Converting EKG Signals Into 3-D Stereo Sound

by

Frank Seong-Huhn Kim

S.B., Electrical Engineering
Massachusetts Institute of Technology, 1991

Submitted to the Department of Electrical Engineering and
Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science in Electrical Engineering and Computer Science
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1994

© Frank Seong-Huhn Kim, MCMXCIV. All rights reserved.

The author hereby grants to MIT permission to reproduce and
distribute publicly paper and electronic copies of this thesis
document in whole or in part, and to grant others the right to do so.

Author .................................................................
Department of Electrical Engineering and Computer Science

Certified by ...........................................
Gill Pratt
Assistant Professor
Thesis Supervisor

Accepted by ...........................................
Frederic R. Morgenstahler
Chairman, Committee on Graduate Students
Converting EKG Signals Into 3-D Stereo Sound

by

Frank Seong-Huhn Kim

S.B., Electrical Engineering
Massachusetts Institute of Technology, 1991

Submitted to the Department of Electrical Engineering and Computer Science
on January 14, 1994, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

This thesis involved converting an EKG signal into sound. To do this, the three-dimensional EKG vector versus time was found, and then a stereo sinusoidal sound, based on the location of this vector versus time, was created. The stereo sound’s frequency, amplitude, and phase between the left and right sound were the variable factors. Using the stereo sound created, different EKGs could be assessed qualitatively and compared. This research project’s main motivation is not to eventually discover a new method of analyzing EKG signals but rather to determine new methods of analyzing qualitative information using sensory techniques.

Thesis Supervisor: Gill Pratt
Title: Assistant Professor
Acknowledgments

Thanks go to Professor Gill Pratt for his patience, encouragement, and guidance throughout this project. Kirk “UhOh” Johnson, a premier hacker, was invaluable in helping with late night debugging sessions and pointing out several compiler tricks. Barbara Shinn provided the filters and great help in developing the software to use the filters. Mike McCandless provided the software for driving the stereo sound through the headphones. Dr. Ary Goldberger, Dr. Roger Mark, and Dr. George Moody were important sources of information about current EKG research.
4.2 Using Digital Signal Processing Chips .................. 38

5 Conclusions 40
5.1 Results .................................................. 40
5.2 Conclusions .............................................. 44

A Program Code 45
A.1 Horizontal Sweep Using Sound Localization Theory .... 45
A.2 Horizontal Sweep Using Measured Head-Related Transfer Functions 48
A.3 Producing Three-Dimensional EKG Sound .................. 57
A.4 Playing Binary Sound Files ................................ 75
A.5 Convolution using FFT .................................... 79
A.6 FFT of Real Signal ........................................ 81
A.7 Fast Fourier Transform (FFT) ............................. 83
List of Figures

1-1 Typical EKG wave. ........................................ 12
1-2 EKG leads. .................................................. 14

3-1 Graphical Representation of EKG Chain File. ............. 20
3-2 EKG 18. ..................................................... 21
3-3 Plots of the magnitude of the Fourier transforms of the right and left sound for a horizontal sweep using theoretical method. The sampling rate is 44100 Hz. The large spikes correspond to the component frequencies of the original sinusoidal sound at 70, 88, 100, 117, and 351 Hz. ...................................................... 24
3-4 Plots of a typical impulse response for right and left pinnae. $n$ is the sample number. The sampling rate is 44100 Hz. Notice that the right impulse response is smaller in amplitude since the sound source was behind and left of the head (150°) when this particular pair of impulse responses were measured. ............................................ 25
3-5 Changing big endian to little endian. ............................ 26
3-6 Plots of the magnitude of the Fourier transforms of the right and left sound for horizontal and vertical sweeps using HRTFs. The sampling rate is 44100 Hz. The spikes correspond to the component frequencies of the original sinusoidal sound at 70, 88, 100, 117, and 351 Hz. Notice that the convolution of the original sound with the HRTFs caused a greater emphasis of the high frequency sounds. This is because the high frequency is where most of the localization information is found. 28
3-7 Vector Cardiogram of a typical EKG. \( n \) is the sample number. The sampling rate is 306 Hz. The frequency of this particular EKG is 1.52 Hz. The FFTs of the different leads shows other higher frequency components, up to 20 Hz, corresponding to the bumps in the EKG, and the DC offset.

5-1 EKG 1765

5-2 EKG 94

5-3 Plots of the magnitude of the Fourier transforms of the right and left EKG sounds for EKG 18, EKG 1765, and EKG 94. The sampling rate is 44100 Hz. Notice that the frequency spectrum is spread out within a certain range due to frequency modulation. EKG 18’s sounds have frequencies concentrated around 100 Hz, 150 Hz, and 250 Hz. EKG 1765’s sounds have a similar spectrum but its lower and higher frequency components are relatively stronger, resulting in a longer beat and higher pitch. EKG 94’s sounds have many more high frequency components, resulting in a high pitch and fast beat.
List of Tables

4.1 Performance of Overlap-Save Algorithm Based On Value of $L$ . . . . 37
Chapter 1

Introduction

1.1 Motivation

Recently there has been an increased amount of research on utilizing sound as a method of communicating qualitative information. Continuing this direction of research, we are attempting to convert EKG signals into sound. This research project’s main motivation is not to eventually discover a new method of analyzing EKG signals but rather to determine new methods of analyzing qualitative information using sensory techniques.

The human ear is particularly adept at noticing both patterns and changes in patterns. This property has been the motivation for turning large amounts of data into sound or “music”. Some examples of data that have been turned into music are the genetic code of a mouse cell, the atomic structure of air pollutants, the way gas flows through a jet engine, the constituents of human urine [7].

Evidence of the growing interest in this field could be seen at the First International Conference on Auditory Display in Santa Fe, N.M. in late 1992. Researchers from many of the top corporate labs including AT&T, Xerox, Exxon, IBM, Apple Computers and other corporate labs attended this inaugural conference [7].
1.2 Previous Work

The EKG has been used for decades as a noninvasive diagnostic tool for analyzing and monitoring the heart. Since its emergence as a major medical tool, much research has been done into learning how to fully utilize the EKG. One example is the Holter EKG which is a long-term electrocardiogram recording. The clinical ability of the Holter EKG is continuous, long-term examination of the patient. The Holter EKG is either done using a conventional tape (Holter) recording and appropriate playback instrumentation systems or solid-state technology employing a real-time analysis microcomputer and report generator [6]. Frequency analysis of EKG signals has also been researched and found to benefit in distinguishing patients with prior myocardial infarction from those without sustained ventricular arrhythmias and distinguishing patients with and without sustained ventricular tachycardia (VT) [1].

Sound localization research has been done for decades but recently there has been increased activity in this field. Dr. Mark F. Davis’s 1981 Doctoral thesis, Computer Simulation of Static (fixed-head) Localization Cues, focused on determining the arrival process of a single source sound. He developed a computer system which measured peripheral binaural impulse responses at a series of target source positions using small microphones in the subject’s ear canals. This system then prepares test stimuli by convolving these impulse responses with the stimuli and playing the results to the subject over headphones [2].

Wightman, Kistler, Wenzler, and Foster continued doing research on sound localization using binaural impulse responses which they called head-related transfer functions, or HRTFs. The goal of their research was to produce a virtual display system for conveying three-dimensional acoustic information [13] [14] [15]. Durlach, Shinn-Cunningham, and company continued this work, trying to improve on localization [4] and externalization of the sound [3].

Professor Gill Pratt’s 1987 Master’s thesis, A blind man’s compass, used sound to help locate magnetic north. The motivation of this thesis was to provide blind people some sort of directional cue since they obviously could not use visual cues for direc-
tion. The blind man's compass consisted of a sound generator, stereo headphones, and a solid state magnetometer. After measuring the relative direction of magnetic north using the magnetometer, the sound generator created an acoustic image which was perceived to be originating from this measured direction. Since the magnetometer was placed on the subject's head, the sound coming out of the headphone was dependent on the subject's head position relative to magnetic north. Such a device allowed a person to easily locate magnetic north without needing a visual compass thus providing a directional cue [9].

1.3 Background on Electrocardiograms

The electrical potential of cells is related to the difference between the concentrations of intracellular and extracellular electrolytes. In the resting state, myocardial cells (those that make up the thick muscular middle layer of the heart wall and most of the heart mass) are negatively charged with respect to their surroundings. During a process called depolarization, these cells lose their negativity, resulting in the cells contracting. This fundamental electrical event of the heart propagates from cell to cell, producing a wave of depolarization that can be transmitted across the entire heart. Once depolarization is complete, a similar process called repolarization occurs which just reverses what the previous process did. The result is that the myocardial cells return to their original resting potential.

The EKG's main clinical value is that of a diagnostic tool. Some examples of its medical use are

- Interpretation of arrhythmias (alterations in the rhythm of the heartbeat either in time or force).
- Diagnosis and management of ischemic heart disease (localized tissue anemia due to obstruction of the inflow of arterial blood).
- Assessment of ventricular hypertrophy (exaggerated growth or complexity).
Figure 1-1: Typical EKG wave.

- Diagnosis of evolving myocardial infarction (area of necrosis, localized death of living tissue, in middle muscular tissue layer of heart. This results from obstruction of the local circulation by a thrombus, a clot of blood formed within a blood vessel and remaining attached to its place of origin, or an embolus, an abnormal particle such as an air bubble, circulating in the blood).

- Pinpointing the chronic effects of sustained hypertension (abnormally high blood pressure, especially arterial blood pressure) or the acute effects of a massive pulmonary embolus.

- Providing simply a measure of reassurance to someone who wants to begin an exercise program [5] [12].

A number of abnormalities of conduction such as bundle branch block, which cannot easily be diagnosed by other methods, are revealed by the electrocardiogram [5].

Depolarization is initiated at the sinus node of the heart. This activity is not seen on the EKG. This event is followed by a wave of depolarization through first the right atrium and then the left atrium. The contraction of the atria is recorded in the P wave portion of the EKG wave (see Figure 1-1). After the wave of depolarization goes
through the atria, it reaches the atrioventricular (AV) node, and the EKG falls silent. Then the wave of depolarization spreads along the ventricular conducting system and out into the ventricular myocardium. The left and right ventricle depolarize at about the same time. However most of what we see on the EKG is due to the left ventricle since its muscle mass is about three times as large as that of the right ventricle. Ventricular depolarization generates the QRS complex (see Figure 1-1).

After the myocardial cells depolarize, there is a brief refractory period during which they are resistant to further stimulation. During this time the EKG is again silent. Then ventricular repolarization occurs and this is seen in the T wave (see Figure 1-1). Finally atrial repolarization occurs but this is not seen in the EKG.

Note that the EKG is divided into three main segments/intervals. The PR interval is the time from when atrial depolarization begins to when ventricular depolarization begins. The QT interval measures the time for ventricular depolarization and repolarization. The ST segment measures the refractory period.

The electrocardiogram consists of twelve leads of information. Six of the leads view the electrical forces moving up and down and left and right in the frontal plane. The other six leads view the electrical forces moving anteriorly and posteriorly. Of these twelve leads, three leads can be used directly to form the three-dimensional plot of the net electrical vector of the heart as it beats (leads I, AVF, V2).
SIX LEADS of the FRONTAL PLANE

SIX PRECORDIAL LEADS

Figure 1-2: EKG leads.
Chapter 2

Sound Localization

2.1 Theoretical

In normal circumstances, humans use as many senses as possible, especially the sense of sight, with the sense of hearing to help localize sound. For example, if one hears an owl hoot in the night, you will first look towards where you think the sound is coming from. You first base your initial guess on where the sound is coming from based on the difference in amplitude and timing/phase between the sounds heard at each ear. Then as you turn your head towards the sound, you try to locate the origin of the sound with your eyes and by noticing relative differences in sound quality between the two ears as your head moves.

