2007

Audio watermarking techniques using singular value decomposition

Joseph Kardamis

Rochester Institute of Technology
Master of Science in Computer Science Thesis

Audio Watermarking Techniques using Singular Value Decomposition

By: Joseph R. Kardamis

June 5, 2007

Signature Block

Author: Joseph R. Kardamis Date

Chair: W. Carithers Date

Reader: J. Biles Date

Observer: S. Radziszowski Date

Acknowledgements

Thanks to Warren Carithers for guidance throughout the course of this work.
Thanks to Hamza Özer for help in implementing his system.
Love to Liz and thanks for tolerating my crazy-person ramblings.

Abstract

In an increasingly digital world, proving ownership of files is more and more difficult. For audio files, many schemes have been put into place to attempt to protect the rights of the digital content owners. In general, these techniques fall under the classification of Digital Rights Management (DRM). Audio watermarking is one of the less invasive schemes which embeds security into the data itself instead of in an outside layer meant to encapsulate and protect the data.

There are many domains in which an audio watermark can be applied. The simplest is that of the time domain; often, however, other domains may be more desirable due to greater imperceptibility and robustness to attack. Common domains include the frequency domain, or domains similar to frequency through functions such as the Wavelet Transform. One domain of particular interest is that of the Singular Value Decomposition.

The goal of this thesis is to propose and test many different watermarking schemes as well as test an existing watermarking scheme operating in the SVD domain in order to assess the viability of the SVD as a watermarking carrier domain. Different carrier matrices as well as bit embedding methods are explored.

The use of a standard set of audio files was used to help test the systems; a standard set of watermarking tests was unavailable, so a comparable test bed was implemented and utilized.

Contents

List of Tables

1 Introduction

As the proliferation of digital media increases, digital content owners will continue to attempt to enforce their ownership and copyright using various means. These means are collectively known as Digital Rights Management (DRM). Many DRM systems are invasive or limit the usability of the files that they are attempting to protect. One such family of DRM techniques involves the use of rootkits to attempt to prevent unauthorized copying or to prevent other illegal activities, but the use of such systems is of a dubious ethical nature. Many others involve encrypting data to limit the devices on which the media can be used. These can be bypassed should the data be decrypted, however, be it through the normal usage of the file, or by breaking the encryption.

Still other techniques, such as audio watermarking, exist which do not attempt to limit the functionality or usability of a file, but rather attempt to help prove ownership. These techniques are far less invasive, and the protection lies within the file itself, not in a layer around the file. Audio watermarking involves altering bits in an audio file such that a message can be embedded in the audio file and later retrieved. It is important that the watermark cannot easily be removed or destroyed, so new domains in which the watermark can be embedded and is more resilient are heavily sought after. One such domain which holds promise and therefore merits further investigation is that of the Singular Value Decomposition (SVD).

The format of this paper is as follows: In section 2, requisite background material will be more fully introduced and explained. In section 3, related work in audio watermarking will be covered. Section 4 discusses the design of the systems that were implemented for this work. Section 5 will cover the attacks used to evaluate these systems, and in section 6 the results of said attacks will be analyzed. The results of the attacks are presented in section 7, and future work that is still to be done is discussed in section 8.

2 Background

2.1 Trends in Digital Content

One of the major trends in the recording industry and the media industries in general is the availability of content in a digital form rather than strictly physical hardcopies. Pictures can be taken and shared digitally via removable media such as SmartCards or via picture sharing websites such as WebShots or Picasa. Broadcasts of television shows are available for viewing at network websites, often at no charge, which allow anyone to watch their programs without schedule restrictions. Single songs and entire albums can be purchased in digital form through the Internet as well, most notably through services such as Apple's iTunes and Napster. Advances in software and hardware devices have allowed for the extraction of media from physical forms to purely digital forms.

These trends have major impacts on the content owners due to the nature of digital

media. Analog forms of media (such as cassette tapes and VHS) record their information on a piece of physical media, which can degrade in quality after extensive use. Coupled with the cost of materials, this loss in quality due to copying made large-scale illegal distribution infeasible. These restrictions do not apply to most digital media, however. Digital media are simply sequences of bits ordered in a specific way, and can be copied without any degradation in signal or corruption of quality. This ease of copying, as well as ease of distribution through methods such as file sharing, has made problems such as large scale copyright infringement far easier. It is for these reasons that copyright holders are looking to find ways to protect their intellectual property from illegal copying.

2.2 Digital Rights Management

Digital Rights Management, as discussed above, attempt to limit the usage of digital files in order to enforce copyright. Most DRM systems today are invasive and restrictive, in that they attempt to embed themselves into a computer in order to prevent any potentially illegal actions done with the media. One particularly invasive technique found on CDs made under the Sony/BMG label used technology called XCD. These CDs, when inserted into the computer, installed a rootkit on the system without the user's knowledge. Among the functions that this rootkit performed without the user's knowledge or agreement was "phoning home." The rootkit monitored the playing of the CD and contacted Sony servers regarding usage. To remain hidden, it modified system calls in order to hide files and processes. However, instead of hiding a specific list of processes and files for use only by the rootkit (a direct manifest), the program hid all files and processes beginning with a "magic bullet" prefix. This opened affected systems to potentially large security risks, as this "magic bullet" could then be exploited by other outside attackers. Malicious programs could install their own files and run their own processes under the cover of the XCD rootkit [14, 18].

Apple's iTunes allows users to purchase songs encoded in Advanced Audio Coding (AAC) format using Apple's FairPlay system. These songs can then be transferred to a FairPlay-compliant player (currently only other Apple iPod devices), or burned to CD. This restricts the range of audio devices users can use to listen to their music; however, no copy or access restrictions are transferred with the file once it has been burned to disc [6]. Data in AAC files are encrypted; however, to burn the CD the data must then be decrypted, and as the only method of protection is through the encryption; once the CD is burned the protection of the data is lost. The song could then theoretically be ripped from the CD and distributed.

High Definition DVD (HD DVD) and Blu-ray both protect their content through hardware authentication and data encryption as well, using High-bandwidth Digital Content Protection (HDCP) for the former and implementations of the Advanced Access Content System (AACS) for the latter. If the hardware is confirmed as being valid, the disc passes its decryption key to the hardware player, allowing it to play the media. If the decryption

key is extracted, it is possible to bypass the protection for the disc. This does not compromise the entire AACS system, but rather compromises specific titles using the AACS system; as the titles are pressed and immutable after manufacture, compromising a single title cannot be undone [10]. However, using HDCP, it is possible to revoke the playback functionality for hardware caught transmitting non-encrypted high definition content [3, 4].

2.3 Audio Watermarking

Digital watermarking is a technique by which the bits of a digital carrier, often a piece of media, are altered in a verifiable fashion. The idea parallels that of physical paper watermarking. A paper watermark can be seen when held up to the light, thereby authenticating the source of the paper. The watermark cannot be removed without destroying the carrier (paper), and the presence of the watermark does not obscure or otherwise hinder the normal usage of the media. Digital watermarking attempts to alter bits so that the media is not perceptibly altered, but the watermark can be detected.

There are many ways by which a watermark can be evaluated, but there are three major criteria used: transparency, robustness, and capacity. *Transparency* measures how perceptible or imperceptible a watermark is. This is either measured with human participants of a survey, or by attempting to model the Human Auditory System (HAS) to get an easily repeatable measure of the effects of the watermark on the audio quality. *Robustness* denotes the watermark's resistance to modification, either benign or malicious. This is tested by embedding watermarks in a variety of audio carrier signals and subsequently altering the file in some fashion, such as by adding noise or by changing the amplitude. *Capacity* marks how much information can be stored in a watermark. This is determined by the design of the watermarking algorithm [20]. Often these three criteria are linked, such that in order to improve robustness transparency must be sacrificed by making the watermark presence stronger, or capacity is sacrificed by adding redundancy, thereby increasing the required space for each bit. Similarly, increasing transparency generally negatively impacts robustness, and increasing capacity can hinder transparency.

Other important criteria used to note the effectiveness or usefulness of a watermark include: the processing time required to embed a watermark, the processing time required to extract the watermark, and the amount of information required to extract the watermark. Schemes that require no knowledge of the original unmarked file are known as "oblivious" or "blind". Systems that require no information specific to the original file, but do require extra knowledge beyond the marked file (such as a secret key) are "semi-oblivious". Alternately, systems that require the marked file as well as the original file and any secret keys are "non-oblivious" [11]. Generally oblivious or semi-oblivious schemes are preferable to non-oblivious, because they tend to reduce the required complexity of the detection system by reducing the amount of processing required and the amount of prior knowledge required for watermark detection.

Digital watermarking can occur over a variety of media, such as pictures or movies, as

evidenced by usage of the watermarking schemes mentioned in the previous section, but audio watermarking is of particular interest as companies continue to attempt to protect audio content. There are several methods that can embed a watermark in an audio file. The watermark can be applied in the time domain, that is, altering the original content of the file. The content can alternatively be transformed to some other domain by an invertible process, after which the watermark can be applied and transformed back, yielding watermarked audio.

2.4 Application of Watermarking in DRM

Audio watermarking is a technology that could be employed in attempting to protect the copyright holders of audio files. The original media intended for distribution is watermarked with a secret key, yielding a new watermarked file. This file is then distributed as intended. The watermark cannot be detected by parties who do not know the secret key, so attackers who attempt to destroy the watermark do not know if they have succeeded, or even if a watermark exists. However, to parties with the secret key (the copyright owner and perhaps an impartial adjudicator), the watermark serves as proof of ownership by the copyright holder or digital media owner, thereby providing evidence in their favor in the case of a copyright dispute [16]. When used in this fashion it creates thorough passive protection, in that the illegal actions are not prevented, but rather the distribution of the marked files can be tracked, as the protection is embedded within the file itself rather than around the file.

This in particular makes digital watermarking a viable alternative to other DRM systems and therefore worthy as a topic of research. Provided the watermark is sufficiently transparent, watermarking does not impair the usability of the files being protected, while most DRM systems greatly limit how files can be used. Watermarking itself does not prevent illicit copying of data, as DRM attempts to; rather, watermarking allows for illegal copying to be detected in a passive manner. If a user has a watermarked file without proof of license, then the file has been obtained illegally, and appropriate action can be taken. It is possible that the mere existence or even the possibility of a watermark can serve as a deterrent against copyright infringement.

Watermarking has been applied by the film industry with a degree of success. When Oscar® screening copies are distributed to members of the voting panel, they are watermarked with unique identifiers that allow the specific copy to be tracked to its source should it become illegally available. This system has met with some success, as the watermark allowed authorities to track a breach to the culprit, a brother of an Oscars® voter. [1]

[1] Oscar and Oscars are registered trademarks of the Academy of Motion Picture Arts and Sciences

3 Related Work

A large amount of work has already been conducted in the realm of watermarking, and there are several techniques by which watermarks can be applied. Watermarks can be applied in the time domain, in which the raw data of the audio is directly manipulated. Other techniques involve transforming the audio data to some other domain, such as frequency, and performing data manipulation before inverting the data back to the time domain. Simple techniques are usually insufficient, so new techniques and new domains are constantly being sought out.

3.1 Watermarking in the Time Domain

The simplest technique for watermarking audio is to simply consider the least significant bits of the song [12]. This method is easy to implement, and easy to detect, as specific sets of low order bits will be interpreted as particular values in the watermark. However, this technique is not resilient against attack; common audio manipulation functions, such as MPEG compression and resampling, can alter the watermark bits, destroying the watermark, without affecting the usability of the file.

A more complicated time-based method proposed by Xu et al [25] involves analyzing feature vectors of the original audio, and using those vectors to perform bit hiding. Essentially, echoes are added to the frames at varying delays and amplitudes, with specific combinations of parameters mapping to either a zero or a one. The parameters are selected so as to not be detected by the Human Auditory System (HAS).

3.2 Watermarking in the Frequency Domain

Many other watermarking techniques involve transforming the audio into another domain, most commonly the frequency domain. Information about the frequencies of an audio file is obtained, generally by variants of the Fourier transform. The watermark then can be embedded in a variety of fashions. Methods detailed by Haitsma et al [8] and Megías et al [15] detail a process that creates a watermark signal in the frequency domain based on the carrier signal, but scales the signal strength to ensure its imperceptibility in the temporal domain. Each frequency is analyzed so that the watermark occurs with a higher strength at frequencies with more presence.

The frequency domain can also be useful for dividing the audio signal into different subsignals using band-pass filters [1]. These band-limited signals can then be reconstructed in another domain, such as inverting back to the time-domain, where the watermark is applied. The bands are then returned to the frequency domain, recombined, and the marked audio file is then constructed. Similar techniques have been used for image watermarking as well [21].

3.3 Watermarking in Other Domains

Other techniques involve shifting the time-based information to some domain other than frequency. These methods are often very similar in function to the frequency domain techniques. Quan et al [17] describe a system in which the audio is transformed to the wavelet packet domain. The watermark is hidden via statistical methods, such that the effect of the watermark is below the temporal masking threshold, beyond which the mark cannot be perceived.

One method explored by Wang et al [23] involves using the Integer Lifting Wavelet Transform. Salient points are extracted from the original audio sequence to determine areas in the audio that are particularly noticeable. This is used to identify where to embed the watermark. The theory is, if the watermark can be successfully embedded in a highly salient area, then its removal by an attacker will be noticeable, reducing the quality of the audio file. These salient sections are transformed using the Integer Lifting Wavelet Transform, in which the watermark is embedded.

3.4 Watermarking in the SVD Domain

The Singular Value Decomposition (SVD) is the result of a process that decomposes generic rectangular matrices. It can be viewed as a byproduct of a generalization of the spectral theorem, which says that normal matrices can be diagonalized using a basis of eigenvectors. The SVD, however, exists in matrices of any size, and can be used for several applications, including finding a pseudoinverse for a noninvertible matrix, or for approximating the value of a matrix [9].

$$SVD(A) = UDV^T \tag{1}$$

As seen in equation 1, A, an m-by-n matrix, is decomposed into three matrices, U, D, and V. U and V are orthogonal matrices of dimension m-by-m and n-by-n, respectively, and D is an m-by-n matrix containing the singular values along the diagonal, and zeroes elsewhere.

The SVD can be applied to the problem of audio watermarking, as proposed in a process by Özer et al [16]. Using the Short Time Fourier Transform (STFT), a matrix of values is derived from the original audio. This matrix, with dimensions frequency-by-time, is decomposed, yielding the SVD matrices. The watermark is embedded within the singular value matrix. This embedding process is over the entire matrix, and as such is no longer pseudo-diagonal, so the marked singular value matrix is itself decomposed, yielding a new singular value matrix. This new matrix is then used with the original U and V matrices as inputs to reconstruct the STFT frame, which is then used with the inverse STFT to yield a watermarked audio signal. This method is particularly attractive as it is resilient to many forms of attack; singular values are invariant under orthogonal changes to a matrix. This process is technically semi-oblivious, but it requires an amount of information

commensurate with the size of the original audio signal to detect the watermark. This particular system will be further investigated in the course of this paper.

3.5 Psychoacoustics and Watermarking

Often it is beneficial to consider the way in which the Human Auditory System (HAS) perceives sound; this area of study is known as psychoacoustics. An understanding of psychoacoustics usually leads to designing watermarking systems that are formulated in such a way that the watermark is strong, but the HAS cannot detect it. The principles of auditory masking are often taken into consideration; more information or distortions can be hidden in a more intense carrier signal. Temporal masking allows for weak signals to be hidden if positions directly before or after strong signals [13, 22].

Psychoacoustics is not only important in the design of a watermark system, but also when measuring the effects of a watermark on a carrier signal. It is possible to conduct aural surveys and tests to measure the perceptual distortion on a sample of audio, but this is often time consuming, expensive, and not easily repeatable. Humans are by nature subjective, so the opinions of two people in the survey may disagree. However, an objective measure quantifying the amount of distortion between two signals based on psychoacoustics is both repeatable and less costly. The Perceptual Audio Quality Measure (PAQM) is such a tool for determining the relative quality of two audio signals [2]. The process of the measurement is as follows:

The two signals are transformed into the time-frequency domain using a Fast Fourier Transform and a time windowing function. The spectral power density is then computed, and transformed from the frequency scale to the bark scale, which is a psychoacoustic measure of pitch rather than frequency [26]. This bark spectrum is then filtered by a function that mimics the behavior of audio transferring from the outer ear to the inner ear of the HAS. Time and frequency spreading is performed, and excitation levels are computed over the time-pitch domain. Compressed loudness is computed, at which point the two signals are compared. The momentary noise disturbance of the absolute difference yields the measure of difference between the two signals.

Beerends and Stemerdink successfully showed that the PAQM does have a strong correlation to the Mean Opinion Score (MOS) of surveys, and as such can be used in their place [2]. A noise disturbance of 0.01 correlates to roughly 4.7 on the MOS scale (out of 5, 5 being imperceptible, 1 being annoying), which is barely perceptible. This makes an ideal value for scaling watermark strengths as the watermark cannot adversely impact the quality of signal.

4 Implemented Systems

Özer's watermarking scheme was implemented to better understand his approach and to allow verification of his results against other schemes in the SVD. New watermarking

systems operating in the SVD domain were designed in an attempt to determine their feasibility and to better understand what can and cannot be done in this domain. The watermarking schemes were developed in families, with each family differing significantly in operation. The families are further divided into schemes that differ from other schemes in the parameters used.

All of the families have some common steps, however, which are in fact identical to the process put forth by Özer. The STFT of the audio signal is first computed, yielding a matrix of frequency-by-time. Next, depending on the bit-encoding rate desired, frames of the STFT matrix are decomposed, yielding the SVD, yielding three matrices, U, D and V^T. The bit-encoding rate used for all of the following schemes is 16 bits per second. Each frame is the target location of a single watermark bit. Depending on the watermarking scheme, these matrices are altered and recombined, yielding a new STFT frame. After all frames have been marked, the inverse STFT computes the newly marked audio signal. The first steps of watermark detection are identical to the steps described above, yielding the SVD matrices for each STFT frame of the marked audio.

Random values for all families come from a pseudo-random number generator seeded with a particular key. That key, along with other data, is part of the information required to successfully detect the watermark.

4.1 Family 1: Diagonal

The Diagonal family of watermarking schemes targets the pseudo-diagonal D matrix from the SVD. This matrix has dimensions to those of the STFT frame originally decomposed with singular values along the main diagonal (where the row index equals the column index) and zeroes elsewhere.

The singular values are then altered by equation 2, where σ_W is the marked singular value, σ is the original singular value, a is the watermark strength, bit is the bit being encoded, and $rand$ is a random value generated over a particular range as defined by the parameter variations below. The amount by which the singular values are altered is scaled to the value of the singular value itself. A signal frame with high singular values will be better able to mask the watermark, so the watermark can itself be stronger and, therefore, more easily detected.

$$\sigma_W = \sigma + (\sigma \times a \times bit \times rand) \tag{2}$$

This is a simpler encoding scheme than that of Özer in that after the watermark has been applied; the matrix is still pseudo-diagonal, meaning a second decomposition is not needed. This also reduces the amount of information required to detect the watermark. However, a characteristic of the Singular Value Decomposition brings forth a potential issue: the singular values always occur in non-increasing order. The process of watermarking could potentially violate this restriction, so we must take note of the order of the new singular values.

The detection of this scheme is essentially the inverse of the watermarking process. Once the SVD matrices are computed, the singular values are reordered to match the original ordering from the watermarking step. Then, an estimate of the value of the bit is made for each singular value by equation 3. σ_A is the singular value of the new audio file, and σ is the original singular value. The median of these "bit guesses" is then compared with the candidate valid bit values (either {0,1} or {-1,1}), and the closest valid bit is taken to be the bit in the message.

$$bit_{guess} = \frac{\sigma_A - \sigma}{\sigma \times a \times rand} \tag{3}$$

The information required to extract the watermark is the set of original singular values and the order of the altered singular values.

In total, sixteen unique schemes were proposed for this family, based on the values of three different parameters. The first parameter determines the number of singular values to be marked. The variations are: mark all singular values, mark one singular value (selected at random), mark two singular values (selected at random), and mark a random number of singular values (selected at random). The selection of the singular values is randomized by the number generator, but they must be selected so as to be reproducible at detection. The second parameter determines the range of the random values: either a uniform distribution over the range [0,1) ($0 \leq \text{rand} < 1$), or a uniform distribution over the range [-1,1) ($-1 \leq \text{rand} < 1$). The final parameter determines the domain of the watermarking bits. The bits may come from the range {0,1}, or the range {-1,1}, where a 0 bit in the message is mapped to a -1 bit in the watermark.

These parameter combinations are embedded into a single number for ease of identification as follows. Schemes 0-3 embed all singular values, schemes 4-7 embed in a single singular value, schemes 8-11 embed in two singular values, and schemes 12-15 embed in a random number of values. The ranges of random values are [0,1) for scheme $\% \ 4 \leq 1$, and [-1,1) for scheme $\% \ 4 > 1$. The domains of bit values are {0,1} for scheme $\% \ 2 = 0$ and {-1,1} for scheme $\% \ 2 = 1$.

