therefore give us ample room for the clip size to fit within the recording.

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 user. The waveform is plotted using MATLAB to indicate where in the record the onset time is, and what the signal look like.

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 5.11. The steps presented in Section 4.2 (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 4.2).

The signal in Figure 5.10 looks different from the one in Figure 5.12. 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`-script has completed we have two new arrays that are based on the previous \( P \) and \( T \); \( \text{new}P \) and \( \text{new}T \). For each peak and trough we have twin tuples marking the beginning and end of the pause around a peak.

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 5.13.

Finally the statistical computations are display for the user. This demonstrates the application now working on a simulated 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 with Figure 5.14.
Figure 5.10: Onset time marked both in the input field and on plot

Figure 5.11: Initial screen for respiration analysis
Figure 5.12: Visual feedback to the user when checking validity of peaks and troughs

Figure 5.13: Pauses identified in the troughs
Figure 5.14: Statistical computations from a 10 second clip. Signal with 3 second respiration loops
Chapter 6

Re-purposing the application for real time analysis

guideline for sections:

• Describe goal and point of realtimification
  – Strategies for realtimification
  – Discussion about which solution (pros and cons)
• Description of chosen solution (implementation steps)
• Challenges with this solution (implementation) –> move to eval
• Testing and verification –> move to eval

6.1 Describe goal and point of realtimification

Since the algorithm used for respiration analysis in puka has been implemented to be run on a pre recorded signal we will have to look into what changes to be done in order to provide a data stream with data that TRIO can use to derive useful information about the sleep quality and detect deviations from normal sleep. Looking into
what makes the algorithm work, we identify the parts that can be removed, modified and added in order to be able to supply TRIO with useful events.

The overall goal of doing the analysis real-time is to be able to detect sleep disorders report it to a different system that can act upon 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.

The statistical calculations done in the end of the analysis of a given clip gives us different insights to the qualities of the signal. But for a real-time system, these calculations are not required, as they are aggregations on a static signal. We need to identify the changes in the signal in order to convert them into events that can be sent to TRIO.

6.1.1 Strategies for real-time solution / Initial approach

• **First iteration**: naive implementation, use the existing analysis, adapt the surrounding program into receiving data and executing the analysis on different sized window

  – **What we need**:
    – Dynamic window sizes in order to compensate for long respiration pauses
    – Use historical data in the application and compare to adjust window size

• **Future work**: Modify the algorithm and optimize it for real-time usage.... but how

  – Different algorithm, describe potential candidates.
Since the algorithm for detecting the pause around a peak or trough originally tries to find the entire pause, we need to modify it in order to be able to detect events as data arrives from sensors.

puka is designed to analyse one clip within a record, then discard the rest of the remainder of the record. We therefore need to make the window/clip move along the whole record. The record will in our real-time implementation be a buffer where we store the incoming signal.

Some lag will be present in the system since the time series is generated while the peak detection and respiration analysis is running, but as long as we can report accurately the time of each event we will be able to detect the types of complex events we are interested in shortly after they have occurred.

The GUI that accompanies puka can fairly easily be removed, making a system that takes a stream input and then applies the respiration analysis to it. For our purposes we create a program that feeds data into the stripped down version of puka which on contains the respiration analysis.

We will need to create a controller class which will control the execution of the puka respiration analysis. This controller handles communication with simulated sensor(s) and initiates the analysis. This controller also contains the historical data needed to be able to make sense of the current window.

### 6.2 Implementation

As described in Subsection 6.1.1, the real-time version of puka will have to be able to detect changes and report these events to TRIO using some form of change point detection. Both the algorithm for detecting peaks and troughs and the algorithm for calculating pauses in respiration has to be modified in order to detect events in as close to real-time as possible.

We create a simple program that reads files containing time series which can be served to a the modified version of puka. This
version of puka is stripped of the GUI components, and is made to work in conjunction with the data serving program. The events detected by the modified version of puka is then compared with the results of a respiration analysis done with the original version of puka.

Figure 6.1: Structure of the real-time test application
6.2.1 puka reduced

This application controls the user-application interaction. A user can initiate the respiration analysis either from a local file or from a data serving service from a shell. The local option is intended to mimic the original puka, but fully automated and without the GUI found in the original. 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.

6.2.2 Respiration Analyser

We create a minimal version of the respiration analysis found in puka. 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.

