A mathematical approach to digital sound production, and musical interpolation

Our project focused on applications of the Discrete Fourier Transform (DFT) to digital audio production and music interpolation (sometimes known as 'digital music production'). For the audio production aspect we implemented a graphic equalizer and used it to boost or cut different frequency ranges of sound tracks. We showed how to re-sample and scale the length of audio tracks. We also show how the sample rate of audio tracks can be raised and lowered so that mixing can occur at a higher sample rate than the final output and how to generate musical notes by synthesis and re-sampling of recorded notes. With the software we wrote in Matlab we had all the necessary tools to create music and mix multiple audio tracks together, each having their own equalization parameters and amplitude parameters.

Sound is characterized by the properties of sound waves, which are frequency, wavelength, period, and amplitude (frequency, wavelength and period are interrelated). A general wave function is $y(t) = a(t)\, sin(f(t)\, x(t))$. The function $f(t)$ is the time varying frequency function which can be thought of as the song function because a change of notes in a song is a change in frequency with respect to time. The function $a(t)$ is the amplitude function which, for a note produced by a musical instrument, is a function of the force applied to the instrument. The use of the *sin* function in $y(t)$ is not required; other periodic functions could be used instead, such as *cos* or the "*saw*" and "*sq*" wave function (digital *saw* and *sq* defined here in the inventory of code).

The occurrence of two or more waves at the same time is known as the superposition of waves. The superposition is found simply by taking the sum of the wave functions involved. More formally, let $w_i(t)$ be a sequence of n wave functions, then $s_n(t) = w_1(t) + w_2(t) + \ldots + w_n(t)$ is the resulting wave when all n waves occur at the same time (and in effect from he same spatial source).

The unit of frequency generally used in physics and engineering is the Hertz (Hz); the base unit of the hertz is one cycle per second, or 2π radians per second, where radians are the native units of a *sin* function. A *sin* wave at 1 Hz will go though one period in one second. For example, the A4 note on a piano resonates at a frequency of 440 Hz, which means that the wave goes through 440 periods in one second. The function $x(t)$ is the conversion function from Hz to radians: $x(t) = 2\pi t$; observe that $x(1 \text{ second}) = 2\pi = 1$ period.

Sounds, occurring for certain durations and at certain frequencies can be organized, along with silence, to form music. Music can be expressed in terms of pitch, rhythm and dynamics. The term pitch refers to the frequency of the sound; as the pitch varies over time, melodies and harmonies can be formed. Patterns of changes between frequencies were found which seem to define pleasant melodies. These patterns are known as scales. The two fundamental types of scales are the major and minor scales. There is a major and minor scale for every type of note (by letter). Also, every major scale is the minor scale of some other type of note. When a song is written in a certain scale, the song is said to be in the key of that scale; if the scale were C major or minor it would be in the key of C. The rhythm of a song determines the tempo and meter: the relative durations of notes and their groupings. So if a sequence of notes is played in some fixed scale (i.e. in a key) with an exact rhythm the notes define a pleasant melody.

The dynamics of music determine how hard the notes are hit or, in the case of digital music, would just be generally how loud the notes are played at any given time; in the MIDI standard (the most common standard for storing musical notes in digital form) the dynamics are stored as velocity values, where velocity in piano terms is just a measure of how hard the musician hit the piano key.

The quality of musical sounds (timbre, articulation, and texture) in physical settings are determined by the actual instrument used; for instance there are string and wind instruments and their sounds are quite distinct. In a digital setting, notes with various different qualities similar to physical instruments can be synthesized using a technique know as frequency modulation synthesis (i.e. FM synthesis). We did not go as far as implementing a full FM synthesizer, we just implemented a function to generate waves at arbitrary frequencies and sampling rates for arbitrary durations.

