Mobilizing Victims of Sleep Paralysis Using Small Embedded Devices

Xavier Villa[1]

***Abstract*— Breathing patterns during Sleep Paralysis can be controlled in a manner that is detectable to humans, and now by small embedded devices. This study is on the ability of small embedded systems to detect Sleep Paralysis, a frightening condition that is common among the general population, and chronic among many who experience it [1]. Sleep paralysis is a 30 second to 5 minute episode where somebody suddenly becomes conscious in the middle of REM sleep but is unable to speak or control movement with the exception of breathing and eye movement. It is most often accompanied with mild to severe auditory and visual hallucinations. Currently, the only way for an individual to be woken from sleep paralysis, is by either waiting for the event to naturally subside, or to be woken up by an external event, like a loved one who witnesses it happening. Using real-time accelerometer readings of sleepers' chests, signal processing techniques, feature extraction, and machine learning techniques, a classifier was designed and implemented on hardware with astonishing results. Utilizing the ARM Cortex m4 SoC [9] and the MPU6050 gyroscope and accelerometer unit [10], Sleep Paralysis detection was achieved with high accuracy and low false-positives by monitoring breathing patterns. Results of this study indicate that future improvements can include additional compact hardware that provides a method of waking individuals when sleep paralysis is detected. This study indicates the effectiveness and usability of small embedded systems in the psychiatric field, and more importantly, suggests further improvements can alleviate these frightful events for chronic victims during the night, and provide a more energized waking life during the day.**

I. INTRODUCTION

The main agenda for this study was to build an embedded device prototype that when attached to the chest of sleepers can successfully detect Sleep Paralysis (SP) episodes, and also to quantify and improve the effectiveness of the device on test subjects, and discuss the potential for deploying additional hardware to the device as a mechanism for waking individuals upon detection.

Sleep paralysis (SP) is defined as a period of consciousness during sleep or the process of falling asleep where voluntary muscle movement is inhibited, but ocular (eye) and respiratory movements are intact [1]. These characteristics can be detected in SP victims by monitoring breathing patterns alone. Because subjects are conscious and aware of their surroundings, one could intentionally trigger an alarm by altering their breathing pattern- in fact, some report breathing heavily in an attempt to try and wake their loved one next to them. Waking up subjects from this state is as simple as placing a hand on them and telling them to wake up. This alone stops any auditory and visual hallucinations the victim may be experiencing, and they will become mobile. In the case of SP where sleepers are unable to be externally woken by another person, over 90 percent report a feeling of overwhelming fear. These individuals would greatly benefit from a device that could detect SP events and wake them.

[1]Xavier is a student and member of The Adaptive Networked Distributed Embedded Systems (ANDES) research group at the University of California, Merced.

II. PRELIMINARY DESIGN

To create a working, testable prototype of a device that can detect SP, various challenges were addressed and overcome.

A. Preliminary Hardware

The hardware used for the initial prototype was an Atmega328 chip on an Arduino Nano prototype board and the MPU6050 6-axis accelerometer and gyroscope. The Arduino Nano was capable of sampling very quickly-1Khz using interrupts triggered by the MPU6050's built in Digital Motion Processor (DMP). A reasonable high-end estimate of the frequency range of human motion was assumed to be 16 Hz. In order to detect this range, the sample frequency of 32Hz was determined to be sufficient for this application. Setting the interrupt sample rate was done by writing to the configuration register of the MPU6050's built in Digital Motion Processor (DMP). Although this can also be done through the use of the i2cdevlin MPU6050 library [11], the library is not useful unless you fully understand the MPU6050 datasheet [12], as you are not setting a sample rate directly to the register. Instead, you must set a divider. To calculate the sample rate, the following function is used:

$$SampleRate = GyroscopeRate/(1 + SMPLRT\_DIV)$$

By default, the Gyroscope Output Rate is 8KHz, but but non-intuitively, enabling low-pass filtering in the DMP changes the Gyroscope Output Rate to 1Khz. Figure 1 shows the low pass filter settings. Further adding to confusion, Depending on the value of another seemingly arbitrary register that I have yet to determine, the desired $SMPLRT\_DIV$ value may need to be halved. Because a low pass filter is later enabled, and because a divider of 31 resulted in half the

The DLPF is configured by *DLPF_CFG*. The accelerometer and gyroscope are filtered according to the value of *DLPF_CFG* as shown in the table below.