The effects that these parameters have on the audio file will vary, so different watermark strengths are required for each embedding scheme. Table 1 lists the watermark strengths which yield a PAQM noise disturbance of 0.01 for each scheme. It can be inferred from the table that marking more singular values has a greater effect on the distortion of the marked audio, and as such the watermark strength must be lower. Also using the bit domain {-1,1} distorts the audio more than the domain {0,1}, as encoding a zero bit does not have any effect on the quality of the audio for that frame. Also the random range plays a role, in that the range [0,1) affects the audio more than the range [-1,1), as evidenced by the lower watermark strengths for the schemes using [0,1).

The viability of these schemes is tested by encoding a message into a series of audio samples and attempting to retrieve the message without altering the audio at all. The number of incorrect bits is then recorded. Table 2 shows the error rates in encoding and

Scheme	Marks	Random	Bits	Strength
0	All	[0,1)	{0,1}	.0234
1	All	[0,1)	{-1,1}	.0188
2	All	[-1,1)	{0,1}	.0328
3	All	[-1,1)	{-1,1}	.0211
4	One	[0,1)	{0,1}	.1125
5	One	[0,1)	{-1,1}	.0750
6	One	[-1,1)	{0,1}	.1500
7	One	[-1,1)	{-1,1}	.0750
8	Two	[0,1)	{0,1}	.0656
9	Two	[0,1)	{-1,1}	.0469
10	Two	[-1,1)	{0,1}	.0750
11	Two	[-1,1)	{-1,1}	.0469
12	Some	[0,1)	{0,1}	.0375
13	Some	[0,1)	{-1,1}	.0281
14	Some	[-1,1)	{0,1}	.0469
15	Some	[-1,1)	{-1,1}	.0281

Table 1: Diagonal Watermark Strengths

decoding. A total of 30,428 bits are embedded and retrieved, and audio samples are taken from single instrument tracks (14,157 bits), pure vocal tracks (3,795 bits), contemporary music (10,606 bits), and speech (1,870 bits).

The first two schemes have the lowest base error rate, which yields the following conclusions:

It is more viable to mark all singular values instead of only a few. The watermark strength for these schemes is lower than the schemes which only mark a few singular values, as marking more values has a greater effect on the quality of the resultant audio. Therefore, a weaker watermark must be used to maintain imperceptibility. However, the redundancy inherent in marking all values, albeit with a weaker watermark, is more effective than marking fewer values with a stronger signal.

Marking with a random range in [0,1) was overall better than marking in [-1,1), which potentially merits further investigation, as the watermarks for [-1,1) were stronger.

Using bits from {-1,1} tended to have fewer errors than bits from {0,1}. This is due to the fact that there is a greater amount of difference between the effects of embedding a -1 and a 1 as compared to 0 and 1, so the encodings of -1 and 1 are more easily distinguishable.

Scheme	Marks	Random	Bits	Single Inst	Vocal	Contemp	Speech	All
0	All	[0,1)	{0,1}	.0661	.0127	.0011	.0706	.0370
1	All	[0,1)	{-1,1}	.0872	.0142	.0010	.0818	.0478
2	All	[-1,1)	{0,1}	.1186	.0588	.0592	.1348	.0914
3	All	[-1,1)	{-1,1}	.0946	.0153	.0025	.0893	.0523
4	One	[0,1)	{0,1}	.1870	.0772	.0779	.1786	.1348
5	One	[0,1)	{-1,1}	.1445	.0292	.0118	.1037	.0814
6	One	[-1,1)	{0,1}	.1882	.0848	.0881	.1807	.1399
7	One	[-1,1)	{-1,1}	.1443	.0311	.0110	.1139	.0819
8	Two	[0,1)	{0,1}	.1348	.0314	.0270	.1369	.0844
9	Two	[0,1)	{-1,1}	.1366	.0232	.0089	.1043	.0760
10	Two	[-1,1)	{0,1}	.1707	.0806	.0830	.1834	.1297
11	Two	[-1,1)	{-1,1}	.1492	.0356	.0254	.1198	.0900
12	Some	[0,1)	{0,1}	.1038	.0314	.0169	.1080	.0647
13	Some	[0,1)	{-1,1}	.1167	.0237	.0091	.1059	.0669
14	Some	[-1,1)	{0,1}	.1486	.0622	.0657	.1578	.1095
15	Some	[-1,1)	{-1,1}	.1235	.0221	.0096	.1016	.0698

Table 2: Diagonal Bit Encode-Decode Error

4.2 Family 2: Orthogonal

The Orthogonal family of watermarking schemes targets the U and V^T matrices from the SVD. These matrices are square, orthogonal, and, due to the nature of the STFT, populated with complex values. These schemes were developed in order to ascertain the viability of using the orthogonal matrix components of the SVD as a watermarking carrier signal. As mentioned before, the singular values of a matrix are relatively stable; they are invariant under orthogonal alterations. This would seem to suggest that the changes to the signal, if not reflected in the singular values, would be apparent in the orthogonal matrices, making potential watermark retrieval difficult.

The process of applying the watermark is as follows. To ensure we only mark certain random values, a binary matrix is generated with the watermark positions set to one. A second random matrix is generated reflecting the random range as described with the parameters below. Then, for every position in the watermarking mask where there is a one, that corresponding position is altered by the product of the original orthogonal value, the corresponding random value, the bit, and the watermarking strength. This process is formalized in equation 4. The original value in the orthogonal matrix is denoted o, the watermarked value is o_W, the bit being encoded is bit, and the random value is $rand$. The watermark strength a is scaled to yield watermarks with a PAQM of 0.01.

$$o_W = o + o \times a \times bit \times rand \tag{4}$$

Detection is, again, very much related to the inverse of the watermarking function. For each orthogonal value that was marked, we can extract a raw value that corresponds to the encoded bit. These raw bit values may be in error, so to mitigate that risk we take the median of the raw values and convert that to a bit. The difference between the original orthogonal value and the marked orthogonal value is found and divided by the original orthogonal value. In an ideal situation where the marked signal had not been altered, this will result in a complex number with a zero-valued imaginary component. However, as this will likely not be the case, to convert from imaginary numbers to reals, we take the magnitude of this value. This converts to the correct domain; however, it has the added effect of removing any sign information, making it impossible to distinguish between -1 and 1, and thereby limiting our bit domain to {0,1}. Once the magnitude of the division is found, the bit can be extracted by dividing by the product of the watermark strength and the random value used for embedding. Because of the loss of the sign information, the absolute value of the random value must be used, and not the actual random value; in this way we ensure our bit is positive. This process is further explained in equation 5. The original orthogonal value as extracted from the original audio is o; the new orthogonal value is o_A.

$$bit_{guess} = \frac{|\frac{o_A - o}{o}|}{a \times |rand|} \tag{5}$$

The information required to extract the watermark is the original orthogonal matrices from the original audio file, so to retrieve this, the original audio is required at detection, making this a non-oblivious watermarking technique.

In total twenty-four unique schemes were proposed for this family, based on the values of three different parameters. The first parameter determines the target matrices to be marked: either U, V^T, or both. The second parameter determines the number of values in the orthogonal matrices to be marked. The possible frequencies of marking are 25%, 50%, 75%, and 100%. In the cases where fewer than all of the values are being marked, the values to be marked are determined at random. The final parameter determines the range of the random values: either a uniform distribution over the range [0,1), or a uniform distribution over the range [-1,1).

These parameter combinations are embedded into a single number for ease of identification as follows. Schemes 0-7 embed in the U matrix, schemes 8-15 embed in V^T, and schemes 16-23 embed in both matrices. 25% of the values are marked for scheme = 0 or 1 % 8, 50% are marked for scheme = 2 or 3 % 8, 75% for scheme = 4 or 5 % 8, and 100

Table 3 lists the watermark strengths required for each parameter configuration to yield the appropriate noise disturbance level. Marking only U or only V^T have roughly the same effect, and, as expected, marking both matrices has a greater effect on the quality

of the audio, so the strength for schemes 16-23 is lower to compensate. As more values are marked, the effect is higher, so the strengths for schemes drop as the number of values marked increases. Finally, using the random range [0,1), as in the diagonal watermarking scheme described earlier, has a greater effect on the audio than does the range [-1,1).

Scheme	Target	Marks	Random	Strength
0	U	25%	[0,1)	.0563
1	U	25%	[-1,1)	.0750
2	U	50%	[0,1)	.0375
3	U	50%	[-1,1)	.0563
4	U	75%	[0,1)	.0281
5	U	75%	[-1,1)	.0445
6	U	100%	[0,1)	.0211
7	U	100%	[-1,1)	.0375
8	V^T	25%	[0,1)	.0656
9	V^T	25%	[-1,1)	.0844
10	V^T	50%	[0,1)	.0375
11	V^T	50%	[-1,1)	.0563
12	V^T	75%	[0,1)	.0300
13	V^T	75%	[-1,1)	.0500
14	V^T	100%	[0,1)	.0200
15	V^T	100%	[-1,1)	.0400
16	Both	25%	[0,1)	.0363
17	Both	25%	[-1,1)	.0525
18	Both	50%	[0,1)	.0200
19	Both	50%	[-1,1)	.0400
20	Both	75%	[0,1)	.0150
21	Both	75%	[-1,1)	.0300
22	Both	100%	[0,1)	.0113
23	Both	100%	[-1,1)	.0300

Table 3: Orthogonal Watermark Strengths

The audio encoding and decoding tests are run as they are for the Diagonal family of watermarking schemes. Table 4 shows the results of the testing.

Scheme 1 has the lowest base error rate, but overall the error rates are far higher than those of the Diagonal family of watermarking schemes. However, some important conclusions can still be drawn from these results.

In an interesting reversal from the Diagonal family, it is actually more beneficial to mark fewer values. The watermarking strengths for the schemes which mark only 25% of the values are higher than other schemes, and the redundancy of marking more values does

Scheme	Target	Marks	Random	Single Inst	Vocal	Contemp	Speech	All
0	U	25%	[0,1)	.1091	.0693	.0657	.0968	.0882
1	U	25%	[-1,1)	.0975	.0640	.0569	.0914	.0788
2	U	50%	[0,1)	.1440	.1080	.1079	.1171	.1252
3	U	50%	[-1,1)	.1332	.0964	.0899	.1032	.1117
4	U	75%	[0,1)	.1726	.1455	.1507	.1396	.1595
5	U	75%	[-1,1)	.1644	.1410	.1342	.1305	.1488
6	U	100%	[0,1)	.2101	.1921	.1986	.1684	.2017
7	U	100%	[-1,1)	.1990	.1797	.1759	.1576	.1860
8	V^T	25%	[0,1)	.1689	.1004	.0922	.1203	.1306
9	V^T	25%	[-1,1)	.1625	.0899	.0747	.1144	.1199
10	V^T	50%	[0,1)	.2361	.1939	.1914	.1620	.2107
11	V^T	50%	[-1,1)	.2296	.1813	.1598	.1604	.1950
12	V^T	75%	[0,1)	.2608	.2424	.2379	.1909	.2462
13	V^T	75%	[-1,1)	.2561	.2277	.2141	.1824	.2343
14	V^T	100%	[0,1)	.2719	.2596	.2565	.2107	.2612
15	V^T	100%	[-1,1)	.2683	.2540	.2409	.2000	.2528
16	Both	25%	[0,1)	.2048	.1615	.1587	.1465	.1797
17	Both	25%	[-1,1)	.1911	.1526	.1452	.1487	.1677
18	Both	50%	[0,1)	.2625	.2535	.2502	.2000	.2533
19	Both	50%	[-1,1)	.2552	.2359	.2268	.1888	.2388
20	Both	75%	[0,1)	.2715	.2688	.2661	.2214	.2662
21	Both	75%	[-1,1)	.2685	.2585	.2544	.2037	.2583
22	Both	100%	[0,1)	.2754	.2717	.2686	.2374	.2702
23	Both	100%	[-1,1)	.2723	.2656	.2627	.2235	.2651

Table 4: Orthogonal Bit Encode-Decode Error

not provide a benefit.

Additionally, it seems that the U matrix is the only matrix viable as a watermark carrier; the error rates for schemes marking in the V^T matrix were on average 60% higher.

The random range also had an effect; marking with random values from the range [-1,1) slightly outperformed the schemes marking from the range [0,1), which is also reversed from the behavior of the Diagonal family schemes.

4.3 Family 3: Diagonal Bit Spreading

Diagonal Bit Spreading is very similar in operation to the Diagonal family with one important distinction: where in the Diagonal family the bit used in the watermarking equation (equation 2) was the same for every singular value being altered; in Diagonal Bit Spreading the values of the bit being encoded are randomly flipped. For instance, when encoding the bit 0, for roughly half of the singular values a 0 will be used in the watermarking equation and for the other half a 1 will be used. The same is true for all bits being encoded in each decomposed STFT frame. In the Diagonal family the encoding of a zero bit had no effect on the bit frame; however, due to the bit flipping, each encoded bit will have a similar effect on the audio quality. The watermark strengths for the Diagonal Bit Spreading family are on average higher than the watermark strengths for the Diagonal family.

The only parameter setup that will experience no appreciable change in watermark encoding output quality is that with a bit domain of {-1,1} and a random range of [-1,1). The bit flipping in the case of the bit domain of {-1,1} is a matter of simply changing the sign, which is the same effect produced by having a random range include negative values. The times at which the bits are flipped could correspond to when the random values are negative, so in this instance bit flipping does not introduce any new behavior.

This change in embedding requires an alteration in the way in which the watermark is extracted. Where in the Diagonal family all of the candidate bits are taken together, and the median of that collection is then interpreted as a bit, that technique is not available, as each value may have been marked with a different bit. This forces the candidate bits to be interpreted as bits and flipped as needed, yielding a collection of bits as opposed to a collection of raw values. The median of the collection is taken as before, but the order of operations required to detect the watermark is altered to account for the flipping of bits.

The parameter variations and naming conventions are identical to those of the Diagonal family as previously described. The requisite watermark strengths are listed in Table 5; they follow very similar trends as the Diagonal family listed in Table 1.

The encoding-decoding tests are run as described above; the results follow in Table 6 shows the results of the testing.

These results are similar to those of the Diagonal family, in that marking all singular values is better than marking fewer singular values, and marking with bits from the domain {-1,1} is better than bits from the domain {0,1}. There was no discernable benefit from using one range of random values over another however.

Scheme	Marks	Random	Bits	Strength
0	All	[0,1)	{0,1}	.0200
1	All	[0,1)	{-1,1}	.0200
2	All	[-1,1)	{0,1}	.0350
3	All	[-1,1)	{-1,1}	.0200
4	One	[0,1)	{0,1}	.1200
5	One	[0,1)	{-1,1}	.0850
6	One	[-1,1)	{0,1}	.1600
7	One	[-1,1)	{-1,1}	.0850
8	Two	[0,1)	{0,1}	.0700
9	Two	[0,1)	{-1,1}	.0500
10	Two	[-1,1)	{0,1}	.0800
11	Two	[-1,1)	{-1,1}	.0500
12	Some	[0,1)	{0,1}	.0350
13	Some	[0,1)	{-1,1}	.0300
14	Some	[-1,1)	{0,1}	.0500
15	Some	[-1,1)	{-1,1}	.0300

Table 5: Diagonal Bit Spread Watermark Strengths

Scheme	Marks	Random	Bits	Single Inst	Vocal	Contemp	Speech	All
0	All	[0,1)	{0,1}	.1122	.0224	.0035	.1016	.0625
1	All	[0,1)	{-1,1}	.1014	.0153	.0019	.0888	.0552
2	All	[-1,1)	{0,1}	.1135	.0332	.0306	.1139	.0746
3	All	[-1,1)	{-1,1}	.0989	.0148	.0027	.0947	.0546
4	One	[0,1)	{0,1}	.1956	.0928	.0909	.1888	.1459
5	One	[0,1)	{-1,1}	.1327	.0203	.0071	.1053	.0732
6	One	[-1,1)	{0,1}	.1937	.1020	.0969	.1947	.1485
7	One	[-1,1)	{-1,1}	.1357	.0235	.0088	.0989	.0752
8	Two	[0,1)	{0,1}	.2060	.0988	.0962	.2000	.1540
9	Two	[0,1)	{-1,1}	.1598	.0472	.0377	.1364	.1017
10	Two	[-1,1)	{0,1}	.2195	.1228	.1157	.2070	.1705
11	Two	[-1,1)	{-1,1}	.1538	.0437	.0271	.1267	.0943
12	Some	[0,1)	{0,1}	.1425	.0408	.0295	.1294	.0896
13	Some	[0,1)	{-1,1}	.1249	.0227	.0088	.0973	.0700
14	Some	[-1,1)	{0,1}	.1533	.0743	.0647	.1604	.1130
15	Some	[-1,1)	{-1,1}	.1270	.0358	.0159	.1086	.0757

Table 6: Diagonal Bit Spreading Bit Encode-Decode Error

Compared to the performance of the Diagonal family, Diagonal Bit Spreading had more errors due to the change in the detection algorithm. The Diagonal scheme is able to mitigate more errors by only interpreting the median of the extracted raw values as a bit. In Diagonal Bit Spreading more interpretations occur which in turn introduces slightly more errors.

4.4 Family 4: Orthogonal Bit Spreading

The Orthogonal Bit Spreading family is related to the Orthogonal watermarking schemes in the same way that the other two schemes are related. For each orthogonal value being marked, the bit is flipped at random to the other value before being applied by the watermarking equation.

However, as in Diagonal Bit Spreading, the order of watermark detection must be altered as well, meaning that the extracted raw values must be converted to bits before the median of the collection is taken. Where this resulted in slightly higher error rates when bit spreading was applied to the Diagonal family, the Orthogonal family became completely nonfunctional. As so many more values were being marked, there is a higher propagation of error as the bit value interpretation is occurring more frequently.

In an attempt to reduce the number of interpretations and therefore the amount of error, the extracted raw values are placed into two lists, one list referring to bits that were unflipped, and the second referring to bits that were flipped. In theory, the bits of one list correspond to a 1, while the bits of the other list correspond to a 0. The median of the 1-bit list should therefore be higher than the median of the 0-bit list. So, if a 1 is being encoded, the unflipped list will correspond to the 1 and the flipped list will correspond to the 0, so the median of the unflipped list should be higher than that of the flipped list, and the opposite should be the case when encoding a 0.

However, this proved to not be the case. No discernable relation between the two medians could be found. Encoding zeroes and ones were, by comparison of medians, indistinguishable. There was roughly a 40% error for identifying bits, which is hardly better than simply guessing. As the watermark detection algorithm could not be successfully crafted, the schemes could not be tested, and as such this family was deemed not suitable for watermarking.

4.5 Özer

The system put forward by Özer performs a different form of watermark embedding on the singular value matrix than do the Diagonal and Diagonal Bit Spreading families. The watermarking is different in that, instead of only altering the singular values, all values in the singular value matrix, including the zeroes, are marked. The method by which this occurs is described in equation 6. The watermarked matrix W_D is generated by the matrix of singular values D and the random value from the watermark signal w. Each row in W_D

is altered by using the singular value of that row ($D_{(i,i)}$), and not the corresponding value in the singular matrix.

$$W_D(i,j) = D(i,j) + a \times bit \times D(i,i) \times W(i,j) \tag{6}$$

A random signal matrix W is generated with dimension equal to the singular value matrix. Then, each value in the singular matrix is altered by the addition of the product of the singular value for that row, the bit being embedded, the watermark strength, and the appropriate random value. This results in a non-pseudo-diagonal matrix, and as such, must be decomposed again via the SVD. This is an important distinction as the watermark is spread among all values of the matrix, so it is spread throughout all frequencies and times in the STFT sample.

Watermark detection is also sufficiently different. Instead of attempting to extract the bit directly, detecting the watermark involved attempting to recover the random signal matrix W. From the new audio signal A', we generate U', D'_w, and V'^T. Given at detection are the matrices U_w, V_w^T, and D. Using these matrices and the new singular value matrix from the sample audio signal, D'_w, we can recover an approximate of the pseudo-random matrix, W'. This process is further explained in equations 7-9. W' is compared to the genuine pseudo-random matrix, W. If the inner product of W' and W is positive, the bit encoded was 1, otherwise the bit was -1.

$$A' = U' D'_w V'^T \tag{7}$$

$$W'_D = U_w D'_w V_w^T \tag{8}$$

$$W' = \frac{D^{-1}(W'_D - D)}{a} \tag{9}$$

Özer asserts that a watermark strength of $a = 0.15$ will yield a PAQM of 0.01, which corresponds to an MOS of around 4.7 which is in turn nearly imperceptible [16]. However, it was found that for this implementation, a strength of 0.15 actually yielded a far higher amount of noise disturbance, closer to 0.64. This indicates that the watermark should be audible, and moreover, distracting. In actuality, upon listening to marked audio files, a constant low-volume static is present and audible, which is at odds with its definition of being inaudible. Running tests to scale the watermark such that the PAQM was 0.01 resulted in a watermark strength of 0.015, which is somewhat closer to the strengths of the other systems, and is in fact imperceptible.