While the original version of puka has manual steps for verifying onset time we need to automate the process. We can change the source code in either the control code (Java) or the scripts (MATLAB):

- Adaptive window, always start a recording when we have a 0 intersection
- Ignore the requirement in the script (TODO: test changes in classifyPeaks.m)

Listing 6.1: Calls to MATLAB scripts from Java application

```java
frmLoadData.engMatLab.engEvalString("[P,T,th,Qd] = newPT(y, .1, 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);");
...

6.2.3 Historical data

- What do we need to keep track of
- For how long? Buffer size/historical comparison

The controller contains

6.2.4 Data serving

We create an application which reads the data files and serves them to the puka reduced application, simulating a sensor. We also add the option to insert an explicit time stamp.

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

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, 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"[45]. 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.

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. These methods will have to be timed in order to discover how much time that can reasonably be used on the signal processing.

We will also need monitoring of the buffers in order to detect any overflows, indicating that the processing is slower than data entry rate, suggesting we need to down-sample the signal.

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Status code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request file list</td>
<td>&lt;empty&gt;</td>
<td>REQ</td>
</tr>
<tr>
<td>Request file</td>
<td>&lt;file name&gt;,&lt;num&gt;</td>
<td>REQ</td>
</tr>
<tr>
<td>Acknowledge OK</td>
<td>&lt;human readable message&gt;</td>
<td>200</td>
</tr>
<tr>
<td>Request new rate</td>
<td>&lt;requested rate&gt;</td>
<td>300</td>
</tr>
<tr>
<td>Abort</td>
<td>&lt;empty&gt;</td>
<td>400</td>
</tr>
</tbody>
</table>
Table 6.1: Calls from client to Data Feeder service protocol

Table 6.1 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.

6.3 Modifying puka for online usage

6.3.1 Changes to puka

![Figure 6.2: Place holder flow of RT impl](image)

We use windows of data in the same way as puka already uses a clip to section of a part of a larger record. The smaller the windows are, the better since we get closer to real-time but there are limits to how small a given window can be and still letting us derive useful information. The signal resolution 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 resolution 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.

How large must the window size be to be able to identify a peak or trough? We need to experiment using different window sizes in order to find a balance between lag (the time we detect the pt) and accuracy (the number of false pt’s)

We identify which parts of the application define the window size (which scripts, and if there are anything in the Java code that needs to be changed). What variables can be parametrized in order to make the experiment easier to conduct?

The classification of peaks and troughs as it is done in the implementation of puka will have to replaced with an automatic alternative or removed. An automatic implementation can use statistical data to generate a threshold or a score criteria that each event has to fulfill in order to be classified as an actual event.

1. create window
   - adaptive window, adaptive onset, create clips (windows) with a suitable onset time
   - try different window sizes
   - window: overlapping, sliding?

2. analyse while filling the next window – parallel or

**6.3.2 Window sizes**

As puka uses a *clip* in a similar way to a window. That is the size of the selection of a time series we want to analyse. In order to create a window puka looks at the *clip* and finds the first point where the signal crosses 0. Vanilla puka uses this as an indicator called *onset time*.

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 and negatives will necessarily increase as each pass of the algorithm will have less data to base its analysis on.

6.4 TODOs and NOTES


- **Real-time:**
- (Step 2 has to be automatic (in puka this is done manually by end user) or disregarded for a real-time)
- split input-signal into smaller window sizes (What size?)
- recording of statistical information for real-time analysis
• normalization of signal (based on experience with getting the program to run)

• try to make it into a one pass algorithm for realtimeness

• Keep track of an average amplitude, use this as a threshold for detecting peaks and troughs.

• Implement threshold variable into existing program.

Create `frmRespiration`, `loadData`. \( y \) in MATLAB contains the signal (start to stop only). –>

must be adjusted – discussed in implementation? The respiration analysis analyses each peak and through, and for this to work it first has to define a `window` to validate the peak or through. The `classifyPeaks` function found in the MATLAB scripts contains hard coded values that indicate the size of the window based upon the sampling rate. This should be parametrized and consistent throughout the program by passing the value stored in `preferences` in the control part of the application.
Part III

Results
Chapter 7

Evaluation

We now have both the original puka and the real time adaption working.

7.1 Metrics

How do we measure whether the realtime system is accurate.

What do we compare.

Timing.

Accuracy vs window size

Historical data, what to preserve.

7.2 Experiments

Run both programs on the same signal, overlay resulting plots. Do we have the same pauses, peaks and troughs?

Signals:
7.3 Results and Adjustments to puka

When we run pukas respiration analysis on the simulated data found in ?? we get different errors and exceptions based on what signal is used and the clip length.

TODO: table with results for different clips We need to look for

- missing peaks/troughs
- false positives

based on the simulated signal content. Our gold standard is the fact that we generate the signal to contain a certain number of respiratory cycles.

7.3.1 Results

Based on the different signals in ?? Meta about the testing done with the vanilla puka version...

7.3.2 Decimate

The first MATLAB script to be called is newPT.m. In this script, which is based on The Identification of Peaks in Physiological Signals[64], 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 7.1: Downsample code in newPT.m line 7

```matlab
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.