DLPF_CFG	Accelerometer (F_s = 1kHz)		Gyroscope		
	Bandwidth (Hz)	Delay (ms)	Bandwidth (Hz)	Delay (ms)	Fs (kHz)
0	260	0	256	0.98	8
1	184	2.0	188	1.9	1
2	94	3.0	98	2.8	1
3	44	4.9	42	4.8	1
4	21	8.5	20	8.3	1
5	10	13.8	10	13.4	1
6	5	19.0	5	18.6	1
7	RESERVED		RESERVED		8

Fig. 1: $DLPF\_CFG$ options from the MPU6050 Datasheet

frequency I desired, A $SMPLRT\_DIV$ of 15 was used to achieve 32 Hz sample rate.

```
// set the SMPLRT_DIV divider to 15 to ...
    get sample rate of 32Hz
mpu.setRate(15);

//set digital low pass filter to 20Hz
mpu.setDLPFMode(MPU6050_DLPF_BW_20);
```

B. Data Gathering

Raw accelerometer and gyroscope data from the Arduino was then recorded through serial communication to files on a computer for interpreting later. Recording data through serial proved to be a challenge on my build of Linux (more on this later), so recordings at this stage were copy and pasted from Serial Monitor in Arduino IDE. In order to have a quick proof-of-concept, the data recorded during this stage was all artificially generated by controlling the MPU unit by hand. In addition, the classifier to be trained was intended to detect walking as a label of "1", and the compliment of walking would be labeled "0". This was done for a proof of concept because of the more easily distinguished features of walking versus not walking. Walking-labeled recordings were made by holding the sensor to my chest, and walking at different speeds for long durations while being cautious to not change the orientation of the sensor on my chest. Non walking activities like sitting, laying, standing, Etc. were recorded in the same fashion. All walking-labeled recordings were stored in a folder titled "1", and all non-walking-labeled recordings were stored in a folder titled "0".

C. Feature Extraction

Machine learning with high resolution sensor data like an accelerometer is computationally intensive, so the embedded device needed to contain pre-trained algorithms from a host machine that has the resources required. There are many frameworks and tools to build a classifier using machine learning. MATLAB[2] is the computational environment chosen for this research due to it's abundance of support and documentation for signal processing and machine learning toolkits. Machine learning with raw data from the accelerometer and gyroscope data is not practical due to the computation required and noise in the signals. The activities can be learned with far less data-points and computational power when using key features of the activities rather than using raw sensor data from the activities. The data was preprocessed by the MPU6050's built-in Digital Motion Processor (DMP) with a low-pass filter by writing to the DMP's configuration registers. The low-pass filter used was 21Hz. Known activities (walking, sitting, sleeping) were examined with a spectrum analyzer. Key characteristics of these activities that differ from walking were selected. The only features used for training and detection were initially, mean, principle component analysis (PCA) coefficients, and standard deviation. After the initial proof-of-concept, additional features were added to train models with much higher accuracy. These characteristics are computed from a sample size of 64. 64 samples at 32Hz means approximately two seconds worth of measurements. The reason for choosing this value, is that walking (and SP) signals are low frequency, around 2-3Hz. Detecting these low frequencies are more easily done with a longer sample size. Although frequency features weren't initially extracted for this proof-of-concept, it was also helpful to measure two seconds worth of data to ensure that any delay caused by the prediction algorithm did not introduce additional noise or interfere with the accelerometer readings. If the delay was long enough such that the fresh samples were dropped, at least the frequency of the dropped samples would be lower and further spread apart.