The duration of a note is determined by the time signature, tempo (measured in beats per minute), and note type. Songs are generally defined as sequences of bars, or measures. The time signature is of the form N/D; the numerator N states how many notes make up a bar and the denominator *D*, which must be a power of 2, states the type of notes. The most common time signature is 4/4. In this case, one measure is four quarter notes long (i.e. type 4 notes). The tempo is the given pace of the music; it specifies a particular note value as “the beat” and indicates that a certain number of these beats must be played per minute. For instance, slide 6 contains a figure showing “eighth note = 120," which means that 120 eighth notes must be played per minute, or two eighth notes per second. In general, given a BPM of *B* and time signature *N*/*D*, a note of type *D* gets the beat, *B*/60 *D* notes get played per second and one *D* note lasts 60/*B* seconds. In our case, *N* has little or no bearing on our results since we are not focused on music composition here.

As mentioned, each note resonates at a different frequency. Slide 7 shows the notes on a piano keyboard and their corresponding frequencies that characterize an ideal piano. Each note is roughly a *sin* function with a frequency in the range of 27.5 Hz to 4186 Hz. The notes of the piano exhibit a distinct amplitude function. The amplitude is initially at some peak and slowly fades out to 0. Physically, pianos have an ability to sustain notes for extended periods of time with a foot pedal but their sustain is always degrading. On the computer we can generate simulated waves of arbitrary length which is beyond the piano's capabilities. All other physical instruments have similar amplitude functions. In a digital setting, those functions can be simulated or approximated.

Humans can generally hear sounds with frequencies between 20 Hz and 20,000 Hz. The piano notes are all well within that range. Every octave (or 12 keys) on the piano is a has twice the frequency of the previous octave. This is easy to see as the A notes are (in Hz) 27.5, 55, 110, 220, 440, 880, and so on. The highest frequency on the piano is 4186hz and two higher than that would be 16744 Hz, so at most there could only be two more octaves added beyond what we have within human hearing. (See slide 8 to see a graph of an idealized A4 note wave).

Sound is represented in digital form by a sequence of amplitude values. An analog signal is converted to or from a digital signal at a given sampling rate and bit depth. The sampling rate is the number of samples taken from the analog signal per second to form a discrete waveform i.e. the digital audio signal. The bit depth defines the number of increments of amplitude in each sample. For instance, if the sampling rate of a song is 8192 Hz with bit depth of 8, then one second of sound at that rate is represented by 8192 evenly spaced samples and the amplitudes of the samples take on values from -127 to 127. The sample rate and bits per sample define a two-dimensional grid. If we have *k* bits then the amplitude range is broken up into a grid of 2^k nodes. Half of the nodes describe the positive amplitude range and half describe the negative range. As an example, CD audio uses a sampling rate of 44.1 kHz with 16 bits per sample and presently defines the standard in high quality audio. However, sound engineering equipment can deal with rates such as 96khz and 24 bits per sample (or higher). When sound engineers mix audio at a high sampling rate it needs to be re-sampled before it can be played on regular CD players.

In analog systems, amplitude is measured in dB, but in digital systems, the amplitude is a percentage of the maximum for the input or output device. If a digital 8 bit sample has a value of 127 it's telling the output hardware to output at the loudest amplitude it can, and for sound input, receiving a value of 127 would mean the actual input signal was exactly at the max for the device or exceeded the max. Recordings of amplitudes in excess of the input device's capabilities manifest as distortion in the analog recordings. In digital systems, excessive amplitudes lead to "clipping." Clipping means the amplitude went beyond the representable range and is left at its maximum or minimum value depending on where it went out of range. For instance, if one continues to scale a digital representation of a *sin* wave eventually the result is a square wave because as the wave amplitude increases it goes beyond the amplitude range in the positive and negative and gets cut off to a square, but since $sin(x) = 0$ for some x, it stays 0 at those points and thus we are left with a square wave.

In Matlab sound can be played with the command `sound(vec,r,b)`, which takes a vector *vec* of samples, each in (or truncated to) the range [-1,1], with r being the sample rate and b being the bit resolution. For CD quality, $r = 44.1$ kHz and $b = 16$. A sound that lasts one second at rate r is represented by a vector with length r (Refer to slide 12 for a picture).