### 7.3.3 Mark pause peak and trough location

When running `markPeakPauses.m` with a 10 second clip on a 30 second record using the clipped sine generated signal we end up with a deadlock in the MATLAB script. The amplitude of the initial clipped sine signal was 0.8.

It checks the number of valid peaks created `classifyPeaks.m`

"newP" and "newT" are variables in the script that both contain the new, centred peak or trough based on the pause one each side. During the execution of this script we run into a problem when checking each side of the original peak/trough. The signal passed to the script is called "Qd" and is the signal in which the peaks and troughs have been found.

When peaks or troughs are too close to the beginning or end of the clip, the MATLAB script freezes during `markPeakPauses` or `markTroughPauses`. These functions iterates through the peaks and troughs analysing a windows around each event. There is no fail safe implemented, but in puka a user can choose what peaks
to analyse. Here we can de-select detected events that border too close to the beginning or end at the cost of the accuracy of the analysis.

This can also be avoided by adding an automatic discarding of these events by checking the index (sample number) of the peak/trough, making sure it is not positioned as to make the window move outside the clip array.

7.3.4 Peak and trough classification

Notes:

- As it stands, the code does not contain any reference to questionable peaks
- A further analysis of the classification code can be found in ?? TODO: describe potential errors in this one, the decimation, assumed frequency etc

7.4 Result/Analysis online

Meta about the testing done with the vanilla puka version...
Chapter 8

Discussion

... 

• zero-crossing algorithm not suitable for baseline shift!!
Chapter 9

Future Work

Other non invasive approaches

• Respiration rate using Novelda radar
• wake/sleep state
• oxygen saturation
• other bio markers?

• Calculate respiratory flow in addition to bpm

• http://www.aastweb.org/resources/focusgroups/rip_intro.pdf p 6

• Other sensors, $so^2$ $sco^2$.

In order to make the platform that has been created more descriptive and useful for validating respiration analysis done by puka (or potentially other applications), the following points can be explored.
1. Combine the events generated by pukaReduced (midlertidig navn) with other sensors.

2. Recreate the respiration analysis as an CQL or some other stream favourable system.

3. Now that we have a framework for applying algorithms to a respiratory signal, novel algorithms can be implemented in order to detect new types of events or improvements to the existing ones.

4. Test on flow as well as respiration effort. Focus here have been on RIP, but this should be applicable to other sensors such as thermistors, which are able to detect changes in flow, not only respiration effort.

5. Use the system for detecting other respiratory abnormalities, such as tachypnea, bradypnea, Cheyne-Stokes, hypoventilation, hyperpnea or hyperventilation[30].


7. Create an application that contains the "answer" we want to validate against that consumes the Esper events and creates a report.

8. standard deviation for peak detection?

9. Deviation detection: Expect changes, but how to define baseline, threshold?

Figure 9.1: Caption
9.0.1 Signal processing

The real world data that is found in the different Physionet databases prepares us for what kind of challenges can be found when using these types of sensors. Certain signal processing has to be done to prepare the signal for usage in puka.

**Post/pre processing real world data:**

- fft? high pass/low pass filter?
- smooth signal:
  - moving average -really slow, removes more noise
  - SG - much faster, keeps more of the features
  - initial
- normalization:
  -

9.0.2 beyond puka

After having implemented a close to real time version of pukas respiration analysis we have some insight as to possible paths ahead.

