Audification And Sonification Of Power Grid Data

Patrick Cowden

ECE 499, Capstone Design Project

Supervisor: Luke Dosiek

March 16, 2017

Report Summary:

In the power grid, the dynamics of the system are constantly fluctuating. These fluctuations/changes of the system are referred to as the response of the power system. For this project, focus on voltage magnitude and frequency fluctuation was taken. There are currently no applications for analyzing these system responses as audio signals. The main objective of this project was to develop algorithms for processing and conversion of power grid data into audio signals. The goal of this project was to determine the usefulness of audio representation for power system response analysis. Two approaches were taken for this method. A sonification approach was taken to convert the data into MIDI notes that are played back through an audio synthesizer. The voltage of the power system data controls the volume of the MIDI note and the frequency controls the pitch. An audification approach was also taken that processed the sinusoidal power system voltage data and directly converted the signal into an audible signal. The sonification approach was completed through algorithms written in python while the process of audification was written in MATLAB. To finish the project a survey was conducted with Union professors and students. The results of this survey revealed that the audio generated from audification gave clear representation of system responses for listeners. The survey also revealed sonification was able to generate music that was pleasing to the listener.

Throughout this report the general approach to this project is discussed. Background information is given to inform the reader of the necessary information and components for understanding of the concepts discussed. Design requirements, alternatives, and preliminary design are also discussed in great detail to explain the proposed project implementation and planning. The final design is discussed thoroughly along with the results, production schedule, cost analysis, and instruction manual for the project.

Table of Contents

Report Summary **2**
Table of Figures **4**
Introduction **5**
Background **8**
Design Requirements **11**
Design Alternatives **15**
Preliminary Proposed Design **18**
Final Design And Implementation **23**
Performance Estimates And Results **30**
Production Schedule **37**
Cost Analysis **39**
User's Manual **40**
Discussion, Conclusions, And Recommendations **41**
References **43**
Appendix A: Bit Scope Data Capture Python Code **44**
Appendix B: MIDI Pitch Parameter Generator MATLAB Code **49**
Appendix C: MIDI Velocity Parameter Generator MATLAB Code **50**
Appendix D: MIDI Velocity Parameter Generator MATLAB Code **51**
Appendix E: MIDI Velocity Parameter Generator MATLAB Code **52**
Appendix F: MIDI Velocity Parameter Generator MATLAB Code **53**
Appendix G: MIDI Velocity Parameter Generator MATLAB Code **54**

Table of Figures and Tables

Figure 1: U.S. Power Grid Section Distribution 5
Figure 2: Forced Event Station Frequency Response 6
Figure 3: Forced Event Station Voltage Magnitude Response 6
Figure 4: System General Block Diagram 11
Figure 5: Virtual Synthesizer Interface 14
Figure 6: Open Pipe Midi USB Shield 16
Figure 7: Historical Data Parameter Extraction Example MATLAB 18
Figure 8: Synchrophasor Measurement System 19
Figure 9: Text File Data Parameter Extraction Example MATLAB 19
Figure 10: Synchrophasor Measurement Device Measured Local Frequency 20
Figure 11: Synchrophasor Measurement Device Measured Local Frequency 20
Figure 12: Preliminary Audification Block Diagram 22
Figure 13: Audification Workflow Diagram 23
Figure 14: Sonification Workflow Diagram 26
Figure 15: MIDI File Generation Python Code 26
Figure 16: Ambient Data Frequency Deviations Normalization 31
Figure 17: Ambient Data Voltage Magnitude Normalization 31
Figure 18: Ambient Data Signal Phase 32
Figure 19: Ambient Data Signal Before And After Audification 32
Figure 20: Forced Oscillation Normalized Frequencies 33
Figure 21a: Note Velocity Mapping Figure 34
Figure 21b: Transient Response Voltage Magnitude 34
Figure 22a: Note Pitch Mapping 34
Figure 22b: Transient Response Frequency 34
Figure 23: Ambient Data Sonification With Randomized Note Length in Bb key 35
Table 1: Union Student/Faculty Survey Results 36
Table 2: Project Software Costs 39

Introduction:

In power grid systems, power is distributed across power stations using synchronized phasors or synchrophasors. Synchrophasors are extremely useful for examining power system functionality and stability [1]. The distributed signal consists of a sinusoidal waveform with magnitude and frequency. In the U.S., the distribution of voltage across the power grid varies in magnitude depending on the power grid station but ranges in orders of kV magnitude. The distributed signal is at a frequency value of 60 Hz, with a local stepped down socket voltage magnitude of 120V. For the U.S. power grid, there are three main sections of the main grid as shown in Figure 1.

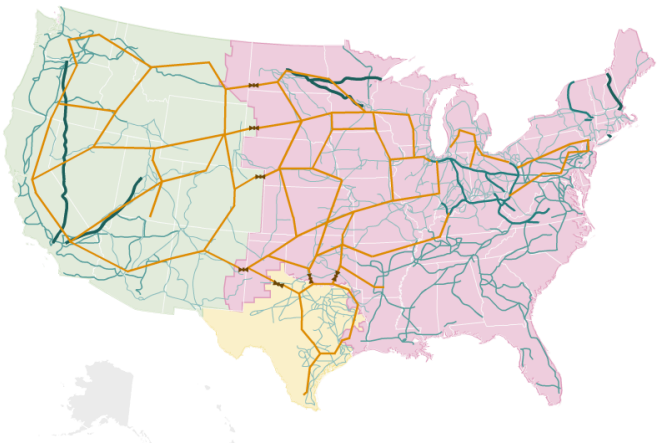


Figure 1: U.S. Power Grid Section Distribution [2]

The left green section is the Western section, the yellow is the Texas section and the pink is the Eastern section. The frequency across each individual section of the grid is the same across every point in that section. Ideally frequency and voltage magnitude are constant, but due to changes in applied load, the values constantly fluctuate over time. Power grid system operators use this effect to complete examination on system responses to high loads and forced voltage oscillations. Forced events result in system responses similar to what is shown below in Figures 2 and 3.

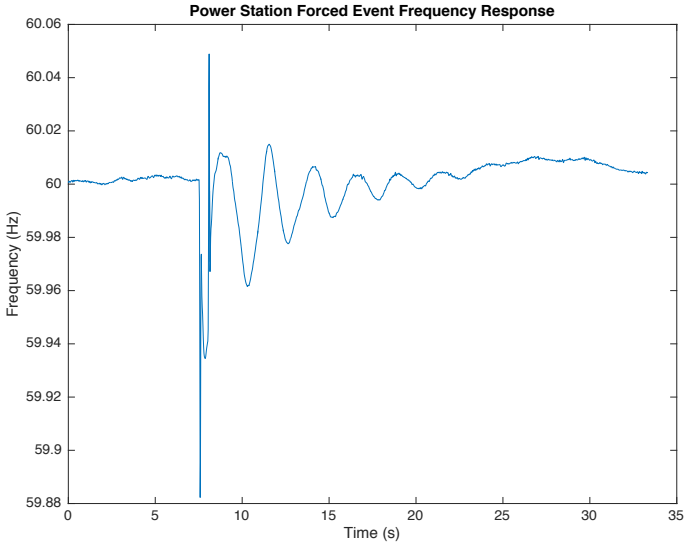


Figure 2: Forced Event Station Frequency Response

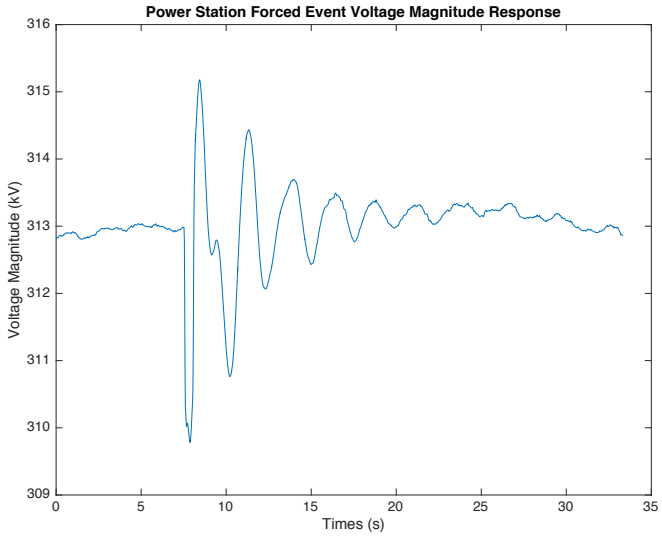


Figure 3: Forced Event Station Voltage Magnitude Response

These plots display the frequency and voltage magnitude fluctuations as a result of the Forced event. Forced events occur when an operator turns on a resistor of GΩ magnitude for a few seconds and the response of the system is called the transient response [3]. Currently, there are no existing methods that process these parameter fluctuations to generate audio representations for the various system parameter fluctuations. Audio representation of power grid data could be very useful for various audiences. The audio signals would represent the various system individual parameters and how they fluctuate over time. Generated power grid audio signals

could be used for education of the general public, as well as an introduction of power engineering to students. Power grid station employees could also use the generated audio to analyze system responses to forced voltage spike events occurring across individual power grid stations. The objective of this project is to create signal-processing algorithms that will generate audio representing power grid data. A sonification and audification approach will be taken towards processing and audio representation of the data. Sonification will serve as a musical mapping of the frequency and voltage fluctuations. For this approach, note parameters controlled by the voltage magnitude and frequency will be played through an audio synthesizer. The audification algorithm will examine the signal parameters directly and rebuild the mathematical representation of the signal as an audible audio signal. The algorithms will analyze each parameter of the power grid voltage signal and generate audio signal's corresponding to the data. The audio will accurately represent voltage magnitude and frequency fluctuations of various periods of time.

Background:

The power grid instantaneous voltage is a sinusoidal cosine with the equation

$$v(t) = \sqrt{2}V(t) * cos(2\pi f_c t + \theta(t)) \quad (1)$$

where, V(t) is the voltage magnitude, θ(t) is phase angle, and f_c is the 60 Hz carrier frequency of the signal [4]. The frequency deviation from the carrier can be extracted from this equation by taking the derivative of the phase angle. Extraction of the frequency is how the fluctuations of frequency can be processed and analyzed.

Previous examples of sonification and audification have not been performed on power grid data before, but the approach has been taken towards various types of other data. A group of engineers in California developed an audification algorithm for generating audio representations of seismograph data called “Sounds Of Seismic”. The generated audio samples are very interesting and allow for a clear audible representation of various earthquakes that have occurred across the world [5]. In regards to the sonification of data, A German engineer, David Worrall performed the sonification of network metadata. Worrall was able to accurately measure the network data flow rate using musical note pitch and time [6].

Although the two approaches have not been taken towards power grid data, analysis of frequency fluctuations has been completed before. Frequency fluctuation processing has been studied for possible forensic applications by various scientists/engineers. My advisor, Professor Luke Dosiek, performed research on this topic and extracting frequency fluctuation from digital recordings. His research examined a forensic technique for using the frequency fluctuations to time stamp digital recordings [4]. Since frequency fluctuations are consistent across each section

of the power grid, he aimed at determining the time at which digital recordings were recorded. His attempt was to extract the frequency from the recordings and compare it to power grid frequency data. With various other examples of audification/sonification, this project is a very interesting area of research. When planning the design and implementation of this project there were various areas of concern that needed to be considered.