The human auditory system appears to act as if it makes independent estimates of the horizontal angle, vertical angle, and distance of the sound source in question, relative to the current position and orientation of the listener’s head. Distance can be determined by the auditory system using amplitude and reverberation content. Vertical position can, to a limited degree, be deduced from head motion and spectral cues imparted by the pinnae [2].

When using headphones, one can only use two single valued time functions to help localize sound, i.e. the effective incremental positions of the eardrums as functions of time. Both the difference in sound arrival times at the two ears (“interaural time delay”, or ITD) and the ratio of the amplitudes at the ears (“interaural amplitude
differences" or IAD) were readily identified as affecting horizontal localization (cf. Rayleigh, 1876). ITD comes about because a sound originating off center arrives at one ear before the other. IAD occurs due to the shadowing of sounds by the body of the head [9].

When using headphones it is very difficult to localize sound in the vertical plane. The static ITD and IAD cues give no information about the vertical angle. Theoretically, dynamic ITD and IAD cues can give very accurate estimates of the vertical angle, since these cues change as the head moves. But since the headphones are tied to the head, head motion is useless and it is difficult to quantify head motion. Also spectral cues for vertical localization are difficult to communicate. Distance localization can be done but not as successfully as horizontal localization because you can only use amplitude; reverberation content is difficult to simulate.

A crude approximation for the static IAD cue is

\[ IAD_1 = \frac{(1 + \sin(\theta))}{2} \tag{2.1} \]
\[ IAD_2 = \frac{(1 - \sin(\theta))}{2} \tag{2.2} \]

\( IAD_1 \) modulates the amplitude of one ear while \( IAD_2 \) modulates the amplitude of the other ear [9].

A simple formula for the static ITD cue, phase-derived is

\[ ITD_{p-lf} = \frac{3a}{c} \ast \sin(\theta) \tag{2.3} \]
\[ ITD_{p-hf} = \frac{2a}{c} \ast \sin(\theta) \tag{2.4} \]

\( a \) = radius of the sphere
\( \theta \) = angle relative to the pole (perpendicular) facing the incident angle
\( c \) = ambient speed of sound

\( ITD_{p-lf} \) is the ITD cue for low-frequency while \( ITD_{p-hf} \) is the ITD cue for high-frequency. [16].

It was found that the ear is most accurate at determining the horizontal location of a source emitting a continuous sinewave for stimulus frequencies either below 1 kHz or above 3 kHz [2].
2.2 Using Measured Head-Related Transfer Functions

Two primary assumptions about the nature of the auditory system are a general assumption of linearity and dual time function:

I The air and reflecting surfaces coupling sound to the eardrum from an external source at a given angle comprise a linear, time-invariant system.

II The position of a source is inferred on the basis of the two signals at the eardrums. There is no "direct" sensing of wavefront arrival angle done by the auditory system, and only the auditory system is involved in the localization process.

Of course, in a natural environment one is likely to augment the auditory localization process with other sensory modes, especially vision and movements of the head [2].

The impulse responses, also known as head-related transfer functions or HRTFs, at the right and left pinnae can be measured for various azimuth and elevation angles. The closer ear's impulse response will be sharper and less delayed (i.e. more spread out) than that of the other ear. To then determine the sound heard at the ears, one can convolve the HRTFs with the original sound to produce the output sounds for the right and left pinnae. To determine the HRTFs for the angles that were not directly measured, a simple linear interpolation can be used to determine the correct impulse response. Such a method will produce only small amounts of error if the angular interval between measured impulse responses is small.

Using the HRTFs, as filters to synthesize spatial cues does not actually give us true three-dimensional spatial sound. This is simply because the HRTFs depend only on two dimensions, azimuth and elevation. These filters fail to provide any differential distance cues. Also, the actual spatialization of the sound is limited to "in-head spatialization", i.e. the images often appear to be located inside the head [4].

Mathematically we can express how we use the impulse responses to determine the sound originating from the direction of a single source in anechoic (free from echoes
and reverberations) space.

\[ Y_L(\omega, \theta, \phi) = X(\omega)S_L(\omega, \theta, \phi) \] \hspace{1cm} (2.5)

\[ Y_R(\omega, \theta, \phi) = X(\omega)S_R(\omega, \theta, \phi) \] \hspace{1cm} (2.6)

- \( Y_L(\omega, \theta, \phi) \) = complex spectrum of signal received in left ear,
- \( Y_R(\omega, \theta, \phi) \) = complex spectrum of signal received in right ear,
- \( X(\omega) \) = complex spectrum of transmitted signal,
- \( S_L(\omega, \theta, \phi) \) = space filter for path to left ear,
- \( S_R(\omega, \theta, \phi) \) = space filter for path to right ear,
- \( \omega \) = angular frequency,
- \( \theta \) = azimuth,
- \( \phi \) = elevation.

These equations assume that the distance from the head to the source is sufficiently great that it affects only overall level of \( Y_L \) and \( Y_R \) (and this dependence is ignored), and that the source can be assumed isotropic (exhibiting properties with the same values when measured along axes in all directions) [4].
Chapter 3

Methodology

Several steps were involved in implementing the sound localization of the EKG signals.

1. Acquire digitized EKG signals.

2. Do background research on sound localization theory.

3. Create demonstration of three-dimensional sound localization using sound localization theory.

4. Acquire digitized impulse responses (HRTFs) for various azimuth and elevation angles.

5. Create demonstration of three-dimensional sound localization by convolving variable impulse responses (HRTFs) with input sound.

6. Compare two demonstrations and determine the best method. Using the best method, convert EKG signals into three-dimensional sound.

3.1 Acquiring digitized EKG signals

We were fortunate in being able to acquire digitized EKG signals, free of charge, from Clifford Goldsmith with the assistance of Dr. Ary Goldberger. Dr. Goldberger selected some interesting EKG signals and then sent them to Mr. Goldsmith who
digitized them for us. The format of the digitized EKG signals was a simple ASCII file called an EKG Chain File. The EKG signal has 300 samples/inch and it was digitized at a rate of 25 mm/sec, for a sampling rate of 306 Hz.

The Chain File is a digitized representation of a 12 lead EKG. The 12 lead EKG is divided into 13 regions. 12 regions represent a time sampling of a different lead each. The bottom region represents the time sampling of lead II which is done while all the other leads are being sampled. x0 through x3 are the values of the zero axis for each horizontal section. y0 represents the time value when data is first sampled for leads I through III, y1 represents the time value when data is first sampled for leads aVR through aVF, etc. Figure 3-1 shows a graphical representation of a sample Chain File. Figure 3-2 shows a typical 12 lead EKG.
put EKG here!

Figure 3-2: EKG 18.
3.2 Creating demonstration using sound localization theory

First we created a demonstration of a horizontal sweep around the head using sound localization theory (refer to A.1). A horizontal sweep should sound like the sound source is moving around the head at a constant speed and at a constant distance from the head. To do this we simply created a sinusoidal sound of a desired length (usually about five seconds) but how we created the sound depended on where we were in the sweep i.e. what the azimuth angle is. Equations 3.1, 3.2, and 3.3, previously explained in Chapter 2, were used to accomplish this.

\[ IAD_R = \frac{(1 + \sin(\theta))}{2} \] (3.1)

\[ IAD_L = \frac{(1 - \sin(\theta))}{2} \] (3.2)

\[ ITD_p = \frac{3a}{c} \sin(\theta) \] (3.3)

\( IAD_R \) modulates the right ear while \( IAD_L \) modulates the amplitude of the left ear [9].

\( a = \) radius of the sphere
\( \theta = \) azimuth angle
\( c = \) ambient speed of sound [16]

For our program, we used \( a = 0.10 \) meters, \( c = 334 \) m/s. Also we multiplied the ITD by 44100 to convert it from seconds into samples (the output sound play rate is 44100 Hz). In keeping the above equations consistent with the equations of Chapter 2, the azimuth angle increases positively in the counter-clockwise direction and an azimuth angle of 0° means the sound is directly in front of the subject.

The resulting right and left sound signals were saved to a file and then played through the stereo headphones using the PLAY program (see A.4).
One can see in Figure 3-3 the magnitude of the Fourier transforms of the output sounds for a horizontal sweep. We attempted to move the original sinusoidal sound around the head, beginning directly behind the head and moving in a circular fashion towards the left side of the head and then to the right side of the head. Listening to the sound, it was determined that the sound didn't seem realistic nor pleasing. We then decided to concentrate our efforts on using the HRTFs.

3.3 Acquiring digitized impulse responses for various azimuth and elevation angles

We learned of Barbara Shinn-Cunningham's research on three-dimensional sound localization fortuitously through the MIT Tech Talk. The article only mentioned that the research was being done at MIT but not where or by whom. So after a month of phone calls and messages, we managed to track down the research to Professor Lou Braida's lab.

Ms. Shinn-Cunningham was exceedingly helpful in explaining her research to us. She had set up a system on an IBM PC using a commercial DSP board (Convolvotron) to produce real-time three-dimensional sounds through stereo headphones. Interestingly, when she demonstrated this to us, we did not notice any change in the sound in terms of the elevation angle, only in terms of the azimuth angle and distance.

As explained before in Chapter 2, the impulse responses, HRTFs, at the right and left pinnae can be measured for various azimuth and elevation angles. Figure 3-4 shows a typical pair of impulse responses. These impulse responses can then be convolved with the original sound. The output sounds of these convolutions are the sounds that the eardrum would receive if the original sound was coming from the orientation at which the impulse responses were measured.

Ms. Shinn-Cunningham's system did real-time convolution of the impulse responses with the original sound to produce her three-dimensional sounds. These sets of measured impulse responses (HRTFs), which she freely gave to us, were sampled at 44100 Hz. Since they had been recorded using a IBM PC, they were recorded in
Figure 3-3: Plots of the magnitude of the Fourier transforms of the right and left sound for a horizontal sweep using theoretical method. The sampling rate is 44100 Hz. The large spikes correspond to the component frequencies of the original sinusoidal sound at 70, 88, 100, 117, and 351 Hz.
Figure 3-4: Plots of a typical impulse response for right and left pinnae. $n$ is the sample number. The sampling rate is 44100 Hz. Notice that the right impulse response is smaller in amplitude since the sound source was behind and left of the head (150°) when this particular pair of impulse responses were measured.
big endian format. Each data point was two bytes long and since the SUN used little endian format, we had to convert the data format. This was simply done by switching the order of every pair of bytes. Figure 3-5 graphically shows how this was done.

The files she gave us not only had measured HRTFs but also had for every azimuth and elevation angle pair the corresponding interaural time delay in samples. The files were arranged such that the azimuth angle increased positively in the counterclockwise direction, the elevation angle increased positively in the upward direction, a positive ITD meant the sound arrived at the left ear first, and when both angles were zero, the sound originated in front of the head.

3.4 Creating demonstration using impulse responses

First we created a demonstration of a horizontal sweep around the head using the impulse responses. The impulse response file would be read into arrays. Then we would create a sound of the desired length (usually about five seconds). Finally we would sweep along a circle, starting behind the head and moving clockwise. At every step along the circle, we would calculate the angle of the sweep and use that angle to find the appropriate impulse response. Since the impulse response was only measured for azimuth angles which were multiples of 30°, for angles that weren’t multiples of 30°
we found the correct impulse response by interpolating between the impulse responses at multiples of 30° just below and above the angle. After finding the correct impulse response, we convolved it with the sound to produce the output sound. This was done for both the left and right ear. After computing the sounds, the sounds were appropriated delayed (ITD) relative to each other based on the correct delay for the given angle. The output sound was saved to a file and later played with the PLAY program through the stereo headphones (refer to A.4).

One can see in Figure 3-6 the magnitude of the Fourier transforms of the output sounds for a horizontal sweep using this empirical method. It was found that the sounds were more realistic and swept more smoothly around the head than the sounds created using the theoretical approach.