D. Training

Once these features were computationally selected from the raw data, it was fed into a series of machine learning models in the MATLAB Statistics and Machine Learning Toolbox[4]. Parameters and features were then tuned iterative in order to increase the classifier's accuracy.

E. Simulating In Real-Time

To ensure that the newly-trained classifier was capable of real time sensor data analysis and labeling, real time environment was simulated prior to deployment. This was done by classifying one to two second samples in a loop and evaluating the resulting labels against the known labels. Predicting in this manner worked with no issues. The Real-time test is shown below:

```
% X, Y, and Z are all matrices with a width ...
    of 64 samples (2sec)
for i = 1:size(trainActivity)
  label(i) = predict(X(i,:), Y(i,:), Z(i,:));
end
```

F. C/C++ Code Generation

The predict function contains the signal processing and feature extraction code. It then computes a label with those features and returns it. A benefit of calling a single function to perform every task required to predict, is that MATLAB can generate complete C/C++ code of a function (with many restraints discussed later). MATLAB even allows C code to be generated optimized for a wide range of specific platforms, including the 8-bit Atmel AVR architecture used by the Arduino Nano. The MATLAB toolbox for this called MATLAB Coder[7]. Using Matlab Coder, the entire predict function was exported to C files and included into the the Arduino IDE as a library. This made writing the C code on the Arduino extremely easy, since all that needed to be done, was to collect 64 samples of x, y, and z accelerometer data over the course of two seconds, and then send the pointer to the data to the predict function so that a label could be returned. Exporting to C/C++ from MATLAB was not intuitive at first. The documentation for how to do this is available online, however many MATLAB libraries use objects and datatypes

that are not supported for Code Generation. This required quite a bit of manual labor, especially later when frequency analysis was later introduced as a feature for extraction. Many tables being used needed to be converted to matrices. In addition, functions and references to those tables had to be changed to accommodate regular matrices. After much trial and error, C code was able to be generated from the predict function.

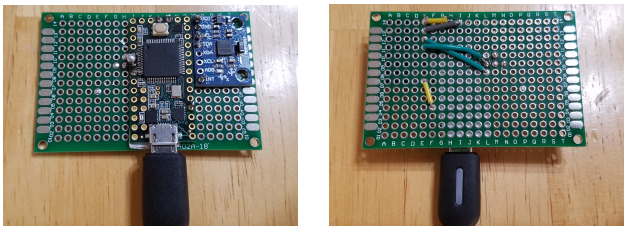
G. Compiling To The Microcontroller

Including Libraries in the Arduino IDE was not as intuitive as one might expect. After hours of frustrating errors trying to compile my simple Arduino sketch with the newly included library, I finally found the solution. All of the 40-something generated C code files had to be renamed to the *.cpp file extension, and placed inside the sketch libraries folder. I was unable to include the library by placing the files in the same directory as my sketch, and never was able to get it to compile with the *.c file extension. This meant that every time C code was regenerated from MATLAB (which was very frequent), additional labor was required to compile the library to the Arduino. I alleviated this issue by using a script to perform these tasks for me each time I regenerated C code from MATLAB. The script contents are shown below.

```
1  rm -r mod-predictor
2  mv predictor mod-predictor
3  cp -a mod-predictor predictor
4  rm predictor/buildInfo.mat
5  rm predictor/predictor_ref.rsp
6  rm predictor/predictor_rtw.mk
7  rm predictor/codeInfo.mat
8  rm predictor/gcGuiReport.mat
9  rm predictor/rtw_proj.tmw
10 rm -r predictor/examples
11 rm -r predictor/html
12 rm -r predictor/interface
13 for x in predictor/*.c;do mv $x ${x}pp;done
14 rm -r ...
       /home/username/Arduino/libraries/predictor
15 cp -a predictor ...
       /home/username/Arduino/libraries/predictor
```

H. Atmega328 Is Not Enough

After finally clearing all of the errors in the Arduino IDE about library references, I immediately discovered that there was not enough program space for the sketch to fit on the Atmega328 that the Arduino was using. Determined to make the Arduino work for the project, I began reducing everything. I removed PCA as a feature for extraction, removed Gyroscope readings and computations completely, and even reduced the sample rate to 8Hz. Only then did the sketch fit on the Arduino with less than 20 Bytes to spare. Surprisingly enough, even with just this setup, the Arduino was able to fairly reliably detect walking against other types of movements. All detection of walking was indicated by a lit LED on the Teensy3.2. It was clear that the Atmega328 was not enough to detect SP, but served as a good proof-of-concept for what was to come.