Bit encoding-decoding tests were run on Özer, yielding the following results as listed in Table 7. The errors are overall far lower than the watermarking schemes already described. It also clearly shows some trends that match those of the other watermarking schemes. The error rate for the single instrument and speech tracks are far higher than that of the vocal

and contemporary. This is because there is far more pauses and areas of silence in these samples. If the audio has no signal, or a very weak signal, the watermark, which is scaled to the strength of the signal, if be very weak if existent. Simply put, quiet samples and samples with pauses are far more difficult to watermark and detect successfully. These issues are not present for vocal or contemporary samples as there is generally much more noise and much stronger signals present. This makes it far easier to inaudibly embed a strong watermark and successfully detect it later.

Özer	Single Inst	Vocal	Contemp	Speech	All
	.0297	.0063	.0004	.0396	.0171

Table 7: Özer Bit Encode-Decode Error

4.6 Özer Bit Spreading

Let us consider the application of the bit spreading technique described in Family 3 to the scheme proposed by Özer and described in the previous section. Recall that Özer works on the bit domain {-1,1}, so flipping a bit has the same effect as multiplying by a negative one. Referring to equation 6, it is easy to see that via equations 10 and 11 that multiplication of a negative one to the watermarking bit can easily be transferred to the random value, as multiplication is commutative. The effect of multiplying over a random range has the capability to alter the range, and therefore potentially affect the success of the embedding and recovery. However, as the random range in use is [-1,1), multiplication by a negative one will result in (-1,1], so the combination of these two ranges (both are potentially in use, as the bit is only flipped 50% of the time) will yield [-1,1]. This is not an appreciable difference in operation, so it is unlikely that this will have any effect at all. If the random range in use were [0,1), we would see a change in operation, as the new effective random range would be (-1,1), and would likely have similar behavior to the system currently in place.

$$W_D(i,j) = D(i,j) + a \times (-1 \times bit) \times D(i,i) \times W(i,j) \tag{10}$$

$$W_D(i,j) = D(i,j) + a \times bit \times D(i,i) \times (-1 \times W(i,j)) \tag{11}$$

Just as the transition from Diagonal to Diagonal Bit Spreading made for some alteration in the successful operation of the detection algorithm, it is still important to consider the detection algorithm. In this case however, very little would need to be changed to apply bit spreading. Since each position in which the bit was embedded is not separately extracted and analyzed (see equations 7-9), the result is the pseudo-random signal W' with some of the signs flipped as a result of the bit flipping. This can easily be undone be recreating the

sequence of flipping signals, and changing the signs where a bit flip occurred. The rest of the operation can proceed as described before; no adverse functional behavior is added by applying the bit spreading process.

As detailed above, the application of bit spreading would have no effect on the Özer system as it is implemented. However, we have argued that bit spreading can in fact be added to a system will no ill effects, making it a potentially viable option to consider when building a watermarking system.

5 Attacks

One major criterion for a successful watermarking scheme (arguably the most important criterion) is robustness. The resistance of a watermark against destruction upon the alteration of a file, whether the rationale for alteration is malicious or benign, is extremely important. Therefore, it is important to devise tests that either are similar to common audio processing techniques (benign alterations not meant to destroy the watermark), or are designed specifically break the watermark. Additionally, watermarks may be vulnerable to second-watermark attacks in which re-marking the audio file hinders the detection of the original watermark.

A common suite of tests used by many researchers is the StirMark Benchmark for Audio (SMBA). It contains a suite of audio attacks for just such a purpose. However, as the authors of StirMark are currently working under a Non-Disclosure Agreement, the StirMark suite of robustness attacks was not available for testing purposes [19, 7].

Original tests were developed to build a similar suite of robustness attacks. Some tests were created based on the specifications of the StirMark tests while others were original. The more complex signal processing attacks were processed by the audio processing program Audacity. Audacity version 1.3 has introduced the ability to string commands together and perform batch processing on multiple audio files in sequence (termed "chains"), which is ideal for these processes. All attacks are described in detail in the remainder of this section. New values outside of the allowable range (16-bit signed integers) are clipped to the appropriate boundaries. Attacks which alter the length of the audio file will either reduce the detection information or reduce the length of the attacked audio file to ensure that the lengths agree.

AddNoise The AddNoise attack adds a uniformly distributed noise to the audio signal. The random values come from the range [-500,500), which resulted in a PAQM of roughly 0.01, a barely perceptible distortion commensurate with the distortion of the watermarking schemes.

Amp067 The Amp067 attack reduces the volume of the audio file by 33%. This is designed as an inverse to the Amp150 attack, as the usage of one will theoretically cancel out the usage of another, barring any clipping errors.

Amp150 The Amp150 attack increases the volume of the audio file by 50%. As noted above, the usage of Amp150 after the application of Amp067 will result in the original audio file.

BassBoost The BassBoost attack enhances and increases the strengths of the bass frequencies.

Compressor The Compressor attack compresses the dynamic range of the audio file. This has the effect of making loud parts softer while not affecting the volume of the soft parts.

DelayStart The DelayStart attack appends one tenth of a second of silence to the audio file; this is designed to result in a misalignment of the STFT frames and therefore impact the accuracy of the watermark retrieval without actually affecting the quality of the audio.

Diagonal 0 The Diagonal 0 attack marks the audio file with scheme 0 of the Diagonal family of watermarking schemes to test the resistance against second-watermark attacks. The key and message are different from the key and message originally used to mark the audio.

Diagonal 1 The Diagonal 1 attack marks the audio file with scheme 1 of the Diagonal family of watermarking schemes to test the robustness against second-watermark attacks. The key and message are different from the key and message originally used to mark the audio.

Echo The Echo attack repeats the selection with a delay time and a decay rate to sound like an echo. The delay time is 1 second, and the decay rate is 50%. This will not affect the length of the audio, so if an echo will continue past the end of the audio, it will not be added to the file.

FFTNoise The FFTNoise attack takes the Fast Fourier Transform of the audio file, applies a noise signal with a random strength in the real and imaginary components, and recovers the altered audio file through the inverse Fast Fourier Transform. The magnitude of each component ranges up to 200,000 and was selected to result in a PAQM of roughly 0.01.

FlippSample The FlippSample behaves in the same way as it appears in the StirMark for Audio Benchmark tests. Samples of the audio are swapped periodically as determined by three parameters. Sections of audio 8 samples long are swapped from 400 samples away. This process occurs every 16000 samples (one second). The parameters were chosen to yield a PAQM of around 0.01.

HighPass The HighPass attack is based on a first-order analog high pass filter in that the presence of frequencies below a certain threshold is reduced. The rate of decay is such that the power is halved for every octave below the threshold frequency, which is 993 Hz.

Invert The Invert attack flips the sign of every sample in the audio file.

Levelling The Levelling attack equalizes the volumes of the loud and soft sections. Above the noise threshold, soft parts are made louder and loud parts are made softer, and below the noise threshold, the signal strength is reduced. The noise threshold is -70db.

LowPass The LowPass attack is based on a first-order analog low pass filter in that the presence of frequencies above a certain threshold is reduced. The rate of decay is such that the power is halved for every octave above the threshold frequency, which is 993 Hz.

NoiseRemoval The NoiseRemoval attack removes the presence of noise from the audio sample. The noise profile used to remove the noise comes from a sample of band limited pink noise (Track 2 of the SQAM audio samples).

Normalize The Normalize attack displaces all samples of the audio track by a constant vertical amount such that the center of the audio is around zero.

Orthogonal 1 The Orthogonal 1 attack marks the audio file with scheme 1 of the Orthogonal family of watermarking schemes to test the resistance against second-watermark attacks. The key and message are different from the key and message originally used to mark the audio.

Özer The Özer attack marks the audio file with the watermarking scheme proposed by Özer to test the resistance against second-watermark attacks. The key and message are different from the key and message originally used to mark the audio.

PitchDown The PitchDown attack lowers the pitch of the audio sample by a quarter tone. It does not affect the speed or the length of the audio file.

PitchUp The PitchUp attack raises the pitch of the audio sample by a quarter tone. It does not affect the speed or the length of the audio file.

SpeedDown The SpeedDown attack decreases the speed of the audio sample by 2%. The result of this attack is a slightly longer and slightly lower pitched audio file.

SpeedStart The SpeedStart attack removes the first tenth of a second from the audio file; this is designed to result in a misalignment of the STFT frames and therefore impact the accuracy of the watermark retrieval without actually affecting the quality of the audio.

SpeedUp The SpeedUp attack increases the speed of the audio sample by 2%. The result of this attack is a slightly shorter and slightly higher pitched audio file.

Spread 1 The Spread 1 attack marks the audio file with scheme 1 of the Diagonal Bit Spreading family of watermarking schemes to test the resistance against second-watermark attacks. The key and message are different from the key and message originally used to mark the audio.

Spread 3 The Spread 3 attack marks the audio file with scheme 3 of the Diagonal Bit Spreading family of watermarking schemes to test the resistance against second-watermark attacks. The key and message are different from the key and message originally used to mark the audio.

SVDNoise The SVDNoise attack adds uniformly distributed random values in the range [-600,600) to the singular values of an audio file. The STFT frames are selected to yield square matrices using a sliding Hamming window of roughly 25 ms and 50% overlap. The random value range was scaled to yield PAQM disturbance values of around 0.01.

TempoDown The TempoDown attack reduces the tempo of the audio file by 2%, resulting in a slightly longer audio file, but in such a way such that the pitch of the audio is not affected.

TempoUp The TempoUp attack increases the tempo of the audio file by 2%, resulting in a slightly shorter audio file, but in such a way such that the pitch of the audio is not affected.

6 Evaluation

The test base of audio files included files from the Sound Quality Assessment Materials (SQAM) as well as various studio recorded audio files [24]. As per the process used by Özer [16], the files were sampled down from 44100 Hz to 16000 Hz. The files used from SQAM were the single instrument, vocal, and speech. Line-up (e.g. sine waves) and artificial signals (e.g. pink noise) were not processed as they are less likely to be in need of protection, and as such are less likely to be actual targets of watermarking. Additionally, as these files are artificially generated, they are a purer signal, so they are far more difficult to watermark without creating noticeable distortion.

The studio recorded audio files consist of Aerosmith's "Walk This Way" and Queen's "Don't Stop Me Now" (fast popular music), The Wallflower's "Invisible City" (a slower song), and "What Kind of Fool Am I?" as performed by Surround Sound, an all-male barbershop and a cappella group (all vocal). These songs were split into 20-second samples and processed as individual samples.

The best schemes from each watermarking family were tested along with an implementation of the system proposed by Özer [16]. These schemes are listed along with descriptions of their parameters in Table 8.

Name	Diagonal Scheme 0
Values marked	Pseudo-diagonal matrix D - all singular values
Bit domain	{0,1}
Random range	[0,1)
Name	Diagonal Scheme 1
Values marked	Pseudo-diagonal matrix D - all singular values
Bit domain	{-1,1}
Random range	[0,1)
Name	Orthogonal Scheme 1
Values marked	Orthogonal matrix U - 25% of values
Bit domain	{0,1}
Random range	[-1,1)
Name	Diagonal Bit Spreading Scheme 1
Values marked	Pseudo-diagonal matrix D - all singular values
Bit domain	{-1,1}
Random range	[0,1)
Name	Diagonal Bit Spreading Scheme 3
Values marked	Pseudo-diagonal matrix D - all singular values
Bit domain	{-1,1}
Random range	[-1,1)
Name	Özer
Values marked	Pseudo-diagonal matrix D - all values
Bit domain	{-1,1}
Random range	[-1,1)

Table 8: Watermarking Scheme Parameters

These schemes were used to watermark the all audio files, which were then subjected to all 29 robustness tests. A total of 30,428 bits were embedded and retrieved for each scheme and attack, with 14,157 bits from single instrument samples, 3,795 from pure vocals, 10,606 from contemporary samples, and 1,870 from speech. The error rates for each scheme are listed in tables 9-14. A row labeled "Nothing" has been added to each of the tables so meaningful comparisons between the bit encode-decode error rates and the attack error rates can be made.

Attack	Single Inst	Vocal	Contemp	Speech	All
Add Noise	.4771	.2796	.2590	.4679	.3759
Amp067	.4995	.4954	.4942	.5027	.4974
Amp150	.4962	.5007	.5055	.4952	.4999
BassBoost	.4955	.5007	.5054	.4936	.4995
Compressor	.4962	.4962	.4414	.4952	.4770
DelayStart	.4970	.5012	.5003	.4941	.4985
Diagonal 0	.1327	.0899	.0845	.1299	.1104
Diagonal 1	.2903	.3041	.3037	.3134	.2981
Echo	.4580	.4659	.4713	.4695	.4643
FFTNoise	.4805	.2748	.2533	.4636	.3746
FlippSample	.1072	.0385	.0234	.0920	.0685
HighPass	.4692	.4920	.4940	.4818	.4814
Invert	.1555	.0179	.0014	.1086	.0817
Levelling	.4962	.5007	.5055	.4952	.4999
LowPass	.4703	.4946	.4936	.4930	.4828
NoiseRemoval	.4088	.2648	.2366	.4941	.3360
Normalize	.4962	.5007	.3795	.4952	.4560
Nothing	.0661	.0127	.0011	.0706	.0370
Orthogonal 1	.0990	.0422	.0324	.1021	.0689
Özer	.2515	.2242	.2236	.2305	.2371
PitchDown	.4762	.4675	.4736	.4840	.4747
PitchUp	.4768	.4677	.4737	.4845	.4751
SpeedDown	.4959	.5080	.4971	.5070	.4985
SpeedStart	.4916	.4896	.5016	.4882	.4946
SpeedUp	.4882	.4922	.4897	.4963	.4897
Spread 1	.1882	.1312	.1337	.1824	.1618
Spread 3	.2031	.1402	.1275	.1850	.1678
SVDNoise	.4469	.2632	.2181	.4321	.3433
TempoDown	.4886	.4970	.4928	.4936	.4914
TempoUp	.4863	.4838	.4927	.4909	.4885

Table 9: Diagonal Scheme 0 Attack Error

Attack	Single Inst	Vocal	Contemp	Speech	All
Add Noise	.4773	.2282	.1797	.4551	.3411
Amp067	.5112	.4996	.4947	.5123	.5041
Amp150	.5076	.5049	.5059	.5059	.5066
BassBoost	.5057	.5049	.5058	.5032	.5055
Compressor	.5076	.5007	.4427	.5059	.4840
DelayStart	.5091	.5072	.5008	.5048	.5057
Diagonal 0	.1303	.0685	.0592	.1198	.0972
Diagonal 1	.1974	.1634	.1592	.2016	.1801
Echo	.4663	.4680	.4700	.4743	.4683
FFTNoise	.4857	.2369	.1758	.4545	.3447
FlippSample	.1180	.0347	.0196	.0925	.0718
HighPass	.4751	.4909	.4941	.4909	.4847
Invert	.1558	.0216	.0013	.1171	.0828
Levelling	.5076	.5049	.5059	.5059	.5066
LowPass	.4738	.4967	.4935	.5005	.4852
NoiseRemoval	.3893	.2308	.1715	.5027	.3006
Normalize	.5076	.5049	.3787	.5059	.4622
Nothing	.0872	.0142	.0010	.0818	.0478
Orthogonal 1	.1025	.0256	.0119	.0973	.0610
Özer	.1729	.1030	.0918	.1487	.1344
PitchDown	.4798	.4627	.4653	.4909	.4733
PitchUp	.4803	.4630	.4652	.4909	.4735
SpeedDown	.5072	.5109	.4959	.5155	.5042
SpeedStart	.5032	.4989	.5025	.4963	.5016
SpeedUp	.5026	.4954	.4898	.5043	.4974
Spread 1	.1402	.0554	.0459	.1257	.0959
Spread 3	.1442	.0672	.0514	.1278	.1012
SVDNoise	.4345	.1987	.1312	.4273	.2989
TempoDown	.4969	.4967	.4899	.5032	.4948
TempoUp	.4990	.4909	.4930	.5005	.4960

Table 10: Diagonal Scheme 1 Attack Error

Attack	Single Inst	Vocal	Contemp	Speech	All
Add Noise	.5042	.2796	.4975	.5053	.5008
Amp067	.1944	.4954	.0443	.1775	.1266
Amp150	.1684	.5007	.0687	.1422	.1250
BassBoost	.5065	.5007	.5058	.5064	.5060
Compressor	.1631	.4962	.1532	.1444	.1595
DelayStart	.4964	.5012	.5054	.4942	.4999
Diagonal 0	.1490	.0899	.1023	.1289	.1256
Diagonal 1	.2355	.3041	.2201	.2219	.2257
Echo	.4782	.4659	.4785	.4861	.4785
FFTNoise	.5060	.2748	.4969	.5048	.5014
FlippSample	.1344	.0385	.0811	.1144	.1097
HighPass	.4870	.4920	.5053	.4920	.4951
Invert	.1839	.0179	.0493	.1743	.1229
Levelling	.3074	.5007	.3934	.3235	.3427
LowPass	.4695	.4946	.5027	.4898	.4858
NoiseRemoval	.5046	.2648	.4727	.5064	.4925
Normalize	.3244	.5007	.0536	.3369	.2088
Nothing	.0975	.0640	.0569	.0914	.0788
Orthogonal 1	.1764	.0422	.1526	.1642	.1649
Özer	.2469	.2242	.2773	.2824	.2617
PitchDown	.5066	.4675	.5058	.5059	.5061
PitchUp	.5066	.4677	.5058	.5069	.5061
SpeedDown	.5069	.5080	.5058	.5064	.5062
SpeedStart	.4935	.4896	.5025	.4973	.4979
SpeedUp	.4979	.4922	.4967	.4968	.4972
Spread 1	.3137	.1312	.3665	.3465	.3377
Spread 3	.3136	.1402	.3605	.3283	.3354
SVDNoise	.4945	.2632	.4672	.5011	.4830
TempoDown	.5069	.4970	.5056	.5064	.5061
TempoUp	.4976	.4838	.4967	.4968	.4970

Table 11: Orthogonal Scheme 1 Attack Error

Attack	Single Inst	Vocal	Contemp	Speech	All
Add Noise	.3778	.1697	.1090	.3599	.2570
Amp067	.5120	.5117	.5128	.5086	.5120
Amp150	.4803	.4830	.4950	.4930	.4866
BassBoost	.4562	.4661	.4924	.4861	.4719
Compressor	.4803	.4719	.4221	.4930	.4598
DelayStart	.5024	.5057	.5062	.4979	.5038
Diagonal 0	.1500	.0819	.0771	.1524	.1163
Diagonal 1	.2182	.1584	.1656	.2278	.1930
Echo	.4378	.4430	.4698	.4455	.4501
FFTNoise	.3892	.1739	.1028	.3658	.2611
FlippSample	.1161	.0348	.0219	.1080	.0726
HighPass	.4425	.4588	.5076	.4743	.4692
Invert	.1153	.0195	.0038	.1107	.0642
Levelling	.4817	.4825	.4951	.4904	.4870
LowPass	.4331	.4846	.4663	.4412	.4516
NoiseRemoval	.3477	.2097	.1355	.4856	.2650
Normalize	.4803	.4830	.3557	.4936	.4380
Nothing	.1014	.0153	.0019	.0888	.0552
Orthogonal 1	.1192	.0311	.0222	.1139	.0741
Özer	.1709	.0901	.0836	.1589	.1297
PitchDown	.4731	.4909	.4882	.4866	.4814
PitchUp	.4730	.4909	.4882	.4882	.4815
SpeedDown	.4949	.4983	.5060	.5064	.4999
SpeedStart	.4922	.5088	.5044	.4909	.4984
SpeedUp	.4867	.4975	.4961	.4963	.4919
Spread 1	.1775	.0851	.0836	.1695	.1328
Spread 3	.1672	.0846	.0797	.1594	.1259
SVDNoise	.3796	.1589	.0994	.3775	.2543
TempoDown	.4854	.4906	.4945	.5011	.4902
TempoUp	.4909	.4922	.4909	.4989	.4915

Table 12: Diagonal Bit Spread Scheme 1 Attack Error

Attack	Single Inst	Vocal	Contemp	Speech	All
Add Noise	.3720	.1676	.1097	.3610	.2544
Amp067	.5089	.5275	.5384	.5310	.5229
Amp150	.4738	.4725	.4793	.4802	.4759
BassBoost	.4503	.4564	.4731	.4663	.4600
Compressor	.4738	.4627	.4030	.4802	.4481
DelayStart	.4880	.4880	.5096	.4877	.4955
Diagonal 0	.1347	.0664	.0641	.1369	.1017
Diagonal 1	.2123	.1808	.1884	.2337	.2014
Echo	.4171	.4250	.4414	.4246	.4270
FFTNoise	.3905	.1713	.1121	.3508	.2637
FlippSample	.1196	.0385	.0243	.1070	.0755
HighPass	.4439	.4685	.5314	.5091	.4814
Invert	.1158	.0184	.0043	.0947	.0635
Levelling	.4754	.4727	.4791	.4791	.4766
LowPass	.4277	.4889	.4830	.4620	.4567
NoiseRemoval	.3477	.2116	.1445	.5064	.2697
Normalize	.4743	.4725	.3461	.4840	.4300
Nothing	.0989	.0148	.0027	.0947	.0546
Orthogonal 1	.1213	.0295	.0227	.1144	.0751
Özer	.1550	.0698	.0729	.1422	.1150
PitchDown	.4907	.4896	.4834	.5139	.4895
PitchUp	.4914	.4896	.4833	.5134	.4897
SpeedDown	.4863	.4928	.5026	.4882	.4929
SpeedStart	.5002	.5091	.5052	.5096	.5036
SpeedUp	.5043	.4920	.4982	.4930	.5000
Spread 1	.1873	.1020	.0980	.1706	.1445
Spread 3	.1711	.0833	.0824	.1652	.1289
SVDNoise	.3790	.1692	.1048	.3668	.2565
TempoDown	.4815	.4777	.4969	.4914	.4870
TempoUp	.4976	.4917	.4996	.4893	.4971

Table 13: Diagonal Bit Spread Scheme 3 Attack Error

Attack	Single Inst	Vocal	Contemp	Speech	All
Add Noise	.2096	.0300	.0017	.1658	.1120
Amp067	.0578	.0361	.0292	.0631	.0455
Amp150	.0463	.0087	.0005	.0417	.0253
BassBoost	.0911	.0124	.0206	.0631	.0550
Compressor	.1560	.0345	.0009	.0428	.0798
DelayStart	.0559	.0482	.0529	.2588	.0664
Diagonal 0	.0297	.0063	.0004	.0396	.0172
Diagonal 1	.0297	.0063	.0004	.0396	.0172
Echo	.1543	.0553	.0108	.1979	.0946
FFTNoise	.2151	.0316	.0017	.1642	.1147
FlippSample	.0325	.0071	.0004	.0401	.0185
HighPass	.2023	.0345	.0836	.1380	.1361
Invert	.0402	.0087	.0005	.0406	.0224
Levelling	.0445	.0008	.0005	.0401	.0244
LowPass	.1712	.1494	.1118	.1294	.1452
NoiseRemoval	.0755	.0250	.0010	.0925	.0443
Normalize	.1523	.0345	.0010	.0417	.0781
Nothing	.0297	.0063	.0004	.0396	.0172
Orthogonal 1	.0297	.0063	.0004	.0396	.0172
Özer	.0298	.0063	.0004	.0396	.0172
PitchDown	.0421	.0095	.0023	.0963	.0275
PitchUp	.0422	.0095	.0023	.0968	.0276
SpeedDown	.0730	.0606	.0483	.2578	.0742
SpeedStart	.0668	.0564	.0465	.2540	.0699
SpeedUp	.0744	.0719	.0442	.2567	.0748
Spread 1	.0299	.0063	.0004	.0396	.0173
Spread 3	.0299	.0063	.0005	.0396	.0173
SVDNoise	.1700	.0200	.0007	.1358	.0901
TempoDown	.0619	.0543	.0448	.2417	.0661
TempoUp	.0800	.0751	.0465	.2674	.0792S

Table 14: Özer Attack Error

7 Results

From the data detailed above, many conclusions can be drawn and conjectures can be made. We next detail each pertinent point and attempt to explain the origin of the behavior.