The first issue is the manufacturability of the project. Since the project is entirely written using software, the algorithms could be distributed to other engineers for use but not to the general public. The algorithms of this project were planned to be written in MATLAB. Users therefore can only use the algorithms, with a MATLAB license including all of the incorporated toolboxes. There are only certain components of this project that will be available to the public. The first component is the actual audio signals that will be generated by the audification algorithm. The second component is the MIDI file generated by the sonification algorithm. This MIDI file could be distributed to the public for playback use in an audio synthesizer. As this project is purely for research purposes and not for economic gain, these components can be distributed to the public at no cost. The uses of these distributed components also connect to the social impact of this project.

The social aspect of this project is an area that is very important to consider for development. If ordinary people can identify the fluctuations in the generated audio signals, this project is extremely useful for education purposes as well as power grid station use. In a submitted ICAD paper regarding real-time data-agnostic sonification the authors discuss the effect of data sonification by stating, "All sonifications, however, may have musical effects on listeners, as our trained ears with daily exposure to music tend to naturally distinguish musical and non-musical sound relationships, such as harmony, rhythmic stability, or timbral balance"

[7]. This quote displays the effect that sonified data can have on the listener. Even without technical or musical knowledge any person can process musical components of audio. Humans naturally can interpret the musical changes discussed in the quote and this makes sonification very useful. Since the audification process is similar the aspects of this process are also related. Without any knowledge of the power grid parameters and it's distribution, the sonification and audification could be used to display the various fluctuations that occur over time. Along with educating the public the generated audio could also be used to educate engineering students in an intro to power systems class. In this class the theory of the power transmission could be introduced by the instructor and then audibly displayed by playing the audified/sonified data. This interesting introduction would allow for students to be more drawn to the subject and also have a better understanding of how the fluctuations occur over time. Power systems can be a very difficult concept to understand at first and this audible representation could greatly improve student understanding. Along with the social aspect, the political issues regarding this project must also be considered.

Political issues can arise since the Western Electricity Coordinating Council does want unauthorized grid data to be publicized. For this project historical data and data captured from a local AC socket will be used. The WECC data is proprietary data shared with me by Professor Dosiek. Since this data is proprietary I cannot distribute to the public. To care to this issue I had to make sure that the historical data that I used for processing was kept between my supervisor and I. Although this is true there is no concern towards the local grid data that I capture from an AC socket. The only components of the project that will be shared with the public are the generated audio signals and not the raw data measurements. The generated audio signals are just representations of the power grid data and could not be used for extraction of the processed grid

data.

Design Requirements:

When designing the algorithms for sonification and audification of power grid data there are various design specifications that needed to be considered. A propesed general block diagram for both approaches is shown below in Figure 4.

Figure 4: System General Block Diagram

For the implementation of this project the data must first be captured for processing of the algorithms. Various sources for data can be used for implementation of this project. Data can be captured from a local AC socket or historical data can be used. This data then needs to be processed by the sonification and audification algorithms for generation of audio. Sonification will generate parameters for the MIDI signal corresponding to the various data fluctuations and then generate a MIDI signal. The generated MIDI signal are then be played through a virtual audio synthesizer and played through a speaker. A MIDI signal is a Musical Integrated Digital interface signal that has main parameters of velocity, pitch, and note length. Each of these parameters can be assigned values ranging from 0 to 127. The pitch values correspond to the range of pitches that can be assigned to a MIDI note with 0 being C0 and 127 being G10. The velocity values correspond to the volume of each midi note with 0 being barely audible and 127 being loud. For the sonification approach, the velocity and frequency parameters will be

controlled by the power grid data. The note length was proposed to stay constant for all grid data points for consistency in the time that each note is played. For Audification, the algorithm processes the gird data and converts it into an audible signal. The resulting generated signal is then played through a speaker.

For sonification, the main sub functions are the pitch generation, velocity generation, MIDI file generation, and sound producing source component. The pitch generation needs to process the frequency data and accurately convert the each data into a MIDI pitch value. A range of pitch values needs to be determined for implementation of this sub function. The range needs to be large enough so that the data fluctuations are clear to the listener. The center of the range also needs to be at an appropriate pitch. Middle C was chosen s an appropriate pitch to represent the 60Hz frequency. Deviations above the 60Hz frequency have higher MIDI pitch values and frequencies below having lower pitch values. The number of octaves above/below this frequency pitch is also important for consideration of the range. The octave number above and below should be the same so the Middle C is the center point of the pitch values. A proper range for MIDI velocity generation sub function is also necessary for proper representation of voltage magnitude fluctuations. For both of these functions it is important to also consider how each MIDI value will be assigned. To assign MIDI values, the maximum and minimum fluctuations and their difference can be used to assign MIDI pitch value. With the max/min difference, the frequency ranges for each pitch can be determined. To accomplish this the difference can be divided by the number of pitch values in the set total fluctuation range. This sub-range can then be used for assignment of the midi parameter to each component of the processed data. If a parameter values falls into a certain sub-range a MIDI values corresponding to that sub-range will be assigned as the MIDI value for that parameter value.

For the MIDI file generation sub function there were not many design concerns. The MIDI file generation can be created using various types of pre-existing open source code. MIDI file generation is a complicated algorithm in itself and it would not be possible to implement the algorithm on top of the other components of the system on the short time spent on this project. For this reason open-source preexisting code is better for generation of MIDI files.

For the sound producing source sub function, selection of a device is important. A synthesizer is one of the best options for implementing a sonification algorithm, since synthesizers are extremely modular and can create very complicated sounds. Most synthesizers can be controlled by MIDI signals, which also make them very useful for sonification. The MIDI signals are inputted to the synthesizer and the contained MIDI data is played through the synthesizer device. A synthesizer composes sound by combining oscillating audio waveform signals. The most common types of oscillator signals are the square, saw tooth, triangle, and sine waveform. Saw tooth waves generate some of the most interesting synthesizer sounds and are the most common type of synthesizer oscillators for main melodies. After choosing an oscillator for the synthesizer, filtering and modulation of various parameters allow for more complicated signals. When considering the use of synthesizer, the type of synthesizer is also very important to consider. Synthesizers can be in virtual, analog, or digital-analog form. Virtual synthesizers are software-based synthesizers that generate the audio signals entirely on a computer. Virtual synthesizers have an interactive interface that allows for ease in audio synthesis as shown in Figure 5.

Figure 5: Sylenth 1 Virtual Synthesizer Plug-in Interface

The various components of synthesizer generation discussed before can be examined in this figure. Oscillator waveform, filtering, volume, and other components are all easy controlled by adjusting the parameters in the synthesizer interface. Virtual synthesizers are also much cheaper applications of audio generation then the other types of synthesizers since they are entirely software based. MIDI files can be uploaded and inputted directly to virtual synthesizers with any type of mixing software. Analog synthesizers differ in the fact that the generation is entirely through hardware with no software component. A digital-analog synthesizer is a combination of the virtual and analog synthesizer and is the most commonly used synthesizer in music performance today. The digital-analog synthesizer can also receive MIDI information but external MIDI interface devices are needed to input the signal into the synthesizer.

There are also various components that needed to be considered for audification. Audification is implemented by amplifying components of sinusoidal data to generate an audible signal. The data is usually sped up to higher frequency so that it as at an audible frequency. Amplification of amplitude can also be completed to make the variations in the data more noticeable. For audification of power grid data, the carrier frequency can be amplified so that the data is at a pitch corresponding to a music note pitch, and the fluctuations of frequency will

deviate from this pitch. A good pitch to center the frequency on would be middle C at 261.6 Hz. Middle C is a good center frequency since it is the middle of commonly used pitches for music.

In general, the length of the algorithms needs to be as concise as possible and properly commented. This is important for processing speed, since large data sets will be processed. If the algorithms were to be used by others, the users would need to be able to easily follow what the code is doing. The algorithms need to also be easy to implement by the user for audio generation. Testing of the various gain values applied for audification needs to be completed so that the generated audio fluctuations are noticeable.

The algorithms must be able to function on all frequency standards so that it could be used on any type of power grid. The U.S. uses a standard of 60 Hz while other countries use a standard of 50 Hz. The system must be able to process and perform the discussed methods on both types of carrier frequencies.

Design Alternatives:

The implementation of converting power grid parameter data could be performed in various ways. For sonification, MIDI parameters could be controlled by various components of the power grid data. Frequency was chosen for representing pitch, since the frequency deviations are very small and are very close in magnitude. This similarity in magnitude allows for better representation of the fluctuations as a note pitch. The frequency pitch could also easily be centered at a specific middle note that corresponds to the carrier frequency of 60 Hz. MIDI note velocity is the best option for voltage magnitude since the voltage magnitude spikes can be very large and dramatic. Velocity in music is often used to represent drama since it corresponds to the volume of the notes. High/low velocity allows for great dramatic effect in music and therefore is

much more useful to represent large voltage magnitude spikes. The sonification approach was the original plan of this project but after careful consideration and contact with the seismograph audification co-creator [5], sonification and audification were chosen for implementation. Discussion with Ryan McGee, the main creator of the Sounds of Seismic audification algorithm greatly helped distinguish the appropriate approaches. Sonification is very useful for representation of the fluctuation parameters over a long period of time. Audification can be extremely useful for examining forced power grid station events. The audio resulting from audification is a continuous audio signal without note interruptions while each successive note interrupts the continuity of the sonification audio signal. The sonification approach was originally going to use a microcontroller to process the data and generate midi signals. The generated signals were then going to be sent directly to a virtual synthesizer in a computer. This approach would have implemented an Arduino and an existing MIDI-USB shield shown below in Figure 6.

Figure 6: Open Pipe Midi USB Shield [8]

This Arduino shield allows for ease of generating audio signals using the Arduino and sending the signals through a USB to a computer [8]. This implementation would have involved unnecessary amounts of wiring and coding. The implementation would be much more efficiently conducted entirely on a computer using a programming language like MATLAB. After this and

other careful consideration it was determined that a software-based approach would be better since it would have little to no cost and implementation was much more useful. Various programming languages could be used for implementation of sonification and audification of the grid data but MATLAB was chosen due to personal familiarity. Python has various audio generating capabilities that would allow for implementation of this project but MATLAB has specific features that make it a better language for my project implementation. MATLAB also contains various toolboxes that are extremely useful for signal processing and generation of audio signals. The soundsc function was extremely influential in using MATLAB for audification since it can be used to generate audio signals from pre-existing sinusoidal waveforms [9]. The function scales the inputted waveform from -1 to 1 and allows for audible output of the data. This function was suggested for use by Ryan McGee and after further research was deemed to be extremely useful for audification. Examination of the audification outputs from McGee's sound of seismic further supported the necessity of using MATLAB and the soundsc function. McGee generated all of the audio samples used in his recording with the soundsc function [5].

