How does Shazam work

Have you ever wondered how Shazam works? I asked myself this question a few years ago and I read a research article written by Avery Li-Chun Wang, the confounder of Shazam, to understand the magic behind Shazam. The quick answer is audio fingerprinting, which leads to another question: what is audio fingerprinting?

When I was student, I never took a course in signal processing. To really understand Shazam (and not just have a vague idea) I had to start with the basics. This article is a summary of the search I did to understand Shazam.

I'll start with the basics of music theory, present some signal processing stuff and end with the mechanisms behind Shazam. You don't need any knowledge to read this article but since it involves computer science and mathematics it's better to have a good scientific background (especially for the last parts). If you already know what the words "octaves", "frequencies", "sampling" and "spectral leakage" mean you can skip the first parts.

Since it's a long and technical article (11k words) feel free to read each part at different times.

Music and physics

A sound is a vibration that propagates through air (or water) and can be decrypted by ears. For example, when you listen to your mp3 player the earphones produce vibrations that propagate through air until they reach your ears. The light is also a vibration but you can't hear it because your ears can't decrypt it (but your eyes can).

A vibration can be modeled by sinusoidal waveforms. In this chapter, we'll see how music can be physically/technically described.

Pure tones vs real sounds

A pure tone is a tone with a sinusoidal waveform. A sine wave is characterized by:

- Its frequency: the number of cycles per second. Its unit is the Hertz (Hz), for example 100Hz = 100 cycles per second.

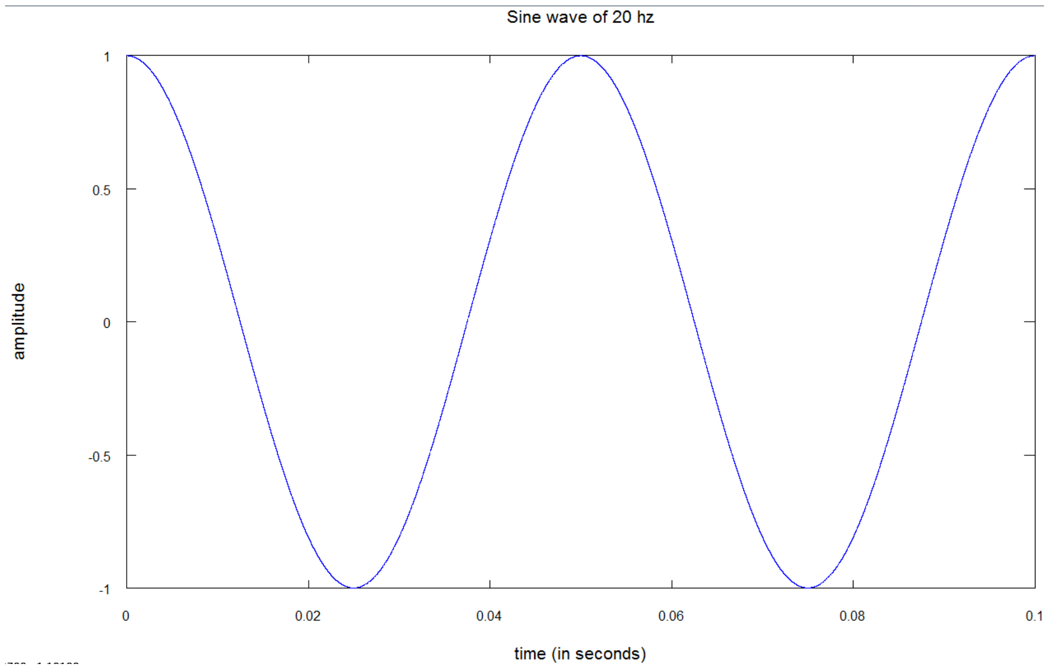
- Its amplitude (related to loudness for sounds): the size of each cycle.

Those characteristics are decrypted by the human ear to form a sound. Human can hear pure tones from 20 Hz to 20 000 Hz (for the best ears) and this range decreases over age. By comparison, the light you see is composed of sinewaves from 4*10^14 Hz to 7.9*10^14 Hz.

You can check the range of your ears with youtube videos like this one that displays all the pure tones from 20 Hz to 20k Hz, in my case I can't hear anything above 15 kHz.

The human perception of loudness depends on the frequency of the pure tone. For instance, a pure tone at amplitude 10 of frequency 30Hz will be quieter than a pure tone at amplitude 10 of frequency 1000Hz. Humans ears follow a psychoacoustic model, you can check this article on Wikipedia for more information.