7.1 Orthogonal Schemes

As evidenced by both the bit encoding-decoding rates in Table 4 and the robustness attack results in Table 11, the Orthogonal family of watermarking schemes is unsuitable for actual watermarking applications. This is due to the nature of the Singular Value Decomposition. If the singular values are relatively invariant under orthogonal changes to the signal matrix, it can be inferred that the change will then be reflected in the orthogonal matrices, making these matrices unfit to hold information that must be preserved. In addition, each column in the orthogonal matrix is related to a single singular value. Because the singular values are extracted in an enforced non-increasing order, should any transformation affect the relative ordering of the singular values, that change in ordering will also be reflected by the swapping of columns in the orthogonal matrices. As the detection of the watermark is highly dependent on the position of the orthogonal values, swapping of columns will destroy the ability to successfully extract the information of the watermark.

A possible remedy that could address the problem of the reordering of the columns but not the inherent volatility of the domain itself is to devise a watermarking scheme that is not position dependent. That is, once the STFT frame is decomposed, resort the signal carrier such that the order of the columns is guaranteed. This will remove the dependence on the ordering of the singular values; the columns will be in the same order in which they were marked, and only the values within the columns may have changed.

Additionally, one major drawback to the Orthogonal Family of schemes was in the operation of the watermark detection. By not considering the direction of the difference between the old and new complex values and simply considering the magnitude, the bit domain {-1,1} could not be used. The use of the bit domain {-1,1} appears to be beneficial; the distance between valid bit values is greater, so it is theoretically easier to successfully categorize a raw value as its original bit value.

Overall, the Orthogonal system as it is described is insufficient, and the previous modifications would need to be made for it to be made a viable solution for watermarking.

7.2 Bit Spreading

Bit Spreading was only successfully implemented in one type of system. Recall that Orthogonal Bit Spreading could not be made to successfully identify bits in the absence of attack, and in Özer the modification was dismissed as not providing a tangible mathematical change to the operation of the system, thereby providing no likely benefit. Diagonal Bit Spreading, however, was implemented and tested, and some interesting conclusions can be drawn from comparing its performance to the tested Diagonal schemes.

The Diagonal Bit Spreading schemes performed notably better than Diagonal in the noise-based attacks (AddNoise, AddFFTNoise, AddSVDNoise). The bits embedded into the singular values of each STFT frame were flipped at random, such that both valid bit values were present in each frame. Should the audio signal be altered in such a way that all of the singular values are distorted in the same fashion, if there was only one type of bit in the values it would be more difficult to successfully classify the bit. Because both types of bits are present, if they are all altered, the two types of bits can still be compared against each other. They can be used as a frame of reference for each other, thereby making bit spreading more robust against certain types of attacks, including, as suggested by the results, noise.

This was the only major disparity between the two types of system; overall the systems behaved in a very similar fashion. However, it suggests that bit spreading is in fact a viable potential extension to existing watermarking schemes, provided that it can be integrated into the embedding and detection process without adversely altering its behavior and can be shown to otherwise change how the values are altered.

7.3 Diagonal Schemes

The Diagonal schemes (Diagonal and Diagonal Bit Spreading) were ineffective against most of the robustness attacks, and are, therefore, unsuitable as watermarking schemes. Their implementations were very similar to that of Özer's system but they differed in an important fashion: the Diagonal schemes affected only the singular values, whereas Özer affected all values in the singular value matrix. Each bit in Özer's system had a much greater spread among the entire spectrum, and as such it was far more robust against attack. The overall failure of the Diagonal schemes indicates that simple value manipulations are not sufficient for watermarking schemes in the Singular Value Decomposition domain. Such watermarking schemes are not spread throughout the spectrum sufficiently, and as such are not viable.

Upon looking for other potential reasons why the Diagonal schemes underperformed Özer, the detection scheme is also brought into question. Each bit was individually extracted from the marked singular value, instead of all at once. One question to consider is whether or not detecting the watermark bit by reconstructing the random signal used to embed the bit is any more accurate. Empirically, altering the detection scheme in the Diagonal family to mimic more closely that of Özer's did not have any appreciable affect. This suggests that the two detection schemes are more or less equivalent in functionality, and that any shortcomings of the Diagonal schemes are due solely to a deficient embedding scheme and not because of a deficient detection scheme.

7.4 Özer

The watermarking scheme proposed by Özer vastly outperformed all other tested watermarking schemes. However, it did not appear to perform nearly as well as published [16], which leads to questions regarding the disparity. One possibility is that the implementation used for testing was somehow flawed. This seems relatively unlikely, however, given how well the system performed. If there was a mistake in the implementation, it would seem far more likely that the system would not function at all. Another possibility is that the values returned from the PAQM were flawed, resulting in an inaccurate scaling of the watermark strength. This is, perhaps, more likely, but the lack of any PAQM benchmark tools precluded me from performing any verifiable tests on the PAQM short of listening to the files and determining the level of annoyance. As mentioned before, the watermark using a strength of 0.15 (the published value) resulted in a constant low-volume static, which does not fit the description as being "nearly imperceptible" [16]. The watermark at 0.015 however introduced no audible artifacts, which lends credence to the values produced by the PAQM. This leaves the possibility that the constant suggested by Özer is incorrect, and that an error was made in the paper.

At any rate, the tested implementation of Özer, while performing the best among all tested systems, did not perform as published. In particular, the system seems susceptible to noise-based attacks, but only in the audio samples with more pauses and quiet sections (single instrument and speech), and the system also had problems with attacks which changed the length or otherwise altered the alignment of the output file relative to the original audio (DelayStart, SpeedStart, TempoUp, TempoDown, SpeedUp, SpeedDown). This illustrates a major problem inherent will all of these systems: to function correctly, the detection data and the marked audio need to be aligned properly. These systems rely on the STFT frames being taken at the same relative portions, which determine where the watermark is applied and detected. Should the frames be misaligned, the system will see increased failure. This illustrates the need for some way to adaptively identify portions of the audio or otherwise ensure that the frames are properly aligned.

7.5 Music Type

As mentioned above, and as illustrated by the results, the type of music being marked certainly has an effect on how well the watermark is retrieved. The audio samples with more sounds (vocal and contemporary) were far easier to watermark. There is more cover noise, so the singular values are higher, which results in a more powerful watermark. As the watermark is stronger, it is also far easier to detect, resulting overall in far fewer errors. On the other hand, the samples with more silence were far more difficult to watermark. There was not as much cover, so the watermark is inherently weaker, and attacks that alter the sound of the file have a far greater effect, relative to the original signal, destroying the watermark.

7.6 Attacks

Some interesting conclusions could be drawn based on the tests as applied to all of the watermarking schemes in general. As discussed above, the attacks that altered the length or alignment of the output file (DelayStart, SpeedStart, TempoUp, TempoDown, SpeedUp, SpeedDown) were particularly effective, illustrating a fundamental weakness in the operation of these watermarking systems.

The amplitude attacks (Amp067 and Amp150) were found to have varying effects on detection. Namely, Amp067 caused far more errors than Amp150. The watermark is already embedded at the edge of perception; by making the audio track quieter, the parts of the watermark pass through all detection, human or otherwise. Similar issues did not occur on the Amp150 attack, as, if anything, the watermark signal is getting stronger.

Invert does not appear to have any effect on the SVD-type watermarking schemes, illustrating a fundamental resistance to these types of attacks. Also, as applied to samples watermarked by Özer, the second-watermark attacks overall were unsuccessful at destroying the original watermarks. This means that, in a potential copyright dispute, should a copyright infringer watermark the file with their own watermark to attempt to claim ownership, the rightful owner will still be able to detect the original watermark in the other file, while the culprit will not be able to detect theirs in the original.

8 Conclusions and Future Work

The surface of this topic has merely been scratched. Many potential techniques have been shown to be unsuitable for watermarking. Of the potential watermark signal carriers, the orthogonal matrices have been mostly shown as insufficient for the needs of a robust watermark, so future research should focus on the pseudo-diagonal matrix as a watermark carrier. Simple watermarking schemes have been shown to be inadequate, so systems that develop the entirety of the pseudo-diagonal matrix, like the one developed by Özer, should be further developed and explored. Additionally, bit spreading could be a viable alteration to suitable watermarking systems and as such deserves further investigation.

With respect to music samples, a large amount of work needs yet to be done. A broader set of music samples should be used, including purely instrumental pieces, in order to determine any differences in the suitability of watermarking by genre. The music samples should be analyzed based on other characteristics as well, including volume, to determine what else contributes to or detracts from the effectiveness of a watermarking scheme.

Above all, far more work must be done to research the effects of the frame-adjusting attacks as described in this paper. Given particular frame widths, how much must be added to an audio signal before the watermark is destroyed? Once the actual consequences of these attacks are known, ways to mitigate the attacks are needed. As the detection of all watermarking techniques presented in this paper are dependent on using the STFT frame to build the two-dimensional matrix for bit embedding, a frame-adjusting attack of

sufficient size may be possible to completely destroy a watermark. There are potentially many methods which hold promise and are worthy of further research. It may be possible to create a marker to denote when the actual watermarking begins so the detection frames can be properly aligned, but this requires that the marker is itself resistant to attack and does not adversely affect the quality of the audio. Frames of varying size could be used at embedding to place wider "marker" bits which can then be used to properly calibrate the alignment of the narrow frames for detecting the real watermark bits. This could potentially create a more complex detection scheme, and the marker bits could adversely affect the genuine watermark, but as most of the schemes presented in this paper are relatively resistant to the second-watermark attack, this will likely not be a problem. Alternately, the embedding, and therefore, the detection, can be modified to not use frames with the STFT at all, instead developing some other method to embed multiple bits in a signal carrier. This, however, could potentially have drawbacks relating to capacity and processing time which should be taken into account.

References

[1] Antoniou, A. *Digital Filters: Analysis, Design, and Applications*. McGraw-Hill, 1993.

[2] Beerends, J., and Stemerdink, J. A perceptual audio quality measure based on a psychoacoustic sound representation. *Journal of Audio Engineering Society 40*, 12 (December 1992), 963–978.

[3] Bennett, H. The authoritative blu-ray disc (bd) faq. *EMediaLive* (June 2006).

[4] Bennett, H. The authoritative hd dvd faq. *EMediaLive* (July 2006).

[5] Burns, E. Intervals, scales, and tuning. In *The Psychology of Music second edition*. Academic Press, 1999.

[6] Dilger, D. E. How fairplay works: Apple's itunes drm dilemma. *Roughly Drafted* (February 2007).

[7] Dittmann, J., Lang, A., Steinebach, M., and Petitcolas, F. *StirMark Benchmark for Audio (SMBA): Evaluation of watermarking schemes for audio*.

[8] Haitsma, J., van der Veen, M., Kalker, T., and Bruekers, F. Audio watermarking for monitoring and copy protection. In *Proceedings on the 2000 ACM workshops on Multimedia* (2000), pp. 119–122.

[9] Horn, R., and Johnson, C. *Matrix Analysis*. Cambridge University Press, 1985, ch. 7.3.

[10] Jin, H., Lotspiech, J., and Megiddo, N. Efficient traitor tracing. Tech. rep., IBM, October 2006.

[11] Kusyk, J., and Eskicioglu, A. A semi-blind logo watermarking scheme for color images by comparison and modification of dft coefficients. In *Proceedings of SPIE* (2005), pp. 1–15.

[12] Lang, A., and Dittman, J. Transparency and complexity benchmarking of audio watermarking algorithms issues. In *Proceedings of the 8th workshop on Multimedia and security* (2006), pp. 190–201.

[13] Larsen, E., and Aarts, R. *Audio Bandwidth Extension: Application of Psychoacoustics, Signal Processing, and Loudspeaker Design*. Wiley, 2004.

[14] McMillan, R. Settlement ends sony rootkit case. *PC World* (May 2006).

[15] Megías, D., Herrera-Joancomartí, J., and Minguillón, J. An audio watermarking scheme robust against stereo attacks. In *Proceedings of the 2004 workshop on Multimedia and security* (2004), pp. 206–213.

[16] Özer, H., Sankur, B., and Memon, N. An svd-based audio watermarking technique. In *Proceedings of the 7th workshop on Multimedia and security* (2005), pp. 51–56.

[17] Quan, X., and Zhang, H. Statistical audio watermarking algorithm based on perceptual analysis. In *Proceedings of the 5th ACM workshop on Digital rights management* (2005), pp. 112–118.

[18] Russinovich, M. Sony, rootkits and digital rights management gone too far. *Mark's Blog* (October 2006).

[19] Steinebach, M., Dittmann, J., Seibel, C., Ferri, L. C., Petitcolas, F., Fates, N., Fontaine, C., and Raynal, F. Stirmark benchmark: Audio watermarking attacks. In *International Conference on Information Technology: Coding and Computing* (2001).

[20] Steinebach, M., Zmudzinski, S., and Chen, F. The digital watermarking container: secure and efficient embedding. In *Proceedings of the 2004 workshop on Multimedia and security* (2004), pp. 199–205.

[21] Terzija, N., and Geisselhardt, W. Digital image watermarking using complex wavelet transform. In *Proceedings of the 2004 workshop on Multimedia and security* (2004), pp. 193–198.

[22] von Békésy, G. *Experiments in Hearing.* Acoustical Society of America Press, 1960.

[23] Wang, X.-Y., Cui, Y.-R., Yang, H.-Y., and Zhao, H. A new content-based digital audio watermarking algorithm for copyright protection. In *Proceedings of the 3rd international conference on Information security* (2004), pp. 62–68.

[24] Waters, G. Tech 3253 - sound quality assessment material (sqam). Tech. rep., European Broadcasting Union, July 2004.

[25] Xu, C., Zhu, Y., and Feng, D. D. Digital audio watermarking based-on multiple-bit hopping and human auditory system. In *Proceedings of the ninth ACM international conference on Multimedia* (2001), pp. 568–571.

[26] Zwicker, E. Subdivision of the audible frequency range into critical bands. *The Journal of the Acoustical Society of America 33* (February 1961).

A Glossary

AAC Advanced Audio Coding: A lossy compression and encoding scheme for digital audio. AAC is the most prevalent format for encoding audio CDs for Apple's iTunes and iPod, and is used in Sony's Playstation 3 and as the default audio codec for iTunes video files. AAC can only be played by authorized hardware [6].

AACS Advanced Access Content System: A method for content distribution and a form of digital rights management. AACS is used to restrict access to the next generation of DVDs and optical storage disks, most notably HD DVD and Blu-ray Disc [10].

Bark scale A psychoacoustical scale proposed by Earl Zwicker. The scale ranges from 1 to 24 and corresponds to the first 24 critical bands of hearing. The bark scale is used in the Perceptual Audio Quality Measure [26].

Critical bands of hearing The frequency bandwidths on the basilar membrane in the cochlea of the inner ear which facilitate the sense of hearing. Critical bands are fundamental in human perception of loudness [22].

Diagonal A square matrix with values over the diagonal (where the row and column index are equal), and zeroes elsewhere.

DRM Digital Rights Management: A term for the collective schemes and technologies designed to protect the rights of the copyright holder of digital content.

FFT Fast Fourier Transform: An invertible process to transform data from the time domain to the frequency domain.

HAS Human Auditory System: The system of nerves and organs through which humans perceive audio.

HDCP High-bandwidth Digital Content Protection: A form of digital rights management used to control digital multimedia content as it travels across connections between devices. Products using HDCP agree to limit the functionality of their devices to help ensure the content cannot be misused [3, 4].

MOS Mean Opinion Score: A survey-based method of determining the amount of distortion between two different signals. MOS ranges over a scale from 1 (annoying) to 5 (imperceptible).

MPEG Moving Picture Experts Group, or more commonly, the data compression standards created by the group.

Orthogonal A matrix whose columns are pairwise orthogonal, meaning the dot product (the sum of an element-wise multiplication) of any two columns is equal to zero.

PAQM Perceptual Audio Quality Measure: A psychoacoustic system which measures the amount of disturbance between two audio signals. This process correlates strongly with the Mean Opinion Score [2].

Pseudo-diagonal A non-square matrix with values over the diagonal (where the row and column index are equal), and zeroes elsewhere.

Psychoacoustics The study of the subjective human perception of sounds.

Quarter tone One half of the smallest interval in 12 tone equal temperament. A quarter tone is equivalent to 50 cents, or a frequency ratio of $2^(1/24)$ [5].

Rootkit A group of programs designed to gain and maintain undetectable access and control over the highest level of access in a computer system.

SMBA StirMark Benchmark for Audio: A set of standardized attacks used to measure the robustness of audio watermarking schemes.

Square A matrix with an equal number of rows and columns.

STFT Short Time Fourier Transform (alternately, Short Term Fourier Transform): An invertible process which combines a Fourier transform with a sliding window, yielding a matrix of values corresponding to time-by-frequency. The size of the windowing function and the amount of overlap determines the frequency and time resolution.

SVD Singular Value Decomposition: A process by which the singular values of a matrix are extracted, yielding three matrices, U, D, and V^T. U and V^T are square orthogonal matrices sharing the first and second dimension, respectively, with the original matrix. D is a pseudo-diagonal matrix with the same dimensions as the input matrix populated by the singular values of the original matrix along the diagonal in non-increasing order.