Preliminary Proposed Design:

For implementation of the sonification and audification approach a design approach was implemented for both algorithms. For both algorithms, the first part of the design is the capture and upload of the power grid data into MATLAB. Professor Dosiek shared the national grid historical data for which I upload into MATLAB using the code shown below in figure 7.

```
fs = 60;
Ts = 1/fs;
V = InData(11).Data;
N = length(V);
tPMU = [0:Ts:(N-1)*Ts]';


Vmag = abs(V); %mag in kV RMS
Vangle = unwrap(angle(V)); %angle in rads
Vfreq = [0; diff(Vangle)/Ts/2/pi] + 60; %Frequency in Hz
```

Figure 7: Historical Data Parameter Extraction Example MATLAB

The code in the previous figure uploads the historic data, and creates an array of voltage magnitude values and frequency values. The voltage magnitude is found by taking the absolute value of the historic signal data. The phase angle of the signal is captured by using the unwrap MATLAB function. As discussed before, the derivative of the phase angle is the frequency deviation. This deviation is in radians, so to calculate in Hz the values are divided by 2π. Adding 60 to this calculated deviation then results in a historic data frequency array.

The other data source for processing and analysis is the data that is captured from a Local AC socket. To capture this data, a system designed over the 2016 summer by Professor Dosiek and his research student Pranav Shrestha will be used shown below in Figure 8.

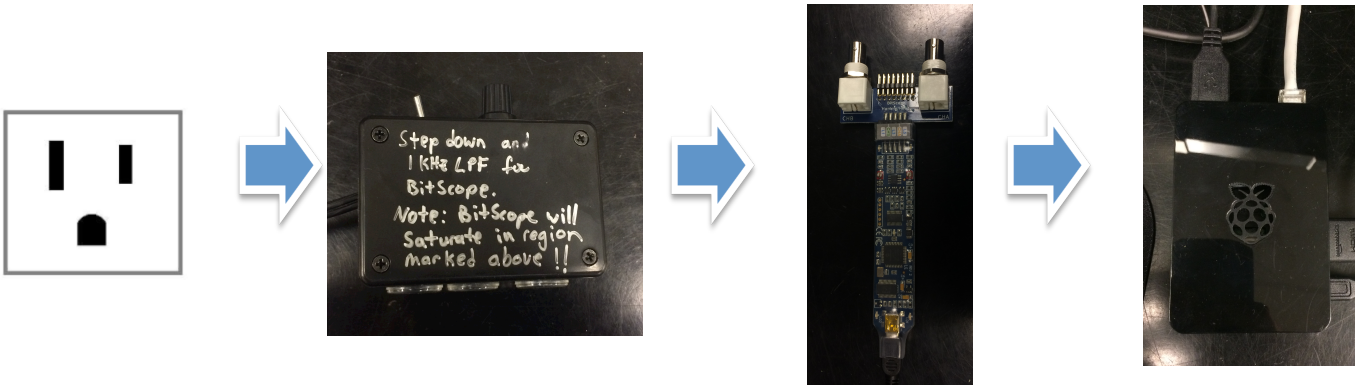


Figure 8: Synchrophasor Measurement System

Data from an AC socket can be gathered over a period of time using this system. The voltage signal from the AC socket is stepped down to around 3V and filtered by a 1 kHz low-pass filter. The stepped down signal is then measured using a Bitscope I/O measurement device and uploaded to a raspberry pi computer. The python code to initiate data recording and capture is shown in appendix A. After the code processes and records the data text files containing the Voltage magnitude, and Frequency are saved as text files. These text files are then uploaded to MATLAB and plotted using the code shown below in Figure 9.

```
Vfreq2= importdata('/Volumes/WAR MACHINE/Vfreq2.txt');
Vmag2 = importdata('/Volumes/WAR MACHINE/Vmag2.txt');

Nfreq = length(Vfreq2);
tEnd = Nfreq/60.0;
s = 1/60.0;
t = 0:(s):tEnd-s;

plot(t,Vfreq2)
xlabel('Time')
ylabel('Frequency')
figure

t2 = 0:(s):tEnd;
plot(t2,Vmag2)
xlabel('Time')
ylabel('Voltage')
```

Figure 9: Text File Data Parameter Extraction Example MATLAB

Two different time vectors are used for plotting the data because a sample is lost when calculating the frequency deviation. The acquired data is then filtered using a 2 Hz low-pass filter for a smoother signal. The resulting plots are shown below in Figures 10 and 11.

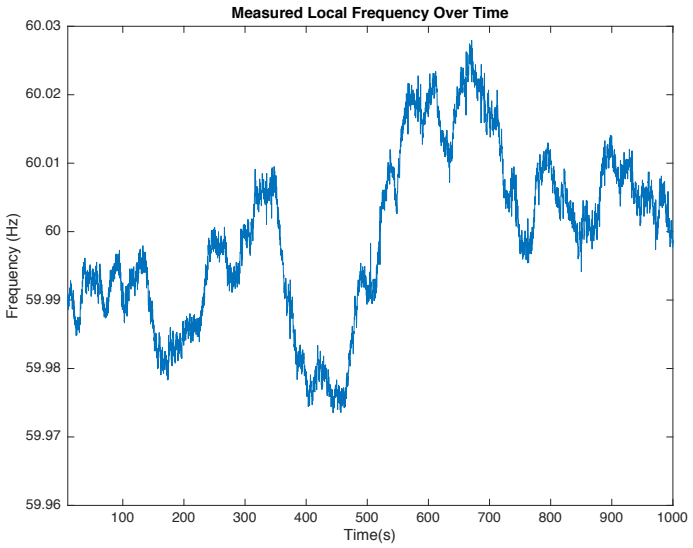


Figure 10: Synchrophasor Measurement Device Measured Local Frequency

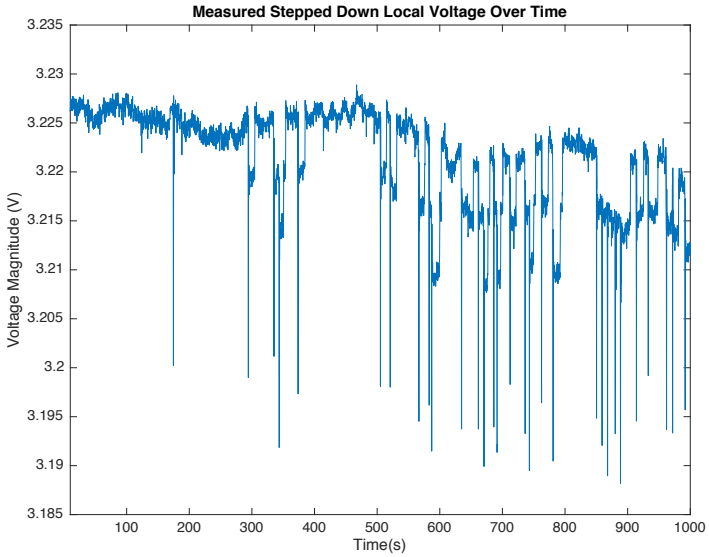


Figure 11: Synchrophasor Measurement Device Measured Stepped Down Local Voltage

When examining these figures, the voltage magnitude and frequency fluctuations over time can be seen clearly. These fluctuations are of very small magnitude and if the ac signal was played

back in MATLAB the fluctuations would not be heard. The frequency fluctuations are in mHz magnitude and voltage magnitude is in mV magnitude. Amplification of these fluctuations is necessary in order to properly hear the fluctuations of these parameters.

For the design of sonification, the functions shown in Appendix B and Appendix C were proposed for generation of the MIDI values. Both functions take the corresponding data parameter array as an input and generate the MIDI pitch/velocity values for each sample within the inputted array. For velocity assignment all of the possible velocity values are used so low voltages will be very quiet and high voltages will be loud. Both functions work by creating an empty array with the same length as the inputted data array. The code then runs through each sample and assigns a MIDI pitch/velocity value corresponding to the signal parameter value. For pitch range, two octaves above and below middle C were chosen. For the frequency assignment, the function assigns the middle C MIDI value to any sample that is exactly 60 Hz. The parameters generated from these functions will then been used as inputs for pre-existing MIDI file generation code. I found code that creates functions for generation of a MIDI file in python for which I plan to edit and convert to MATLAB syntax. When the final MIDI generation code is created, the MIDI values will be used with the code to create a MIDI file. The generated MIDI file will then be uploaded to a Digital Audio Workstation called Ableton Live so that the file can be played back in audio synthesizer within the software. The generated audio signal can then be extracted as an audio sample for portable playback of the audio signal.

For the audification approach, a block diagram of the preliminary design is shown in Figure 12.

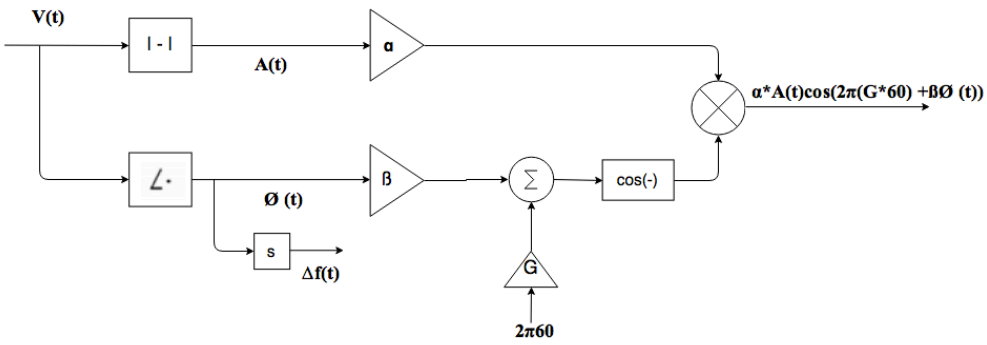


Figure 12: Preliminary Audification Block Diagram

As seen in the figure, the main approach of this algorithm is to analyze each component of the signal and apply gains to each parameter. Using data gathered from the historical data or from the local measurement device, each component of the signal is extracted for amplification. Each parameter is then amplified by a gain specific to that parameter. Testing was proposed for the proper value of amplification for each parameter. The new parameters are then used to create a new representation of the signal as a cosine signal with parameter amplification. The resulting signal is then be played using the soundsc function in MATLAB for testing of the audio output. For the created signal Voltage magnitude will be represented as the volume of the sinusoidal signal. The gain for the frequency parameter was proposed to center the audio signal at a frequency of 60 Hz with increases/decreases in frequency corresponding to the deviation from this frequency.

After implementation of the algorithms, testing of the audio signal accuracy in representing the fluctuations is important. I had goals successfully implement fully running drafts both algorithms along with test samples for the student survey. I also hoped to incorporate students of all disciplines including musicians to gather feedback for the generated audio samples.

Through the process of generation and implementation of these algorithms there are

various roadblocks that I predicted. The audification audio may only be of pure-tone and not interesting for auditory display. Small fluctuations in parameters may also not be noticeable for the generated audio signal. For sonification, the musical dissonance may be very annoying between each note and not musical interesting. The main goal of this project is to tend to these roadblocks by making adjustments in code or general process for the audio generation.

Final Design And Implementation:

Audification:

For audification, the process was completely written in MATLAB. A diagram of the audification workflow is shown below in Figure 13.