We then created a demonstration for a vertical sweep using the same method. Unfortunately, as we expected, we could not notice any sound movement in the vertical plane. Figure 3-6 shows the magnitude of the Fourier transforms of the right and left sounds as we swept from above the head to below the head.

### 3.5 Convert EKG Signal into Three-Dimensional Sound

After trying the two different methods, it was determined that using the HRTFs to create the stereo sound was definitely the better method. However, since the stereo sound from the convolutions apparently seemed to impart very little audio information in the vertical plane, we needed to do some improvisations to make the sound more qualitatively rich. Such improvisations became apparent and practical after optimizing the method described below and are explained in Chapter 4.

The following steps are taken to convert an EKG signal into a three dimensional sound:

1. Open and read impulse response, HRTF, file.

2. Open and read EKG Chain File.
Figure 3-6: Plots of the magnitude of the Fourier transforms of the right and left sound for horizontal and vertical sweeps using HRTFs. The sampling rate is 44100 Hz. The spikes correspond to the component frequencies of the original sinusoidal sound at 70, 88, 100, 117, and 351 Hz. Notice that the convolution of the original sound with the HRTFs caused a greater emphasis of the high frequency sounds. This is because the high frequency is where most of the localization information is found.
3. Determine EKG vector cardiogram.

4. Create three-dimensional sound of a sinusoidal sound source moving along the vector cardiogram

5. Save to a file.

6. Play sound through stereo headphones.

After opening and reading the HRTF file, it was necessary to read the EKG Chain File and extract one heartbeat period from it. The Chain File represents data being extracted over a period of time (approximately a few seconds). As one can see in Figure 3-1 and Figure 3-2, the data is divided into quarters in which the data of different leads is recorded. However during the entire data recording period lead II’s data is always recorded.

We use the peaks of the long lead II to mark the start of each heartbeat period. Then we pick out the first heartbeat period for each different lead. However due to sampling errors, the lead data between the different rows was not exactly synchronized. To fix this we used the boundary markers between the different regions of leads to synchronize them.

Now it was necessary to create the three-dimensional locus of a single heartbeat, otherwise known as the EKG vector cardiogram. To do this we simply used lead V2 as the projection of the vector cardiogram on the x axis (pointing out of the chest), lead I as the projection along the y axis (pointing horizontally on the chest from the heart’s right atrium to left atrium), and lead aVF as the negative z axis (the negative z axis points downwards from the head to the foot). In Figures 3-7 one can see the plots of the three leads, the vector cardiogram projected on the three different planes, and a three dimensional plot of the vector cardiogram.

To create the three-dimensional sound of a sinusoidal sound source moving along the vector cardiogram, we first calculate the original sound. Then as we move along the vector cardiogram, projected on the yz plane, we convolve the original sound with the impulse response, which is determined based on the azimuth angle (angle in the yz plane, 0° points along the z axis, along the body up towards the head).
Figure 3-7: Vector Cardiogram of a typical EKG. $n$ is the sample number. The sampling rate is 306 Hz. The frequency of this particular EKG is 1.52 Hz. The FFTs of the different leads show other low frequency components, up to 20 Hz, corresponding to the bumps in the EKG, and the DC offset.
The sound is also modified by the distance from the source in the yz plane and the distance along the x plane. The distance in the x plane is used to change the intensity of the sound. To do this we simply multiplied the amplitude of the original sound, before convolving it, by a constant which is inversely proportional to the distance in the x plane.

The distance in the yz plane is used to change the frequency of the sound. While creating the original sound, we take into account the distance in the yz plane when figuring out the frequency components. The greater the distance in the yz plane, the lower the frequency of the sound.

Chapter 4 talks about how this algorithm was optimized. Chapter 5 shows the results of this algorithm on different EKG’s.
Chapter 4

Optimization and Real Time Study

4.1 Optimization

4.1.1 Initial Optimizations

When we first did the horizontal sweep demo, we noticed that it took about two minutes to calculate just a second of sound. Obviously we felt major improvements were necessary if this system was ever to become part of a practical real-time system for analyzing EKG signals. Several steps were taken to reduce this calculation time.

Since we couldn't use the vertical plane to convey sound information, we threw out all the impulse responses with elevation angles not equal to zero from the data file. Next we realized that we were not efficiently compiling the C program. We simply fixed this by using an efficient compiler which also took advantage of the SUN Sparcstation 10's hardware floating point multiply and divide hardware (normally on most SUN work stations multiply and divide is done with software routines). Doing this resulted in about a 30% speed-up.

Next we realized that it was unnecessary to calculate by interpolation a new impulse response, HRTF, for every azimuth angle not equal to a multiple of 30°. Assuming that the ear was not accurate enough to distinguish between sounds coming
from sources less than 15° apart, we created a new impulse response file with impulse responses for angles equal to a multiple of 15°. The impulse responses at 15°, 45°, etc. were calculated by taking the average between the impulse responses just above and below it. For example, the impulse response at 45° was calculated by averaging the known impulse responses at 30° and 60°. Now it was unnecessary to calculate any new impulse responses during the sound calculation since we just used the impulse response whose azimuth angle was closest to the angle of the sound source. This modification resulted in about a 70% speed-up.

4.1.2 Fast Fourier Transform (FFT)

Instead of doing the convolution of the HRTFs with the original sound the direct way, we decided instead to do the following steps:

1. Take the discrete Fourier transforms of the left HRTF, right HRTF, and original sound.

2. Multiply the left HRTF transform with the original sound transform.

3. Take the inverse of this product to get the left sound.

4. Multiply the right HRTF transform with the original sound transform.

5. Take the inverse of this product to get the right sound.

Intuitively it would seem this would be more work but in fact by using the Fast Fourier transform algorithm and the overlap save method we managed about a 95% speed-up.

The \(N\)-point Discrete Fourier transform (DFT), \(X[k]\), of a discrete time function, \(x[n]\), is defined as

\[
X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}, \quad k = 0, 1, \ldots, N - 1,
\]

(4.1)

where \(W_N\) is defined as

\[
W_N = e^{j\frac{2\pi}{N}}.[8]
\]

(4.2)
Note that $X[k] = 0$ for $k$ outside the interval $0 \leq k \leq N - 1$ and $x[n] = 0$ for $n$ outside the interval $0 \leq n \leq N - 1$.

If $N$ is an even integer, we can compute $X[k]$ by separating $x[n]$ into its even- and odd-numbered points. Doing this we obtain

\[
X[k] = \sum_{n=0}^{N/2-1} x[2n]W_N^{2nk} + \sum_{n=0}^{N/2-1} x[2n+1]W_N^{(2n+1)k}, \quad N \text{ is even}
\]

\[
= \sum_{n=0}^{N/2-1} x[2n]W_N^{nk} + W_N^k \sum_{n=0}^{N/2-1} x[2n+1]W_N^{nk}
\]

\[
= G[k] + W_N^k H[k]. [8]
\]

(4.3)

In Equation 4.3 we see how an $N$-point DFT computation can be split up into two $(N/2)$-point DFT computations where $G[k]$ and $H[k]$ are the $(N/2)$-point DFTs of the even-numbered points and the odd-numbered points of the original sequence, respectively. This property of an even $N$-point DFT is called the Danielson-Lanczos Lemma [10].

As with the $N$-point DFT, we can similarly transform the $(N/2)$-point DFTs into two $(N/4)$-point DFTs if $N/2$ is even. When $N$ is a power of 2, this splitting up can be done successively on each smaller transform until we have $N/2$ 2-point DFTs. Using this recursive property, the total calculation time of the transform is now $O(N \lg N)$ rather than $O(N^2)$. This $O(N \lg N)$ algorithm is known as the Fast Fourier transform or FFT.

The Numerical Recipes in C book gives an elegant and very efficient program for doing the FFT using the Danielson-Lanczos Lemma (see A.7). We used this code along with another program from the Numerical Recipes in C book (see A.6) to make a simple convolution program which uses the FFT (see A.5).

### 4.1.3 Overlap-Save Method

Now that we have a simple way to do the FFT, we need to decide how to use it in our EKG sound program. A naive yet simple way to use the FFT would be to take the $N$-point FFT of the original sound ($N$ being the length of the sound) and an $N$-point FFT of the HRTF. Then multiply the two FFTs and find the inverse FFT.
of the product to get the output sound. There are two reasons why one would not want to use the FFT in this naive manner.

1. Considering that $M = 256$ and $N$ is usually on the order of a few hundred thousand, it would be inefficient to compute an $N$-point FFT of the impulse response.

2. This is the more important reason and really the main reason. During the duration of the output sound, the azimuth angle is constantly changing. Therefore the corresponding impulse response is changing. Since the impulse response is time-variant, one cannot just take the FFT of one impulse response and use that to find the sound of a horizontal sweep or of any sound which is moving in three-dimensional space.

As one can see, it is necessary to use the overlap-save method for correctness as well as efficiency.

Instead of multiplying two $N$-point FFT's, we can instead divide up the original sound into sections of length $L$, (for example $L$ could be 512 samples long). For each section of length $L$, we can take the $L$-point FFT of it and multiply it by the $L$-point FFT of the correct impulse response (which is determined based on the azimuth angle). After taking the inverse transform of the product of these FFTs, we need to identify the part of the circular convolution that corresponds to a linear convolution. Note that the product of two FFTs multiplied together is the FFT of the circular convolution of the original time signals. Since the impulse response is length $M < L$, the first $(M - 1)$ points of the inverse transform are incorrect (this is called time aliasing) while the remaining points are identical to those had we implemented a linear convolution. After doing this with all the $L$-point sections of the original sound, we can put together the correct points to form the output sound.

Let the original sound be $x[n]$. We define the sections of $x[n]$ as $x_r[n]$ to be

$$x_r[n] = x[n + r(L - M)], \quad 0 \leq n \leq L - 1.$$  \hspace{1cm} (4.4)
Notice that for simplicity we are discarding the first \( M \) points rather than the first \( M - 1 \) points. This does not make a difference as long as we piece the correct points together in the proper manner.

Let \( h[n] \) be the \( M \)-point impulse response, \( y[n] \) be the \( N \)-point output sound, and \( y_{rp}[n] \) be the \( L \)-point result of a circular convolution of \( x_r[n] \) with \( h[n] \) in which time aliasing has occurred. Knowing that we are discarding the first \( M \) points of the \( L \)-point result segments we have

\[
y[n] = \sum_{r=0}^{N/(L-M)} y_r[n - r(L - M)],
\]

(4.5)

where

\[
y_r[n] = \begin{cases} y_{rp}[n], & M - 1 \leq n \leq L - 1, \\ 0, & \text{otherwise.} \end{cases}
\]

(4.6)

### 4.1.4 Implementing FFT and Overlap-Save Method

Using the FFT and the overlap-save method, we arrived at the following algorithm - the following steps are done continuously in a loop for each ear:

1. Create input sound for a section of length \( L \). How this input sound was created depended on the distance in the \( yz \) plane and \( x \) plane as described in Chapter 3.

2. Calculate azimuth angle of output sound. Use this to find appropriate impulse response.

3. Find the \( L \)-point FFT of impulse response and \( L \)-point FFT of sound section.

4. Multiply \( L \)-point FFT’s.

5. Find inverse \( L \)-point FFT of product, which is the output sound.

After each loop, we need to delay the right and left output sounds relative to each other depending on the azimuth angle. When the loop was completed the output sound was passed through a low-pass filter to smooth the output sound.
Table 4.1: Performance of Overlap-Save Algorithm Based On Value of $L$

<table>
<thead>
<tr>
<th>$L$</th>
<th>time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>3.66</td>
</tr>
<tr>
<td>1024</td>
<td>2.99</td>
</tr>
<tr>
<td>2048</td>
<td>2.94</td>
</tr>
<tr>
<td>4096</td>
<td>3.03</td>
</tr>
</tbody>
</table>

The input sound and output sounds are of length $N$. We begin the above loop at sample $M$, calculate the sound for samples $M$ to $M + L - 1$, and then do the next five steps to arrive at the output sound segment for samples $M$ to $M + L - 1 - M$. Next time through the loop (ignoring any delays), we calculate the input sound for samples $M + (L - M)$ to $M + (L - M) + L$. We continue to do this until we have computed the last output sound sample at $N - 1$.

