Lecture 3
Quantization

Most people have a good idea how music becomes a groove in an LP record: a microphone converts sound waves to electrical waves, and these are duplicated in the grooves of the phonograph. It's very similar to the process on a tape recorder, but the impulses are converted to magnetic "waves" on the tape.

The CD has to be wildly different: it represents music as numbers, and the numbers are somehow written to the CD. Let's start with the "writing" process. The amplitude of a sound, \( f(t) \), has been sampled, once every \( T \) seconds. Here's what I imagine a CD signal would look like.

The time is evenly spaced on the CD (somehow), and the height represents the amplitude of the sound at that instant. Unfortunately, this doesn't make a lot of sense. The signal that’s put on the CD is either zero or one. It’s represented by a pit, laser-burned into the CD: photo below.

FYI on this: the pit has to be flat on the bottom. In fact, the system works by shining a laser light on the surface of the CD. If there's a pit, the light is reflected back and interferes with itself. The depth of the pit is rigged so that the interference cancels out the light beam, and a detector picks up the sudden decrease in light intensity.

For this to work, the depth of the pit has to be one-quarter the wavelength of the laser light used to read the CD. Check it out here.
The moral is, though: a single pit doesn't allow any expression of amplitude. The amplitude must be coded, then expressed in terms of "pits'. This takes several steps.

The first step is to **quantize** the signal. Amplitudes are represented in binary:

\[
(a_2a_1a_0,b_1b_2b_3)_2 = a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0 + \frac{b_1}{2^1} + \frac{b_2}{2^2} + \frac{b_3}{2^3}
\]

so that

\[
(101.011)_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 + \frac{1}{2^2} + \frac{1}{2^3}
\]

\[= (5.375)_{10}
\]

This reduces amplitudes to zeroes and ones. Now there's a problem: not every number has a finite binary expansion. But CD's definitely only have finite data -- like 16 songs at most!

So -- the trick is, limit the number of bits used in the binary expansions. Only sixteen bits are used, giving sixteen spots on the CD, reserved for each binary number.

Finally, the spots are written onto the CD something like this:

The space between \((n-1)T\) and \(nT\) is subdivided into sixteen portions; each portion holds a one or zero. Together, they represent a sixteen-bit binary number. The particular number above is 1010101000000000.

This system of representing amplitude is called pulse-code modulation, but in fact, this isn't what is actually used. PCM assumes that digital circuits can handle any string of zero’s or ones’s equally well. This is not the case; constant switching between 0 and 1 can cause errors. Chip's site tells us, "CDs use an encoding scheme called **Eight to Fourteen Modulation** (EFM) which avoids really short pits or having pits too close together by ensuring that there are always at least two 0 bits between any 1 bit."
Actually, even this isn't quite right. The data written to CD is encoded first, using an error-correcting code. That way, the system can detect glitches and correct them automatically. We won't go into the theory of error-correcting codes, but it's a fascinating mathematical subject.

It does bring up a question, though: why are we going through this? Because each of the steps we've talked about introduces some problems, and Fourier ideas can be used to think about them.

First, a small problem: PCM as described can't represent any negative numbers. This could be handled by assigning one bit to represent sign.

A second issue: how do we actually quantize? How do we distribute positive and negative numbers among the $2^{16} = 65,536$ levels of sound allowed us in a 16 bit scheme?

One scheme is to divide amplitudes between $-A$ and $A$, with level $-A$ corresponding to 0, then 0 to 32,768, and finally $A$ to 65,536. The advantage to this is that no matter what the signal, zero is always at the same place. Then, the region between amplitude 0 and amplitude $A$ is subdivided into 32,768 equally sized regions.

The problem with such a scheme is that amplitudes greater than $A$ cannot be represented at all; the best such a system can do is to code them the same as $A$ itself is coded. This effect is called clipping; of course the same thing happens for amplitudes below $-A$. Situating our zero amplitude half way between 0 and 65,536 guarantees that large positive and large negative amplitudes will be clipped equally.

The picture below shows the effect of clipping a sine signal with $A = 1.5$.

![Clip of sine signal](image)

We can represent the clipping of arbitrary functions via the the \textit{clip} operator, defined as:

\[
\text{clip}_A[f](x) = \begin{cases} 
  f(x) & \text{if } |f(x)| \leq A \\
  A & \text{if } f(x) > A \\
  -A & \text{if } f(x) < -A 
\end{cases}
\]
**Lab Project:** Try implementing `clip` in Matlab, then using it on a variety of music, but particularly on a nice sine signal. One very very cheap way to do this is to construct a sine wave in Matlab, with a frequency you can actually hear, then give it a fairly large amplitude, and use `wavwrite` to save it. Matlab will give you an error message, that the signal has been clipped: your A is larger than the coding Matlab handles.