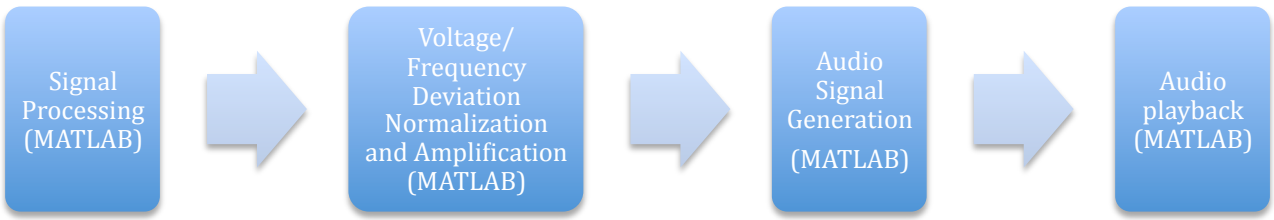


Figure 13: Audification Workflow Diagram

The first part of this process is the signal processing of the power grid data. The voltage magnitude and frequency data are filtered with a 2 Hz low pass filter to remove noise from the signal. The data acquisition process results in amplification of signal noise, so filtering is necessary for accurate data. After the data is filtered, the transient response of the filter has to be removed from the filtered data. For this process I used a 6^{th} order Butterworth filter that had a filter transient of about 120 samples. The filter transients need to be removed, so the first 120 samples of the filtered frequency data are removed. When the frequency of power grid signal is derived using differentiation of the signal angle, the first sample of the signal is lost. Because of

this, the first 121 samples of the filtered voltage magnitude data are removed. This allows for the voltage magnitude and frequency parameter data to correctly be synchronized with each other. After the data is filtered, an interesting period of the data has to be selected for audification. The frequency and voltage magnitude signals are plotted over time and the fluctuations are analyzed. Once an interesting period of data is determined, new frequency and voltage magnitude data arrays are created that contain the interesting period data. This interesting period array is the final signal data that will be converted to audio. Subtracting 60 from the frequency array creates the frequency deviation array. The voltage magnitude array is then normalized from values of 0 to 1 so that the resulting audio file does not have audio clipping. To normalize the voltage magnitude data, the following equation is used:

$$VmagNorm(i) = \frac{Vmag(i) - min(Vmag)}{max(Vmag) - min(Vmag)} \tag{2}$$

This equation takes a voltage magnitude sample, subtracts the minimum magnitude sample value, and divides this difference by the range of the voltage magnitude data array. A MATLAB function, “Normalize”, was written to implement this normalization of an entire voltage magnitude array as shown in Appendix E. This function takes a voltage magnitude array as an input, and first calculates the maximum and minimum voltage magnitude. The function then iterates through the entire inputted data array, and using equation 2, the function normalizes each sample value.

After normalization of voltage magnitude, the frequency deviation array is also normalized. The frequency deviation array is normalized from values of -1 to 1 and amplified since the deviations above and below 60 Hz are very small. To complete the normalization of the

frequency deviations, the following equation is used:

$$\Delta FreqNorm(i) = 2 * \frac{\Delta Freq(i) - min(\Delta Freq)}{max(\Delta Freq) - min(\Delta Freq)} - 1 \quad (3)$$

This equation takes a frequency fluctuation sample, completing a similar process as equation 2. The process differs in the fact that the equation multiplies the 0 to 1 normalized value by 2 and subtracts 1 for a normalized value from -1 to 1. A MATLAB function, “Normalize_1”, was written to implement this normalization of an entire frequency deviation array as shown in Appendix E.

Once the frequency deviations and voltage magnitude are normalized, the audio signal generation step is completed. The new normalized data arrays are up-sampled to the common audio sampling rate of 44100 using the resample function. This function takes an array and two values as inputs. The function re-samples the inputted array to a new rate equal to the original sampling rate multiplied by the ratio of the two inputted values [10]. If a data array had an original sampling rate of 60 Hz, the resample function would be used with command “resample(data,44100,60)” to convert the data array to a 44100 sampling rate. The use of this function can be seen in the audification code example shown in Appendix D. After up-sampling the data, the frequency deviations need to be integrated to determine the audio signal phase shift array. To do this the following equation is used:

$$\Delta\theta(i) = 2\pi * T_s \Delta FreqNorm(i) + \theta(i-1) \quad (4)$$

In this equation Ts is the sampling period of the data. The equation allows for numerical

integration of the entire frequency deviation data array into the resulting audio signal phase shift. A MATLAB function, “integrateFreqDevInteresting”, was written to implement this normalization of an entire frequency deviation array as shown in Appendix F. After creating the new data array for the audio signal phase, the audio signal is creating by using equation 1 and amplifying the carrier frequency to 261.6 and the signal phase by a tunable gain G as shown below in equation 5.

$$Vaudio(t) = \sqrt{2}VmagNorm(t)cos((2\pi * 261.6))t + G\Delta\theta(t)) \quad (5)$$

This equation yields an audified signal of the power grid data. This signal is then played back in MATLAB using the sound function to determine if the fluctuations of frequency can be heard. The user then changes the phase gain G until the fluctuations of frequency are heard. An example of the implementation of the audification process is shown in Appendix D.

Sonification:

For sonification, the process was conducted in MATLAB and python. A diagram of the sonification workflow is shown below in Figure 14.

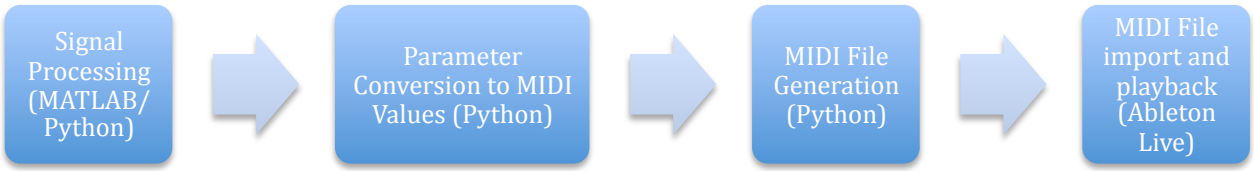


Figure 14: Sonification Workflow Diagram

The first part of this process is the signal processing of the power grid data. The same procedure is taken as for audification to find an interesting period of data. Once an interesting period of

data is found, the data is imported to python. After import to python, the frequency and voltage magnitude data are filtered and the transient's are removed from the signal. New arrays are then created that contain the data samples corresponding to the interesting time period.

After creating the new interesting voltage magnitude and frequency arrays, the parameters are converted to MIDI values. The frequency data is converted to an array of MIDI pitches and the voltage magnitude is converted to MIDI velocities (volumes). For conversion from voltage magnitude to velocity, all possible velocity values from 0 to 127 are used. To convert the values the function "midiVelocityGen" is used and is shown in Appendix G. This function works by first calculating the range of the inputted voltage magnitude data by subtracting the maximum and minimum voltage magnitudes. This range is divided by 128, since there are 128 possible velocity values (0-127) to calculate the step range vStep. To assign a velocity value, a 'while' loop is used where the minimum magnitude voltage is increased by vStep until the voltage magnitude value falls within a range between the new minimum magnitude and the new minimum magnitude plus vStep. Every time the 'while' loop iterates, the velocity value increases by 1 and the new velocity corresponds to the new range of voltage magnitudes. Once a corresponding velocity value is determined, the value is appended to a new velocity array. Once each voltage magnitude is assigned a velocity value, the final velocity array is returned by the function.

The process of converting frequency values to corresponding pitches was similar to the voltage conversion process but musical key can be specified. The function midiPitchGen is used for the conversion and is shown in Appendix G. This function takes 3 parameters: a frequency array, the musical key as a string, and an integer number of octaves. The number of octaves parameter specifies how many octaves above/below pitch 60 (middle C) the possible MIDI

pitches will be. The musical key parameter can be any of the 12 musical keys. If the key string input is not a musical key then the generated pitches will be chromatic. To create an array of the frequency MIDI pitches, the notes of that key are first determined. If the key is not specified then the note values are all integers between 0 and 11 and an array containing these values is created. For a musical key, the possible note values are 0, 2, 4, 5, 7, 9, and 11 with 0 being the first note of the scale and 11 being the last. An array containing these note values is created and adjusted based on the musical key as shown by the "generateKey" function in Appendix G. This function adjusts the note value array by the key index where the key of C is index 0. The possible keys and their key index are shown in Appendix G by the function "createKeys". This function creates a dictionary of all of the possible keys with their corresponding adjustment value or "key index". There are 11 keys above middle C and the notes array is adjusted by the value of the musical key. Once the key is determined, the lowest pitch is found by subtracting the 12 times the number of octaves using the "findLowestNote" function shown in Appendix G. The number 12 is used since there are 12 notes in one octave. Using the "generateNotes" function shown in Appendix G, final note pitches array is created. This function takes an array of musical keynotes, a number of octaves above/below integer, and a lowest note integer as parameters. This function determines the total number of octaves above the lowest note by multiplying the number of octaves above and below the parameter by 2. The function then uses a 'for' loop and generates an array containing the notes of the key above the lowest note, spanned over the total number of octaves. This final notes array is then used in the "createPitches" function shown in Appendix G. This function completes a similar process to the "midiVelocityGen" function where each pitch value corresponds to a range of frequency values. The function generates a fStep value by dividing the range of the frequency values by the total number of notes. The function then uses a

'while' loop to assign a frequency value a corresponding pitch. This corresponding pitch is contained in the final notes array where the smallest frequency is assigned the lowest note pitch and the largest frequency is assigned the highest note pitch.

Before creating the MIDI file the note length of each sample must also be determined. The note length can be set to any value in beats. To playback the data in real time the following equation is used to determine the set note length:

$$Notelength = \frac{Tempo(BPM)}{SamplingRate(Hz) * \frac{60seconds}{minute}} \quad (6)$$

In this equation the tempo is in beats per minute, and the sampling rate is in Hz. The resulting note length value is a magnitude with no units. To further improve the musical quality of the sonification, the note length can be randomized using the python choice method. This method chooses a value from an array randomly.

Following the creation of the MIDI values, the MIDI file is then created using the code shown in Figure 15.

```
#midi parameter values
track    = 0
channel  = 0
time     = 0   # In beats
noteLengths = [1/4.0,1/2.0,1]   # In beats, array of possible values for randomized note
length
tempo    = 120  # In BPM
MyMIDI = MIDIFile(1) # One track
MyMIDI.addTempo(track, time, tempo)
for i in range(len(vfreqInteresting)):
    duration = choice(noteLengths)
    MyMIDI.addNote(track, channel, pitch[i], time, duration, velocity[i])
    time = time + duration
with open("AmbientDataBbRandomLength.mid", "wb") as output_file:
    MyMIDI.writeFile(output_file)
```

Figure 15: MIDI File Generation Python Code

The example in Figure 15 shows MIDI file generation with randomized note length. The entire process of MIDI file generation from the power grid data with randomized note length is fully shown in the example shown Appendix G.