It was necessary to decide which value of $L$ would give the best performance. We needed $L$ to be a power of two since we were using it in an FFT program and $L > M$ where $M = 256$. A simple formula for the order of the running time is

$$
\text{running time} = L \lg \frac{N}{L - M}
$$

With $M = 256$, we calculated using Equation 4.7 that for $L = 2048$ we would get the best performance. Table 4.1.4 shows the empirical results of how fast it took to calculate one second with different values of $L$. From this table one can see that $L = 2048$ was indeed optimal.

4.1.5 Final Optimizations

As a final improvement on the program, we realized that since we knew the impulse responses ahead of time, we could calculate its FFT ahead of time. So we calculated the FFT of the impulse responses and stored this in a separate file. Now it was unnecessary to calculate the FFT of the impulse response anymore. Also, we noticed that since the sound slices were overlapping, it was not necessary to calculate the sounds for the overlapping sections. These two improvements resulted in about a 33% improvement so that one second of sound for the right ear and one second of sound for the left ear took about 2.01 seconds to calculate (see A.3).
4.2 Using Digital Signal Processing Chips

In an actual system, even a time performance of 2.01 seconds to calculate one second of sound for the left and right ear would be unacceptable. The question still remained about whether a real-time system could be possible.

Digital Signal Processing (DSP) chips have specific hardware to efficiently handle FFTs, low pass filtering, and other signal processing functions. We studied Texas Instruments TMS320C3x DSP chips to see whether a real-time system could be built using one or more DSP chips.

According to the TMS320C3x User’s Guide:

The TMS320 family optimizes speed by implementing functions in hardware that other processors implement through software or microcode. It can perform parallel multiply and ALU operations on integer or floating-point data in a single cycle.

Certain TMS320C3x features that increase efficient implementation of numerically intensive algorithms are particularly well-suited for FFTs. The high speed of the device (50-ns cycle time) makes the implementation of real-time algorithms easier, while the floating-point capability eliminates the problems associated with dynamic range. The powerful indexing scheme in indirect addressing facilitates the access of FFT butterfly legs that have different spans. A construct that reduces the looping overhead in algorithms heavily dependent on loops (such as the FFTs) is the repeat block implemented by the RPTB instruction. This construct gives the efficiency of in-line coding but has the form of a loop. Since the output of the FFT is in scrambled (bit-reversed) order when the input is in regular order, it must be restored to the proper order. This rearrangement does not require extra cycles. The device has a special form of indirect addressing (bit-reversed addressing mode) that can be used when the FFT output is needed. This mode permits accessing the FFT output in the proper order.
The TMS320C3x quickly executes FFT lengths up to 1024 points (complex) or 2048 (real), covering most applications, because it can do so almost entirely in on-chip memory [11].

The TMS320C3x User’s Guide comes with assembly language routines which can be called from a C program as C functions.

Talking with DSP engineers at TI, we learned of TI’s newest DSP chip, the TMS320C40 which has a 50 MHz clock. A radix-2 real FFT of 1024 real points takes 1.01 milliseconds on the C40. A simple formula for the running time of an FFT on an N-point sound train, modifying equation 4.7, is

\[
running \ time = (1.01 \text{ ms}) * \frac{N}{1024 - 256} \tag{4.8}
\]

For \( N = 2^{16} = 65536 \) = 1.48 seconds of sound, we get a running time of 86.2 milliseconds for one FFT on a 65536 point sound train using the overlap-save method. Now we need to do this FFT once on the input sound and an inverse FFT twice to produce the left and right sounds using the overlap-save method. Then we do an N-point FFT on the left and right sounds, multiply by a low pass filter, and then do the inverse N-point FFTs on the products to get the smoothed left and right sounds.

In total we do five FFTs so a naive answer would be that with the C40 chip, we can do all of these in 431 ms or approximately half a second. But the C40 chip does FFTs optimally when they are small so that it can take advantage of using on-chip memory. A 65536-point FFT will take much longer than 65536/1048 * 1.01 ms. However the C40 has hardware and assembly routines to efficiently do low pass filtering.

An added advantage of using C40 chips is that more than one chip can be used so that the signal processing tasks can be run in parallel across all the DSP chips. The TMS320C40 was designed to be easily run in parallel with very little overhead. So for example, if four C40 chips were networked together in parallel, a radix-2 real FFT of 1024 real points will take not much more than 1.01/4 = 0.25 milliseconds.

One can see that by using DSP chips, especially a network of DSP chips, it is realistic to expect that we can implement a real-time system.
Chapter 5

Conclusions

5.1 Results

The EKG program was tested on several different EKG recordings. Three of the more interesting ones are presented in this chapter.

The first EKG we analyzed, EKG 18, is shown in Figure 3-2. We analyzed two other EKGs, EKG 1765 and EKG 94. Their EKG waveforms are shown in Figures 5-1, 5-2. The magnitude of the Fourier transforms of the resultant sounds for these EKGs are shown in Figure 5-3.

EKG 18's stereo sound had a deep bass-like quality and sounded very beaty, like a heartbeat. In comparison, EKG 1765's stereo sound had a somewhat higher pitch, a longer beat, and less dynamic range. EKG 94's stereo sound had the highest pitch of the three sounds, a faster beat, and seemed to move more in sound space with a greater dynamic range.

Notice that the frequency spectrum (see Figure 5-3) is spread out within a certain range due to frequency modulation. As the EKG vector cardiogram moves further away from the origin in the yz plane, the component frequencies of the output sounds become smaller. At the origin the component frequencies of the output sounds are 70, 117, 140, 175, and 351 Hz. But the EKG vector cardiogram will rarely be found at the origin of the yz plane. As one can see in Figure 5-3 EKG 18's sounds have frequencies concentrated around 100 Hz, 150 Hz, and 250 Hz. The 150 Hz and 250
Figure 5-1: EKG 1765.

put EKG here!
Figure 5-2: EKG 94.
Figure 5-3: Plots of the magnitude of the Fourier transforms of the right and left EKG sounds for EKG 18, EKG 1765, and EKG 94. The sampling rate is 44100 Hz. Notice that the frequency spectrum is spread out within a certain range due to frequency modulation. EKG 18’s sounds have frequencies concentrated around 100 Hz, 150 Hz, and 250 Hz. EKG 1765’s sounds have a similar spectrum but its lower and higher frequency components are relatively stronger, resulting in a longer beat and higher pitch. EKG 94’s sounds have many more high frequency components, resulting in a high pitch and fast beat.
Hz components are probably from the 351 Hz component being modulated to a lower value. EKG 1765’s sounds have a similar spectrum but its lower and higher frequency components are relatively stronger, resulting in a longer beat and higher pitch. EKG 94’s sounds have many more high frequency components, resulting in a high pitch and fast beat.

5.2 Conclusions

The goal of this thesis was to convert EKG signals into stereo sound. Based on that goal, we were obviously successful. How useful the sounds we produced remains debatable. Only a cardiologist with an open mind to new techniques or an EKG researcher could truly determine how useful the stereo sound is.

The possibilities of this technique becoming something of clinical value remains somewhat certain. What needs to be done most likely to improve this technique is to work on externalizing the sound, moving it through vertical space (using head motion for example), and improving the sound quality. Cues could also be added to the sound which would signal various heart conditions. The field of utilizing sound to communicate qualitative information still is young but has exciting prospects. This research as well as others can also be used to help design virtual audio display systems, virtual reality game machines, etc.
Appendix A

Program Code

A.1 Horizontal Sweep Using Sound Localization

Theory

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <libmine.h>

#define LENGTH 44100

// global variables
short left[LENGTH+1000], right[LENGTH+1000];

// procedure declarations
void create();
short interpolate(short y1, short y2, double xint, int x1, int x2);

void main() {
```

FILE *fp_rt, *fp_lt;
int count;
char filename[50];
int length = LENGTH;

/********** create horizontal sweep signal ***********/
create();

/***** CREATE OUTPUT SOUND GRAPH FILES *****/
printf("Creating right, and left sound graph files.
");
fp = fopen("Graphs/th_right_h", "wb");
fwrite((char *) &length, sizeof(int), 1, fp);
fwrite((char *) right, sizeof(short), length, fp);
fclose(fp);

fp = fopen("Graphs/th_left_h", "wb");
fwrite((char *) &length, sizeof(int), 1, fp);
fwrite((char *) left, sizeof(short), length, fp);
fclose(fp);

void create() {

    short rt, lt;
    int count, index, r_index = 0, l_index = 0;
    int delay = 0, delta_delay, desired_time_delay;
    double R, L;
    double az;

    for (count = 0; count < LENGTH; count++) {

        az = (-2.0 * M_PI * count / LENGTH) + M_PI;

        /********** calculate right and left amplitude adjustment, 
        (i.e. the IAD or interaural amplitude differences), 
        and delay adjustment (i.e. the ITD or interaural time delay). 
        ***********/
    }
L = (1.0 + sin(az)) / 2.0;
R = (1.0 - sin(az)) / 2.0;

desired_time_delay = (3.0 * 0.10 / 334.0 * sin(az) * 44100.0 + 0.5);

//calculate what goes in right and left ear.

rt = R * (sin(count/60) + sin(count/70) + sin(count/80) +
          sin(count/40) + sin(count/100) + sin(count/20))
     * 4000;

lt = L * (sin(count/60) + sin(count/70) + sin(count/80) +
          sin(count/40) + sin(count/100) + sin(count/20))
     * 4000;

//DETERMINE IF NEED TO DELAY RIGHT OR LEFT SIGNAL

delta_delay = desired_time_delay - delay;
if (delta_delay > 0) {
    for (index = r_index; index < r_index+delta_delay; index++) {
        right[index] = interpolate(right[r_index-1], rt, index,
                                    r_index-1, r_index+delta_delay);
    }
    r_index += delta_delay;
    delay = desired_time_delay;
}
else if (delta_delay < 0) {
    for (index = l_index; index < l_index-delta_delay; index++) {
        left[index] = interpolate(left[l_index-1], lt, index,
                                   l_index-1, l_index-delta_delay);
    }
    l_index += delta_delay;
    delay = desired_time_delay;
}

right[r_index] = rt;
left[l_index] = lt;

r_index++;
l_index++;
in this section, I find the interpolated value between \((x_1, y_1)\)
and \((x_2, y_2)\) at \(x_{\text{int}}\).