(a) front side (b) bottom (chest) side

Fig. 2: Teensy3.2 board soldered to MPU6050

III. Final Design

A. Final Hardware

The Arduino Nano's Atmega328 Microcontroller was not an option for the amount of data and program size required for the purpose of SP detection. The ARM Cortex M4 was chosen instead due to it's compact size, low power requirements, fast processing, and most importantly, available memory and program storage. The prototype board used is called the Teensy3.2 [13] which can be seen in figure 2. The specifications of the Teensy3.2 are shown below.

- 32 bit ARM Cortex-M4 72 MHz CPU (M4 = DSP extensions)
- 256K Flash Memory, 64K RAM, 2K EEPROM
- 21 High Resolution Analog Inputs (13 bits usable, 16 bit hardware)
- 34 Digital I/O Pins (5V tolerance on Digital Inputs)
- 12 PWM outputs
- 7 Timers for intervals/delays, separate from PWM
- USB with dedicated DMA memory transfers
- 3 UARTs (serial ports)
- SPI, I2C, I2S,CAN Bus, IR modulator
- I2S (for high quality audio interface)
- Real Time Clock (with user-added 32.768 crystal and battery)

B. Data Gathering Done Right

With the assembled hardware in a compact, flat form-factor, the system was attached to my left lowest rib with a fabric medical wrapped around my chest. This location was chosen because during rapid breathing, it appears to move the most distance compared to my chest center line, or stomach. It is also a convenient location that doesn't interfere with normal sleeping positions. In fact, the thin medical wrap alone added enough padding to the device that I woke up on top of it and forgot I was wearing it while I laid in bed for several minutes awake. The device remained tethered to a PC with a 10 foot USB cable for serial logging overnight and throughout the day. Non-SP recordings were made by performing ordinary daily activities like laying, sitting, standing, and walking, and of course, sleeping overnight. To record real sleep paralysis episodes would require a partner to mark the data being recorded as an episode, however, best effort was made to simulate rapid breathing motions for multiple SP recordings in various different sleeping positions. Having experienced SP thousands of times, and waking my fiancée