After the MIDI file is created, the file is then dragged from the computer document library to a MIDI channel in Ableton live for playback. The virtual synthesizer Massive is then opened in the MIDI channel. In Massive, a synthesizer sound pre-set can be used or a unique sound can be created. Once the synthesizer sound is chosen, the MIDI file is then played through the synthesizer by pressing the play button within the Ableton workstation.

Performance Estimates And Results:

Before the implementation of this project there were various goals of the project. The first goal was to generate clear representations of different types of system responses. These responses include transient, ambient, and forced oscillation responses. The second goal was to generate easy to use algorithms. The third goal of this project was to incorporate musical aspect to sonification approach for more pleasing playback. The final goal was to determine possible use in power systems introduction and for power grid operators.

Through the implementation of this project the goals were all successfully completed. The generated algorithms generated are easy to understand and use. They are well commented and the user only needs to adjust a few lines of the script code for proper audification and sonification. The sonification and audification algorithms were able to represent the discussed types of system response. I was able to successfully audify ambient and forced oscillation data. The process of parameter normalization and up sampling for ambient data can be seen in figures 16 and 17. These plots show the ambient data frequency fluctuations and voltage magnitude

before and after normalization.

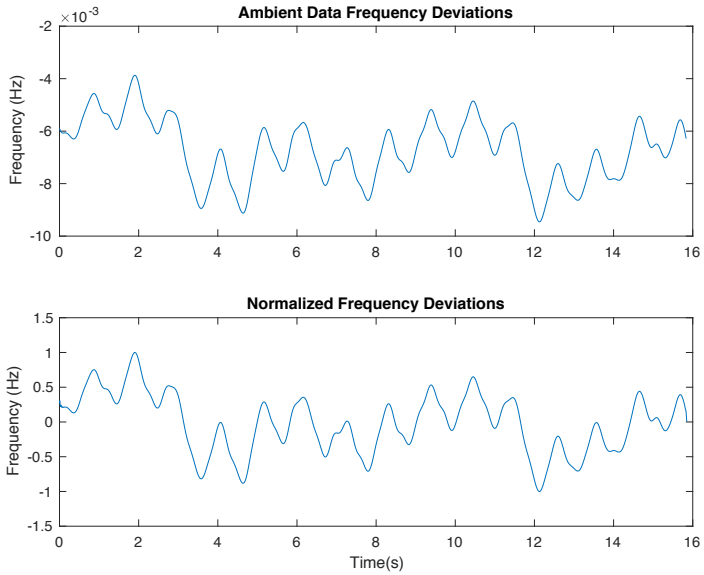


Figure 16: Ambient Data Frequency Deviations Normalization

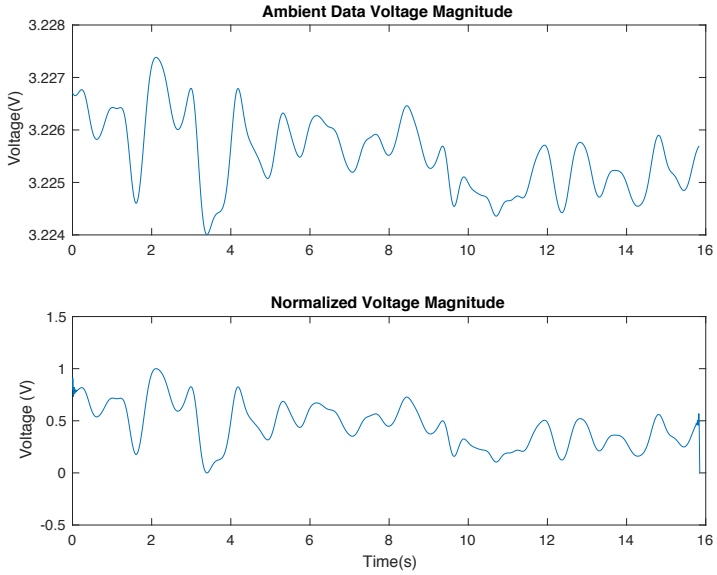


Figure 17: Ambient Data Voltage Magnitude Normalization

In figure 16 it is clear that the data was properly normalized since the final frequency deviations span a range of -1 to 1. The normalization is also clear in Figure 17 since the final voltage magnitude spans a range of 0 to 1. As described previously, integration of the normalized frequency deviations results in the final phase angle, which can be seen in Figure 18.

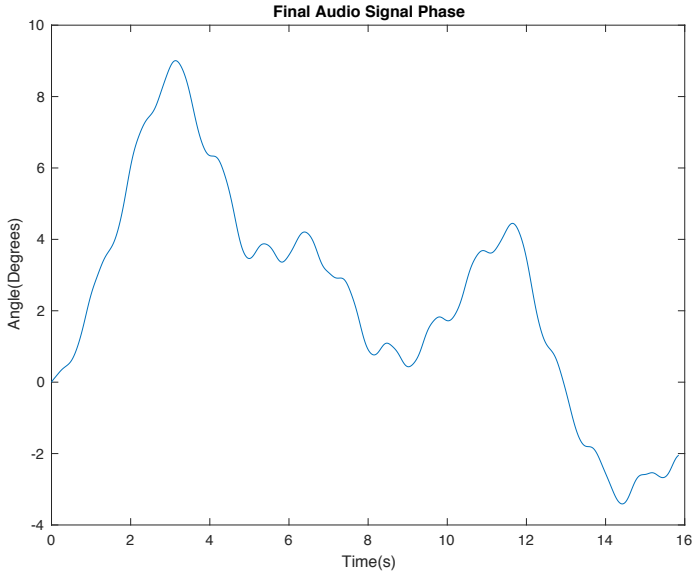


Figure 18: Ambient Data Signal Phase

After amplification of this final phase, the audio signal is built using the new parameters in equation 5. The power grid AC signal before and after audification is shown below in figure 19.

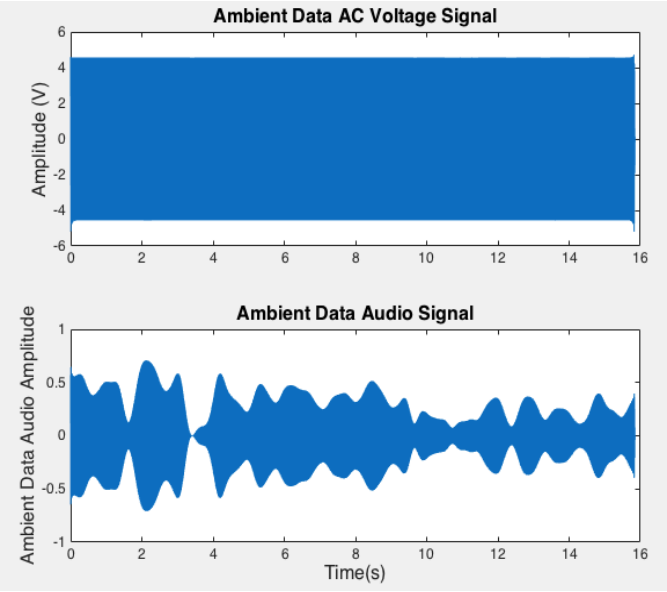


Figure 19: Ambient Data Signal Before And After Audification

When examining this figure the voltage control over volume can clearly be seen. The voltage magnitude plot in figure 16 serves as the volume envelope of the audio signal shown in figure 19. The fluctuations of frequency cannot be seen since the frequency is high, but can be heard

with audio playback of the signal. Audification was also successfully completed on 3 different magnitudes of forced oscillations. These oscillations were simulated power grid data were a square wave was applied to simulated ambient data. This data was for analysis of the frequency fluctuations with the voltage remaining constant. The oscillations were of small, medium, and large magnitude. These oscillations were normalized to each other for playback and determination if the oscillations could be heard in each magnitude. The normalized frequency oscillations are shown below in figure 20.

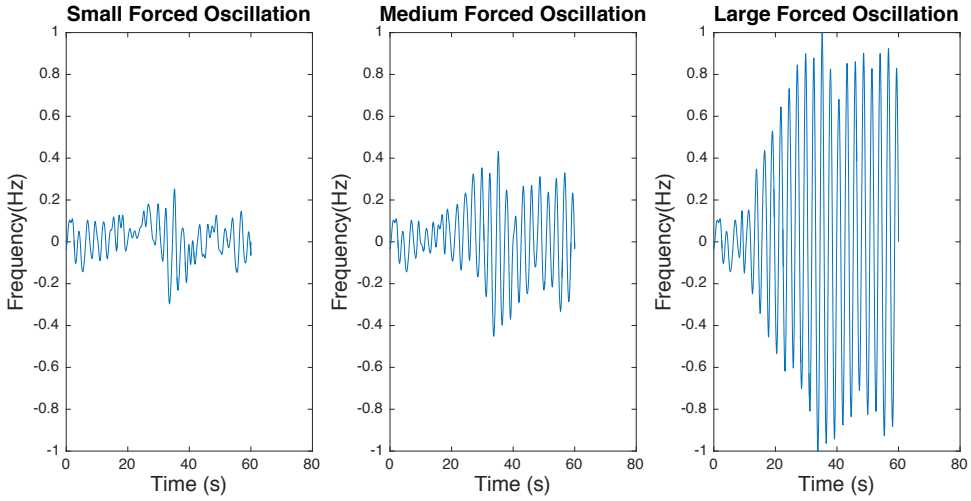


Figure 20: Forced Oscillation Normalized Frequencies

As seen in the figure, the small forced oscillations were properly normalized to one another. The large oscillation frequencies span values of -1 to 1 and the medium and small oscillations span negative to positive values that are a smaller factor than the large oscillations. These normalized oscillations were later used to rebuild audified representation of each magnitude forced oscillation.

Sonification of various system responses was also very successful and the algorithms allow for more pleasing musical playback. Transient response data was sonified in real time as

shown in figures 21 and 22.