```c
short interpolate(short y1, short y2, double xint, int x1, int x2) {
    double slope;
    short intermediate;

    slope = ( (double) (y2 - y1) ) / ( (double) (x2 - x1) );
    intermediate = round_off(slope*(xint - x1) + y1);

    return intermediate;
}
```

### A.2 Horizontal Sweep Using Measured Head-Related Transfer Functions

```c
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <libmine.h>
#include <sys/time.h>
#include <sys/resource.h>
```

```c
/* the sampling rate is 44.1 KHz, so 44100 samples is 1 sec */
#define UNIT_LENGTH 44100

/* there are of course two ears, ear 0 is the right ear */
#define EARS 2

/* the number of responses per ear is 24 */
#define NUM_OF_RESPONSES 24

/* the number of samples in each impulse response is 256 */
#define NUM_OF_SAMPLES 256

/* length of sound slice (the length of the FFT),
answer sound slice, and impulse response ***/

```c
#define N 2048
#define L N*2
#define M 256

/****
** EXTERNAL PROCEDURE **
****/
void myconvlv(float data[], unsigned long n, float H[], float ans[]);
void realft(float data[], unsigned long n, int isign);

/****
** GLOBAL VARIABLES **
****/
float H[ EARS ][ NUM_OF_RESPONSES ][ N ];
short time_delay[ NUM_OF_RESPONSES ];
short *left, *right;
float sound[N];

/****
** PROCEDURES **
****/
void open_impulse_response_file();
void convolve(int length);
void create_sound(float sound[], int start, int end, int period, int index);
void copy_end_to_beginning_f(float x[], int length, int slice);
void slice(float x[], int start, int length, short z[], int index);

/****
** SIGNAL PROCESSING FUNCTIONS **
****/
void zero(short x[], int start, int end);
short interpolate(short y1, short y2, double xint, int x1, int x2);
void lpf(int k, float x[], int length);

/****
** OTHER FUNCTIONS **
****/
int powerof2ceiling(int num);

/****************************
 MAIN
 MAIN
****************************/
void main()
{
    int length;

    printf("How long do you want the sound to be in seconds? ");
    scanf("%d", &length);
    length = powerof2ceiling(length * UNIT_LENGTH);

```
printf("The length is \%d\n", length);

printf("N: \%d, M: \%d, L: \%d\n", N, M, L);
/***** allocate blocks of memory dynamically for large arrays.
     CALLOC also initializes these blocks of
     memory by setting everything to zero.  *****/
left = (short *) calloc(length*2, sizeof(short));
right = (short *) calloc(length*2, sizeof(short));

/*** OPEN THE IMPULSE RESPONSE FILE AND READ IT ****/
open_impulse_response_file();

/***** BEGIN THE NEAT STUFF  *****/  
convolve(length);
}

/********************************************************************************
OPEN_IMPULSE_RESPONSE_FILE
1. open binary map file for reading
2. read in the 24 pairs of FFT of impulse responses,
   each of length 2048.
3. read in the 24 time delays, which are in units of samples,
   sampled at 44.1 kHz.
********************************************************************************/

void open_impulse_response_file() {

FILE *fp;
int az, ear, counter = 0;

fp = fopen("mapoazH.dat","rb"); /* open binary map file for reading */

/********************************************************************************
the outer loop contains the AZIMUTH ANGLE, starting at 180 degrees
(right behind the subject) and decrementing by 15 degrees at a time.
the inner loop contains the EAR number, depending on right
or left.
zero degrees means the sound is right in front of the subject.
there are 24 filter pairs, because there are 24 distinct AZIMUTH
ANGLES \{180, 165, 150, 135, 120, 105, 0, -15, -30, -45, -60, -75, -90, -105, -120, -135, -150, -165\},
and at each point in space, there is a left and right impulse
response. each impulse response array has 2048 elements.
********************************************************************************/
for (az = 180; az > -170; az -= 15) {
    for (ear = 0; ear < EARS; ear++) {
        /****************************************************************************
         * reads from the binary map file 2048 objects of short size     
         * into the filter array. this is the FFT of the impulse response.       
         ****************************************************************************/
        fread((char *)H[ear][counter], sizeof(float), N, fp);
    } /* end ear alternation */
    counter++;
} /* end az alternation */

****************************************************************************
* reads from the binary map file 24 time delays of short size      
* into the time_delay array.                                      
*****************************************************************************
fread((char *)time_delay, sizeof(short), NUM_OF_RESPONSES, fp);
fclose(fp);
}

*****************************************************************************
CONVOLVE
1. do the following loop:
   A. create sound
   B. convolve sound using impulse response FFT.
   C. delay sound appropriately. if delay > 0, delay right.
2. low pass filter the output sounds.
3. save to file output sounds so that PLAY program can use them.
*****************************************************************************

void convolve(int length) {

    FILE *fp;
    float ans[L], Left[length], Right[length];
    double az;
    int r_index = NUM_OF_SAMPLES, l_index = NUM_OF_SAMPLES;
    int i_index;
    int count, index, delay = 0, delta_delay, desired_time_delay;
    int PERIOD = N-M;
    int no2 = length>>1;
    struct rusage start, stop;
    double utime, stime;

    printf("%nConvolving Sound of Length: %d\n", length);

    for (az = 180; az > -170; az -= 15) {
        for (ear = 0; ear < EARS; ear++) {
            
            /****************************************************************************
             * reads from the binary map file 2048 objects of short size     
             * into the filter array. this is the FFT of the impulse response.       
             ****************************************************************************/
            fread((char *)H[ear][counter], sizeof(float), N, fp);
            } /* end ear alternation */
            counter++;
            
        } /* end az alternation */

        /****************************************************************************
         * reads from the binary map file 24 time delays of short size      
         * into the time_delay array.                                      
         ****************************************************************************/
        fread((char *)time_delay, sizeof(short), NUM_OF_RESPONSES, fp);
        fclose(fp);
        
    }
            

            printf("n\nConvolving Sound of Length: %d\n", length);


} 51
/* get CPU time before entering loop */
getrusage(RUSAGE_SELF, &start);

***** create initial slice that gets copied to beginning *****
create_sound(sound, N-M, N-1, PERIOD, -1);
for (count = NUM_OF_SAMPLES; count < length; count += PERIOD) {
    az = -360.0*( (double)count - NUM_OF_SAMPLES)
        /(length - NUM_OF_SAMPLES) + 180.0;

    ***** CREATE THE SINE WAVE SOUND *****
copy_end_to_beginningf(sound, N, M);
create_sound(sound, M, N-1, PERIOD, count/PERIOD);

    ***** DETERMINING INDEX FOR CORRECT IMPULSE RESPONSE AND TIME DELAY *****
    if (az < -165-7)
        i_index = 0;
    else if (az >= 0)
        i_index = -( (int) az + 7)/15 - 12;
    else
        i_index = -( (int) az - 7)/15 - 12;

    ***** CONVOLVE SOUND WITH RIGHT IMPULSE RESPONSE *****
    myconvlv(sound, N, H[ 0 ][ i_index ], ans);

    ***** DETERMINE IF NEED TO DELAY RIGHT SIGNAL *****
    desired_time_delay = time_delay[ i_index ];
    delta_delay = desired_time_delay - delay;
    if (delta_delay > 0) {
        for (index = r_index; index < r_index + delta_delay; index++) {
            right[index] = interpolate(right[r_index-1], ans[256], index,
            r_index-1, r_index + delta_delay);
        }
        r_index += delta_delay;
        delay = desired_time_delay;
    }

    ***** SLICE OUT ANSWER INTO RIGHT SOUND *****
slice(ans, M, PERIOD, right, r_index);

    ***** CONVOLVE SOUND WITH LEFT IMPULSE RESPONSE *****
    myconvlv(sound, N, H[ 1 ][ i_index ], ans);
/***** DETERMINE IF NEED TO DELAY LEFT SIGNAL ****/  
if (delta_delay < 0) {
    for (index = l_index; index < l_index - delta_delay; index++) {
        left[index] = interpolate(left[l_index-1], ans[256], index,  
                                 l_index-1, l_index - delta_delay);
    }
    l_index += delta_delay;  
    delay = desired_time_delay;
}

/***** SLICE OUT ANSWER INTO LEFT SOUND ****/  
slice(ans, M, PERIOD, left, r_index);

    r_index += PERIOD;
    l_index += PERIOD;
}

/***** ZERO OUT GARBAGE AT BEGINNING OF SOUND ****/  
zero(left, M, M+100);
zero(right, M, M+100);

for (count = 0; count < length; count++) {
    Left[count] = (float) left[count] / no2;
    Right[count] = (float) right[count] / no2;
}

/**** printf("FFT and Low Pass of Left, Right\n"); ****/  
realft(Left-1, length, 1);
lpf(64000, Left, length);

realft(Right-1, length, 1);
lpf(64000, Right, length);

/**** printf("Inverse FFT\n"); ****/  
realft(Left-1, length, -1);
realft(Right-1, length, -1);

for (count = 0; count < length; count++) {
    left[count] = Left[count];
    right[count] = Right[count];
}

/* get CPU time after loop is done */  
getrusage(RUSAGE_SELF, &stop);
/* compute elapsed time */
  utime = (stop.ru_utime.tv_sec - start.ru_utime.tv_sec);
  utime += (stop.ru_utime.tv_usec - start.ru_utime.tv_usec) * 1e-6;
  stime = (stop.ru_stime.tv_sec - start.ru_stime.tv_sec);
  stime += (stop.ru_stime.tv_usec - start.ru_stime.tv_usec) * 1e-6;

  printf("%.21f user, %.21f system\n", utime, stime);

/***** CREATE OUTPUT SOUND GRAPH FILES *****/
printf("\nCreating right, and left sound graph files.\n");
fp = fopen("Graphs/rt_h", "wb");
fwrite((char *) &length, sizeof(int), 1, fp);
fwrite((char *) right, sizeof(short), length, fp);
fclose(fp);

fp = fopen("Graphs/lt_h", "wb");
fwrite((char *) &length, sizeof(int), 1, fp);
fwrite((char *) left, sizeof(short), length, fp);
fclose(fp);

/**** CREATE_SOUND ****/
void create_sound(float sound[], int start, int end, int period, int index) {

  int count;
  double i;

  for (count = start; count <= end; count++) {
    i = count + period*index;
    sound[count] = 0.5 * (sin(i/60) + sin(i/70) + sin(i/80) +
                           sin(i/40) + sin(i/100) + sin(i/20));
  }
}

COPY_END_TO_BEGINNING
copies X[LENGTH-SLICE...LENGTH-1] to X[0..SLICE-1]
void copy_end_to_beginning(float x[], int length, int slice) {
    int i;
    for (i = length - slice; i < length; i++) {
        x[i + slice - length] = x[i];
    }
}

 SLICE
copies elements in float array X, starting at X[START] to
X[START+LENGTH-1], into short array Z.

void slice(float x[], int start, int length, short z[], int index) {
    int count;
    for (count = 0; count < length; count++) {
        z[index + count] = x[start + count];
    }
}

 SIGNAL PROCESSING FUNCTIONS

void zero(short x[], int start, int end) {
    int i;
    for (i = start; i <= end; i++)
        x[i] = 0;
}

 ZERO
in the array of shorts X, it sets X[START] to X[END] all to zero.