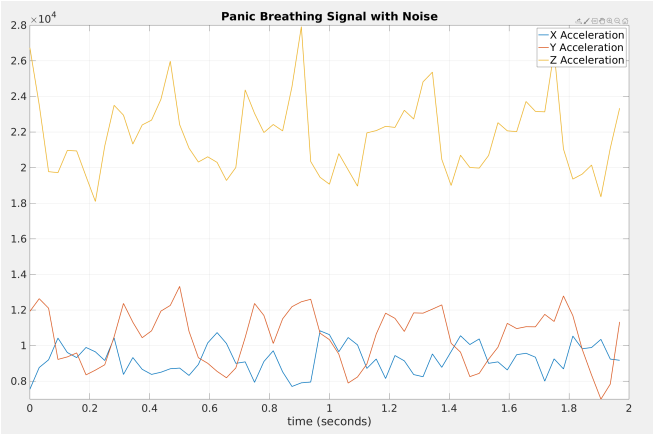


Fig. 3: Raw accelerometer data of an SP episode

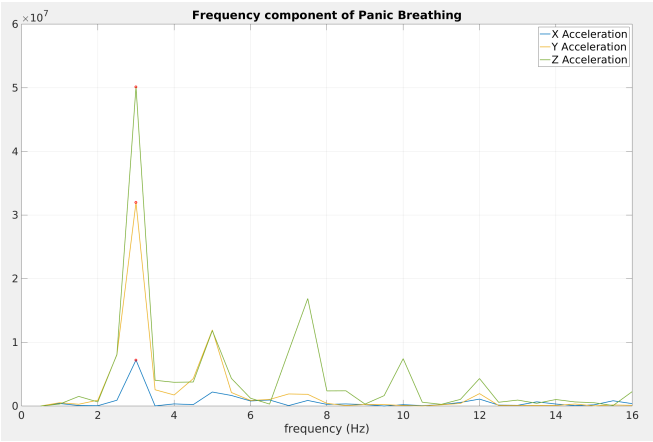


Fig. 4: Frequency component of an SP episode with peaks marked in red

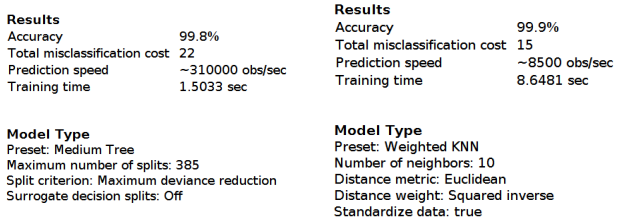
up with this ”panic breathing” very frequently, I have high confidence that this data accurately represents a real SP episode. All SP recordings were stored in a folder titled ”1”, and the non-SP recordings were stored in a folder titled ”0”.

C. Feature Extraction Done Right

With the new burly ARM Cortex M4, the features that were previously disabled were re-enabled, and more were added for extraction. The most useful feature is the fundamental frequency. Using the 64-sample, raw accelerometer buffer as shown in figure 3, the Fast Fourier Transform was computed after subtracting the mean from the contents of the buffer to standardize the samples. Using the Fast Fourier Transform, the Power Spectrum was computed as shown in figure 4. The index of the highest point of the Power Spectrum is chosen as the most prominent frequency in the sample. For SP, this value will almost always be 2.5-3Hz. Rarely, other frequencies appear with a higher amplitude during SP, and more often, 2.5-3Hz signals are prominent in non-SP activities. A future improvement that I suspect will improve accuracy, is computing the power contained in the 2.5-3Hz range for extraction. Especially because the Z axis contains the highest power at that frequency during SP, since the Z axis is normal to the sleeper’s chest. As previously stated, these computations can be done easily using the Signal Processing Toolbox[3] in MATLAB, however these functions are not currently supported by MATLAB Coder for C code generation. Thus, I wrote a function to find the prominent frequency.

D. A Classifier Trained Right

A machine learning classifier was designed to classify accelerometer data from the sleeper’s chests. The classifier contains only two prediction labels: ”1” which represents SP detection, and ”0” corresponding to the compliment of the SP detection set. Many types of classification models were tested, but primarily Decision Trees and K Nearest Neighbor (KNN) models. The results of training the best Tree and KNN classifiers are shown in Figure 5. The final



(a) Tree Classifier

(b) KNN Classifier

Fig. 5: Trained Classification Models in Matlab

model chosen was a decision tree due to it’s much faster training, high accuracy, and low memory consumption which is ideal for a small embedded device. Allowing MATLAB’s built-in iterative model tester to choose optimal parameters for the tree, the following values were used.

```
% Train a classifier
% This code specifies all the classifier ...
    options and trains the classifier.
classificationTree = fitctree(...
    predictors, ...
    response, ...
    'SplitCriterion', 'gdi', ...
    'MaxNumSplits', 100, ...
    'Surrogate', 'off', ...
    'ClassNames', [0; 1]);

% Perform cross-validation
partitionedModel = crossval( ...
    trainedClassifier.ClassificationTree, ...
    'KFold', 10);
```

The results of the testing after training with overnight data and wearing the device properly were astonishing. Running the real-time simulator, the output of the preduction results are shown in figure 6