B Source Code

```
"""
attacks.py
This file contains functions which alter the contents of input audio files to
test the robustness of the watermark.  If possible, the attack effect is
scaled to yield a PAQM of 0.01.
"""

from pylab import *
from FFT import real_fft, inverse_real_fft
from singular_value import full_svd
from short_time import stft, inverse_stft
import random
import copy


def add_noise( audio_data, strength=500 ):
    '''
    Adds a uniformly distributed random signal
    to the audio_data with the given strength.
    '''

    rand = random.Random()
    noise = zeros( len(audio_data), 'i' )
    for i in range(len(noise)):
        noise[i] = int(rand.random() * (2 * (strength+1)) - (strength+1))

    return ensure_range( audio_data + noise )


def delay_start( audio_data, samples=1600 ):
    '''
    Adds the specified number of zeroes to
    the front of the audio file.
    '''

    return array( list(zeros(samples, 'i')) + list(audio_data) )


def speed_start( audio_data, samples=1600 ):
    '''
    Removes the specified number of samples
    from the front of the audio file.
    '''

    return audio_data[samples:]


def add_fft_noise( audio_data, strength = 200000 ):
    '''
    Adds a uniformly distributed random signal
    in the FFT domain.
    '''

    rand = random.Random()
    Fxx = real_fft(audio_data)
    for i in range(len(Fxx)):
        real_rand = rand.random() * (2*strength) - strength
        imag_rand = (0+1j)*(rand.random() * (2*strength) - strength)
        Fxx[i] += ( real_rand + imag_rand )
    return ensure_range(inverse_real_fft(Fxx))


def flipp_sample( audio_data, period=16000, distance=400, count=8 ):
    '''
    Every [period] samples, swaps sections of samples of [count]
    length with sections [distance] samples away.
    '''

    new_aud = copy.deepcopy(audio_data)
    for i in range(0, len(new_aud)-(distance+count)+1, period):
        new_aud[i:i+count] = audio_data[i+distance:i+distance+count]
        new_aud[i+distance:i+distance+count] = audio_data[i:i+count]
    return new_aud


def add_svd_noise( audio_data, strength=600 ):
```

```
    '''
    Adds a uniformly distributed random signal
    in the SVD domain.
    '''

    rand = random.Random()
    Pxx = stft(audio_data)
    ind = arange(0, len(Pxx)-len(Pxx[0])+1, len(Pxx[0]))
    for f in range(len(ind)):
        frame = Pxx[ind[f]:ind[f]+len(Pxx[0])]
        u, d, vt = full_svd( frame )
        for i in range(len(d)):
            d[i,i] = d[i,i] + rand.random() * (2*strength) - strength
            if d[i,i] < 0:
                d[i,i] = 0
        Pxx[ind[f]:ind[f]+len(Pxx[0])] = dot( dot(u, d), vt )
    return ensure_range(inverse_stft( Pxx ))


def low_pass( audio_data, cutoff=993., Fs=16000. ):
    '''
    Mimics the functioning of a low pass filter.
    Filters out high frequencies.
    '''

    Fxx = real_fft( audio_data )
    freqs = Fs / len(Fxx) * arange(len(Fxx))

    starter = int( ceil( cutoff * len(Fxx) / Fs ) )
    for i in range( starter, len(Fxx) ):
        Fxx[i] *= cutoff / freqs[i]

    return ensure_range(inverse_real_fft( Fxx ))


def high_pass( audio_data, cutoff=993., Fs=16000. ):
    '''
    Mimics the functioning of a high pass filter.
    Filters out low frequencies.
    '''

    Fxx = real_fft( audio_data )
    freqs = Fs / len(Fxx) * arange(len(Fxx))

    ender = int( cutoff * len(Fxx) / Fs )
    for i in range( ender ):
        Fxx[i] *= freqs[i] / cutoff

    return ensure_range(inverse_real_fft( Fxx ))


def ensure_range( audio ):
    '''
    Ensures that the signals in audio are within
    the legal bounds for 16-bit.
    '''

    retVal = copy.deepcopy(audio)
    for i in range(len(retVal)):
       if retVal[i] < -(2**15):
            retVal[i] = 0
       elif retVal[i] >= 2**15:
            retVal[i] = 0
    return retVal


"""
audio.py
Functions which load in and write to file audio data.
"""

from pylab import *
import wave
import struct

def load_wave_array(filename):
    '''
    Load a sound file in WAV format and return as an array.
```

```
    Supports 8 and 16 bit sample resolution, single channel (mono).
    Sample rate is not determined or returned.

    Uses the left channel in stereo files.
    '''

    wav = wave.open(filename)

    if wav.getnchannels() == 1:
        data = wav.readframes(wav.getnframes())

        if wav.getsampwidth() == 1:
            typecode = UInt8
            return array(data, typecode)

        elif wav.getsampwidth() == 2:
            signal = []
            for i in range(len(data)/2):
                frame = data[i*2:(i+1)*2]
                signal.append(struct.unpack('h', frame))
            return squeeze(array(signal))

        else:
            raise 'loadwavearray Error', '%s byte sample width not supported' % wav.getsampwidth()

    elif wav.getnchannels() == 2:
        data = wav.readframes(wav.getnframes())

        if wav.getsampwidth() == 1:
            typecode = UInt8
            return array([data[2*i] for i in range(len(data)/2)], typecode)

        elif wav.getsampwidth() == 2:
            signal = []
            for i in range(len(data)/4):
                frame = data[i*4:i*4+2]
                signal.append(struct.unpack('h', frame))
            return squeeze(array(signal))

    else:
        raise 'loadwavearray Error', '%s channels not supported' % wav.getnchannels()



def pack_wave_array(aud_out, file_to_write, Fs=16000):
    '''
    Write the given audio data to a wav file.
    '''
    wav = wave.open(file_to_write, 'w')
    wav.setnchannels(1)
    wav.setsampwidth(2)
    wav.setframerate(Fs)

    signal = ''
    for i in range(len(aud_out)):
        signal += struct.pack('h', aud_out[i:i+1])

    wav.writeframes(signal)
    wav.close()


"""
bark.py
Functions and data corresponding to the psychoacoustical bark scale.
"""

# Bark scale interval edges
bark_edges = [0, 100, 200, 300, 400, 510, 630, 770, 920, 1080, 1270, 1480, 1720, 2000, 2320, 2700, 3150, 3700,
    4400, 5300, 6400, 7700, 9500, 12000, 15500]

# Bark scale interval widths
bark_widths = []

# Bark scale interval centers
bark_centers = []
for i in range(1,25):
    bark_widths.append( bark_edges[i] - bark_edges[i-1] )
```

```
    bark_centers.append( (bark_edges[i] + bark_edges[i-1])/2 )


def freq_to_bark(freq):
    '''
    Converts from a frequency to the corresponding value
    on the bark scale.
    '''

    if freq < bark_edges[0] or freq > bark_edges[-1]:
        return None

    bark_val = 1
    while bark_edges[bark_val] < freq:
        bark_val += 1

    return bark_val


"""
bit_utils.py
Helper functions to convert from strings to arrays of bits and
back, and also for Hamming7-11 encoding/decoding.
"""

from pylab import *
import struct

def extract_bits(message):
    '''
    Given a string, return the array of bits.
    '''
    bits = []
    for i in range(len(message)):
        byte = struct.unpack('B', message[i])[0]
        for j in range(8):
            tmp = byte / (2**(7-j))
            bits += [tmp]
            byte -= tmp * (2**(7-j))
    return array(bits)


def construct_message(bits):
    '''
    Given an array of bits, return the string.
    '''
    message = ''
    for i in range( len(bits) / 8 ):
        byte = zeros(1, UInt8)
        val = 0
        for j in range(8):
            val *= 2
            val += bits[ i*8 + j ]
        byte[0] = val
        message += struct.unpack( 'c', byte )[0]
    return message


def extract_hamming_unicode_bits(message):
    '''
    Given a string, return the hamming7-11
    encoded bits (ignore the first bit)
    '''
    h_bits = []
    for i in range(len(message)):
        bits = []
        byte = struct.unpack('B', message[i])[0]
        for j in range(7):
            tmp = byte / (2**(6-j))
            bits += [tmp]
            byte -= tmp * (2**(6-j))
        h_bits += list(encode_hamming_11_7(bits))
    return array(h_bits)


def construct_hamming_unicode_message(h_bits):
    '''
    Given hamming encoded bits, return the string.
```

```
    '''
    message = ''
    for i in range( len(h_bits) / 11 ):
        bits = decode_hamming_11_7(h_bits[i*11:(i+1)*11])
        byte = zeros(1, UInt8)
        val = 0
        for j in range(7):
            val *= 2
            val += bits[j]
        byte[0] = val
        message += struct.unpack( 'c', byte )[0]
    return message



def encode_hamming_11_7(bits):
    '''
    Given 7 bits, return the 11 bit hamming code.
    '''
    h_bits = zeros(11)
    h_bits[2] = bits[0]
    h_bits[4] = bits[1]
    h_bits[5] = bits[2]
    h_bits[6] = bits[3]
    h_bits[8] = bits[4]
    h_bits[9] = bits[5]
    h_bits[10] = bits[6]

    h_bits[0] = (h_bits[2]+h_bits[4]+h_bits[6]+h_bits[8]+h_bits[10])%2
    h_bits[1] = (h_bits[2]+h_bits[5]+h_bits[6]+h_bits[9]+h_bits[10])%2
    h_bits[3] = (h_bits[4]+h_bits[5]+h_bits[6])%2
    h_bits[7] = (h_bits[8]+h_bits[9]+h_bits[10])%2

    return h_bits


def decode_hamming_11_7(h_bits):
    '''
    Given 11 bits, return the 7 bit codeword.
    '''
    wrong_pos = 0 #this is 1-indexed
    if( (h_bits[0]+h_bits[2]+h_bits[4]+h_bits[6]+h_bits[8]+h_bits[10])%2 ):
        wrong_pos += 1
    if( (h_bits[1]+h_bits[2]+h_bits[5]+h_bits[6]+h_bits[9]+h_bits[10])%2 ):
        wrong_pos += 2
    if( (h_bits[3]+h_bits[4]+h_bits[5]+h_bits[6])%2 ):
        wrong_pos += 4
    if( (h_bits[7]+h_bits[8]+h_bits[9]+h_bits[10])%2 ):
        wrong_pos += 8

    if( wrong_pos > 0 and wrong_pos <= 11 ):
        h_bits[wrong_pos-1] = (h_bits[wrong_pos-1]+1)%2

    bits = []
    bits += [h_bits[2]]
    bits += [h_bits[4]]
    bits += [h_bits[5]]
    bits += [h_bits[6]]
    bits += [h_bits[8]]
    bits += [h_bits[9]]
    bits += [h_bits[10]]

    return bits


"""
check_attack_error.py
This script runs over all of the attacked files generated from run_attacks.py,
determines the number of bits in each file that were incorrectly identified,
and prints the results to attack_error.txt.
"""

import ozer
import diagonal
import diag_bit_spread
import orthogonal

import os
```

```
import audio
import bit_utils

orig_path = '../audio/all16'
orig_files = os.listdir(orig_path)
attack_path = '../attacked'
attack_dir = os.listdir(attack_path)
key = 1
message = 'Hello'
bits = bit_utils.extract_bits( message )

out_file = open('attack_error.txt', "w")

for f in range(len(orig_files)):

    print orig_files[f]
    # get the original audio
    orig_audio = audio.load_wave_array( orig_path + '/' + orig_files[f] )

    # mark with d0
    print '  di_0_'
    tmp, old_diag, old_ind = diagonal.watermark_audio_data( orig_audio, message, key, 0 )
    out_file.write( 'di_0_' )
    out_file.write( orig_files[f] )
    out_file.write( ', ' )
    # check all attacks
    for a in range( len( attack_dir ) ):
        print '    attack', a, 'of', len(attack_dir)
        file = attack_path + '/' + attack_dir[a] + '/di_0_' + orig_files[f]
        attacked_audio = audio.load_wave_array( file )
        new_bits = diagonal.detect_watermark( attacked_audio[:len(orig_audio)], old_diag, old_ind, key, 0 )

        error = 0
        for i in range(len(new_bits)):
            if( new_bits[i] != bits[i % len(bits)] ):
                error += 1
        out_file.write( str(error) )
        out_file.write( ", " )
    out_file.write('\n')

    # mark with d1
    print '  di_1_'
    tmp, old_diag, old_ind = diagonal.watermark_audio_data( orig_audio, message, key, 1 )
    out_file.write( 'di_1_' )
    out_file.write( orig_files[f] )
    out_file.write( ', ' )
    # check all attacks
    for a in range( len( attack_dir ) ):
        print '    attack', a, 'of', len(attack_dir)
        file = attack_path + '/' + attack_dir[a] + '/di_1_' + orig_files[f]
        attacked_audio = audio.load_wave_array( file )
        new_bits = diagonal.detect_watermark( attacked_audio[:len(orig_audio)], old_diag, old_ind, key, 1 )

        error = 0
        for i in range(len(new_bits)):
            if( new_bits[i] != bits[i % len(bits)] ):
                error += 1
        out_file.write( str(error) )
        out_file.write( ", " )
    out_file.write('\n')

    # don't need to mark with orth 1
    print '  or_1_'
    out_file.write( 'or_1_' )
    out_file.write( orig_files[f] )
    out_file.write( ', ' )
    # check all attacks
    for a in range( len( attack_dir ) ):
        print '    attack', a, 'of', len(attack_dir)
        file = attack_path + '/' + attack_dir[a] + '/or_1_' + orig_files[f]
        attacked_audio = audio.load_wave_array( file )
        new_bits = orthogonal.detect_watermark( attacked_audio[:len(orig_audio)], orig_audio, key, 1 )

        error = 0
        for i in range(len(new_bits)):
            if( new_bits[i] != bits[i % len(bits)] ):
                error += 1
        out_file.write( str(error) )
```

```
            out_file.write( ", " )
        out_file.write('\n')

        # mark with ozer
        print '  oz_'
        tmp, detect = ozer.watermark_audio_data( orig_audio, message, key )
        out_file.write( 'oz_' )
        out_file.write( orig_files[f] )
        out_file.write( ', ' )
        # check all attacks
        for a in range( len( attack_dir ) ):
            print '    attack', a, 'of', len(attack_dir)
            file = attack_path + '/' + attack_dir[a] + '/oz_' + orig_files[f]
            attacked_audio = audio.load_wave_array( file )
            new_bits = ozer.detect_watermark( attacked_audio[:len(orig_audio)], detect, key )

            error = 0
            for i in range(len(new_bits)):
                if( new_bits[i] != bits[i % len(bits)] ):
                    error += 1
            out_file.write( str(error) )
            out_file.write( ", " )
        out_file.write('\n')

        # mark with spread 1
        print '  sp_1_'
        tmp, old_diag, old_ind = diag_bit_spread.watermark_audio_data( orig_audio, message, key, 1 )
        out_file.write( 'sp_1_' )
        out_file.write( orig_files[f] )
        out_file.write( ', ' )
        # check all attacks
        for a in range( len( attack_dir ) ):
            print '    attack', a, 'of', len(attack_dir)
            file = attack_path + '/' + attack_dir[a] + '/sp_1_' + orig_files[f]
            attacked_audio = audio.load_wave_array( file )
            new_bits = diag_bit_spread.detect_watermark( attacked_audio[:len(orig_audio)], old_diag, old_ind, key, 1 )

            error = 0
            for i in range(len(new_bits)):
                if( new_bits[i] != bits[i % len(bits)] ):
                    error += 1
            out_file.write( str(error) )
            out_file.write( ", " )
        out_file.write('\n')

        # mark with spread 3
        print '  sp_3_'
        tmp, old_diag, old_ind = diag_bit_spread.watermark_audio_data( orig_audio, message, key, 3 )
        out_file.write( 'sp_3_' )
        out_file.write( orig_files[f] )
        out_file.write( ', ' )
        # check all attacks
        for a in range( len( attack_dir ) ):
            print '    attack', a, 'of', len(attack_dir)
            file = attack_path + '/' + attack_dir[a] + '/sp_3_' + orig_files[f]
            attacked_audio = audio.load_wave_array( file )
            new_bits = diag_bit_spread.detect_watermark( attacked_audio[:len(orig_audio)], old_diag, old_ind, key, 3 )

            error = 0
            for i in range(len(new_bits)):
                if( new_bits[i] != bits[i % len(bits)] ):
                    error += 1
            out_file.write( str(error) )
            out_file.write( ", " )
        out_file.write('\n')

out_file.close()


"""
diag_bit_spread.py
Functions to embed watermarks into an audio file
using the Diagonal Bit Spreading family.
"""

from pylab import *
from singular_value import *
from short_time import stft, inverse_stft
```

```
from bit_utils import *
import random
import copy

def watermark_audio_data(audio_data, message, key, scheme):
    '''
    Embeds an audio carrier with a watermark signal in a
    variety of ways, dependent on the scheme value.

    Embedding Targets:
    Schemes 0-3: All diagonal values
    Schemes 4-7: One diagonal value (picked at random)
    Schemes 8-12: Two diagonal values (at random, must be different)
    Schemes 13-16: A random number of values (at random)

    Random values:
    Schemes 0-1 % 4: [0,1)
    Schemes 2-3 % 4: [-1,1)

    Bits:
    Schemes 0 % 2: {0,1}
    Schemes 1 % 2: {-1,1}
    '''

    prng, randMult, randAdd = get_random_params(key, scheme)
    watermarkBits = get_watermark_bits(message, scheme)

    # run STFT on the audio data to yield
    # matrix of time x freq
    Pxx = stft(audio_data)

    # Magic number: taking 8 time slices
    # from the STFT results yields a bitrate
    # of roughly 16 watermark bits per second of
    # audio data
    bps = 16
    frame_width = 128 / bps

    a = get_watermark_strength(scheme)

    # figure out which indices our frames start at
    ind = arange(0, len(Pxx)-frame_width+1, frame_width)

    # keep the old diagonal values
    old_diag = []

    # keep the old indices
    old_ind = []

    # for each frame:
    for f in range(len(ind)):
        frame = Pxx[ind[f]:ind[f]+frame_width]
        bit = watermarkBits[f % len(watermarkBits)]

        # run SVD on the frame
        u, s, vt = list_svd(frame)

        # determine which diagonal values need to be marked
        target_values = get_target_values(frame_width, scheme, prng)

        new_sing = copy.copy( s )

        # embed
        for val in target_values:
            randVal = (prng.random() * randMult) + randAdd
            new_sing[val] += a * flip(bit, prng, scheme) * s[val] * randVal

        # build the new diagonal values
        dw = bloat_diagonal_list( new_sing, len(u), len(vt) )

        # need to return the old diagonal values
        old_diag += list(s)

        # need to mark whether or not the ordering of the new values
        tmp = sort( new_sing )
        tmp = array([tmp[len(tmp)-i-1] for i in range(len(tmp))])
        for i in range(len(tmp)):
            old_ind += [ find([tmp[i] == new_sing[j] for j in range(len(new_sing))])[0] ]
```

```
        # calculate new frame
        Pxx[ind[f]:ind[f]+frame_width] = dot( dot(u, dw), vt )

    # Run stft inverse
    new_audio_data = inverse_stft( Pxx )

    return new_audio_data, old_diag, old_ind


def detect_watermark(marked_audio, old_diag, old_ind, key, scheme):
    '''
    Given the parameters, detect and return the watermark.
    '''

    prng, randMult, randAdd = get_random_params(key, scheme)

    Pxx = stft(marked_audio)

    bps = 16
    frame_width = 128 / bps
    a = get_watermark_strength(scheme)
    ind = arange(0, len(Pxx)-frame_width+1, frame_width)

    bits = []

    # for each frame:
    for f in range(len(ind)):
        frame = Pxx[ind[f]:ind[f]+frame_width]

        # run SVD on the frame
        u, s, vt = list_svd(frame)

        # reorder the diagonal values to account for value altering
        diag = zeros( len(s), 'f' )
        for i in range(frame_width):
            diag[ old_ind[8*f + i] ] = s[i]

        # determine which diagonal values need to be checked for marks
        target_values = get_target_values(frame_width, scheme, prng)

        # un-embed
        tmp_bits = []
        for val in target_values:
            randVal = (prng.random() * randMult) + randAdd
            if (old_diag[8*f+val] * randVal * a != 0):
                bit = (diag[val] - old_diag[8*f+val]) / (old_diag[8*f+val] * randVal * a)
                bit = val_to_bit( bit, scheme )
                bit = flip(bit, prng, scheme)
                tmp_bits += [bit]
            else:
                # need to call random to keep symmetry
                randVal = prng.random()

        if len(tmp_bits) > 0:
            bits += [median(tmp_bits)]
        else:
            bits += [0]

    return get_watermark_message(bits, scheme)


def get_target_values(num_diagonal, scheme, prng):
    '''
    Returns the indices needed for marking

    Embedding Targets:
    Schemes 0-3: All diagonal values
    Schemes 4-7: One diagonal value (picked at random)
    Schemes 8-12: Two diagonal values (at random, must be different)
    Schemes 13-16: A random number of values (at random)
    '''

    if scheme < 4:
        return arange(num_diagonal)
    elif scheme < 8:
        return array([ int(prng.random() * num_diagonal) ])
    elif scheme < 12:
```

```
        val1 = int(prng.random() * num_diagonal)
        val2 = val1
        while( val2 == val1 ):
            val2 = int(prng.random() * num_diagonal)
        return array([val1, val2])