- Add threshold adaptation in order to classify and adjust for changes in amplitude.
- 

9.1 Other approaches to puka

9.1.1 Recompiling puka

The project uses Java’s `System.loadLibrary(String libName)` to load JMatLink. One unexplored avenue is to recompile puka even though
there is no documentation describing the linking process of the JMatLink classes. When compiling the project we need to find a way of linking the JMatLink classes. A discrepancy discovered when looking into this potential solution is that the puka source code expects an integer returned when calling `engOpen()`, but the source code for JMatLink (both the latest v1.3.0 and v1.1.0) has `engOpen()` implemented as a `void` method.
Appendix
Software

Software delivered:

- JMatLinkAdapter: an adapter for usage in puka to replace JMatLink
- SignalGenerator: lager simulerte greier. Sinegenerator.m lager ideelle og problematiske signaler
- DataFeeder: leser inn og sender sender data til en klient via socket
- pukaReduced: mottar signaler fra dataFeeder og gjennomfører respirasjons analyse.
- SignalPatcher: Rydde opp i signaler fra physionet ettersom de mangler enkeltmålinger. Se Cleaner.java for enkel løsning.

Listing 9.1: 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
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);");

// 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
frmLoadData.engMatLab.engEvalString("[avgPI,stdPI(avgPE,avgPE) = calculatePauses(newP,newT);");
frmLoadData.engMatLab.engEvalString("[avgTI,stdTI,avgTE,avgTE] = 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

NOTES to be removed

Two main approaches:

Either create events for each PI/PE start and stop, or calculate the pause (most similar to the existing implementation) by increasing the size of the clip! (makes more sense?)

WIP:

- Windows - timed, find useful timeframe
- types of events, trough, peak, respiration stops...
- what is required from TRIO?
- update frequency?
- increase decrease in frequency = sacrifice precision?

- main experiments
- window size
- results compared to manual reading
- results compared to ECG algorithm???
other events that are noteworthy?

/NOTES

Troubleshooting

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

MATLAB script execution freeze

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

Listing 9.2: 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 9.3: 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 trial and error we found the problem in one of the parameters given to the MATLAB engine on start up. The application does not work with proxy: setHidden() set to true. Something
in the implementation creates problems. Without any debug information 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 authors 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 variable 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.

**Misc bugs**

List of smaller, syntactical or otherwise easily fixed bugs found in the source code of puka.

- **frmRespiration.java:43** – ‘,” instead of ‘:', might be change in MATLAB, but this results in y being set to a scalar instead of an array.
# List of Figures

2.1 Obstructive Sleep Apnea ........................................ 8
2.2 Sleep stages\[60\] .............................................. 15
2.3 ECG signal illustrated ............................................ 17
2.4 Precision and accuracy\[23\] ...................................... 20
2.5 A simplified Data Stream Management System ............... 23
2.6 Esper components\[59\] ........................................... 24
3.1 Non contact sensor from Nintendo\[42\] ......................... 28
4.1 The main flow of puka (generalized) .......................... 33
4.2 Preferences stored in preferences.txt .......................... 38
5.1 Resulting error message from regsvr32 ....................... 43
5.2 Missing Windows libraries and error message from Dependency Walker .......................... 43
5.3 UML of the respiration analysis aspect of puka ............... 52
5.4 Normal Respiration Rate\[30\] ...................................... 57
5.5 Sine definition ....................................................... 57
5.6 Respiration signal from 1 minute of stage 2 sleep\[16\] ........ 58
5.7 Scenarios to replicate with the simulated signal . . . . 59
5.8 The clips tab in preferences . . . . . . . . . . . . . . . 64
5.9 Load file dialogue . . . . . . . . . . . . . . . . . . . . 65
5.10 Onset time marked both in the input field and on plot 66
5.11 Initial screen for respiration analysis . . . . . . . . . 66
5.12 Visual feedback to the user when checking validity of 67
peaks and troughs . . . . . . . . . . . . . . . . . . . . . . 67
5.13 Pauses identified in the troughs . . . . . . . . . . . . 67
5.14 Statistical computations from a 10 second clip. Signal 68
with 3 second respiration loops . . . . . . . . . . . . . . . 68
6.1 Structure of the real-time test application . . . . . . . 72
6.2 Place holder flow of RT impl . . . . . . . . . . . . . . . 76
6.3 Details for the implementation . . . . . . . . . . . . . 78
6.4 Window sizes used in experiment . . . . . . . . . . . . 78
9.1 Caption . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
List of Tables

2.1 AHI severity scale . . . . . . . . . . . . . . . . . . . . . 12

4.1 External dependencies for puka . . . . . . . . . . . . 35

4.2 WFDB programs used by puka . . . . . . . . . . . . 36

5.1 JMatLink methods used by puka . . . . . . . . . . . . 50

6.1 Calls from client to Data Feeder service protocol . . . 75
Bibliography


[61] Sleepdex. Esper ... [http://www.esper.stuff], TODO.