We can use interpolation on digital sound for a few important applications. In sound engineering, one may want to record or produce sound at a sample rate r to be played on audio systems of sample rate kr, where k is normally a power of 2 or 1/(power of two); in other words there may be a reason to start out with a low sampling rate and convert to a higher one or vice versa. We can also slow down or speed up an audio sample. For instance, we can re-sample a recording of an A4 piano note to any other note by interpolating, thereby increasing or decreasing its frequency; but that can change the length of the note to be shorter or longer and some error is introduced in the process.

Using the DFT interpolation, we can re-sample a one second audio file from rate r to $r/2$ like this: `dftinterp(audio_file,2*r)` or re-sample a one second A4 note to A5 (440 to 880 Hz) like this: `dftinterp(note_a4,r/2)` (we had to modify dftinterp to produce less points). To re-sample the note to other notes such as C4 for instance we need to look at the frequency of A4, which is 440 Hz, and the frequency of the other note whose frequency is x. Then the scaling factor is $440/x$. The A5 note has a frequency of 880; 440/880 = 1/2, which is why we use $r/2$ above (where the audio files involved are 1 second at rate r). The C4 note has a frequency of 261.626;

440/261.626 ~= 1.68179. Therefore, to resample the A4 note to a C4 note we need to use a factor of 1.68179r (and in practice we would take the floor of that since samples are discrete). So interpolating the A4 note into a C4 note means the result has more samples (where as, when we resample to a higher pitched note like A5 we interpolated to fewer samples).

In order to generate music, notes classified by their corresponding frequencies can be inserted into a vector and then grouped by the major scale they belong in. A note interpolation function can take this vector of notes and add notes in between the existing notes using the DFT based interpolation. Based on the time signature, BPM (beats per minute), and choice of note type (whole, half, quarter, etc) the interpolated vector of notes can have the same duration as the original vector when played and thus provide an accompaniment to the original notes. Let

$sg = [n_1\ n_2\ \ldots\ n_k]$ be a k note song at some tempo and time signature and let each n_i be a D note. Each n_i is a natural number in the range of 1 to 50, mapping to a scale on a standard piano. Let the notes be in some key, say C major. Using `round(dftinterp(sg,2*k))` we get a new song with $2k$ notes and the new notes all fall between the original notes. Let the notes in the new song be of type $2D$, then the two songs have the same duration because even though there are more notes, each note is proportionally faster.

A standard piano has 88 keys; all the white keys, which are natural notes, are in the key of C Major. We can map a song in C major from the piano index to the C major scale index. As an example, using *sg* = [24 28 26 31], we can call `scale_song(sg,2)` to double the number of original notes in the vector. Once two sound tracks are created from the vectors, they can be mixed together to form one audio track. (See slide 16 for the resulting graph. Note, in this slide the graph shows the result of creating 4*n* notes where *n* = 4).

Our goals of audio production involved creating multiple music tracks, mixing them into one "final" track, controlling properties of each audio track so that the amplitudes of selected frequency ranges can be controlled, maximizing amplitudes, and tweaking amplitudes of all tracks so that no clipping occurs in the final mix. To mix tracks we use the principle of superposition: audio tracks can be added to create a new track wherein both original sounds can be heard. In order to avoid clipping, tracks can be averaged, for instance, s = (s1 + s2)/2; mixing was done entirely in analog circuits until computers were powerful enough. In analog circuits the signals can degrade and shift in undesirable ways.

When Mixing, if we want certain tracks to be louder than others we can use weights, for instance, $s = .3s_1 + .7s_2$; this is analogous to the operation of a multi-track mixing board where each coefficient represents the master volume slider for that track.