For the music, try clipping it at 1/2, 3/4 and 15/16 of its maximum value. How does each sound? Better still -- what do the spectrograms look like?

An interesting computation, either using the symbolic toolbox, or by hand: compute the exact Fourier Series of `clip[sin(2*pi*x)]`.

There are all sorts of ways to avoid clipping; one is to use unequally spaced quantizing intervals. It’s generally agreed that you shouldn’t mess with the intervals near zero, the low-amplitude parts, but if the very high amplitudes were squished down, you wouldn’t have to clip ‘em. The log function is perfect at such a job; you apply log to compress a sound, quantize with a uniform quantizer, then use an exponential to expand the sound back. The term for compressions/’expansion is “companding”. Variants of the log functions can be used, leading to the [“mu-law compander.”](http://en.wikipedia.org/wiki/Mu_law compander) It’s used as a standard in telephone transmission, at least in North America and Japan.

And of course you’re welcome to examine this in lab; it might be interesting to see what clipping versus companding does to music that varies quickly from quiet to very loud. Which system produces better sound?

The main point of this section is to look at the number of bits used in quantization. The sound on music CD’s is standardized at 16 bits; telephone uses 8 bits. It’s said that you can actually follow conversations with only two bits, which would be an interesting lab thing to try. What’s the effect?

There’s a good analogy to understand quantization: drawing curves on a computer screen. If you draw every object in terms of square pixels, straight lines get quantized into collections of blocks, as in the picture below.

![Quantization Diagram](http://example.com/quantization.png)
Now, take a sine wave of really small amplitude:

![Sine wave diagram](image)

It gets replaced by a signal that’s either zero or one . . a square wave. The square wave is one of those signals that has sudden transitions, hence high frequencies. Whereas the original signal had only a low-frequency sine.

This is an example of the addition of noise to a sound, because of quantization. Below, a signal has been quantized; left, the spectrogram before quantization; right, after.

![Spectrogram comparison](image)

Not hard to see the tremendous amount of high frequency noise added by quantization.

Now, your first task is to imitate this in Matlab. Write a quantizer, that reduces a sound to n bits, where $1 < n < 16$. Then take a variety of sounds, quantize them to small n, and see the effect on the spectrum.

Below, we've done this for you, with decimal digits instead of binary (as would be actually used). First we took **Shakira's Suerte** (English) and used three decimal digit quantization. Below, left, the numbers; right, the spectrogram.

<table>
<thead>
<tr>
<th>0.6820</th>
<th>0.6830</th>
<th>0.6850</th>
<th>0.6860</th>
<th>0.6880</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7220</td>
<td>0.7240</td>
<td>0.7260</td>
<td>0.7280</td>
<td>0.7300</td>
</tr>
<tr>
<td>0.7320</td>
<td>0.7340</td>
<td>0.7360</td>
<td>0.7380</td>
<td>0.7400</td>
</tr>
<tr>
<td>0.7420</td>
<td>0.7440</td>
<td>0.7460</td>
<td>0.7480</td>
<td>0.7500</td>
</tr>
<tr>
<td>0.7520</td>
<td>0.7540</td>
<td>0.7560</td>
<td>0.7580</td>
<td>0.7600</td>
</tr>
</tbody>
</table>

![Numbers and spectrogram](image)
Now, two decimal digits, but this time with a graph of amplitude:
One digit -- you can finally hear the difference:

The graph explains the spectrogram: the graph is becoming like a bunch of square waves, each of which has a spectrum including lots of high frequency components.
finally, just for fun -- zero digits -- that is, either on or off:

The very odd thing is that you can still make out the words in the song!

Finally, look at the difference between the original sound and the quantized. The literature remarks that the difference is, essentially, noise. That is, it is uncorrelated with the original signal, and random -- uncorrelated with itself.

**Lab Project:** Verify this, by quantizing, subtracting, then doing the cross-correlation.

This is part of the general theory of quantization. If the quantization error truly is noise, then you can treat it via probability theory. Then it is fairly easy to show that if you quantize to n binary bits, the signal to error ratio (largest signal to largest quantization error, in db) is roughly $6.02n+1.76$ db. For 8 bits, this is about 50 db -- the kind of noise we only dreamed of with LP or magnetic tape, in the sixties. With sixteen bits, it’s around 98 db -- very very low! This is why CD sound is so superior to LP.