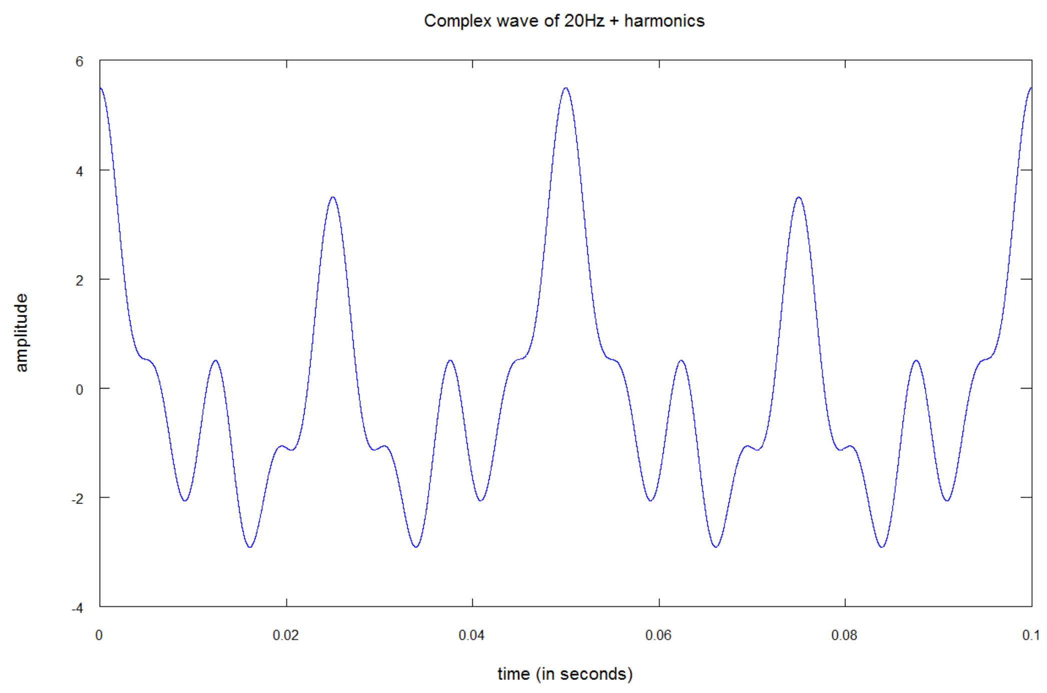
Note: This fun fact will have consequences at the end of the article.



pure sinewave at 20 Hz

In this figure, you can see the representation of a pure sine wave of frequency 20hz and amplitude 1.

Pure tones doesn't naturally exist but every sound in the world is the sum a multiple pure tones at different amplitudes.



composition of sinewaves

In this figure, you can see the representation of a more realistic sound which is the composition of multiple sinewaves:

- a pure sinewave of frequency 20hz and amplitude 1
- a pure sinewave of frequency 40hz and amplitude 2
- a pure sinewave of frequency 80hz and amplitude 1.5
- a pure sinewave of frequency 160hz and amplitude 1

A real sound can be composed of thousands of pure tones.

Musical Notes

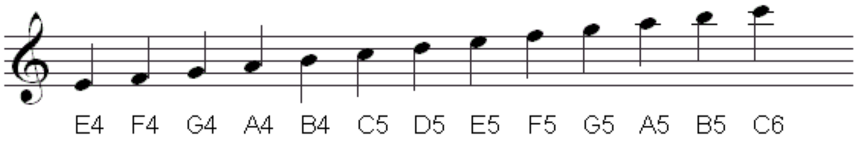
Simple Gifts
Joseph Brackett, Jr., 1848

A music partition is a set of notes executed at a certain moment. Those notes also have a duration and a loudness.

The notes are divided in **octaves**. In most occidental countries, an octave is a set of 8 notes (A, B, C, D, E, F,G in most English-speaking countries and Do, Re, Mi, Fa, Sol, La, Si in most Latin occidental countries) with the following property:

- The frequency of a note in an octave doubles in the next octave. For example, the frequency of A4 (A in the 4th octave) at 440Hz equals 2 times the frequency of A3 (A in the 3rd octave) at 220Hz and 4 times the frequency of A2 (A in the 2nd octave) at 110Hz.

Many instruments provides more than 8 notes by octaves, those notes are called semitone or halfstep.