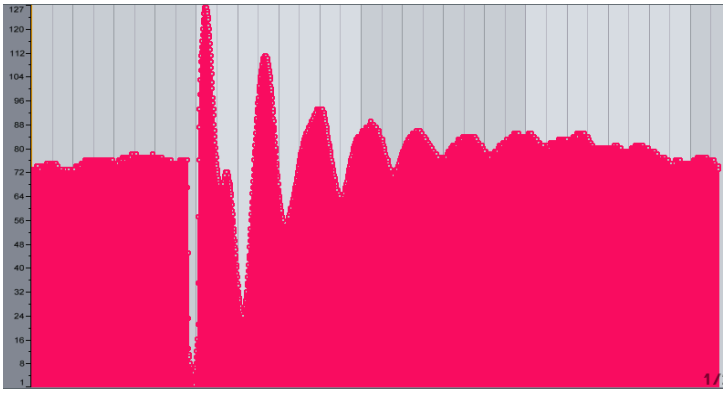
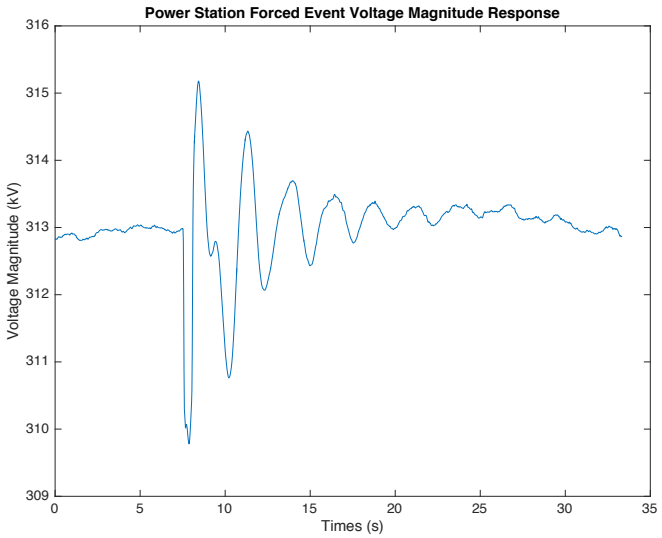


Figure 21a: Note Velocity Mapping Figure



21b: Transient Response Voltage Magnitude

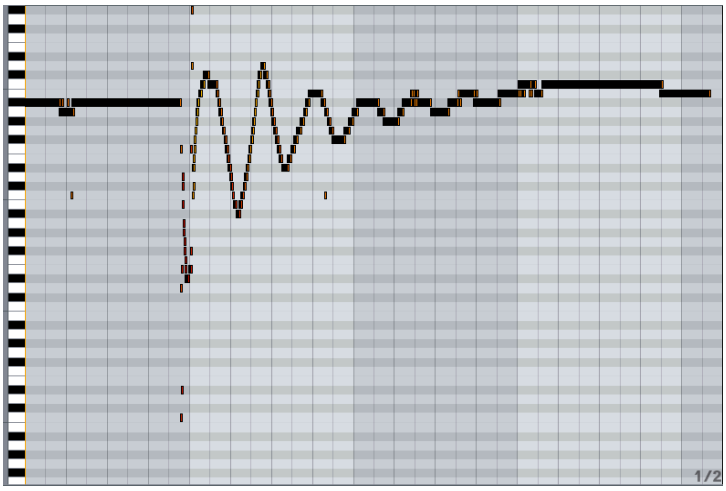
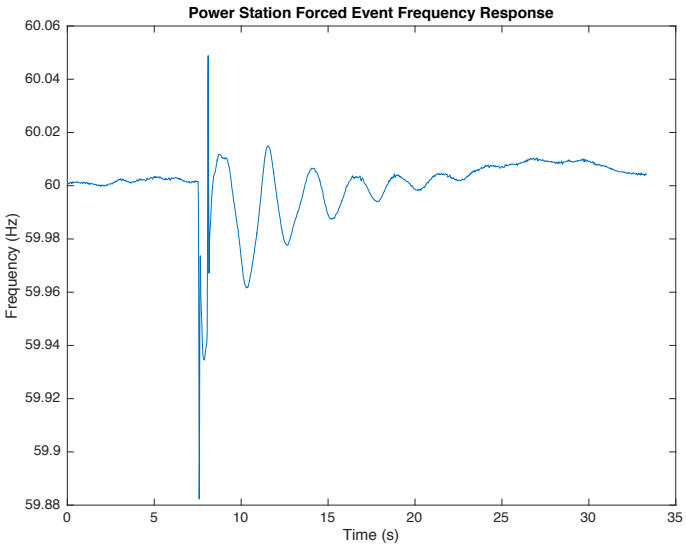


Figure 22a: Note Pitch Mapping



22b: Transient Response Frequency

These figures show how the transient response data was clearly mapped with the MIDI pitch and velocity parameters. The voltage magnitude curve can be matched with the velocity values and the same curve line up can be matched for the frequency to pitch notes. The sonification algorithms were also successful in mapping the simulated forced oscillations as varying note pitches in the synthesizer playback.

The goal of having musically pleasing sonification playback was also completed. The sonification algorithm was able to generate playback of ambient data in the key of Bb with randomized note length. This allowed for the playback to sound better and was more interesting since the note lengths varied and were not consistent. The example showing the randomized note length in the key of Bb is shown in Appendix G. The resulting MIDI mapping can be seen below in figure 23.

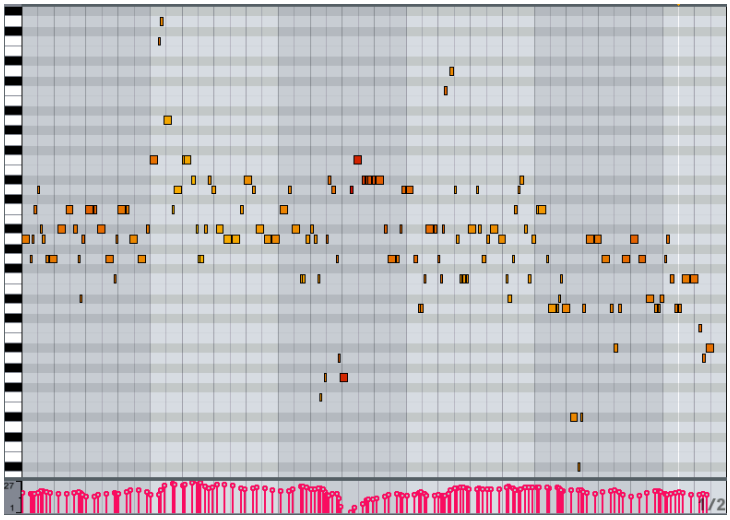


Figure 23: Ambient Data Sonification With Randomized Note Length in Bb key

To determine if the final goal of use for power system analysis/education a survey was conducted with 15 union college professors and students. In the survey the participants listened to various audio samples generated from the audification and sonification algorithms and they were asked to answer questions. The first part of the survey had participants listen to audified forced oscillations and ambient data. There were 3 samples that contained forced oscillations of different magnitudes that started at the same time. A fourth sample in this section was ambient audified data. The participants were asked to comment on if they heard an oscillation in each of the four samples and when they thought the oscillation occurred. A table showing the results of this section of the survey is shown below in table 1.

Response Type	Detection Rate	Est. Start Time (seconds)
Ambient	**20%**	**21± 12**
Small Forced	**73.33%**	**30 ± 10**
Medium Forced	**93.33%**	**25 ± 5**
Large Forced	**100%**	**12 ± 2**

Table 1: Union Student/Faculty Survey Results

When examining the table it can be seen that most participants were able to detect force oscillations in the small, medium, and large forced oscillation audified data. The fluctuations in the small forced oscillations are small, so if the ambient data detection rate were closer in value to the small forced data, then this data would not be useful. This would mean that the participants were mistaking ambient fluctuations for forced oscillations. These small forced oscillations cannot be detected by the algorithms used by power grid operators so since participants could hear the oscillations, there is possible use of power grid audification in power system analysis.

The second part of the survey had participants listen to sonification of transient and ambient data. The transient data sonification sample was in real time and the ambient data sonification contained chromatic, musical key, and random note playback samples. When listening to sonification tracks, many participants mentioned how sonification allowed for clear representation of the different types of power system responses. For sonification, nearly all participants also preferred specified key and random note length. This playback was more musically pleasing to the participants and helped prove that musically pleasing aspect of sonification was achieved.

After completion of the survey further experimentation multi-channel sonification was conducted. For multi-channel sonification, two synchronized power stations were used for

musical playback. Each station data was sonified individually and placed as a different instrument and different note lengths. One channel contained data that skipped every fourth sampling and played half notes that outlined the chords of the song. The other channel contained data that played eighth notes and outlined the melody of the song. There are four eighth notes in a half note, so the power stations are still synchronized with one another in the resulting. Some rhythm instruments were added for musical effect and the resulting music was very interesting and pleasing.

Production Schedule:

The design and implementation of this project was completed over three terms at Union. In the spring, the project was chosen and the engineering design process was studied. A project specifications write up was completed at the end of the spring along with a project proposal presentation. In the original proposal, an Arduino microcontroller with a MIDI-USB shield was proposed to convert the frequency and voltage magnitude fluctuations into MIDI parameters. These parameters would then be sent from the microcontroller within a MIDI signal to a synthesizer on a computer. After further research in the fall term, I learned of the audification and sonification process and changed the project to complete these processes with power grid data. The original proposed design only included sonification but the audification process seemed more useful for power system analysis and education. The sonification process was more for a musical representation of the data. After finding the MIDI-Util open source MIDI file generation library we decided to implement both processes in software. The MIDI file generated from the UTIL library could be placed in audio software for playback, so the microcontroller was no longer needed. The remainder of the fall term was spent acquiring ambient data from the

power systems lab and generating the block diagrams and process of conducting audification and sonification.

During the winter term the sonification and audification algorithms were further designed and written. We determined that frequency and voltage magnitude data needed to be normalized for audification since the fluctuations were very small. The normalized data then needed to be up sampled to 44100 Hz the sampling rate of most audio signals. The final audio signal was to have a new centered frequency of middle C (261.6 Hz) and the normalized frequency fluctuations were amplified for clearer representation in the audio playback. After determining these constraints, the audification script was written in MATLAB and tested with various sets of power grid data. When the audification script was complete, I began the implementation of the sonification algorithm in python. To make the sonification notes more pleasing, I spent a long time computing the logic for musical key notes instead of just chromatic. I also derived an equation to allow for the generated MIDI notes to be played back in real time of the original signal. Once musical key and note length, was derived and coded, the algorithm was tested for functionality on various power grid data sets. Once I knew both algorithms were working properly, I generated final audio pieces that were used for the survey conducted on the Union campus. After completing the survey I began working on conducting sonification on synchronized power stations and playing back the generated MIDI files together as different instruments to create a song.

The only recommendations I have for the design process that I took, was to have completed more research before the fall term. I did not know of sonification and audification until the fall term. If I had researched these processes before the fall term, I would have been more organized with my project design. This would have allowed for my productivity in design

to be faster and I may have had more time to improve my overall design and implementation.

Cost Analysis:

For the implementation of this project, there were no costs. I used software that I already owned or that was provided by Union College. If I did not own the software used in this project, then the cost of the project would have been high. A table displaying the costs of the software used is shown below.

Component	**Cost**
MATLAB Software Personal License	$149.00
Signal Processing Toolbox	$45.00
Python	$0.00
Ableton Live Intro Version	$99.00
Massive Audio Synthesizer Plug-in	$149.00
Total Cost:	$442.00

Table 2: Project Software Costs

The total cost adds to a high value of $442.00, but if the software used was not already available, the signal processing processes for sonification and audification could be completed in GNU octave. Octave is a free scientific programming language that has similar built-in functions and capabilities as MATLAB [11]. To playback the sonification generated MIDI file, I could have used free audio workstations like MuLab and Ardour [12]. These workstations are similar to Ableton, allowing MIDI files to be dropped into a MIDI channel for playback. These workstations also contain various preset instruments for playback of the MIDI file.