We can also do some audio frequency equalization. Ranges of frequency of the audio can generally be grouped; for instance, bass is a general term referring to the low frequency portion of the audio, while treble refers to the high frequencies. If we want higher quality audio production, we use higher granularity; further divide the frequency range into *n* ranges or "bands" for instance. More advanced audio equipment has equalization parameters for many frequency ranges. In software we use the DFT on the audio data to directly edit the frequencies. The elements in the leftmost and right most part of the DFT are low frequencies and high in the middle. Here is a graph of some reference frequencies from $27.5 * 2^k$ where *k* goes from 0 to 9. (See slide 20 for the graph). The graph was created by making sound files of *sin* waves at the frequencies 27.5 to

27.5 * 2^9 Hz. The test frequencies were chosen because they are target frequencies in music starting with the lowest A note and moving up in octaves even beyond the last octave on a piano. The low frequencies appear closest to the right and left edges and the highest frequencies appear closer to the middle. The graph shows redundant frequency information mirrored across the middle of the graph; the height of the spike shows the relative contribution of that that frequency in the original audio track.

To equalize the sound in a slightly naive way, we can scale the areas of the DFT down or up based on the percentages into the graph. To equalize the lowest frequencies we might choose from 0.1% to 10% into the graph, doing the same for the mirrored image on the other side. This would be considered "naive," because it doesn't allow us to select frequency in Hertz; although, some further calculations could be done to figure out a mapping between percentages and frequency ranges.

For testing of the EQ, we used a sound file called `vivaldi` which is a high quality audio sample of a few seconds of Vivaldi's Four Seasons, recorded in DDD and uncompressed (extracted directly from a CD we had). To create the graph shown in slide 22, one would type into Matlab the following: `vivaldi_altered =`

`multibandEQ(vivaldi, [.001 .1 .9 (1-.001)], [.2 .2], 44*1024);`. The second two arguments are vectors; the first vector specifies which percentage ranges to work on and the second specifies the scaling factor to apply to the associated range. A factor of 1 in the second vector will cause the associated range to be not effected. In the second vector we set the EQ to work on 0.1% to 10% of the range and 90% to 99% (the redundant mirrored area). The way it is written, the redundant area always has to be adjusted separately but in most cases it would be equalized the same way as the other half. The third vector argument contains scaling factors which associate directly with the ranges; in this case we are reducing the ranges to 20% of their value.

To make the EQ even better we could have used an arbitrary discrete graph as a diagonal matrix to multiply the DFT by. Instead we used the graph of a piecewise function defined by a set of constant functions. In our example we use the range 0.1% to 10% which turns out to cover most of the frequencies in the Vivaldi piece, even beyond just the bass frequencies.

To make the "graphic" equalizer we use the information outlined in the section on the Weiner filter. The idea of the Weiner filter is to find some diagonal matrix D such that $||F^{-1}DF\mathrm{x} - c||$ (the Euclidian norm) is minimized where $|x| = n$ and c was the noise vector, perpendicular to the sound vector x. The complicated thing about the Weiner is finding D. But it is also possible to define a D in order to have other types of filters. Let $D(j,j) = f(j)$, $j \leq n/2$ and $D(j,j) = f(n/2) - f(j - n/2)$ for $j > n/2$. In other words, $D = Iv$ where $v_i = f(i)$ if $i \leq n/2$ and $v_i = f(n/2) - - f(j - n/2)$ if $i > n/2$ and I is the nxn identity matrix. D can be used to scale the frequency information in the DFT of x based on some arbitrary function f. For instance if $f(j) = j$ then when the diagonal of D is taken in sequence its graph looks like an equilateral triangle. The effect of this filter or "preset" would be to increase frequency values more and more as the frequencies increase. Any choice of f here will yield an equalization graph in this way. We call it a "preset" because some equalizers store and let you choose from a set of already made graphs like this. High end equalizers let you create arbitrarily complex equalization graphs. But standard equalizers work on ranges of frequencies and scale those ranges independently. That is why we choose to have our equalizer let you pick a range by percentage and then a value for the scaling factor of that range, also a percentage.