The features chosen for extraction are best when they have a tendency to correlate strongly with SP activities. One simple and easy to understand example is mean. The average of the X, Y, and Z accelerations will almost always result in the Z value being higher than X and Y, because when Laying down flat, gravity acts strongest with the Z axis. This is why there is an obvious difference between the Z and

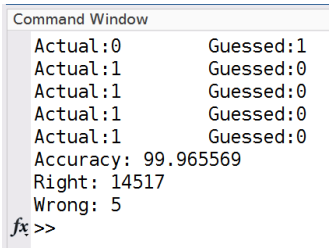


Fig. 6: High accuracy of decision tree classifier

the X and Y signals in figure 3. One way of evaluating the effectiveness or usefulness of features, is by plotting what is called a Parallel Coordinates Plot. The Parallel Coordinates Plot of the chosen classifier can be seen in figure 7. Along the X axis, are the different features used for training and prediction. On the Y axis, is the standard deviation of each of the samples' features. The higher the standard deviation, the less consistent the measurements, and the lower the standard deviation, the more consistent the measurement. Luckily, SP is an activity where SP features tend to have lower standard deviation than non-SP activities, further increasing the accuracy possible. The red lines represent SP activity and the blue lines indicate non-SP activity. The variation in red lines represent different sleeping positions when the episode occurs, different frequencies of SP panic breathing, and other factors. The large variation in blue lines represent a wide range of activities from walking, standing, sitting, sleeping, brushing teeth, and anything else non-SP. The features that have the most overlapping red lines, and the least overlapping blue lines are the best features to use because they set SP apart from non-SP activities, and make the algorithm more accurate and efficient. Notice how the features with the most overlapping red lines correspond to the Z axis. This is because the Z axis is normal to the chest and experiences the most acceleration during SP.

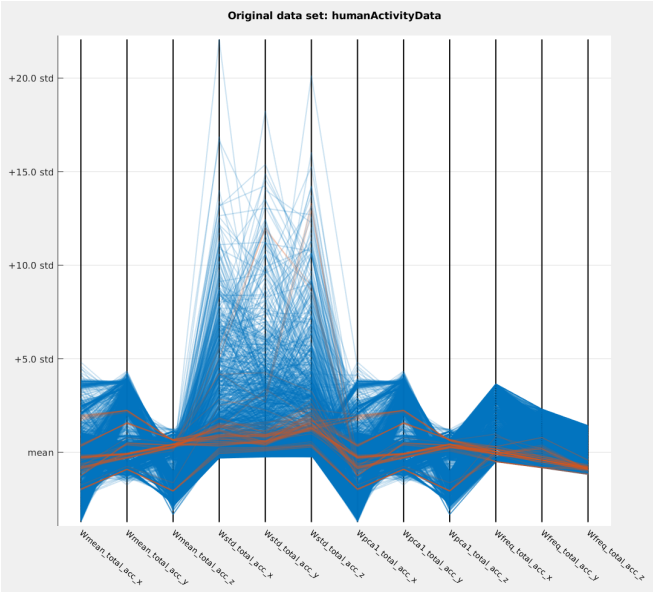


Fig. 7: The Parallel Coordinates Plot of the chosen classifier

E. Final Testing On Hardware

All SP Detections were indicated by the Teensy3.2 With both a lit LED on the device, and a message through serial communication to a computer. The device sits flat on the lower-left rib where the most movement occurs during breathing. It is easily secured in place with a lightweight, thin, fabric medical wrap that goes around the chest once. The 10-foot-long USB cable tether, testing felt wireless and easy. Sleeping with the device, even while tethered, was comfortable, and I forgot I had it on the next morning while laying in bed awake. Wearing this device around the house does not trigger SP detection, and sleeping throughout the night does not result in false positives. When performing SP panic breathing, the device triggers within two seconds every time it has been tested up to this point, regardless the of sleeping orientation.

IV. Results And Suggestions For Improvement

The final product of this study is a low power, unobtrusive, embedded device capable of detecting sleep paralysis throughout a full night of sleep with little-to-no false alarms.