User's Manual:

Audification Manual:

To perform audification of power grid data the user can edit the example script shown in Appendix D. First, the parameter data import lines need to be changed so that the file path corresponds to the data that the user wants to complete audification on. After uploading, the user then needs to plot the filtered frequency and voltage magnitude plots to determine an interesting period of data. Once an interesting period is found, the indexes for creating "freqDev" and "VmagInteresting" should be set to the indexes corresponding to the beginning index and end index of the determined interesting data period. The user then can run the entire script to generate an audified version of the data. The sound function can be used to playback the resulting audio signal in MATLAB. The phase shift gain G can be adjusted to amplify or decrease the fluctuations in order to hear the deviations better. Once the audio signal is satisfactory to the user, they can export the signal as a wav file using the audiowrite function in MATLAB.

Sonification Manual:

To perform sonification of power grid data the user can edit the example code shown in Appendix G. First, the parameter data import lines need to be changed so that the file path corresponds to the data that the user wants to complete sonification on. The interesting period indexes need to be changed to correspond to the data range that the user wants to sonify. The user then can run the entire script to generate a sonified version of the data as a MIDI File. The MIDI file will appear in the same folder that the python script is located. The user then needs to drag the MIDI file into a MIDI channel on an audio workstation like Ableton live. In this MIDI channel, a synthesizer instrument must be opened for playback of the file. Once the MIDI

channel and instrument are configured, the MIDI file can then be played back to hear the sonified data.

Discussion, Conclusions, And Recommendations:

The main goal of this project was to create algorithms for converting power grid data into audio. The purpose of converting the data into audio was to accurately represent the fluctuations of voltage magnitude and frequency that occur in power systems. In completion of this project I hoped to incorporate a musical aspect to sonification for musical playback, determine possible use in power grid system analysis/education, generate easy to use algorithms, and accurately represent different system responses with the converted audio. The two processes that were designed to convert the data into audio were audification and sonification. Audification allowed for conversion of the signal directly into an audible signal while sonification used MIDI parameters to map out the parameter fluctuations in the data as changes in note volume and pitch. The sonification approach was completed through algorithms written in python while the process of audification was written in MATLAB.

The overall performance of the algorithms is very satisfactory. The main goals of this project were all reached. Both algorithms can be used for conversion of any types of power grid data and can accurately represent transient, ambient, and forced oscillation data. These algorithms are very easy to use and small changes in the code allows for audification and sonification of any data. I was successful in incorporating a musical aspect to the sonification algorithms and the final algorithms can generate very interesting melodies within specific musical keys. Participants in my survey found the sonification of data to be musically pleasing and some thought it better helped understand the fluctuations that were occurring in the data. The

audification process has possible use in power systems analysis/education. The audification of data allows for clearer understanding of the fluctuations of the system and participants noticed this when completing my survey.

Future work can be completed on this project to further improve its functionality and use. The first important step would be to increase the survey sample size and have much more participants. This would allow for more accurate results of forced oscillation detection. With more accurate results, determination of audification and sonification use in power systems analysis/education could be completed. Another step would be to determine if minute forced oscillations that are not detectable in time domain analysis could be heard using the audification. If the oscillations are audible with audification, research could be completed to determine why they are audible, and how this can be used for detection algorithm's improvement.

The experience of completing this project was extremely helpful in developing my design skills. I spent almost a full year discussing the design and implementation of this project. There were many difficult blocks that were encountered during this project, but these blocks helped me learn how to better approach the design process. The process was difficult but I am very happy that I completed the work. My final product is much more fascinating than what I expected to design and I plan to keep working on my algorithms after finishing the requirements for this capstone senior project.

References:

[1] Reynaldo Francisco Nuqui, “State Estimation and Voltage Security Monitoring Using Synchronized Phasor Measurements,” Ph.D. dissertation, Electrical Engineering Department, Virginia Polytechnic Institute and State University, July 2, 2001

[2] Dawn Santoianni, Dawn. "The Backbone of the Electric System: A Legacy of Coal and the Challenge of Renewables." Scientific American. N.p., 17 May 12. Web. Nov. 2016.

[3] Jim Follum ,Frank, Tuffner “Power Systems Oscillatory Behaviors: Sources Characteristics Analysis Draft”. Novemebr 28 , 2016.

[4] Luke Dosiek, “Extracting Electrical Network Frequency From Digital Recordings Using Frequency Demodulation,” IEEE SIGNAL PROCESSING LETTERS, VOL. 22, NO. 6, JUNE 2015, pp. 691-695.

[5] Sounds Of Seismic, http://sos.allshookup.org/, 2013

[6] David Worrall, “Real-time Sonification And Visualization Of Network Metadata,” The 21st International Conference Of Auditory Display, Graz, Austria, July 8–10, 2015, pp. 337-339.

[7] Jason Freeman, Lee W. Lerner, and Takahiko Tsuchiya “Data-To-Music API: Real-Time Data-Agnostic Sonification With Musical Structure Models,” The 21st International Conference Of Auditory Display, Graz, Austria, July 8–10, 2015, pp. 244-251.

[8] Openpipe Midi-usb-shield, http://openpipe.cc/product/midi-usb-shield/, 2016.

[9] Matlab soundsc, https://www.mathworks.com/help/matlab/ref/soundsc.html, 2016.

[10] Matlab resample, https://www.mathworks.com/help/signal/ug/resampling.html, 2017.

[11] "GNU Octave." GNU Octave, https://www.gnu.org/software/octave/, 2017.

[12]"The 5 Best Freeware DAWs." MusicTech.net. MusicTech Magazine, 20 Oct. 2015. Web.

Appendix A: Bit Scope Data Capture Python Code

Based on code found at https://bitbucket.org/snippets/prollings/8nbn

```
Luke Dosiek 2016
Edited By Patrick Cowden Fall 2016
"""

import serial
import time
import threading
import struct
from collections import deque
import numpy as np

running = True

""" BS """
ser = serial.Serial("/dev/ttyUSB0", 115200, timeout=10.0)
serWaiting = 0

dataQueue = deque()

volts = deque()

FFTvals = deque()

frameToken = "af"

sampleRate = 3000 # hz per channel

TotSamples = 3000 # total number of samples for 1 second of data

channels = 1 # 1 is ch A alone, 2 is both

bytesPerFrame = channels * 2

""" Serial helpers """
def issue(message):
    global serWaiting
    ser.write(message.encode())
    serWaiting += len(message)

def issueWait(message):
    global serWaiting
    ser.write(message.encode())
```

```
    serWaiting += len(message)
    clearWaiting()

def read(count):
    return ser.read(count)

def readAll():
    return ser.read(ser.inWaiting())

def clearWaiting():
    global serWaiting
    r = ser.read(serWaiting)
    serWaiting = serWaiting - len(r)

""" Utilities """
def freqToHexTicks(freq):
    ticks = int((freq ** -1) / 0.000000025)
    hexTicks = hex(ticks)[2:]
    zeroAdds = "0" * (4 % len(hexTicks))
    combined = zeroAdds + hexTicks
    return combined[2:], combined[:2]

def getToRange(fromRange, toRange):
    fr, tr = fromRange, toRange
    slope = float(tr[1] - tr[0]) / float(fr[1] - fr[0])
    return lambda v : round((tr[0] + slope * float(v - fr[0])), 12)

""" Decoding """
def decodeFrames(data):
    couples = int(len(data) / 2)
    unpackArg = "<" + str(couples) + "h"
    unpacked = struct.unpack(unpackArg, data)
    zeroBottomNibble = lambda x : x >> 4 << 4
    result = list(map(zeroBottomNibble, unpacked))
    return result

def setupBS():
    """ Standard setup procedure. """
    if channels == 1:
        chString = "01"
        modeString = "04"
    else:
        chString = "03"
        modeString = "03"

    issueWait("!")
```

```
    issueWait(
        "[21]@[" + modeString + "]s" # Stream mode (Macro Analogue Chop (is 03))
        + "[37]@[" + chString + "]sn[00]s" # Analogue ch enable (both)
        + "[2e]@[%s]sn[%s]s" % freqToHexTicks(sampleRate) # Clock ticks
        + "[14]@[03]sn[00]s" # Clock scale (Doesn't work for streaming mode)
        + "[36]@[a5]s" # Stream data token
        + "[66]@[5a]sn[b2]s" # High
        + "[64]@[35]sn[1b]s" # Low
    )
    issueWait("U")
    issueWait(">")
    print(freqToHexTicks(sampleRate))
    readAll()
    """NEED CODE TO READ ACTUAL SAMPLE RATE BACK FROM BITSCOPE TO
VERIFY WHAT WE'VE DONE!!"""

def startStream():
    read(2) # ?!?!?! WHY DID AUTHOR PUT SO EMPHASIS HERE?!??!
    issueWait("T")

def readStream(timeout):
    data = bytearray()
    toGet = int(sampleRate * timeout) * bytesPerFrame
    toGet = toGet - (toGet % bytesPerFrame)
    data = ser.read(toGet)
    return data


def readLoop():
    global running
    print 'Capturing Data Stream, press Ctrl-C to terminate'
    while running:
        data = readStream(0.0166666666666666666)
        dataQueue.append(data)
    ser.close()
    print 'BitScope Connection Terminated'

def processAndWriteLoop():
    global running
    toRangeLambda = getToRange((-32768, 32767), (-5, 5)) #map from 32-bit signed int to floats
in range [-5,5]
    time.sleep(1.0)
    #print len(dataQueue)
    nFrames = 0
    try:
        while True:
```

```
            #time.sleep(1.0)
            #print len(dataQueue)
            while len(dataQueue):
                #print 'INSIDE DATA LOOP'
                # Pop data
                data = dataQueue.popleft()
                # Decode
                levelData = decodeFrames(data)
                # Voltify
                voltData = list(map(toRangeLambda, levelData))
                # calculate FFT at 60 Hz ###NEED TO GENERALIZE INDEX
                Nfft = len(voltData)
                TMP = np.fft.fft(voltData)/Nfft
                FFTvals.append(TMP[1])
                # Store in local variable
                #for data_pt in voltData:
                    #volts.append(data_pt)
                nFrames = nFrames + 1
                print str(nFrames)+' frames of data captured\r',
        except KeyboardInterrupt:
            running = False
            print '\nData Capture Terminated by User'
            pass

def main():
    # Stop BS and clear out serial buffer
    issueWait(".")
    readAll()
    time.sleep(1.0)
    # Start BS
    print 'Setting up BitScope'
    setupBS()
    print 'Initializing Stream...'
    startStream()
    # Open read stream thread
    readThread = threading.Thread(target = readLoop)
    readThread.start()
    # NEED THREAD FOR
PLOTTING!!!!#########################################################################
##
    # Start writing loop in main thread
    processAndWriteLoop()

# Go

main()
```

```
#volts = np.array(volts)
FFTvals = np.array(FFTvals)
Vmag = np.abs(FFTvals)*2
Vang = np.unwrap(np.angle(FFTvals))*180.0/np.pi
Vfreq = np.diff(np.unwrap(np.angle(FFTvals)))*60/2.0/np.pi + 60
np.savetxt("/home/pi/Desktop/Pat_Capstone/Vfreq.txt",Vfreq,fmt='%.16e',newline=',')
np.savetxt("/home/pi/Desktop/Pat_Capstone/Vmag.txt",Vmag,fmt='%.16e',newline=',')
np.savetxt("/home/pi/Desktop/Pat_Capstone/Vang.txt",Vang,fmt='%.16e',newline=',')
```