    else:
        vals = []
        num_to_get = int(prng.random() * num_diagonal) + 1
        while( len(vals) < num_to_get ):
            trial = int(prng.random() * num_diagonal)
            while( len(find([vals[i]==trial for i in range(len(vals))])) ):
                trial = int(prng.random() * num_diagonal)
            vals += [trial]
        return array(vals)


def get_watermark_bits(message, scheme):
    '''
    Given a message, return the bits to be embedded.
    '''
    watermarkBits = extract_bits( message )
    if( scheme % 2 ):
        watermarkBits -= watermarkBits == 0
    return watermarkBits


def get_random_params(key, scheme):
    '''
    Seed and return a pseudo-random number generator.
    '''
    prng = random.Random()
    prng.seed(key)
    randMult = 1 + (scheme % 4 > 1)
    randAdd = - (scheme % 4 > 1)
    return prng, randMult, randAdd


def get_watermark_message(bits, scheme):
    '''
    Given raw bit values, return the actual bits
    of the message.
    '''
    if( scheme % 2 ):
        return (array(bits) >= 0)
    else:
        return (array(bits) >= .5)


def get_watermark_strength(scheme):
    '''
    Return the appropriate strength for the watermarking scheme.
    '''
    a = [0.02, 0.02, 0.035, 0.02, 0.12, 0.085, 0.16, 0.085, 0.07, 0.05, 0.08, 0.05, 0.035, 0.03, 0.05, 0.03]

    return a[scheme]


def val_to_bit(val, scheme):
    '''
    Convert a raw value to the nearest legal bit.
    '''
    if scheme % 2:
        return 2 * (val >= 0) - 1
    else:
        return int(val >= .5)


def flip(bit, prng, scheme):
    '''
    Randomly flip the given bit to the
    other value (50% of the time)
    '''
    retBit = bit
    if scheme % 2:               # domain {-1,1}
        if prng.random() < .5:   # flip
            retBit = -bit
    else:                        # domain {0,1}
        if prng.random() < .5:    # flip
```

```
            retBit = 1 - bit
    return retBit


"""
diagonal.py
Functions used to embed watermarks into audio
using the Diagonal family.
"""

from pylab import *
from singular_value import *
from short_time import stft, inverse_stft
from bit_utils import *
import random
import copy

def watermark_audio_data(audio_data, message, key, scheme):
    '''
    Embeds an audio carrier with a watermark signal in a
    variety of ways, dependent on the scheme value.

    Embedding Targets:
    Schemes 0-3: All diagonal values
    Schemes 4-7: One diagonal value (picked at random)
    Schemes 8-12: Two diagonal values (at random, must be different)
    Schemes 13-16: A random number of values (at random)

    Random values:
    Schemes 0-1 % 4: [0,1)
    Schemes 2-3 % 4: [-1,1)

    Bits:
    Schemes 0 % 2: {0,1}
    Schemes 1 % 2: {-1,1}
    '''

    prng, randMult, randAdd = get_random_params(key, scheme)
    watermarkBits = get_watermark_bits(message, scheme)

    # run STFT on the audio data to yield
    # matrix of time x freq
    Pxx = stft(audio_data)

    # Magic number: taking 8 time slices
    # from the STFT results yields a bitrate
    # of roughly 16 watermark bits per second of
    # audio data
    bps = 16
    frame_width = 128 / bps

    a = get_watermark_strength(scheme)

    # figure out which indices our frames start at
    ind = arange(0, len(Pxx)-frame_width+1, frame_width)

    # keep the old diagonal values
    old_diag = []

    # keep the old indices
    old_ind = []

    # for each frame:
    for f in range(len(ind)):
        frame = Pxx[ind[f]:ind[f]+frame_width]
        bit = watermarkBits[f % len(watermarkBits)]

        # run SVD on the frame
        u, s, vt = list_svd(frame)

        # determine which diagonal values need to be marked
        target_values = get_target_values(frame_width, scheme, prng)

        new_sing = copy.copy( s )

        # embed
        for val in target_values:
            randVal = (prng.random() * randMult) + randAdd
```

```
            new_sing[val] += a * bit * s[val] * randVal

        # build the new diagonal values
        dw = bloat_diagonal_list( new_sing, len(u), len(vt) )

        # need to return the old diagonal values
        old_diag += list(s)

        # need to mark whether or not the ordering of the new values
        tmp = sort( new_sing )
        tmp = array([tmp[len(tmp)-i-1] for i in range(len(tmp))])
        for i in range(len(tmp)):
            old_ind += [ find([tmp[i] == new_sing[j] for j in range(len(new_sing))])[0] ]

        # calculate new frame
        Pxx[ind[f]:ind[f]+frame_width] = dot( dot(u, dw), vt )

    # Run stft inverse
    new_audio_data = inverse_stft( Pxx )

    return new_audio_data, old_diag, old_ind


def detect_watermark(marked_audio, old_diag, old_ind, key, scheme):
    '''
    Given the audio, return the watermark bits embedded.
    '''
    prng, randMult, randAdd = get_random_params(key, scheme)

    Pxx = stft(marked_audio)

    bps = 16
    frame_width = 128 / bps
    a = get_watermark_strength(scheme)
    ind = arange(0, len(Pxx)-frame_width+1, frame_width)

    bits = []

    # for each frame:
    for f in range(len(ind)):
        frame = Pxx[ind[f]:ind[f]+frame_width]

        # run SVD on the frame
        u, s, vt = list_svd(frame)

        # reorder the diagonal values to account for value altering
        diag = zeros( len(s), 'f' )
        for i in range(frame_width):
            diag[ old_ind[8*f + i] ] = s[i]

        # determine which diagonal values need to be checked for marks
        target_values = get_target_values(frame_width, scheme, prng)

        # un-embed
        tmp_bits = []
        for val in target_values:
            randVal = (prng.random() * randMult) + randAdd
            if (old_diag[8*f+val] * randVal * a != 0):
                bit = (diag[val] - old_diag[8*f+val]) / (old_diag[8*f+val] * randVal * a)
                tmp_bits += [bit]

        if len(tmp_bits) > 0:
            bits += [median(tmp_bits)]
        else:
            # todo: need to better handle the zero case...
            bits += [0]

    return get_watermark_message(bits, scheme)


def get_target_values(num_diagonal, scheme, prng):
    '''
    Returns the indices needed for marking

    Embedding Targets:
    Schemes 0-3: All diagonal values
    Schemes 4-7: One diagonal value (picked at random)
    Schemes 8-12: Two diagonal values (at random, must be different)
```

```
    Schemes 13-16: A random number of values (at random)
    '''

    if scheme < 4:
        return arange(num_diagonal)
    elif scheme < 8:
        return array([ int(prng.random() * num_diagonal) ])
    elif scheme < 12:
        val1 = int(prng.random() * num_diagonal)
        val2 = val1
        while( val2 == val1 ):
            val2 = int(prng.random() * num_diagonal)
        return array([val1, val2])
    else:
        vals = []
        num_to_get = int(prng.random() * num_diagonal) + 1
        while( len(vals) < num_to_get ):
            trial = int(prng.random() * num_diagonal)
            while( len(find([vals[i]==trial for i in range(len(vals))])) ):
                trial = int(prng.random() * num_diagonal)
            vals += [trial]
        return array(vals)


def get_watermark_bits(message, scheme):
    '''
    Given a message, return the bits
    to embed.
    '''
    watermarkBits = extract_bits( message )
    if( scheme % 2 ):
        watermarkBits -= watermarkBits == 0
    return watermarkBits


def get_random_params(key, scheme):
    '''
    Seed and return a PRNG
    '''
    prng = random.Random()
    prng.seed(key)
    randMult = 1 + (scheme % 4 > 1)
    randAdd = - (scheme % 4 > 1)
    return prng, randMult, randAdd


def get_watermark_message(bits, scheme):
    '''
    Given raw bit values, return the array of
    legal bits.
    '''
    if( scheme % 2 ):
        return (array(bits) >= 0)
    else:
        return (array(bits) >= .5)


def get_watermark_strength(scheme):
    '''
    Apropos strengths for the watermarks based on scheme.
    '''
    a = [0.023438, 0.01875, 0.032813, 0.021094, 0.1125, 0.075, 0.15, 0.075, 0.065625, 0.046875,
        0.075, 0.046875, 0.0375, 0.028125, 0.046875, 0.028125]

    return a[scheme]


"""
diagonal_ozd.py
Functions which embed watermarks like diagonal.py, but
detects the watermarks like ozer.py
"""

from pylab import *
from singular_value import *
from short_time import stft, inverse_stft
from bit_utils import *
import random
```

```
import copy

def watermark_audio_data(audio_data, message, key, scheme):
    '''
    Embeds an audio carrier with a watermark signal in a
    variety of ways, dependent on the scheme value.

    Embedding Targets: All diagonal values

    Random values:
    Schemes 0 % 2: [0,1)
    Schemes 1 % 2: [-1,1)

    Bits: {-1,1}
    '''

    prng, randMult, randAdd = get_random_params(key, scheme)
    watermarkBits = get_watermark_bits(message)

    # run STFT on the audio data to yield
    # matrix of time x freq
    Pxx = stft(audio_data)

    # Magic number: taking 8 time slices
    # from the STFT results yields a bitrate
    # of roughly 16 watermark bits per second of
    # audio data
    bps = 16
    frame_width = 128 / bps

    a = get_watermark_strength(scheme)

    # figure out which indices our frames start at
    ind = arange(0, len(Pxx)-frame_width+1, frame_width)

    # keep the old diagonal values
    old_diag = []

    # keep the old indices
    old_ind = []

    # for each frame:
    for f in range(len(ind)):
        frame = Pxx[ind[f]:ind[f]+frame_width]
        bit = watermarkBits[f % len(watermarkBits)]

        # run SVD on the frame
        u, s, vt = list_svd(frame)

        # determine which diagonal values need to be marked
        target_values = arange(frame_width)

        new_sing = copy.copy( s )

        # embed
        for val in target_values:
            randVal = (prng.random() * randMult) + randAdd
            new_sing[val] += a * bit * s[val] * randVal

        # build the new diagonal values
        dw = bloat_diagonal_list( new_sing, len(u), len(vt) )

        # need to return the old diagonal values
        old_diag += list(s)

        # need to mark whether or not the ordering of the new values
        tmp = sort( new_sing )
        tmp = array([tmp[len(tmp)-i-1] for i in range(len(tmp))])
        for i in range(len(tmp)):
            old_ind += [ find([tmp[i] == new_sing[j] for j in range(len(new_sing))])[0] ]

        # calculate new frame
        Pxx[ind[f]:ind[f]+frame_width] = dot( dot(u, dw), vt )

    # Run stft inverse
    new_audio_data = inverse_stft( Pxx )

    return new_audio_data, old_diag, old_ind
```

```
def detect_watermark(marked_audio, old_diag, old_ind, key, scheme):
    '''
    Given audio, return the watermark bits.
    '''
    prng, randMult, randAdd = get_random_params(key, scheme)

    Pxx = stft(marked_audio)

    bps = 16
    frame_width = 128 / bps
    a = get_watermark_strength(scheme)
    ind = arange(0, len(Pxx)-frame_width+1, frame_width)

    bits = []

    # for each frame:
    for f in range(len(ind)):
        frame = Pxx[ind[f]:ind[f]+frame_width]

        # run SVD on the frame
        u, s, vt = list_svd(frame)

        # reorder the diagonal values to account for value altering
        diag = zeros( len(s), 'f' )
        for i in range(frame_width):
            diag[ old_ind[8*f + i] ] = s[i]

        # determine which diagonal values need to be checked for marks
        target_values = arange(frame_width)

        randVals = []

        # un-embed
        tmp_bits = []
        for val in target_values:
            randVals += [(prng.random() * randMult) + randAdd]

        randVals = array(randVals)
        oldD = copy.deepcopy( array(old_diag[8*f:8*(f+1)]) )

        newRands = (diag - oldD) / (a * oldD)

        if sum(randVals * newRands) > 0:
            bits += [1]
        else:
            bits += [-1]

    return get_watermark_message(bits)


def get_watermark_bits(message):
    '''
    Convert a message to the bits to be embedded
    '''
    watermarkBits = extract_bits( message )
    watermarkBits -= watermarkBits == 0
    return watermarkBits


def get_random_params(key, scheme):
    '''
    Seed and return a PRNG
    '''
    prng = random.Random()
    prng.seed(key)
    randMult = 1 + (scheme % 2 > 0)
    randAdd = - (scheme % 2 > 0)
    return prng, randMult, randAdd


def get_watermark_message(bits):
    '''
    Given raw bit values, return the
    nearest legal bits
    '''
    return (array(bits) >= 0)
```

```
def get_watermark_strength(scheme):
    '''
    Return the apropos strength
    '''
    a = [0.01875, 0.021094]

    return a[scheme]


"""
find_diagonal_a.py
Finds the suitable watermark strengths yielding a
PAQM of 0.01 for the diag* watermarking schemes.
"""

execfile('diagonal.py') #execfile('diag_bit_spread')
import os
import audio
import paqm
path = '../audio/test16'
target_paqm = 0.01
accuracy = 0.001
files = os.listdir(path)
key = 1
message = 'Hello'
try:
    len(audio_data)
except NameError:
    audio_data = []
    for f in files:
        fullpath = path + '/' + f
        print 'Loading', fullpath
        audio_data += [ audio.load_wave_array(fullpath) ]

all_good_a = []
out_file = open('all_good_a.txt', "w")
for scheme in range(16):
    out_file.write("scheme: ")
    out_file.write(str(scheme))
    out_file.write('\n')
    print 'Running scheme', scheme
    good_a = []
    for f in range(len(files)):
        fullpath = path + '/' + files[f]
        print 'Processing:', fullpath
        a = .04

        # run once to get a reference point
        #print '  a =', a
        marked_audio, x,y = watermark_audio_data(audio_data[f], message, key, scheme, a)
        new_paqm = paqm.full_non_log_run(audio_data[f], marked_audio)
        #print '  paqm =', new_paqm

        if( new_paqm < target_paqm ):
            # save the low values
            low_a = a
            low_paqm = new_paqm

            while( new_paqm < target_paqm ):
                # try to find a paqm higher than the target
                a *= 2
                #print '  a =', a
                marked_audio, x,y = watermark_audio_data(audio_data[f], message, key, scheme, a)
                new_paqm = paqm.full_non_log_run(audio_data[f], marked_audio)
                #print '  paqm =', new_paqm
                if( new_paqm < target_paqm ):
                    low_a = a
                    low_paqm = new_paqm
                else:
                    high_a = a
                    high_paqm = new_paqm
        else:
            # save the high values
            high_a = a
            high_paqm = new_paqm

            while( new_paqm >= target_paqm ):
```

```
                # try to find a paqm lower than the target
                a /= 2
                #print '  a =', a
                marked_audio, x,y = watermark_audio_data(audio_data[f], message, key, scheme, a)
                new_paqm = paqm.full_non_log_run(audio_data[f], marked_audio)
                #print '  paqm =', new_paqm
                if( new_paqm < target_paqm ):
                    low_a = a
                    low_paqm = new_paqm
                else:
                    high_a = a
                    high_paqm = new_paqm

        # now we have upper and lower bounds
        # special cases: either the high or low bounds are "close enough" to -1.8 (-1.81,-1.79)
        if high_paqm > (target_paqm-accuracy) and high_paqm < (target_paqm+accuracy):
            good_a += [high_a]
            out_file.write(str(high_a))
            out_file.write(", ")

        elif low_paqm > (target_paqm-accuracy) and low_paqm < (target_paqm+accuracy):
            good_a += [low_a]
            out_file.write(str(low_a))
            out_file.write(", ")

        else:
            # search for an acceptable a...
            runs_left = 10
            while runs_left > 0:
                a = (high_a + low_a)/2
                #print '  a =', a
                marked_audio, x,y = watermark_audio_data(audio_data[f], message, key, scheme, a)
                new_paqm = paqm.full_non_log_run(audio_data[f], marked_audio)
                #print '  paqm =', new_paqm
                if new_paqm > (target_paqm-accuracy) and new_paqm < (target_paqm+accuracy):
                    runs_left = 0
                else:
                    runs_left -= 1
                    if new_paqm < target_paqm:
                        low_a = a
                    else:
                        high_a = a
            good_a += [a]
            out_file.write(str(a))
            out_file.write(", ")

    out_file.write('\n')
    print 'Mean', mean(good_a)
    print 'Median', median(good_a)
    all_good_a += [good_a]

out_file.close()


"""
find_diagonal_biterror.py
This script determines the number of encode-decode
errors for each diag* watermarking scheme and writes
them out to a file.
"""

execfile('diagonal.py') # execfile('diag_bit_spread.py')
import os
import audio
path = '../audio/SQAM16' # '../audio/song16'
files = os.listdir(path)
key = 1
message = 'Hello'
try:
    len(audio_data)
except NameError:
    audio_data = []
    for f in files:
        fullpath = path + '/' + f
        print 'Loading', fullpath
        audio_data += [ audio.load_wave_array(fullpath) ]

out_file = open('bit_error.txt', "w")
```

```
for scheme in range(16):
    out_file.write("scheme: ")
    out_file.write(str(scheme))
    out_file.write('\n')
    print 'Running scheme', scheme
    for f in range(len(files)):
        fullpath = path + '/' + files[f]
        print 'Processing:', fullpath
        bits = get_watermark_bits(message, scheme)
        if( scheme % 2 ):
            bits = bits >= 0
        marked_audio, x,y = watermark_audio_data(audio_data[f], message, key, scheme)
        new_bits = detect_watermark(marked_audio,x,y,key,scheme)

        error = 0
        for i in range(len(new_bits)):
            if( new_bits[i] != bits[i % len(bits)] ):
                error += 1

        uni_bits = []
        for i in range(len(bits)/11):
            uni_bits += decode_hamming_11_7(bits[i*11:(i+1)*11])

        uni_new_bits = []
        for i in range(len(new_bits)/11):
            uni_new_bits += decode_hamming_11_7(new_bits[i*11:(i+1)*11])

        uni_error = 0
        for i in range(len(uni_new_bits)):
            if( uni_new_bits[i] != uni_bits[i % len(uni_bits)] ):
                uni_error += 1

        print error, len(new_bits), uni_error, len(uni_new_bits)
        out_file.write(str(error))
        out_file.write(", ")
        out_file.write(str(len(new_bits)))
        out_file.write(", ")
        out_file.write(str(uni_error))
        out_file.write(", ")
        out_file.write(str(len(uni_new_bits)))
        out_file.write('\n')

out_file.close()


"""
find_orth_thresh.py
This script determines the viability of the two-list
median comparison detection method for orthogonal
bit spreading.
"""

execfile('orth_bit_spread.py')
import os
import audio
import bit_utils
path = '../audio/test16'
files = os.listdir(path)
key = 1
bit = 0
try:
    len(audio_data)
except NameError:
    audio_data = []
    for f in files:
        fullpath = path + '/' + f
        print 'Loading', fullpath
        audio_data += [ audio.load_wave_array(fullpath) ]

out_file = open('orth_thresh_0.txt', "w")
out_file.write("Zeros:")
out_file.write('\n')
for scheme in range(1):
    out_file.write("scheme: ")
    out_file.write(str(scheme))
    out_file.write('\n')
    print 'Running scheme', scheme
    for f in range(len(files)):
```

```
        fullpath = path + '/' + files[f]
        print 'Processing:', fullpath
        marked_audio = watermark_audio_data(audio_data[f], bit, key, scheme, .1)
        pure_bits, flip_bits = detect_watermark(marked_audio,audio_data[f],key,scheme,.1)

        errors = 0
        for i in range(len(pure_bits)):
            if pure_bits[i] > flip_bits[i]:
                errors += 1

        out_file.write(str(errors))
        out_file.write(",")
        out_file.write(str(len(pure_bits)))
        out_file.write('\n')

out_file.close()

bit = 1

out_file = open('orth_thresh_1.txt', "w")
out_file.write("Ones:")
out_file.write('\n')
for scheme in range(1):
    out_file.write("scheme: ")
    out_file.write(str(scheme))
    out_file.write('\n')
    print 'Running scheme', scheme
    for f in range(len(files)):
        fullpath = path + '/' + files[f]
        print 'Processing:', fullpath
        marked_audio = watermark_audio_data(audio_data[f], bit, key, scheme, .1)
        pure_bits, flip_bits = detect_watermark(marked_audio,audio_data[f],key,scheme,.1)