The intuitive computation isn’t very hard. If we use 8 bits for quantization, we’ll get 256 levels. The noise must then be $1/256$ of the signal. This means you have to have a signal to noise ratio of $20\log 2^{8}$ or about $8*20* \log(3)= 6.02*8$db.

mp3 encoding depends on quantization error being noise. This is a good time to check out the mp3.pdf and mp3too.pdf files. Basically, mp3 encoding looks at a small chunk of music, and computes its spectrum. Then it looks at masking curves, and asks, “how many bits do I have to use so the music masks the quantization noise”? Usually it’s much fewer than 16 bits, and each bit I don’t have to use for quantization allows me to compress the signal.

As you can probably guess by now, sometimes this whole process goes wrong. The picture below shows a periodic signal, of very small amplitude, quantized.
If you heard this at slightly higher frequency, it would sound like on/off/on/off and so on. Against a background of silence, it would be very noticeable and very annoying. Try reproducing this, quantizing a small amplitude, low frequency sine.

What this shows is that for small amplitude signals, quantization error may not be noise; it may be correlated with the signal and non-random. The problem is that the ear tends to tune out low-level random noise, but noise that comes in bursts, correlated with the music, is noticeable and annoying.

It can get worse (you knew that). The music on a CD has been filtered (the high frequencies are limited, referred to as band-limiting), so that it meets the conditions of the Nyquist theorem, so that there will be no aliasing when we sample at 44100 Hz. But the quantized signal has been changed to a square wave, so all sorts of high-frequencies are added. Any additional sampling that might be done will produce aliasing, and that may produce beats with whatever is left of the original signal.

For example, a signal, limited to 25 hz, is sampled at 50 hz. Take a 7 hz sine wave, of just the right amplitude to be quantized by one bit. So now it’s either 1 or 0, so it’s a square wave of 7 hz. It has harmonics at 21 hz, 35 hz, 49 hz. With 50 hz sampling, you get aliasing at 1 hz and at 15 hz. If the alias components are near other main frequencies, you’ll get beats. The technical term for this is “granulation noise”. See if you can make it happen by quantizing then resampling.

The solution to all these problems is... well, here’s what the solution looks like in pictures.
At the bottom right is a straight line, all jumpy from quantization (sampled at 72 dots per inch to go onto my computer screen). At the left, a much less jumpy line. In the world of computer graphics, it is called an “anti-aliased” line. You can see what’s been done; pixels have been added to the original, to ‘soften’ the hard edges of the line.

Anti-aliasing is a bad term for sound processing, because “aliasing” has a technical meaning already. The term that’s used in sound processing is “dithering”. After the Old English word, “dither”, related to “dodder”: here’s the Oxford English Dictionary:

[Cf. DODDER, DIDDER, DITHER: the form is that of a frequentative, as in patter, shiver, totter, etc.: but the etymology of the stem dad-, did-, dod-, is obscure; cf. DADE.] v. intr. To quake, tremble.

What’s done is, before quantization and sampling, a small amount of random noise is added to the music. Then the quantization error at low levels is no longer correlated to the music, and the noise that results not as annoying.

Quantization is a necessary evil, a technique we must use to have digital music, digital images, digital anything. But there have been hints that quantization has a bright side -- for example, voice, quantized to four digits, is very understandable. It would make sense for telephone companies to use only four digits -- think of the saving in bandwidth (the amount of information that would have to be transmitted per second) when going from 16 bits to 4 bits!

Quantization, is, in fact, at the basis of mp3 and JPEG compression. The idea is to take a sound or image file, compute the FFT, and quantize the coefficients, then use Huffman or other encoding schemes to store the information. It's actually a lot trickier than it sounds; if your music consists of one violin note, you don't want to reduce the number of digits representing that note. On the flip side, if the violin is playing with an entire orchestra crashing around it, the violin can safely go.

This tells us two things: first, that we have to decide which frequencies get dropped digits and which are left alone. This is usually determined by experiments of people looking at images or listening to music. This kind of decision is at the basis of psychoacoustic encoding methods for music.

The second issue we need to deal with is the fact that the frequencies in a piece of music change from minute to minute. For this reason, mp3 looks at only a short segment of music, and encodes that, before moving on to the next segment. JPEG does the same thing, but in small blocks -- which is why a low-quality jpeg image looks blocky.

Lab Project: Let’s ignore all the subtleties. Back in the "Power" application paper, we gave some suggestions on compressing music based on the amount of power in a given frequency. Let’s redo that. Take the FFT, and again compute the power. Now, instead of dropping frequencies with low contribution to the power, use fewer digits to represent their coefficients. Now take the IFFT, and compare the sound to the original power-only based compression. Of course, you won’t see any compression in size of the signal, as we’re not Huffman-encoding. But, has the sound quality improved?