Appendix B: MIDI Pitch Parameter Generator MATLAB Code

```
function [pitch] = midiPitchGen(Vfreq)
freqDev = Vfreq - 60; % calculates deviation from 60 hertz
pitch = freqDev; %creates array for pitch values
minDev = min(freqDev); %calculates the min dev value
maxDev = max(freqDev); %calculates the max dev value
range = maxDev-minDev; %generates range of frequency values
step = range/48; %generates value of steps. 48 pitches for a total of 48
steps
pMax = 84; %maximum pitch values
sample = 1; %sample number is equal to 1

for i = pitch
    %if the pitch deviation is zero the pitch at that sample is set to 60
    if i == 0
        pitch(sample) = 60;
        sample = sample + 1;
    else
        minStep = minDev;
        pVal = 36;
        while pVal <= pMax
            if  (i >= minStep) && (i < (minStep + step))
                pitch(sample) = pVal;
                sample = sample +1;
                pVal = pMax +1;
            else
                pVal = pVal + 1;
                minStep = minStep + step;
            end
        end
    end
end
```

Appendix C: MIDI Velocity Parameter Generator MATLAB Code

```
function [velocity] = midiVelocityGen(Vmag)
velocity = Vmag; %creates array for voltage values
minV = min(Vmag); %calculates the min  value
maxV = max(Vmag); %calculates the max value
range = maxV-minV; %generates range of volt values
step = range/127; %generates value of steps. 127 velocities for a total of
127 steps
vMax = 127; %maximum velocity value
sample = 1; %sample number is equal to 1

for i = velocity
    %if the pitch deviation is zero the pitch at that sample is set to 60
        minStep = minV;
        vVal = 1;
        while vVal <= vMax
            if  (i >= minStep) && (i < (minStep + step))
                velocity(sample) = vVal;
                sample = sample +1;
                vVal = vMax +1;
            else
                vVal = vVal + 1;
                minStep = minStep + step;
            end
        end
end
```

Appendix D: Ambient Data Audification MATLAB Code

```
% Audification Script
%Imported Data from Text Files
vfreq= importdata('/Users/red2sox4/Desktop/Capstone/Vfreq2.txt');
vmag = importdata('/Users/red2sox4/Desktop/Capstone/Vmag2.txt');

%Create Filter
N1 =6;
Wn1 = 2/30;
[B1,A1]=butter(N1,Wn1);

%filter data
vmagfinal=filter(B1,A1,vmag);
vfreqfinal=filter(B1,A1,vfreq);

%Remove Filter Transients and align data samples
%Sample lost due to differentation so an extra sample is removed when
%removing vmag transients
vmagfinal(1:transient+1) = [];
vfreqfinal(1:transient) = [];

%Create New Vectors for interesting time period
freqDev = vfreqfinal(2050:3000)-60; %subtract 60 for frequency deviation
vmagInteresting = vmagfinal(2050:3000);

%normalize frequency deviation and voltage magnitude
magNormalized = Normalize(vmagInteresting);
freqDevNormalized = (Normalize_1(freqDev);

%upsample data to 44100 fs and create new time arrays
fsAud = 44100;
vmagUp = resample(magNormalized,fsAud,60);
freqDevUp = resample(freqDevNormalized,fsAud,60);
NfreqDevAud = length(freqDevUp);
tEndAud = NfreqDevAud/44100.0;
tsAud = 1/fsAud;
tAud = 0:1/fsAud:tEndAud-tsAud;

%Generate final audio signal phase angle array
vAngFinal = integrateFreqDevInteresting(2*pi*50*freqDevUp,tsAud);

G = 50; %Vang Gain value
%Create Audio signal
vAudio = sqrt(2)*0.5*vmagUp.*cos(2*pi*(261.6).*tAud + G*vAngFinal);
```

Appendix E: Audification MATLAB Normalization Functions

```
%Function takes Data array as input and normalizes the data into values of 0
to 1
function [normalizedData] = Normalize(data)
len = length(data);
normalizedData = zeros(1,len); %creates array for  values
minData = min(data); %calculates the min  value
maxData = max(data); %calculates the max value
range = maxData-minData; %generates range of values
sample = 1;

for i = 1:len
        normalizedVal = (data(i) - minData)/range;
        normalizedData(sample) = normalizedVal;
        sample = sample + 1;
end

%Function takes Data array as input and normalizes the data into values of -1
to 1
function [normalizedData] = Normalize_1(data)
len = length(data);
normalizedData = zeros(1,len); %creates array for  values
minData = min(data); %calculates the min  value
maxData = max(data); %calculates the max value
range = maxData-minData; %generates range of values
sample = 1;

for i = 1:len
        normalizedVal = (((data(i) - minData)/range)-0.5)*2;
        normalizedData(sample) = normalizedVal;
        sample = sample + 1;
end
```

Appendix F: Frequency Numerical Integration Function

```
%Function for numerical integration of frequency for calculation of Angle
function [vAngle] = integrateFreqDevInteresting(data,ts)
len = length(data);
vAngle = zeros (1,len); %creates array for angle values
lowerIntVal = 0;
for i = 2:len
    %integrate samples
    dy = data(i-1);
    vAngle(i)= ts*dy + lowerIntVal;
    lowerIntVal =  vAngle(i);
end
end
```

Appendix G: Ambient Data Sonification Python Code

```
Sonification Script

@author: Pat Cowden
"""
from midiutil.MidiFile import MIDIFile
from scipy import signal
from random import choice
import numpy as np

#generates a list of the note values for a specific Major key
#takes keyIndex as input (0 for C,=1 for C#.....)
def generateKey(keyIndex):
    notes = [0,2,4,5,7,9,11]
    keyNotes = [x + keyIndex for x in notes]
    return keyNotes


#Function used for determing the lowest possible pitch value below the centered middle C
pitch
#Takes number of octaves as a parameter
def findLowestNote(numOctaves):
    lowestNote = 60 - numOctaves*12
    return lowestNote

#function that takes the note values list and number of octaves to generate a list of
#list of the notes in octaves
def generateNotes(keyNotes,numOctaves,lowestNote):
    allNotes = []
    totalOctaves = numOctaves*2
    for i in range(totalOctaves):
        for note in keyNotes:
            noteVal = lowestNote + note + i*12
            allNotes.append(noteVal)
    return allNotes

def createKeys():
    keys = {'B#':0,'C':0,'C#':1,'Db':1,'D':2,'D#':3,'Eb':3,'E':4,'Fb':4,'E#':5,'F':5,'F#'
:6,'Gb':6,'G':7,'G#':8,'Ab':8,'A':9,'A#':10,'Bb':10,'B':11 ,'Cb':11}
    return keys

#function converts set of frequency values into notes within a specified music key
#generates chromatic notes if key does not exist
#numOctaves is the number of octave above/below the center point middle C(pitch val 60)
def midiPitchGen(Vfreq,key,numOctaves):
    keys = createKeys()

    #if key doesnt exist the keyNotes are set to all values for chromatic playback
    #otherwise, the key notes are generated in the else statement
    if keys.has_key(key) == False:
        keyNotes = [0,1,2,3,4,5,6,7,8,9,10,11]
    else:
        keyIndex = keys[key]
        keyNotes = generateKey(keyIndex)
```

```
    #finds lowest notes and generates all of the possible pitches
    lowestNote = findLowestNote(numOctaves)
    notes = generateNotes(keyNotes,numOctaves,lowestNote)

    #converts frequency values into corresponding note pitches
    pitches = createPitches(Vfreq,notes)
    return pitches

#Converts Voltage magnitude data into list of corresponding midi velocities
def midiVelocityGen(Vmag):
    velocity = []; #creates array for voltage values
    minV = min(Vmag); #calculates the min  value
    maxV = max(Vmag); #calculates the max value
    vRange = maxV-minV; #generates range of volt values
    vStep = vRange/128.0; #generates value of steps. 128 velocities for a total of 128 st
eps
    velocityMax = 127; #maximum velocity value

    #for loop for creating the array of velocities corresponding to each data sample
    #sets value if the data sample falls within corresponding voltage magnitude range
    for V in Vmag:
        minVal = minV
        velocityIndex = -1
        while (minVal <= V):
            minVal += vStep
            velocityIndex += 1

        #There's an indexing issue where the maximum value is assigned a veloicty past
        #the max possible velocity, it is a bug that I could not figure out, so I have
        #this if statement to decrease the index by 1 if the value is too high
        if velocityIndex > velocityMax:
            velocityIndex -=1
        velocity.append(velocityIndex)
    return velocity

def createPitches(Vfreq,notes):
    pitch = []
    minFreq = min(Vfreq) #calculates the min dev value
    maxFreq = max(Vfreq) #calculates the max dev value
    rangeFreq = maxFreq-minFreq #generates range of frequency values
    numberOfNotes = len(notes)
    fStep = float(rangeFreq)/numberOfNotes

    #for loop for creating the array of pitches corresponding to each data sample
    #sets pitch value when the data sample value falls within the corresponding frequency
 range
    for freq in Vfreq:
        if freq == 60:
            pitch.append(60)
        else:
            minVal = minFreq
            noteIndex = -1
            while (minVal <= freq):
                minVal += fStep
                noteIndex += 1
```

```
            #There's an indexing issue where the maximum value is assigned an index past
#possible index, it is a bug that I could not figure out, so I have this if statement to
#decrease the index by 1
            if noteIndex == numberOfNotes:
                noteIndex -=1
            pitch.append(notes[noteIndex])
    return pitch

#upload data
Vfreq = np.loadtxt("/Users/red2sox4/Desktop/Capstone/Vfreq2.txt",delimiter = ",")
Vmag = np.loadtxt("/Users/red2sox4/Desktop/Capstone/Vmag2.txt",delimiter = ",")

#create filter and filter data
N =6;
Wn = 2.0/30;

#b = signal.firwin(N,Wn)
b, a = signal.butter(N, Wn, 'low',)

vfreqfinal = signal.lfilter(b,a,Vfreq)
vmagfinal = signal.lfilter(b,a,Vmag)

#Remove Filter Transients
vfreqfinal = vfreqfinal[120:]
vmagfinal = vmagfinal[121:]

#Set Interesting Data range
vfreqInteresting = vfreqfinal[1150:1300];
vmagInteresting = vmagfinal[1150:1300];


#generate pitch and velocity arrays
pitch  = midiPitchGen(vfreqInteresting,'Bb',2)
velocity = midiVelocityGen(vmagInteresting)

#midi parameter values
track    = 0
channel  = 0
time     = 0   # In beats
noteLengths = [1/4.0,1/2.0,1]   # In beats, array of possible values for randomized note
length
tempo    = 120  # In BPM


MyMIDI = MIDIFile(1) # One track
MyMIDI.addTempo(track, time, tempo)

for i in range(len(vfreqInteresting)):
    duration = choice(noteLengths)
    MyMIDI.addNote(track, channel, pitch[i], time, duration, velocity[i])
    time = time + duration

with open("AmbientDataBbRandomLength.mid", "wb") as output_file:
    MyMIDI.writeFile(output_file)
```