For the 4th octave (or 3rd octave in Latin occidental countries), the notes have the following frequency:

- C4 (or Do3) = 261.63Hz
- D4 (or Re3) = 293.67Hz
- E4 (or Mi3) = 329.63Hz
- F4 (or Fa3) = 349.23Hz
- G4 (or Sol3) = 392Hz
- A4 (or La3) = 440Hz
- B4 (or Si3) = 493.88Hz

Though it might be odd, the frequency sensitivity of ears is logarithmic. It means that:

- between 32.70 Hz and 61.74Hz (the 1st octave)
- or between 261.63Hz and 466.16Hz (4th octave)
- or between 2 093 Hz and 3 951.07Hz (7th octave)

Human ears will be able to detect the same number of notes.

FYI, the A4/La3 at 440Hz is a standard reference for the calibration of acoustic equipment and musical instruments.

Timbre

The same note doesn't sound exactly the same if it's played by a guitar, a piano, a violin or a human singer. The reason is that each instrument has its own **timbre** for a given note.

For each instrument, the **sound produced is a multitude of frequencies that sounds like a given note** (the scientific term for a musical note is **pitch**). This sound has a **fundamental** frequency (the lowest frequency) and multiple **overtones** (any frequency higher than the fundamental).

Most instruments produce (close to) **harmonic sounds**. For those instruments, the overtones are multiples of the fundamental frequency called **harmonics**. For example the composition of pure tones A2 (fundamental), A4 and A6 is harmonic whereas the composition of pure tones A2, B3, F5 is **inharmonic**.

Many percussion instruments (like cymbals or drums) create inharmonic sounds.

Note: The pitch (the musical note perceived) might not be present in the sound played by an instrument. For example, if an instrument plays a sound with pure tones A4, A6 and A8, Human brain will interpret the resulting sound has an A2 note. This note/pitch will be an A2 whereas the lowest frequency in the sound is A4 (this fact is called the missing fundamental).

Spectrogram

A music song is played by multiple instruments and singers. All those instruments produce a combination of sinewaves at multiples frequencies and the overall is an even bigger combination of sinewaves.

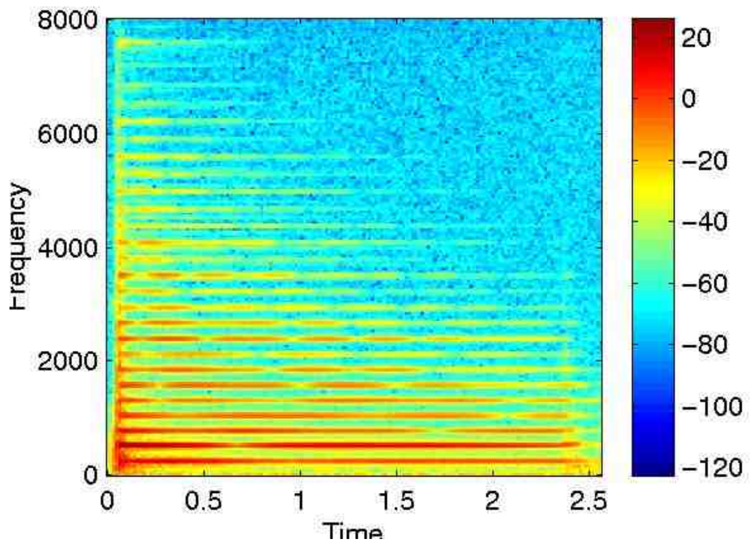
It is possible to see music with a spectrogram. Most of the time, a spectrogram is a 3 dimensions graph where:

- on the horizontal (X) axis, you have the time,
- on the vertical (Y) axis you have the frequency of the pure tone
- the third dimension is described by a color and it represents the amplitude of a frequency at a certain time.

For example, here is a sound of a piano playing of C4 note (whose fundamental frequency is 261.63Hz)

0:00 / 0:00

And here is the associated spectrogram:



The color represents the amplitude in dB (we'll see in a next chapter what it means).

As I told you in the previous chapter, though the note played is a C4 there are other frequencies than 261Hz in this record: the overtones. What's interesting is that the other frequencies are multiple of the first one: the piano is an example of a **harmonic instrument**.

Another interesting fact is that the intensity of the frequencies changes through time. It's another particularity of an instrument that makes it unique. If you take the same artist but you replace the piano, the evolution of frequencies won't behave the same and the resulting sound will be slightly different because each artist/instrument has its own style. Technically speaking, these evolutions of frequencies are modifying the **envelope** of the sound signal (which is a part of the timbre).