 INTERPOLATE
in this section, i try to find the interpolate value between (x1, y1)
short interpolate(short y1, short y2, double xint, int x1, int x2) {

double slope;
short intermediate;

slope = ( (double) (y2 - y1) ) / ( (double) (x2 - x1) );
intermediate = round_off(slope*(xint - x1) + y1);

return intermediate;
}

/**
LPF
Modification 1/10/94
I put in a low pass filter subroutine. This zeroes K high end points
**/

void lpf(int k, float x[], int length) {
    int count;
    for (count = length - k; count < length; count++)
        x[count] = 0;
    x[1] = 0; /* value at PI */
}

/**
POWEROF2CEILING
takes an integer and spits out the smallest integer power of 2 greater
than the input integer.
**/

int powerof2ceiling(int num) {
    int count = 0, original = num;
    if (num == 1) return(num);
    while (num != 1) {

num >>= 1;
count ++;
}

num = num << count;

if (num != original)
    num = num << 1;

return(num);

A.3 Producing Three-Dimensional EKG Sound

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <librnine.h>
#include <sys/time.h>
#include <sys/resource.h>

/***
the sampling rate is 44.1 KHz, so 44100 samples is 1 sec ***/
#define UNIT_LENGTH 44100

/*** there are of course two ears, ear 0 is the right ear ***/
#define EARS 2

/*** the number of responses per ear is 24 ***/
#define NUM_OF_RESPONSES 24

/*** the number of samples in each impulse response is 256 ***/
#define NUM_OF_SAMPLES 256

/*** length of sound slice (the length of the FFT), answer sound slice, and impulse response ***/
#define N 2048
#define L (N*2)
#define M 256

/****
EXTERNAL PROCEDURE
****/
void myconvlvs(float data[], unsigned long n, float H[], float ans[], int K);
void realft(float data[], unsigned long n, int isign);
```
GLOBAL VARIABLES

float H[ EARS ][ NUM_OF_RESPONSES ][ N ];
short time_delay[ NUM_OF_RESPONSES ];
short *left, *right;
float sound[N];
short chain_num, y[5], x[4];
short lead[14][5000];
float leads_avg[14];
int leads_length[6];
short u[500], v[500], w[500], r[500];
double theta[500];

PEAK
    - value of most recent peak.
PEAK_INDEX
    - array of time values of when peaks occur.
PEAK_COUNT
    - number of peaks.

short peak = 0, peak_index[20], peak_count = 0;
short peak_align_index[3];
short min_peak_length = 5000;
short peak1_leads_Is, peak1_leads_aVs, peak1_leads_Vs;

procedure declarations

void open_impulse_response_file();
void ekg(FILE *fp);
void scan_in_initial_info( FILE *fp );
void scan_in_row(FILE *fp, short a[]);
void scan_in_digitized_data( FILE *fp );
int check_if_new_leads(int index, int count);
void put_data_into_lead_arrays(int index, short temp[],
                               int flag, int count);
void peak_check(int delta, int index);
void find_min_peak_to_peak_length();
void peak_align();
void vector_cardiogram();
void determine_u_v_w(short count);
void convolve(int length);
void create_sound(float sound[], int start, int end, int period, int index,
    short x, short r);
void copy_end_to_beginning_f(float x[], int length, int slice);
void slice(float x[], int start, int length, short z[], int index);

/*** SIGNAL PROCESSING FUNCTIONS ***/
void zero(short x[], int start, int end);
short interpolate(short y1, short y2, double xint, int x1, int x2);
void lpf(int k, float x[], int length);

/*** OTHER FUNCTIONS ***/
int powerof2ceiling(int num);

/**** OTHER FUNCTIONS ***/
int powerof2ceiling(int num);

/*****************************/
MAIN
/*****************************/
void main(int argc, char *argv[]) {

    FILE *fp;
    int length;

    printf("How long do you want the sound to be in seconds? ");
    scanf("%d", &length);
    length = powerof2ceiling(length * UNIT_LENGTH);
    printf("The length is \d.\n", length);

    printf("N: \d, M: \d, L: \d\n",N,M,L);
    /**** allocate blocks of memory dynamically for large arrays.
    CALLOC also initializes these blocks of
    memory by setting everything to zero. *****/
    left = (short *) calloc(length*2, sizeof(short));
    right = (short *) calloc(length*2, sizeof(short));

    /*** OPEN THE IMPULSE RESPONSE FILE AND READ IT ***/
    open_impulse_response_file();
}
/** READ EKG FILE AND EXTRACT IMPORTANT DATA ***/
if ((fp = fopen(argv[1], "r")) == NULL) EXIT(argv[1]);
/* if ((fp = fopen("18.VEC", "r")) == NULL) EXIT(argv[1]); */
ekg (fp);

/*** BEGIN THE NEAT STUFF ***/
convolve(length);
}

/****************************
OPEN_IMPULSE_RESPONSE_FILE
1. open binary map file for reading
2. read in the 24 pairs of FFT of impulse responses, each of length
   2048.
3. read in the 24 time delays, which are in units of samples,
   sampled at 44.1 kHz.
*********************************

void open_impulse_response_file() {

    FILE *fp;
    int az, ear, counter = 0;

    fp = fopen("mapoazH.dat","rb"); /**< open binary map file for reading */

    /* the outer loop contains the AZIMUTH ANGLE, starting at 180 degrees
       (right behind the subject) and decrementing by 15 degrees at a time.
       the inner loop contains the EAR number, depending on right
       or left.
       zero degrees means the sound is right in front of the subject.
       there are 24 filter pairs, because there are 24 distinct AZIMUTH
       ANGLES {180, 165, 150, 135, 120, 105, 90, 75, 60, 45, 30, 15, 0,
       -15, -30, -45, -60, -75, -90, -105, -120, -135, -150, -165},
       and at each point in space, there is a left and right impulse
       response. each impulse response has 2048 elements.
       */
    for (az = 180; az > -170; az -= 15) {
        for (ear = 0; ear < EARS; ear++) {

            /* reads from the binary map file 2048 objects of short size
               into the filter array. this is the FFT of the impulse response. */

        }
    }
}
fread((char *)H[ear][counter], sizeof(float), N, fp);
} /* end ear alternation */
counter++;
} /* end az alternation */

/**
 reads from the binary map file 24 time delays of short size
 into the time_delay array.
 ******************
fread((char *)time_delay, sizeof(short), NUM_OF_RESPONSES, fp);
fclose(fp);
*/

/**

void ekg(FILE *fp) {

    //******** scan in initial information **********
    scan_in_initial_info(fp);
    printf("initial scan is done!

");

    //******** scan in digitized data **********
    scan_in_digitized_data(fp);
    printf("digitized data scanned in!

");

    //******** find peak alignments **********
    peak_align();

    //******** determine vector cardiogram **********
    vector_cardiogram();
}

/**

void scan_in_initial_info( FILE *fp ) {


short temp[4];
/***************************************************************
scan in zeroth row that has chain number, y4, and two temporaries.
the first temporary should be a 4.
***************************************************************
fscanf(fp, "%hd", &chain_num);
fscanf(fp, "%hd", &y[4]);
fscanf(fp, "%hd", &temp[0]);
fscanf(fp, "%hd", &temp[1]);
printf("Initial Info - Chain Number: %d\n", chain_num);
printf("Initial Info - y[4]: %d\n", y[4]);

/***************************************************************
the first temporary should be a 4. if not something maybe screwy.
***************************************************************
if (temp[0] != 4) {
    printf("The value in the zeroth row, column #2 is not four!\n");
    exit(0);
}

/***************************************************************
scan in first row of y values, second row of x values,
and third row that has just zero values.
***************************************************************
scan_in_row(fp, y);
scan_in_row(fp, x);
scan_in_row(fp, temp);
}

/***************************************************************
SCAN_IN_ROW
this function scans in the four data points into the array.
***************************************************************

void scan_in_row(FILE *fp, short a[]) {  
    int count;

    for (count = 0; count < 4; count++) {
        fscanf(fp, "%hd", &a[count]);
    }
}
SCAN_IN_DIGITIZED_DATA
this function scans in the digitized data into the appropriate
lead array.
******************************************************************************

void scan_in_digitized_data(FILE *fp) {

    int count = 0, index = 0, delta = 0, lead_set_number = 0, t;
    short temp[4];

    /******************************************************************************
    INDEX tracks the actual time of each data point, starting
    at y0 until y4. COUNT is the array index for LEAD[] and is
    reset everytime a new set of lead data is being read. DELTA is
    the array index for LEAD[13]. LEAD_SET_NUMBER
    tracks which set of data we are reading. T is just a temporary
    variable.
    
    lead[13] = long lead II
    ******************************************************************************

    /******************************************************************************
    read in first field of raw digitized data from EKG file row.
    if this field is EOF or zero (which happens near the end of the file)
    or index > y4 then exit loop.
    ******************************************************************************
    while (((fscanf(fp, "%hd", &lead[13][delta]) != EOF)
        && (lead[13][delta] != 0) && ((delta+y[0]) != y[4])) {

        /******************************************************************************
        read in remaining three fields of raw digitized data from
        EKG file row.
        ******************************************************************************
        for (t = 1; t <= 3; t++) {
            fscanf(fp, "%hd", &temp[t]); }

        /******************************************************************************
        check if new leads. if it is then set leads_length for appropriate
        set of leads, reset COUNT and increment LEAD_SET_NUMBER so
        that the data will be put into the next set of lead arrays.
        ******************************************************************************
    }
index = delta + y[0];
if (index == y[lead_set_number+1]) {
    lead_set_number++;
    leads_length[lead_set_number] = count;
    count = 0;
}

put data, normalized to zero axis, into appropriate lead arrays.

put_data_into_lead_arrays(delta, temp, lead_set_number, count);

count++; delta++;}

/* printf("Done: %d %d %d %d %d\n", lead[13][delta], temp[1],
temp[2], temp[3], count, index); */

leads_length[4] = count;
leads_length[5] = delta;

/*
for (t = 1; t <= 5; t++) {
    printf("leads length #\%d: %d\n", t, leads_length[t]);
    printf("leads #\%d average: %f\n", t, leads_avg[t]);
}
*/

find_min_peak_to_peak_length();

// index = delta + y[0];
if (index == y[lead_set_number+1]) {
    lead_set_number++;
    leads_length[lead_set_number] = count;
    count = 0;
}

put_data, normalized to zero axis, into appropriate lead arrays.

put_data_into_lead_arrays(delta, temp, lead_set_number, count);

count++; delta++;}

/* printf("Done: %d %d %d %d %d\n", lead[13][delta], temp[1],
temp[2], temp[3], count, index); */

leads_length[4] = count;
leads_length[5] = delta;

/*
for (t = 1; t <= 5; t++) {
    printf("leads length #\%d: %d\n", t, leads_length[t]);
    printf("leads #\%d average: %f\n", t, leads_avg[t]);
}
*/

find_min_peak_to_peak_length();

/*
for (t = 1; t <= 5; t++) {
    printf("leads length #\%d: %d\n", t, leads_length[t]);
    printf("leads #\%d average: %f\n", t, leads_avg[t]);
}
*/

find_min_peak_to_peak_length();

/*
for (t = 1; t <= 5; t++) {
    printf("leads length #\%d: %d\n", t, leads_length[t]);
    printf("leads #\%d average: %f\n", t, leads_avg[t]);
}
*/

find_min_peak_to_peak_length();

/*
for (t = 1; t <= 5; t++) {
    printf("leads length #\%d: %d\n", t, leads_length[t]);
    printf("leads #\%d average: %f\n", t, leads_avg[t]);
}
*/

find_min_peak_to_peak_length();

/*
for (t = 1; t <= 5; t++) {
    printf("leads length #\%d: %d\n", t, leads_length[t]);
    printf("leads #\%d average: %f\n", t, leads_avg[t]);
}
*/

find_min_peak_to_peak_length();

64
data among the different leads as well as average the lead values of each array.

************ lead II long ************
lead[13][delta] -= x[0];

peak_check(delta, delta + y[0]);
leads_avg[13] = (leads_avg[13]*delta + lead[13][delta])/(delta+1);

************ lead III, aVF, V3, V6 ************
lead[lead_set_number*3 + 3][count] = temp[1] - x[1];

************ lead II, aVL, V2, V5 ************
lead[lead_set_number*3 + 2][count] = temp[2] - x[2];

************ lead I, aVR, V1, V4 ************
lead[lead_set_number*3 + 1][count] = temp[3] - x[3];

for (c = lead_set_number*3 + 1; c <= (lead_set_number*3 + 3); c++) {
    leads_avg[c] = (leads_avg[c]*count + lead[c][count])/(count+1);
}

PEAK_CHECK
to synchronize all the leads, i use the long lead II which stretches all the way through the samples. i try to find the peaks for each heartbeat.
to find the peaks, i go thru two screens.
1. if the lead II value is greater than the last peak, within 40 units, and this value is greater than the last value (ie. a positive lead II slope), there is a strong positive slope, and that the value is greater than zero.
2. final check: make sure that either this value is larger than the previously found peak or that it is not local to the previous peak in the time period of the heartbeat.

************
void peak_check(int delta, int index) {

/****** first screen *******/
if ( (lead[13][delta] >= peak-40 ) &&
(lead[13][delta] > lead[13][delta-1]) &&
(lead[13][delta] > lead[13][delta-2] + 5) &&
(lead[13][delta] > lead[13][delta-5] + 35) &&
(lead[13][delta] >= 0) ) {

/***** second screen: check that it is definitely a new peak ******/
if (((lead[13][delta] >= peak) || (peak_index[peak_count] + 20 < index)) {
    peak = lead[13][delta];

    /***** if not local then increase peak_count ******/
    if (peak_index[peak_count] + 20 < index) {
        peak_count++;
    }

    peak_index[peak_count] = index;

    printf("peak value: %d at index: %d", peak, index);
    printf(" old value is: %d, peak count is: %d\n",
            lead[13][delta-1], peak_count);
}
}

/*****************************/
FIND_MIN_PEAK_TO_PEAK_LENGTH
finds the smallest length difference between the peaks.
/*****************************/

void find_min_peak_to_peak_length() {

    int length, count;

    count = 1;
    while (peak_index[count] != 0) {
        length = peak_index[count] - peak_index[count-1];
        if (length < min_peak_length)
            min_peak_length = length;
        count++;
    }
}

/*****************************/
PEAK_ALIGN
checks the divide line between the first set of lead data and
the second set of lead data. this divide line is slightly shifted
between the different sets of data. for example, the divide line
between lead I and lead aVR is not in the same place as the divide
line between lead II and lead aVL.
this procedure compares their shifts relative to each other are
and keeps this information for later when it shifts lead data back
and forth.