        errors = 0
        for i in range(len(pure_bits)):
            if pure_bits[i] <= flip_bits[i]:
                errors += 1

        out_file.write(str(errors))
        out_file.write(",")
        out_file.write(str(len(pure_bits)))
        out_file.write('\n')

out_file.close()


"""
find_orthogonal_a.py
This script finds the appropriate watermark strengths
which yield a PAQM of 0.01 for the orthogonal scheme.
"""

import os
import audio
import orthogonal
import paqm
path = '../audio/test16'
target_paqm = 0.01
accuracy = 0.001
files = os.listdir(path)
key = 1
message = 'Hello'
try:
    len(audio_data)
except NameError:
    audio_data = []
    for f in files:
        fullpath = path + '/' + f
        print 'Loading', fullpath
        audio_data += [ audio.load_wave_array(fullpath) ]

all_good_a = []
out_file = open('orth_good_a.txt', "w")
for scheme in range(12,24):
    out_file.write("scheme: ")
    out_file.write(str(scheme))
    out_file.write('\n')
    print 'Running scheme', scheme
```

```
good_a = []
for f in range(len(files)):
    fullpath = path + '/' + files[f]
    print 'Processing:', fullpath
    a = .04

    # run once to get a reference point
    print '  a =', a
    marked_audio = orthogonal.watermark_audio_data(audio_data[f], message, key, scheme, a)
    new_paqm = paqm.full_non_log_run(audio_data[f], marked_audio)
    print '  paqm =', new_paqm

    if( new_paqm < target_paqm ):
        # save the low values
        low_a = a
        low_paqm = new_paqm

        while( new_paqm < target_paqm ):
            # try to find a paqm higher than the target
            a *= 2
            print '  a =', a
            marked_audio = orthogonal.watermark_audio_data(audio_data[f], message, key, scheme, a)
            new_paqm = paqm.full_non_log_run(audio_data[f], marked_audio)
            print '  paqm =', new_paqm
            if( new_paqm < target_paqm ):
                low_a = a
                low_paqm = new_paqm
            else:
                high_a = a
                high_paqm = new_paqm
    else:
        # save the high values
        high_a = a
        high_paqm = new_paqm

        while( new_paqm >= target_paqm ):
            # try to find a paqm lower than the target
            a /= 2
            print '  a =', a
            marked_audio = orthogonal.watermark_audio_data(audio_data[f], message, key, scheme, a)
            new_paqm = paqm.full_non_log_run(audio_data[f], marked_audio)
            print '  paqm =', new_paqm
            if( new_paqm < target_paqm ):
                low_a = a
                low_paqm = new_paqm
            else:
                high_a = a
                high_paqm = new_paqm

    # now we have upper and lower bounds
    # special cases: either the high or low bounds are "close enough" to -1.8 (-1.81,-1.79)
    if high_paqm > (target_paqm-accuracy) and high_paqm < (target_paqm+accuracy):
        good_a += [high_a]
        out_file.write(str(high_a))
        out_file.write(", ")

    elif low_paqm > (target_paqm-accuracy) and low_paqm < (target_paqm+accuracy):
        good_a += [low_a]
        out_file.write(str(low_a))
        out_file.write(", ")

    else:
        # search for an acceptable a...
        runs_left = 10
        while runs_left > 0:
            a = (high_a + low_a)/2
            print '  a =', a
            marked_audio = orthogonal.watermark_audio_data(audio_data[f], message, key, scheme, a)
            new_paqm = paqm.full_non_log_run(audio_data[f], marked_audio)
            print '  paqm =', new_paqm
            if new_paqm > (target_paqm-accuracy) and new_paqm < (target_paqm+accuracy):
                runs_left = 0
            else:
                runs_left -= 1
                if new_paqm < target_paqm:
                    low_a = a
                else:
                    high_a = a
```

```
            good_a += [a]
            out_file.write(str(a))
            out_file.write(", ")

    out_file.write('\n')
    print 'Mean', mean(good_a)
    print 'Median', median(good_a)
    all_good_a += [good_a]

out_file.close()


"""
find_orthogonal_biterror.py
This script determines the number of encode-decode
errors for the orthogonal watermarking scheme and
writes them out to a file.
"""

execfile('orthogonal.py')
import os
import audio
import bit_utils
path = '../audio/SQAM16' # '../audio/song16'
files = os.listdir(path)
key = 1
message = 'Hello'
try:
    len(audio_data)
except NameError:
    audio_data = []
    for f in files:
        fullpath = path + '/' + f
        print 'Loading', fullpath
        audio_data += [ audio.load_wave_array(fullpath) ]

out_file = open('orth_bit_error.txt', "w")
for scheme in range(24):
    out_file.write("scheme: ")
    out_file.write(str(scheme))
    out_file.write('\n')
    print 'Running scheme', scheme
    for f in range(len(files)):
        fullpath = path + '/' + files[f]
        print 'Processing:', fullpath
        bits = get_watermark_bits(message)
        marked_audio = watermark_audio_data(audio_data[f], message, key, scheme)
        new_bits = detect_watermark(marked_audio,audio_data[f],key,scheme)

        error = 0
        for i in range(len(new_bits)):
            if( new_bits[i] != bits[i % len(bits)] ):
                error += 1

        uni_bits = []
        for i in range(len(bits)/11):
            uni_bits += bit_utils.decode_hamming_11_7(bits[i*11:(i+1)*11])

        uni_new_bits = []
        for i in range(len(new_bits)/11):
            uni_new_bits += bit_utils.decode_hamming_11_7(new_bits[i*11:(i+1)*11])

        uni_error = 0
        for i in range(len(uni_new_bits)):
            if( uni_new_bits[i] != uni_bits[i % len(uni_bits)] ):
                uni_error += 1

        out_file.write(str(error))
        out_file.write(", ")
        out_file.write(str(len(new_bits)))
        out_file.write(", ")
        out_file.write(str(uni_error))
        out_file.write(", ")
        out_file.write(str(len(uni_new_bits)))
        out_file.write('\n')

out_file.close()
```

```
"""
find_ozer_a.py
This script finds the appropriate watermark strengths
which yield a PAQM of 0.01 for the ozer scheme.
"""

execfile('ozer.py')
import os
import audio
import orthogonal
import paqm
path = '../audio/test16'
target_paqm = 0.01
accuracy = 0.001
files = os.listdir(path)
key = 1
message = 'Hello'
try:
    len(audio_data)
except NameError:
    audio_data = []
    for f in files:
        fullpath = path + '/' + f
        print 'Loading', fullpath
        audio_data += [ audio.load_wave_array(fullpath) ]

out_file = open('ozer_good_a.txt', "w")
print 'Running ozer'
good_a = []
for f in range(len(files)):
    fullpath = path + '/' + files[f]
    print 'Processing:', fullpath
    a = .04

    # run once to get a reference point
    print '  a =', a
    marked_audio, detect = watermark_audio_data(audio_data[f], message, key, a=a)
    new_paqm = paqm.full_non_log_run(audio_data[f], marked_audio)
    print '  paqm =', new_paqm

    if( new_paqm < target_paqm ):
        # save the low values
        low_a = a
        low_paqm = new_paqm
        while( new_paqm < target_paqm ):
            # try to find a paqm higher than the target
            a *= 2
            print '  a =', a
            marked_audio, detect = watermark_audio_data(audio_data[f], message, key, a=a)
            new_paqm = paqm.full_non_log_run(audio_data[f], marked_audio)
            print '  paqm =', new_paqm
            if( new_paqm < target_paqm ):
                low_a = a
                low_paqm = new_paqm
            else:
                high_a = a
                high_paqm = new_paqm
    else:
        # save the high values
        high_a = a
        high_paqm = new_paqm
        while( new_paqm >= target_paqm ):
            # try to find a paqm lower than the target
            a /= 2
            print '  a =', a
            marked_audio, detect = watermark_audio_data(audio_data[f], message, key, a=a)
            new_paqm = paqm.full_non_log_run(audio_data[f], marked_audio)
            print '  paqm =', new_paqm
            if( new_paqm < target_paqm ):
                low_a = a
                low_paqm = new_paqm
            else:
                high_a = a
                high_paqm = new_paqm

    # now we have upper and lower bounds
    # special cases: either the high or low bounds are "close enough" to -1.8 (-1.81,-1.79)
```

```
    if high_paqm > (target_paqm-accuracy) and high_paqm < (target_paqm+accuracy):
        good_a += [high_a]
        out_file.write(str(high_a))
        out_file.write(", ")
    elif low_paqm > (target_paqm-accuracy) and low_paqm < (target_paqm+accuracy):
        good_a += [low_a]
        out_file.write(str(low_a))
        out_file.write(", ")
    else:
        # search for an acceptable a...
        runs_left = 10
        while runs_left > 0:
            a = (high_a + low_a)/2
            print '  a =', a
            marked_audio, detect = watermark_audio_data(audio_data[f], message, key, a=a)
            new_paqm = paqm.full_non_log_run(audio_data[f], marked_audio)
            print '  paqm =', new_paqm
            if new_paqm > (target_paqm-accuracy) and new_paqm < (target_paqm+accuracy):
                runs_left = 0
            else:
                runs_left -= 1
                if new_paqm < target_paqm:
                    low_a = a
                else:
                    high_a = a
        good_a += [a]
        out_file.write(str(a))
        out_file.write(", ")

out_file.write('\n')
print 'Mean', mean(good_a)
print 'Median', median(good_a)
out_file.write("Mean: ")
out_file.write(str(mean(good_a)))
out_file.write('\n')
out_file.write("Median: ")
out_file.write(str(median(good_a)))
out_file.write('\n')
out_file.close()


"""
find_ozer_biterror.py
Finds the encode-decode error rates
for the ozer system.
"""

execfile('ozer.py')
import os
import audio
import bit_utils
path = '../audio/song16' # '../audio/SQAM16'
files = os.listdir(path)
key = 1
message = 'Hello'
try:
    len(audio_data)
except NameError:
    audio_data = []
    for f in files:
        fullpath = path + '/' + f
        print 'Loading', fullpath
        audio_data += [ audio.load_wave_array(fullpath) ]

out_file = open('ozer_bit_error.txt', "w")
for f in range(len(files)):
    fullpath = path + '/' + files[f]
    print 'Processing:', fullpath
    bits = bit_utils.extract_bits(message)
    marked_audio, detect = watermark_audio_data(audio_data[f], message, key)
    new_bits = detect_watermark(marked_audio, detect, key)

    error = 0
    for i in range(len(new_bits)):
        if( new_bits[i] != bits[i % len(bits)] ):
            error += 1

    print error, 'of', len(new_bits), 'wrong'
```

```
    out_file.write(str(error))
    out_file.write(", ")
    out_file.write(str(len(new_bits)))
    out_file.write('\n')

out_file.close()


"""
generate_marked_audio.py
This script marks all test audio
will all the tested watermarking
schemes.
"""

import ozer
import diagonal
import diag_bit_spread
import orthogonal

import os
import audio
import bit_utils
path1 = '../audio/SQAM16'
path2 = '../audio/song16'
files1 = os.listdir(path1)
files2 = os.listdir(path2)
files = files1 + files2
key = 1
message = 'Hello'

out_path = '../marked_audio'

d_scheme = [0,1]
o_scheme = [1]
d_bs_scheme = [1,3]

for f in range(len(files)):
    print 'Running file', files[f]

    if f < len(files1):
        audio_data = audio.load_wave_array( path1 + '/' + files[f] )
    else:
        audio_data = audio.load_wave_array( path2 + '/' + files[f] )


    for scheme in d_scheme:
        print 'Diagonal scheme', scheme
        marked_audio, diag, ind = diagonal.watermark_audio_data(audio_data, message, key, scheme)
        audio.pack_wave_array( marked_audio, out_path + '/di_' + str(scheme) + '_' + files[f] )

    for scheme in o_scheme:
        print 'Orthogonal scheme', scheme
        marked_audio = orthogonal.watermark_audio_data(audio_data, message, key, scheme)
        audio.pack_wave_array( marked_audio, out_path + '/or_' + str(scheme) + '_' + files[f] )

    for scheme in d_bs_scheme:
        print 'Spread scheme', scheme
        marked_audio, diag, ind = diag_bit_spread.watermark_audio_data(audio_data, message, key, scheme)
        audio.pack_wave_array( marked_audio, out_path + '/sp_' + str(scheme) + '_' + files[f] )

    print 'Ozer'
    marked_audio, detect = ozer.watermark_audio_data(audio_data, message, key)
    audio.pack_wave_array( marked_audio, out_path + '/oz_' + files[f] )


"""
orth_bit_spread.py
Functions to embed and attempt to decode
watermarks for orthogonal bit spreading.
"""

from pylab import *
from singular_value import *
from short_time import stft, inverse_stft
from bit_utils import *
from RandomMatrix import RandomMatrix
```

```
def watermark_audio_data(audio_data, bit, key, scheme, a):
    '''
    Embeds an audio carrier with a watermark signal in a
    variety of ways, dependent on the scheme value.

    Embedding Targets:
    Schemes 0-7: U matrix
    Schemes 8-15: first M rows of VT matrix
    Schemes 16-23: both sets

    Frequency of marking:
    Scheme 0-1 % 8:  25%
    Scheme 2-3 % 8:  50%
    Scheme 4-5 % 8:  75%
    Scheme 6-7 % 8: 100%

    Random values:
    Schemes 0 % 2: [0,1)
    Schemes 1 % 2: [-1,1)
    '''

    randMat = get_random_params(key)
    #watermarkBits = get_watermark_bits(message)
    threshold = .25 * ( ( (scheme/2) % 4) + 1 )

    # run STFT on the audio data to yield
    # matrix of time x freq
    Pxx = stft(audio_data)

    # Magic number: taking 8 time slices
    # from the STFT results yields a bitrate
    # of roughly 16 watermark bits per second of
    # audio data
    bps = 16
    frame_width = 128 / bps

    #a = get_watermark_strength(scheme)

    # figure out which indices our frames start at
    ind = arange(0, len(Pxx)-frame_width+1, frame_width)

    # for each frame:
    for f in range(len(ind)):
        frame = Pxx[ind[f]:ind[f]+frame_width]
        #bit = watermarkBits[f % len(watermarkBits)]

        # run SVD on the frame
        u, d, vt = full_svd(frame)

        # determine which orthogoal values need to be marked
        if scheme < 8:
            mark = randMat.get_next_matrix( shape(u)[0], shape(u)[1] ) < threshold
            randVals = randMat.get_next_matrix( shape(u)[0], shape(u)[1], scheme % 2 )
            bits = randMat.get_next_flipped_matrix( bit, shape(u)[0], shape(u)[1] )
            u += u * a * bits * mark * randVals

        # calculate new frame
        Pxx[ind[f]:ind[f]+frame_width] = dot( dot(u, d), vt )

    # Run stft inverse
    new_audio_data = inverse_stft( Pxx )

    return new_audio_data


def detect_watermark(marked_audio, orig_audio, key, scheme, a):
    '''
    Given an audio signal, return the bits embedded.
    '''
    randMat = get_random_params(key)
    threshold = .25 * ( ( (scheme/2) % 4) + 1 )

    Pxx = stft(marked_audio)
    Oxx = stft(orig_audio)

    bps = 16
    frame_width = 128 / bps
    #a = get_watermark_strength(scheme)
```

```
    ind = arange(0, len(Pxx)-frame_width+1, frame_width)

    pure_bits = []
    flip_bits = []

    # for each frame:
    for f in range(len(ind)):
        frame = Pxx[ind[f]:ind[f]+frame_width]
        orig_frame = Oxx[ind[f]:ind[f]+frame_width]

        # run SVD on the frame
        u, s, vt = full_svd(frame)
        u_o, s_o, vt_o = full_svd(orig_frame)

        # determine which orthogoal values were marked
        if scheme < 8:
            mark = randMat.get_next_matrix( shape(u)[0], shape(u)[1] ) < threshold
            randVals = abs(randMat.get_next_matrix( shape(u)[0], shape(u)[1], scheme % 2 ))
            flipped = randMat.get_next_flipped_matrix( 0, shape(u)[0], shape(u)[0] )

            pure_check = []
            flip_check = []
            for i in range(len(u_o)):
                for j in range(len(u_o[i])):
                    if mark[i][j]:
                        if u_o[i][j] and a*randVals[i][j]:
                            rawVal = abs( (u[i][j] - u_o[i][j]) / u_o[i][j] ) / (a*randVals[i][j])
                            if flipped[i][j]:
                                flip_check += [rawVal]
                            else:
                                pure_check += [rawVal]

            if len(flip_check):
                flip_bits += [median(flip_check)]
            else:
                flip_bits += [0]

            if len(pure_check):
                pure_bits += [median(pure_check)]
            else:
                pure_bits += [0]



    return array(pure_bits), array(flip_bits)
    #return get_watermark_message(bits)


def get_watermark_bits(message):
    '''
    Given a message, return the array of bits.
    '''
    return extract_hamming_unicode_bits(message)


def get_random_params(key):
    '''
    Given a key, initialize a PRNG
    '''
    return RandomMatrix(key)


def get_watermark_message(bits):
    '''
    Given hamming bits, return the decoded message.
    '''
    return construct_hamming_unicode_message( array(bits) >= .5 )



def get_watermark_strength(scheme):
    '''
    Unimplemented.  The system is nonfunctional and as such
    correct strengths were not calculated.
    '''
    a = []

    return a[scheme]
```

```
def val_to_bit(val):
    '''
    Given a raw value, return the nearest bit.
    '''
    return int(val >= .5)


def flip(bit, randMat):
    '''
    Given a bit, randomly flip it.
    '''
    if randMat.get_next_rand_val() < .5:    # flip
        return 1 - bit
    return bit


"""
orthogonal.py
Functions to embed and decode watermarks
using the orthogonal family.
"""

from pylab import *
from singular_value import *
from short_time import stft, inverse_stft
from bit_utils import *
from RandomMatrix import RandomMatrix

def watermark_audio_data(audio_data, message, key, scheme):
    '''
    Embeds an audio carrier with a watermark signal in a
    variety of ways, dependent on the scheme value.

    Embedding Targets:
    Schemes 0-7: U matrix
    Schemes 8-15: first M rows of VT matrix
    Schemes 16-23: both sets

    Frequency of marking:
    Scheme 0-1 % 8:  25%
    Scheme 2-3 % 8:  50%
    Scheme 4-5 % 8:  75%
    Scheme 6-7 % 8: 100%

    Random values:
    Schemes 0 % 2: [0,1)
    Schemes 1 % 2: [-1,1)
    '''

    randMat = get_random_params(key)
    watermarkBits = get_watermark_bits(message)
    threshold = .25 * ( ( (scheme/2) % 4) + 1 )

    # run STFT on the audio data to yield
    # matrix of time x freq
    Pxx = stft(audio_data)

    # Magic number: taking 8 time slices
    # from the STFT results yields a bitrate
    # of roughly 16 watermark bits per second of
    # audio data
    bps = 16
    frame_width = 128 / bps

    a = get_watermark_strength(scheme)

    # figure out which indices our frames start at
    ind = arange(0, len(Pxx)-frame_width+1, frame_width)

    # for each frame:
    for f in range(len(ind)):
        frame = Pxx[ind[f]:ind[f]+frame_width]
        bit = watermarkBits[f % len(watermarkBits)]

        # run SVD on the frame
        u, d, vt = full_svd(frame)
```

```
        # determine which orthogoal values need to be marked
        if scheme < 8:
            mark = randMat.get_next_matrix( shape(u)[0], shape(u)[1] ) < threshold
            randVals = randMat.get_next_matrix( shape(u)[0], shape(u)[1], scheme % 2 )
            u += u * a * bit * mark * randVals
        elif scheme < 16:
            mark = randMat.get_next_clipped_matrix( shape(vt)[0], shape(vt)[1], shape(u)[0], shape(vt)[1] ) < threshold
            randVals = randMat.get_next_clipped_matrix(shape(vt)[0],shape(vt)[1],shape(u)[0],shape(vt)[1],scheme % 2)
            vt += vt * a * bit * mark * randVals
        else:
            mark = randMat.get_next_matrix( shape(u)[0], shape(u)[1] ) < threshold
            randVals = randMat.get_next_matrix( shape(u)[0], shape(u)[1], scheme % 2 )
            u += u * a * bit * mark * randVals

            mark = randMat.get_next_clipped_matrix( shape(vt)[0], shape(vt)[1], shape(u)[0], shape(vt)[1] ) < threshold
            randVals = randMat.get_next_clipped_matrix(shape(vt)[0],shape(vt)[1],shape(u)[0],shape(vt)[1],scheme % 2)
            vt += vt * a * bit * mark * randVals

        # calculate new frame
        Pxx[ind[f]:ind[f]+frame_width] = dot( dot(u, d), vt )

    # Run stft inverse
    new_audio_data = inverse_stft( Pxx )

    return new_audio_data


def detect_watermark(marked_audio, orig_audio, key, scheme):
    '''
    Given audio, return the embedded watermark.
    '''
    randMat = get_random_params(key)
    threshold = .25 * ( ( (scheme/2) % 4) + 1 )