Further testing needs to be conducted to examine the compatibility of the classification algorithm accross multiple individuals with different body types and habits.

Further improvements that need to be addressed is a small, flat case to house the electronics, and include a battery. Rechargeable Lithium Polymer battery cells are the best choice for this research because of their flat form factor and high energy density. Common varieties of small Lithium Polymer cells for individual sell online include 40, 70, and 110 milliampere hour (mAh) capacities. At these capacities, the cells are smaller than a United States Quarter Dollar. Lithium Polymer cells are all 3.7 Volt nominal, and typically operate between 4.2 Volts fully charged, and 2.8 Volts fully discharged, which is ideal for the ARM Cortex M3 SoC that uses 3.3v logic.

Once this has been achieved, a mechanism of waking the individual during an SP event can be implemented. Waking up a subject experiencing SP with an audible tone would be obtrusive, because it could interrupt the sleep of other members in the vicinity. Waking up the subject without generating an audible tone requires touch or motion. In order to keep power consumption to a minimum, it may be wise to wake the subject using a tensioned coil spring that can be set prior to attaching the device to the sleeper's chest. This way, a small electrical output from the device can cause the spring to release the stored energy. The energy stored in the spring should be enough to wake the patient with a strike on the chest from a dime-sized rubber piston. This spring tension should be easily adjustable to accommodate for different striking strengths.

V. Conclusion

Breathing patterns during Sleep Paralysis can be accurately detected and classified by small embedded devices to trigger events by monitoring breathing patterns with an accelerometer and a low powered SoC. This study demonstrates the ability of embedded systems to alleviate sleepers of this frightening, sometimes chronic condition, further indicating the effectiveness and usability of small embedded systems in the psychiatric field. Further research and testing may need to be performed to iteratively improve the classifier if it does not support a wide range of individual body types and sleeping habits. A battery powered, non-tethered version of the device needs to be created. Furthermore, a case to house the components needs to be fabricated. Finally, a real hardware mechanism to wake individuals after an SP detection has yet to be designed, built and tested.

References

[1] B. A. Sharpless and J. P. Barber, "Lifetime prevalence rates of sleep paralysis: A systematic review," *Sleep Medicine Reviews*, vol. 15, no. 5, pp. 311–315, 2011.

[2] "MATLAB - MathWorks", *Mathworks.com*, 2019. [Online]. Available: https://www.mathworks.com/products/matlab.html. [Accessed: 01- Oct- 2019].

[3] "Signal Processing Toolbox", *Mathworks.com*, 2019. [Online]. Available: https://www.mathworks.com/products/signal.html. [Accessed: 01- Oct- 2019].

[4] "Statistics and Machine Learning Toolbox", *Mathworks.com*, 2019. [Online]. Available: https://www.mathworks.com/products/statistics .html. [Accessed: 01- Oct- 2019].

[5] "Deep Learning Toolbox", *Mathworks.com*, 2019. [Online]. Available: https://www.mathworks.com/products/deep-learning.html. [Accessed: 01- Oct- 2019].

[6] "DSP System Toolbox", *Mathworks.com*, 2019. [Online]. Available: https://www.mathworks.com/products/dsp-system.html. [Accessed: 01- Oct- 2019].

[7] "MATLAB Coder", *Mathworks.com*, 2019. [Online]. Available: https://www.mathworks.com/products/matlab-coder.html. [Accessed: 01- Oct- 2019].

[8] G Bunkheila, *Signal Processing and Machine Learning Techniques for Sensor Data Analytics* (2019). Accessed: Sept. 30, 2019. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/53001-code-for-webinar-signal-processing-and-machine-learning-techniques-for-sensor-data-analytics?s_tid=mwa_osa_a

[9] https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4

[10] https://www.invensense.com/products/motion-tracking/6-axis/mpu-6050/

[11] https://github.com/jrowberg/i2cdevlib/tree/master/Arduino/MPU6050

[12] https://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf

[13] https://www.pjrc.com/teensy/teensy31.html