```c
void peak_align() {
    int a, b;
    for (a = 0; a < 3; a++) {  
        peak_align_index[a] = 0;
        peak = 0;

        /* assuming the divide line peak is within 50 of where
         the divide line should be */
        for (b = 0; b < 50; b++) {
            if (lead[a+4][b] > peak) {
                peak = lead[a+4][b];
                peak_align_index[a] = b;
            }
        }
        printf("%d: peak align index %d\n", a, peak_align_index[a]);
    }
}
```

VECTOR_CARDIOGRAM
figure out the EKG vector cardiogram in the vertical plane.

```c
void vector_cardiogram() {
    short count;
    /* first peak after y0 */
    peak1_leads_Is = peak_index[1] - y[0];
```
look for first peak after y1

count = 2;
while (peak_index[count] < y[1]) {
    count++;
}
peak1_leads_aVs = peak_index[count] - y[1];

look for first peak after y2

while (peak_index[count] < y[2]) {
    count++;
}
peak1_leads_Vs = peak_index[count] - y[2];

determine u and v and w

for (count = 0; count <= min_peak_length; count++) {
    determine_u_v_w(count);
}

DETERMINE_U_V_W

New Way:
x axis is lead V2, y axis is lead I, z axis is - lead aVF
theta is the angle on the yz plane. theta = atan(-y/z).

| Z |
|   |
|   |
----------> Y

r is the length on the yz plane.

void determine_u_v_w(short count) {

double ratio;

    u[count] = lead[8][peak1_leads_Vs + count + peak_align_index[1]];
    v[count] = lead[1][peak1_leads_Is + count + peak_align_index[0]];
    w[count] = -lead[6][peak1_leads_aVs + count + peak_align_index[2]];

    ratio = - ((double) v[count]) / ((double) w[count]);
    theta[count] = atan(ratio);
/**** 2nd quadrant including negative z axis but not y axis *****/
if ((-v[count] >= 0) && (w[count] < 0))
    theta[count] = theta[count] + M_PI;
/**** 3rd quadrant; no axes *****/
else if ((-v[count] < 0) && (w[count] < 0))
    theta[count] = theta[count] - M_PI;

r[count] = sqrt( sqr(v[count]) + sqr(w[count]) );

/****************************
CONVOLVE
1. do the following loop:
   A. create sound
   B. convolve sound using impulse response FFT.
   C. delay sound appropriately. if delay > 0, delay right.
2. low pass filter the output sounds.
3. save to file output sounds so that PLAY program can use them.
****************************/

void convolve(int length) {

FILE *fp;
float ans[L], Left[length], Right[length];
int r_index = NUM_OF_SAMPLES, l_index = NUM_OF_SAMPLES;
int i_index;
int ekg_index;
int count, index, delay = 0, delta_delay, desired_time_delay;
int PERIOD = N-M;
int no2 = length>>1;
int lpf_factor;
struct rusage start, stop;
double utime, stime, az;

/** the EKG signals are sampled 306.1 Hz and will be played
   back at 44100 Hz. upscale factor is approximately 144. */
int upscale_factor = 144;

printf("\n\nConvolving Sound of Length: %d.\n", length);

/* get CPU time before entering loop */
getrusage(RUSAGE_SELF, &start);
/**** create initial slice that gets copied to beginning ****/
create_sound(sound, N−M, N−1, PERIOD, −1, u[0], r[0]);
for (count = NUM_OF_SAMPLES; count < length; count += PERIOD) {
  ekg_index = (upscale_factor − 1 + count)/(upscale_factor) % (min_peak_length + 1);
  az = theta[ekg_index];

/**** CREATE THE SINE WAVE SOUND ****/
copy_end_to_beginning_f(sound, N, M);
create_sound(sound, M, N−1, PERIOD, count/PERIOD, u[ekg_index], r[ekg_index]);

/**** DETERMINING INDEX FOR CORRECT IMPULSE RESPONSE AND TIME DELAY ****/
if (az < −165−7)
  i_index = 0;
else if (az >= 0)
  i_index = −((int) az + 7)/15 − 12;
else
  i_index = −((int) az − 7)/15 − 12;

/** CONVOLVE SOUND WITH RIGHT IMPULSE RESPONSE **/
myconvlv(sound, N, H[ 0 ][ i_index ], ans, 0);

/**** DETERMINE IF NEED TO DELAY RIGHT SIGNAL ****/
desired_time_delay = time_delay[ i_index ];
delta_delay = desired_time_delay − delay;
if (delta_delay > 0) {
  for (index = r_index; index < r_index + delta_delay; index++) {
    right[index] = interpolate(right[r_index−1], ans[256], index, r_index−1, r_index + delta_delay);
  }
  r_index += delta_delay;
  delay = desired_time_delay;
}

/**** SLICE OUT ANSWER INTO RIGHT SOUND ****/
slice(ans, M, PERIOD, right, r_index);

/**** CONVOLVE SOUND WITH LEFT IMPULSE RESPONSE ****/
myconvlv(sound, N, H[ 1 ][ i_index ], ans, 0);

/**** DETERMINE IF NEED TO DELAY LEFT SIGNAL ****/
if (delta_delay < 0) {

70
for (index = l_index; index < l_index - delta_delay; index++) {
    left[index] = interpolate(left[l_index-1], ans[256], index,
    l_index-1, l_index - delta_delay);
}

l_index += delta_delay;
delay = desired_time_delay;
}

/***** SLICE OUT ANSWER INTO LEFT SOUND ****/
slice(ans, M, PERIOD, left, l_index);

r_index += PERIOD;
l_index += PERIOD;
}

/***** ZERO OUT GARBAGE AT BEGINNING OF SOUND ****/
zero(left, M, M+100);
zero(right, M, M+100);

/**** Copy From Short to Float and Scale For Inverse FFT ****/
for (count = 0; count < length; count++) {
    Left[count] = (float) left[count] / no2;
    Right[count] = (float) right[count] / no2;
}

/**** FFT and Low Pass of Left, Right ****/
lpf_factor = (double) length * 0.985;
realft(Left-1, length, 1);
lpf(lpf_factor, Left, length);
realft(Right-1, length, 1);
lpf(lpf_factor, Right, length);

/**** Inverse FFT ****/
realft(Left-1, length, -1);
realft(Right-1, length, -1);

/**** Copy From Float to Short ****/
for (count = 0; count < length; count++) {
    left[count] = Left[count];
    right[count] = Right[count];
}

/* get CPU time after loop is done */
getrusage(RUSAGE_SELF, &stop);
/* compute elapsed time */
    utime = (stop.ru_utime.tv_sec - start.ru_utime.tv_sec);
    utime += (stop.ru_utime.tv_usec - start.ru_utime.tv_usec) * le-6;
    stime = (stop.ru_stime.tv_sec - start.ru_stime.tv_sec);
    stime += (stop.ru_stime.tv_usec - start.ru_stime.tv_usec) * le-6;
    printf("%.2f user, %.2f system
", utime, stime);

/***** CREATE OUTPUT SOUND GRAPH FILES *****/
    printf("\nCreating right, and left sound graph files.\n");
    fp = fopen("Graphs/rt_ekg", "wb");
    fwrite((char *) &length, sizeof(int), 1, fp);
    fwrite((char *) right, sizeof(short), length, fp);
    fclose(fp);
    fp = fopen("Graphs/lt_ekg", "wb");
    fwrite((char *) &length, sizeof(int), 1, fp);
    fwrite((char *) left, sizeof(short), length, fp);
    fclose(fp);

/*************************************************************/
    CREATE_SOUND
1. produce an array of sine wave values. the sine wave frequency
    is modulated by the horizontal distance, r, from the origin,
    so that there is a lower pitch for sound further away. also
    the amplitude of the sound is modulated by the vertical distance, x,
    so that the further away, the quieter the sound.
    ************************************************************/

void create_sound(float sound[], int start, int end, int period, int index,
    short x, short r) {

    int count;
    double i = 0;

    /**** produce an array of sine wave values *****/
    for (count = start; count <= end; count++) {
        i = count + period*index;
        sound[count] = (x + 190.0) * (sin(i/(40.0+r)) + sin(i/(50.0+r))
            + sin(i/(80.0+r)) + sin(i/(70.0+r))
            + sin(i/(100.0+r)) + sin(i/(20.0+r)))/ 230.0;
```c
void copy_end_to_beginning_f(float x[], int length, int slice) {
    int i;
    for (i = length - slice; i < length; i++) {
        x[i + slice - length] = x[i];
    }
}

void slice(float x[], int start, int length, short z[], int index) {
    int count;
    for (count = 0; count < length; count++) {
        /*
           printf("%d ", count);
           printf("input[%d] = ", start+ count);
           printf("%4.1f, ", x[start+ count]);
           printf("output[%d] = ", index+ count);
           printf("%d\n", z[index+count]);
        */
        z[index + count] = x[start + count];
    }
}
```

/**** COPY_END_TO-BEGINNING ****/

COPY_END_TO_BEGINNING
copies X[LENGTH-SLICE...LENGTH-1] to X[0..SLICE-1]

/***** SLICE ******/
SLICE
copies elements in float array X, starting at X[START]
to X[START+LENGTH-1], into short array Z.

/******* SIGNAL PROCESSING FUNCTIONS *********/

/******* ZERO *********/
in the array of shorts X, it sets X[START] to X[END] all to zero.

```c
void zero(short x[], int start, int end) {
    int i;
    for (i = start; i <= end; i++)
        x[i] = 0;
}
```

/*_STREAMLINE*/

/*INTERPOLATE*/
in this section, i try to find the interpolate value between \((x_1, y_1)\) and \((x_2, y_2)\) at \(x_{int}\).