    Pxx = stft(marked_audio)
    Oxx = stft(orig_audio)

    bps = 16
    frame_width = 128 / bps
    a = get_watermark_strength(scheme)
    ind = arange(0, len(Pxx)-frame_width+1, frame_width)

    bits = []

    # for each frame:
    for f in range(len(ind)):
        frame = Pxx[ind[f]:ind[f]+frame_width]
        orig_frame = Oxx[ind[f]:ind[f]+frame_width]

        # run SVD on the frame
        u, s, vt = full_svd(frame)
        u_o, s_o, vt_o = full_svd(orig_frame)

        # determine which orthogoal values were marked
        if scheme < 8:
            mark = randMat.get_next_matrix( shape(u)[0], shape(u)[1] ) < threshold
            randVals = abs(randMat.get_next_matrix( shape(u)[0], shape(u)[1], scheme % 2 ))

            check = []
            for i in range(len(u_o)):
                for j in range(len(u_o[i])):
                    if mark[i][j]:
                        if u_o[i][j] and a*randVals[i][j]:
                            check += [abs( (u[i][j] - u_o[i][j]) / u_o[i][j] ) / (a*randVals[i][j])]
            if len(check):
                bits += [median(check)]
            else:
                bits += [0]

        elif scheme < 16:
            mark = randMat.get_next_clipped_matrix( shape(vt)[0], shape(vt)[1], shape(u)[0], shape(vt)[1] ) < threshold
            randVals = abs(randMat.get_next_clipped_matrix(shape(vt)[0],shape(vt)[1],shape(u)[0],shape(vt)[1],scheme % 2))

            check = []
            for i in range(len(u_o)):
                for j in range(len(vt_o[i])):
                    if mark[i][j]:
```

```
                        if vt_o[i][j] and a*randVals[i][j]:
                            check += [abs( (vt[i][j] - vt_o[i][j]) / vt_o[i][j] ) / (a*randVals[i][j])]
            if len(check):
                bits += [median(check)]
            else:
                bits += [0]

        else:
            mark1 = randMat.get_next_matrix( shape(u)[0], shape(u)[1] ) < threshold
            randVals1 = abs(randMat.get_next_matrix( shape(u)[0], shape(u)[1], scheme % 2 ))

            mark2 = randMat.get_next_clipped_matrix( shape(vt)[0], shape(vt)[1], shape(u)[0], shape(vt)[1] ) < threshold
            randVals2 = abs(randMat.get_next_clipped_matrix(shape(vt)[0],shape(vt)[1],shape(u)[0],shape(vt)[1],scheme % 2))

            check = []
            for i in range(len(u_o)):
                for j in range(len(u_o[i])):
                    if mark1[i][j]:
                        if u_o[i][j] and a*randVals1[i][j]:
                            check += [abs( (u[i][j] - u_o[i][j]) / u_o[i][j] ) / (a*randVals1[i][j])]
            for i in range(len(u_o)):
                for j in range(len(vt_o[i])):
                    if mark2[i][j]:
                        if vt_o[i][j] and a*randVals2[i][j]:
                            check += [abs( (vt[i][j] - vt_o[i][j]) / vt_o[i][j] ) / (a*randVals2[i][j])]
            if len(check):
                bits += [median(check)]
            else:
                bits += [0]


    return get_watermark_message(bits)


def get_watermark_bits(message):
    '''
    Given a message, return the bits to encode.
    '''
    return extract_bits( message )


def get_random_params(key):
    '''
    Seed and return a PRNG
    '''
    return RandomMatrix(key)


def get_watermark_message(bits):
    '''
    Given raw bit values, return the
    legal bits.
    '''
    return array(bits) >= .5


def get_watermark_strength(scheme):
    '''
    Return the appropriate strength of the watermark
    based on scheme.
    '''
    a = [0.05625, 0.075, 0.0375, 0.05625, 0.028125, 0.04453125, 0.02109375, 0.0375, 0.065625,
        0.084375, 0.0375, 0.05625, 0.03, 0.05, 0.02, 0.04, 0.03625, 0.0525, 0.02, 0.04, 0.015,
        0.03, 0.01125, 0.03]

    return a[scheme]


"""
ozer.py
Implementation of the watermarking system described by Oezer.
"""

from pylab import *
from singular_value import full_svd
from short_time import stft, inverse_stft
from RandomMatrix import RandomMatrix
from LinearAlgebra import inverse
```

```
from bit_utils import extract_bits, construct_message

def watermark_audio_data(audio_data, message, key, bps=16):
    '''
    Embed the given message into the given audio.
    '''

    # initialize matrix generator with key
    randMat = RandomMatrix(key)

    # get the bits from the message (unicode->binary)
    watermarkBits = extract_bits(message)
    watermarkBits -= (watermarkBits == 0)

    # run STFT on the audio data to yield
    # matrix of time x freq
    Pxx = stft(audio_data)
    Dxx = zeros( shape(Pxx), 'd' )

    # Magic number: taking 8 time slices
    # from the STFT results yields a bitrate
    # of roughly 16 watermark bits per second of
    # audio data
    frame_width = 128 / bps

    # a = Magic number: strength of the watermark
    a = 0.015

    # figure out which indices our frames start at
    ind = arange(0, len(Pxx)-frame_width+1, frame_width)

    # for each frame:
    for f in range(len(ind)):
        frame = Pxx[ind[f]:ind[f]+frame_width]

        # run SVD on the frame
        u, d, vt = full_svd(frame)

        # Get the watermark signal
        w = randMat.get_next_matrix( shape(d)[0], shape(d)[1], True )

        # create wd
        wd = []
        for i in range( len(d) ):
            delta_i = d[i,i]*ones(shape(d)[1],'float')
            bit = watermarkBits[f % len(watermarkBits)]
            wd.append(array(d[i] + a * bit * delta_i * w[i]))

        wd = array(wd)
        # break down wd
        uw, dw, vwt = full_svd( wd )

        # uw, d, and vwt need to be saved - for each frame!
        Dxx[ind[f]:ind[f]+frame_width] = dot( dot(uw, d), vwt )

        # calculate new frame
        Pxx[ind[f]:ind[f]+frame_width] = dot( dot(u, dw), vt )

    # Run stft inverse
    new_audio_data = inverse_stft( Pxx )

    return new_audio_data, Dxx


def detect_watermark(marked_data, Dxx, key, bps=16):
    '''
    Given marked audio, detect and return
    the watermark.
    '''

    # initialize matrix generator with key
    randMat = RandomMatrix(key)

    # run STFT on the audio data to yield
    # matrix of time x freq
    Pxx = stft(marked_data)
    #Dxx = stft(detection_data)
```

```
    frame_width = 128 / bps

    a = 0.015

    # figure out which indices our frames start at
    ind = arange(0, len(Pxx)-frame_width+1, frame_width)

    bits = []

    # for each frame:
    for f in range(len(ind)):
        frame = Pxx[ind[f]:ind[f]+frame_width]
        detection_frame = Dxx[ind[f]:ind[f]+frame_width]

        u, dw, vt = full_svd(frame)
        uw, d, vw = full_svd(detection_frame)

        wd = dot( dot( uw, dw ), vw )

        # build d "inverse"
        d_inv = zeros( [shape(d)[0],shape(d)[0]], 'f' )
        for i in range(len(d)):
            d_inv[i,i] = 1. / d[i,i]

        w = dot( d_inv, wd - d ) / a

        real_w = randMat.get_next_matrix( shape(d)[0], shape(d)[1] )
        detect_marker = sum(sum( w * real_w ))
        bits.append(detect_marker > 0)

    return array(bits)


"""
paqm.py
Implementation of the Perceptual Audio Quality Measure
proposed by Beerends and Stemerdink.
"""

from pylab import *

from bark import freq_to_bark, bark_widths, bark_centers

# Constants
alpha_freq = 0.8
alpha_time = 0.6
delta_z = 0.2


def full_run(audio_data1, audio_data2):
    '''
    Return the log noise disturbance of two signals.
    '''
    return log10(full_non_log_run(audio_data1, audio_data2))


def full_non_log_run(audio_data1, audio_data2):
    '''
    Return the noise disturbance of two signals.
    '''
    minLen = min(len(audio_data1), len(audio_data2))
    Lx = audio_to_loudness(audio_data1[:minLen])
    Ly = audio_to_loudness(audio_data2[:minLen])
    return calculate_non_log_paqm(Lx, Ly)


def audio_to_loudness(audio_data):
    '''
    Convert from audio to loudness values.
    '''
    Bxx, times = pitch_power(audio_data)
    Bsmear = time_smear(Bxx)
    E = freq_spread(Bsmear)
    L = compressed_loudness(E)
    return L


def pitch_power(audio_data):
```

```
    '''
    Bxx is bark x time
    Steps 1 and 2: Get the power spectrum density of the audio file and convert from a
    frequency x time measure to a pitch-centric measure on the bark scale.

    See Appendix A in the Beerends paper.

    Returns pitch power in bark x time
    '''
    # steps 1-2
    Fxx, freqs, times = matplotlib.mlab.specgram(audio_data, NFFT=2048, Fs=44100, noverlap=1024)
    Bxx = zeros( (24,len(times)), 'd' )
    hits = zeros(24, 'i')
    for i in range( len(Fxx) ):
        b = freq_to_bark(freqs[i])
        if b is not None:
            Bxx[b-1] += Fxx[i]
            hits[b-1] += 1

    #adjust based on the number of hits and the width of the bark interval
    # I do not uze delta_z, but it does not affect the scaling adjustment
    for b in range( len(Bxx) ):
        Bxx[b] = Bxx[b] * hits[b] / bark_widths[b]

    return Bxx, times


def time_smear(Bxx):
    '''
    Bxx is bark x time
    Step 3: Perform time smearing to blend the power across times slices to account for the
    lingering effects of sounds from time slices previous.

    See Equation 3 and Figure 6 in the Beerends paper.

    Returns power in bark x time
    '''

    # time-domain smearing constants poorly approximated from the really bad graph in the paper
    tau = array([120,30,15,13,11,10,9,8,7,6,5,4,3,3,3,3,3,3,3,3,3,3,3,3], 'd')
    tau /= 1000.
    scalar = e**( (-1024./44100) / tau ) * alpha_time

    Bsmear = copy.copy(transpose(Bxx))

    for i in range( len(Bsmear)-1 ):
        Bsmear[i+1] += Bsmear[i] * scalar

    return transpose(Bsmear)


def freq_spread(P):
    '''
    P is bark x time
    Step 4: Perform frequency spreading to account for the effects that sounds in frequencies
    have on the excitation levels of other frequencies.

    See Appendix D in the Beerends paper.

    Returns excitation levels in bark x time
    '''
    alpha = alpha_freq
    S1 = 31.

    T1 = 10**( -alpha * S1 * delta_z / 20 )

    Qv = array(zeros(shape(P)), 'd')
    for v in range(len(P)):
        Qv[v] = P[v]**(alpha/2)

    # temporary excitation levels
    Eq = array(zeros(shape(P)), 'd')
    for v in range(len(P)):
        factor = T1
        Q = Qv[v]

        # handle the barks below the current
        for mu in range( v, -1, -1 ):
```

```
            Eq[mu] += Q
            Q *= factor

        factor = T2(v) * Qv[v]**(.2*delta_z)  ### bug?  mult again by second term? ###
        Q = Qv[v] * factor

        # handle the barks above the current
        for mu in range( v+1, len(P) ):
            Eq[mu] += Q
            Q *= factor

    # calculate the final excitation levels
    E = array(zeros(shape(P)), 'd')
    for mu in range(len(P)):
        E[mu] = Eq[mu]**(2./alpha)

    return E


def T2(v):
    '''
    Equations 51 and 41 from the Beerend's paper.  Provides an attenuation
    factor of the frequency spreading based on the frequency affected.
    '''
    return 10 ** (-alpha_freq * (22 + (230. / bark_centers[v])) * delta_z / 200)


def compressed_loudness(E):
    '''
    Given excitation levels, return the compressed loudness.
    '''
    # hacked from a picture
    E0 = [55,30,18,12,8,5,2,1,.01,-.01,-1,-3,-4,-5,-6,-7,-7,-7,-5,-1,5,8,10,15]

    # actual constants
    s = 0.5
    gamma = 0.04

    L = array(zeros(shape(E)), 'd')
    for i in range(len(E)):
        # calculate the middle part first, as if it's negative, Python freaks out
        middle = 1-s+(s*E[i]/E0[i])
        front = E0[i]/s
        L[i] = (abs(front)**gamma) *(front/abs(front)) * (  (abs(middle)**gamma)*(middle/abs(middle)) - 1)

    return L


def calculate_non_log_paqm(Lx, Ly):
    '''
    Calculate the difference in loudnesses.
    '''
    Lys = scaled_y(Lx, Ly)
    Ln = abs( Lys - Lx )
    return mean(sum(Ln))


def scaled_y(Lx, Ly):
    '''
    Scale the loudness of Y to the
    loudness of X
    '''
    # calculate scale factor 1
    x1 = sum(Lx[:2])
    y1 = sum(Ly[:2])

    # ensure no divide by zero errors!
    # this is measuring difference, so if y is zero at any point, it
    # will all be difference, so we can't (and don't want) to scale that away!
    y1 += (y1==0)

    a1 = x1 / y1

    x2 = sum(Lx[2:22])
    y2 = sum(Ly[2:22])
    y2 += (y2==0)

    a2 = x2 / y2
```

```
    x3 = sum(Lx[22:])
    y3 = sum(Ly[22:])
    y3 += (y3==0)

    a3 = x3 / y3

    Lys = array(zeros(shape(Ly)), 'd')
    Lys[:2] = a1 * Ly[:2]
    Lys[2:22] = a2 * Ly[2:22]
    Lys[22:] = a3 * Ly[22:]

    return Lys


"""
RandomMatrix.py
This class generates and returns matrices of specified dimensions
filled with noise-like pseudorandom values in the range [-1, 1).
The RandomMatrix is initialized with a key, used to seed the PRNG.
"""


from pylab import *
import random

class RandomMatrix:

    def __init__(self, key):
        '''
        Creates a RandomMatrix generator seeded with the given key.
        '''
        self.rand = random.Random()
        self.key = key
        self.rand.seed(key)


    def reset_seed(self):
        '''
        Resets the seed of the PRNG.
        '''
        self.rand.seed(self.key)


    def get_next_flipped_matrix(self, bit, dim1, dim2):
        '''
        Returns a matrix populated by randomly
        flipped samples of the given bit.
        '''
        r = zeros((dim1, dim2), 'int')
        for i in range(dim1):
            for j in range(dim2):
                if self.rand.random() < .5:
                    r[i,j] = 1 - bit
                else:
                    r[i,j] = bit

        return r


    def get_next_matrix(self, dim1, dim2, negone=False):
        '''
        Given dimensions dim1 and dim2, return
        a matrix filled with noise.
        '''
        r = zeros((dim1, dim2), 'float')
        for i in range(dim1):
            for j in range(dim2):
                if negone:
                    r[i,j] = (2*self.rand.random()) - 1
                else:
                    r[i,j] = self.rand.random()

        return r


    def get_next_clipped_matrix(self, dim1, dim2, filled_dim1, filled_dim2, negone=False):
        '''
```

```
        Given dimensions dim1 and dim2, return
        a matrix filled with noise up to the
        filled dimensions.
        '''
        r = zeros((dim1, dim2), 'float')
        for i in range(filled_dim1):
            for j in range(filled_dim2):
                if negone:
                    r[i,j] = (2*self.rand.random()) - 1
                else:
                    r[i,j] = self.rand.random()

        return r


"""
run_attacks.py
Runs the attacks described in attacks.py
"""

import ozer
import diagonal
import diag_bit_spread
import orthogonal
import attacks
import os
import audio

path = '../marked_audio'
files = os.listdir('../marked_audio')

for f in range(18,len(files)):
    print 'Running file', f, 'of', len(files)
    audio_data = audio.load_wave_array( path + '/' + files[f] )

    print ' # attack with add_noise'
    tmp = attacks.add_noise( audio_data )
    audio.pack_wave_array( tmp, '../attacked/add_noise/' + files[f] )

    print ' # attack with delay_start'
    tmp = attacks.delay_start( audio_data )
    audio.pack_wave_array( tmp, '../attacked/delay_start/' + files[f] )

    print ' # attack with speed_start'
    tmp = attacks.speed_start( audio_data )
    audio.pack_wave_array( tmp, '../attacked/speed_start/' + files[f] )

    print ' # attack with add_fft_noise'
    tmp = attacks.add_fft_noise( audio_data )
    audio.pack_wave_array( tmp, '../attacked/fft_noise/' + files[f] )

    print ' # attack with flipp_sample'
    tmp = attacks.flipp_sample( audio_data )
    audio.pack_wave_array( tmp, '../attacked/flipp_sample/' + files[f] )

    print ' # attack with add_svd_noise'
    tmp = attacks.add_svd_noise( audio_data )
    audio.pack_wave_array( tmp, '../attacked/svd_noise/' + files[f] )

    print ' # attack with low_pass'
    tmp = attacks.low_pass( audio_data )
    audio.pack_wave_array( tmp, '../attacked/low_pass/' + files[f] )

    print ' # attack with high_pass'
    tmp = attacks.high_pass( audio_data )
    audio.pack_wave_array( tmp, '../attacked/high_pass/' + files[f] )


    # remarking attacks
    message = 'DEADBEEF'
    key = 133

    print ' # attack with ozer'
    tmp, dump = ozer.watermark_audio_data( audio_data, message, key )
    audio.pack_wave_array( tmp, '../attacked/ozer/' + files[f] )

    print ' # attack with diag 0'
    tmp, dump, dump2 = diagonal.watermark_audio_data( audio_data, message, key, 0 )
```

```
    audio.pack_wave_array( tmp, '../attacked/diag_0/' + files[f] )

    print ' # attack with diag 1'
    tmp, dump, dump2 = diagonal.watermark_audio_data( audio_data, message, key, 1 )
    audio.pack_wave_array( tmp, '../attacked/diag_1/' + files[f] )

    print ' # attack with orth'
    tmp = orthogonal.watermark_audio_data( audio_data, message, key, 1 )
    audio.pack_wave_array( tmp, '../attacked/orth/' + files[f] )

    print ' # attack with spread 1'
    tmp, dump, dump2 = diag_bit_spread.watermark_audio_data( audio_data, message, key, 1 )
    audio.pack_wave_array( tmp, '../attacked/spread_1/' + files[f] )

    print ' # attack with spread 3'
    tmp, dump, dump2 = diag_bit_spread.watermark_audio_data( audio_data, message, key, 3 )
    audio.pack_wave_array( tmp, '../attacked/spread_3/' + files[f] )


"""
short_time.py
Implementation of the STFT and inverse STFT.
"""

from pylab import *
from FFT import real_fft, inverse_real_fft

def stft(x, NFFT=256, noverlap=128):
    '''
    Compute the Short Time Fourier Transform of the given data using a Hamming window,
    roughly 25 ms windows with 50% overlap

    x - audio data
    NFFT - window width
    noverlap - amount of overlap for each window

    returns: Pxx - stft complex values (time x freq)
    '''

    numFreqs = NFFT//2+1
    windowVals = hamming(NFFT)
    step = NFFT-noverlap

    # Pad audio with zeros so we process the whole clip.
    actual_frames = (len(x) * 1.0 - NFFT) / step

    num_to_add = 0
    if( floor(actual_frames) != actual_frames ):
        num_to_add = (floor(actual_frames) + 1 - actual_frames) * step

    aud = zeros( len(x)+int(round(num_to_add)), 'i')
    aud[0:len(x)] = x[:]

    # build the stft matrix
    ind = arange(0,len(aud)-NFFT+1,step)
    Pxx = zeros((numFreqs,len(ind)), Complex)

    # do the ffts of the slices
    for i in range(len(ind)):
        thisX = aud[ind[i]:ind[i]+NFFT]
        fx = real_fft(thisX)

        # Scale the spectrum by the norm of the window to compensate for
        # windowing loss; see Bendat & Piersol Sec 11.5.2
        Pxx[:,i] = divide(fx[:numFreqs], norm(windowVals)**2)

    # we want time x freq, not freq x time
    return transpose(Pxx)


def inverse_stft(Pxx, NFFT=256, noverlap=128):
    '''
    Compute the Inverse Short Time Fourier Transform of the given data using a Hamming window,
    roughly 25 ms windows with 50% overlap

    Pxx - stft data (time x freq)
    NFFT - window width
    noverlap - amount of overlap for each window
```

```
    returns - x: reconstructed audio data
    '''

    windowVals = hamming(NFFT)
    n = len(Pxx)
    audio_len = NFFT + (NFFT - noverlap)*(n-1)

    step = NFFT - noverlap
    ind = arange(0,audio_len-NFFT+1,step)

    tmp_data = zeros( audio_len, 'd' )

    for i in range(n):
        tmpA = multiply( Pxx[i,:], norm(windowVals)**2 )
        tmpB = inverse_real_fft(tmpA)
        tmp_data[ind[i]:ind[i]+NFFT] = tmpB

    x = zeros( audio_len, Int16 )
    for i in range(audio_len):
        x[i] = int(round(tmp_data[i]))

    return x


"""
singular_value.py
Implementation of the SVD.
"""

from pylab import zeros
from LinearAlgebra import singular_value_decomposition


def list_svd(a):
    '''
    Returns the SVD of a with the singular
    values in a list.
    '''
    return singular_value_decomposition(a, full_matrices=1)


def full_svd(a):
    '''
    Returns the SVD of a with the singular
    values in a matrix.
    '''
    # run the usual SVD with full matrices
    u, s, vt = singular_value_decomposition(a, full_matrices=1)

    # build the singular values array into a matrix, as expected
    d = bloat_diagonal_list(s, len(u),len(vt))
    return u, d, vt


def bloat_diagonal_list(s, lenx, leny):
    '''
    Turn a list of singular values into a matrix.
    '''
    d = zeros( (lenx, leny), 'float' )
    for i in range(len(s)):
        d[i,i]= s[i]
    return d


def deflate_diagonal_matrix(d):
    '''
    Turn a matrix of singular values into a list.
    '''
    return array([d[i,i] for i in range( min(shape(d)) )])
```