```c
short interpolate(short y1, short y2, double xint, int x1, int x2) {
    double slope;
    short intermediate;
    slope = ( (double) (y2 - y1) ) / ( (double) (x2 - x1) );
    intermediate = round_off(slope*(xint - x1) + y1);
    return intermediate;
}
```

/*LPF*/

Modification 1/10/94
I put in a low pass filter subroutine. This zeroes \(K\) high end points

```c
void lpf(int k, float x[], int length) {
    int count;
    for (count = length - k; count < length; count++)
        x[count] = 0;
    x[1] = 0; /* value at \(PI\) */
}
```
int powerof2ceiling(int num) {
    int count = 0, original = num;

    if (num == 1) return(num);

    while (num != 1) {
        num >>= 1;
        count ++;
    }

    num = num << count;

    if (num != original)
        num = num << 1;

    return(num);
}

A.4 Playing Binary Sound Files

#include <stdio.h>
#include <math.h>
#include <adio.h>

#define UNIT_LENGTH 44100
#define NUM_OF_SAMPLES 256

/****     GLOBAL VARIABLES ****/
short *left, *right;

/****     file descriptor integer, FD ****/
int fd;

/****     chg is a pointer to a channel group structure (CH_G) ****/
CH_G *chg;

/****     params is a structure that contains audio parameters ****/
struct ADIO_PARAMS params;

/* PROCEDURES */
int read_length(FILE *fp);
void read_sound(FILE *fp, short x[], int length);
void play_sound();
void play_loop(int length);
void check_play();

**************************************************************************
MAIN
**************************************************************************

void main()
{
    char filepath[50], filename[50];
    FILE *fp_right, *fp_left;
    int length;

    /* OPEN RIGHT SOUND FILE */
    strcpy(filepath, "/home/nul/frankkim/EKG/Graphs/");
    printf("Filename of the right sound file: ");
    scanf("%s", filename);
    strcat(filepath, filename);
    if ((fp_right = fopen(filepath,"rb"))== NULL) EXIT(filename);

    /* DETERMINE LENGTH OF SOUND FILES */
    fread((char *) &length, sizeof(int), 1, fp_right);
    printf("length is %d which is in seconds %d\n", length,length/UNIT_LENGTH);

    /* OPEN LEFT SOUND FILE */
    strcpy(filepath, "/home/nul/frankkim/EKG/Graphs/");
    printf("Filename of the left sound file: ");
    scanf("%s", filename);
    strcat(filepath, filename);
    if ((fp_left = fopen(filepath,"rb"))== NULL) EXIT(filename);
    fread((char *) &length, sizeof(int), 1, fp_left);

    /* ALLOCATE BLOCKS OF MEMORY DYNAMICALLY FOR LARGE ARRAYS */
    left = (short *) calloc(length, sizeof(short));
    right = (short *) calloc(length, sizeof(short));
**** READ IN RIGHT AND LEFT SOUND FILES ****
printf("\nreading sound\n");
read_sound(fp_right, right, length);
read_sound(fp_left, left, length);

**** PLAY SOUNDS THROUGH STEREO HEADPHONES ****
printf("\nplaying sound of length %d\n", length);
play_sound(length);
}

REP SOUND
1. reads binary sound value element into array of shorts.
*************************************************************************

void read_sound(FILE *fp, short x[], int length) {
    fread((char *)x, sizeof(short), length, fp);
}

PLAY SOUND
1. set up to begin playing.
2. plays the sinewave by calling PLAY_LOOP.
*************************************************************************

void play_sound(int length) {

*************************************************************************
in this section, the following things are done:
1. ADIO_HOST, the Sun's DBRI (Dual Basic Rate ISDN Interface), is
opened using ADIO_OPEN. The argument 0 is irrelevant since the
ariel/proport system is not being used. ADIO_OPEN returns a
file descriptor integer, FD.
2. ADIO_GET_PARAMS uses the file descriptor to put the
audio parameters in the structure PARAMS.
3. The output device is set to the Headphones. Sampling rate set to
44100 Hz.
4. Playback function is started.
5. CHG_NEW(2) allocates two channels in the channel group structure
CHG for stero playback.
6. Play the sound continuously.
*************************************************************************
fd = adio_open(ADIO_HOST, 0);
adio_get_params(fd, &params);
params.output_device = "headphone";
params.sampling_rate_hz = 44100;
adio_set_params(fd, &params);
adio_start_play(fd);
chg = chg_new(2);

while(1) {
    play_loop(length);
    usleep(500000);
}

/***************************************************************************/
PLAYLOOP
1. send sound wave to headphone buffer.
2. check if playing of sound wave is complete.
***************************************************************************/

void play_loop(int length) {

int ns = NUM_OF_SAMPLES, inc;

/***************************************************************************/
Below is a loop to write the sinewave to the headphone buffer where it
is then extracted and played. It does the following:
1. A counter, i, is set to zero. This is repeatedly compared to
   the LENGTH of the array, to determine if writing to the buffer
   is complete.
2. Using CHG_SET, the data in left and right arrays are set to the
   left and right parts of the headphone output. left+ns+i is the
   address of left[ns+i].
3. Send a certain number of data using ADIO_WRITE which returns
   the number of shorts that have been written to the output.
4. The ADIO software grabs the stuff in the buffer and plays it.
5. Increment the NS counter by the amount returned by ADIO_WRITE.
6. USLEEP(10000) suspends execution for 10000 microseconds to give
   plenty of time to the ADIO software for grabbing and playing
   the sinewave.
***************************************************************************/

while(ns < length) {
A.5 Convolution using FFT

The code for this program is a modification of the original convolution code presented in Numerical Recipes in C. This code takes advantage of the fact that since the impulse responses were known ahead of time, we could calculate the FFT’s of the impulse responses beforehand and save them to a file for later use [10].

This was tested December 6, 1993 for correctness, versus the convolution program from Numerical Recipes and found to run perfectly. However this program runs faster since it gets...
as an input the FFT of the impulse response, which is stored on
disk, and therefore saves the times of having to compute it.

**********************************************************************

MYCONV\_LV
Convolves a real data set DATA\[0...N-1\] (including any user-supplied
zero padding) with a response function.
The FFT of the response function, H\[0...N-1\] is given in the
format outputted by REALFT (ie. only the positive frequency half of the
complex spectrum is given with the real-value last component, ie.
the value at theta = PI, found at H[1]. the real-value first
component is at H[0].)
The FFT of DATA is found using REALFT and placed in FFT.
Finally, H and FFT are multiplied taking into account the
structure of H and FFT. The product is stored in ANS\[0...N-1\].
This product is then sent to REALFT for an inverse FFT.
ANS, now the circular convolution, is returned.
N MUST be an integer power of two.
**********************************************************************

void myconvlv(float data[], unsigned long n, float H[], float ans[])
{
    void realft(float data[], unsigned long n, int isign);
    unsigned long i,no2;
    float fft[n];

    for (i = 0; i < n; i++)
    fft[i] = data[i]; /* copy DATA into FFT array */

realft(fft-1,n,1);

    no2=n>>1;

    for (i=2; i< n; i+=2) {
        /* real part */
        ans[i] = (fft[i]*H[i] - fft[i+1]*H[i+1])/no2;
        /* imaginary part */
        ans[i+1] = (fft[i+1]*H[i] + fft[i]*H[i+1])/no2;
    }

    /* special cases for the first and last real-valued components */
    ans[0] = (fft[0]*H[0] - fft[1]*H[1])/no2;
    ans[1] = (fft[1]*H[0] + fft[0]*H[1])/no2;
}

}
ans[0] = fft[0] * H[0]/no2;

realft(ans-1,n,-1); /** Inverse transform back to time domain. **/

A.6 FFT of Real Signal

The code for this program comes from Numerical Recipes in C with added personal comments inserted [10].

#include <math.h>

/************************************************************************
REALFT
If ISIGN = 1:
Calculates the Fourier Transform of a real data set DATA[1...N]
(including any user—supplied zero padding) and returns the
positive frequency half of its complex Fourier Transform
in DATA[1...N]. DATA[1...N] stores the complex data
by alternating real and imaginary parts. However DATA[2]
contains the real valued last component, i.e. the value
at theta = PI.
If ISIGN = -1”
Calculates the inverse Fourier Transform, scaled by a factor
of n/2.
************************************************************************/

void realft(float data[], unsigned long n, int isign)
{
    void fourl(float data[], unsigned long nn, int isign);
    unsigned long i,i1,i2,i3,i4,np3;
    float c1=0.5,c2,h1r,h1i,h2r,h2i;
    /** Double precision for the trionometric recurrences ***/
    double wr,wi,wpr,wpi,wtemp,theta;

    /** Initialize the recurrence ***/
    theta=3.141592653589793/(double) (n>>1);

    if (isign == 1) {
        /** Do forward Fourier transform **/
        c2 = -0.5;
    }
fourl(data,n>>1,1);

} else {
    /** Otherwise set up for an inverse Fourier transform **/  
c2=0.5;
    theta = -theta;
}

wtemp=sin(0.5*theta);
wpr = -2.0*wtemp*wtemp;
wpi=sin(theta);
wr=1.0+wpr;
wi=wpi;
np3=n+3;

for (i=2;i<=(n>>2);i++) {
    /** Case i=1 done separately below **/  
i4=1+(i3=np3-(i2=1+(i1=i+i-1));

    /** The two separate transforms  
are separated out of data. **/  
    hlr=cl*(data[i1]+data[i3]);
    hli=cl*(data[i2]-data[i4]);
    h2r = -c2*(data[i2]+data[i4]);
    h2i=c2*(data[i1]-data[i3]);

    /** Here they are recombined to form the  
true transform of the original real data. **/  
data[i1]=hlr+wr*h2r-wi*h2i;
data[i2]=hli+wr*h2i+wi*h2r;
data[i3] =hlr-wr*h2r+wi*h2i;
data[i4] = -hli+wr*h2i+wi*h2r;

    /** The recurrence. **/  
    wr=(wtemp=wr)*wpr-2.0*wtemp*wpi+wr;
    wi=wi*wpr+wtemp*wpi+wi;
}

if (isign == 1) {
    /** Squeeze the first and last data together  
to get them all within the original array. **/  
data[1] = (hlr=data[1])+data[2];
} else {
    data[1]=cl*((hlr=data[1])+data[2]);
A.7 Fast Fourier Transform (FFT)

The code for this program comes from Numerical Recipes in C with added personal comments inserted [10].

```c
#include <math.h>
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

void fourl(float data[], unsigned long nn, int isign)
{
    unsigned long n,mmax,m,j,istep,i;
    double wtemp,wr,wpr,wpi,wi,theta; /* Double precision for the *
    * trignometric recurrences */
    float tempr,tempi;

    n=nn << 1;

    if (j > i) {
        SWAP(data[j],data[i]); /* Exchange the two */
        SWAP(data[j+l],data[i+l]); /* complex numbers. */
    }
    m=n >> 1;
    while (m >= 2 && j > m) {
        j -= m;
        m >>= 1;
    }

    /******** This is the bit—reversal section of the routine. ****/
    j=1;
    for (i=1;i<n;i+=2) {
        if (j > i) {
            SWAP(data[j],data[i]); /* Exchange the two */
            SWAP(data[j+1],data[i+1]); /* complex numbers. */
        }
    }
```

/** This is the inverse transform
   for the case isign = -1. **/
fourl(data,n>>1,-1);

```
Here begins the Danielson—Lanczos section of the routine. */
mmax=2;

/*** Outer loop executed log2(nn) times. ***/
while (n > mmax) {
    istep=mmax << 1;

    /*** Initialize for the trigonometric recurrence. ***/
    theta = isign*(6.28318530717959/mmax);
    wtemp = sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi = sin(theta);
    wr = 1.0;
    wi = 0.0;

    /*** Here are the two nested inner loops. ***/
    for (m=1;m<mmax;m+=2) {
        for (i=m;i<n;i+=istep) {
            /* Danielson—Lanczos formula */
            j = i + mmax;
            tempr = wr*data[j] - wi*data[j+1];
            tempi = wr*data[j+1] + wi*data[j];
            data[j] = data[i] - tempr;
            data[j+1] = data[i+1] - tempi;
            j = i + mmax;
            data[i] += tempr;
            data[i+1] += tempi;
        }
    }

    /*** Trigonometric recurrence ***/
    wr = (wtemp=wr)*wpr - wi*wpi+wr;
    wi = wi*wpr + wtemp*wpi + wi;

    mmax = istep;
}

#define SWAP

84
Bibliography