To give you a first idea of Shazam music fingerprinting algorithm, you can see in this spectrogram that some frequencies (the lowest ones) are more important than others. What if we kept just the strongest ones?

Digitalization

Unless you're a vinyl disk lover, when you listen to music you're using a digital file (mp3, apple lossless, ogg, audio CD, whatever). But when artists produce music, it is analogical (not represented by bits). The music is **digitalized** in order to be stored and played by electronics devices (like computers, phones, mp3 players, cd players …). In this part we'll see how to pass from an analog sound to a digital one. Knowing how a digital music is made will help us to analyse and manipulate this digital music in the next parts.

Sampling

Analog signals are continuous signals, which means if you take one second of an analog signal, you can divide this second into [put the greatest number you can think of and I hope it's a big one !] parts that last a fraction of second. In the digital world, you can't afford to store an infinite amount of information. You need to have a minimum unit, for example 1 millisecond. During this unit of time the sound cannot change so this unit needs to be short enough so that the digital song sounds like the analog one and big enough to limit the space needed for storing the music.

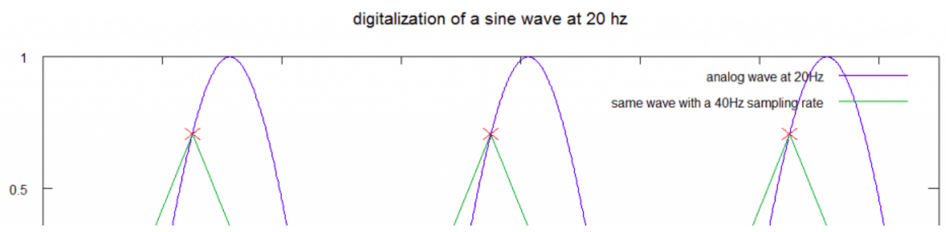
For example, think about your favorite music. Now think about it with the sound changing only every 2 seconds, it sounds like nothing. Technically speaking the sound is **aliased**. In order to be sure that your song sounds great you can choose a very small unit like a nano (10^-9) second. This time your music sounds great but you don't have enough disk space to store it, too bad.

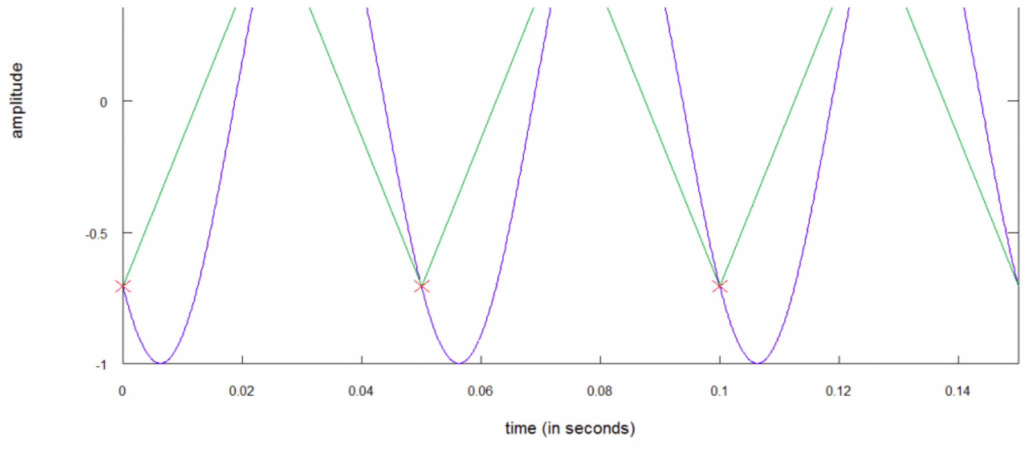
This problem is called **sampling**.

The standard unit of time in digital music is 44 100 units (or **samples**) **per second**. But where does this 44,1kHz come from? Well, some dude thought it would be good to put 44100 units per second and that all … I'm kidding of course.

In the first chapter I told you that humans can hear sounds from 20Hz to 20kHz. A theorem from Nyquist and Shannon states that if you want to digitalize a signal from 0Hz to 20kHz you need at least 40 000 samples per second. The main idea is that a sine wave signal at a frequency F needs at least 2 points per cycle to be identified. If the frequency of your sampling is at least twice than the frequency of your signal, you'll end up with at least 2 points per cycle of the original signal.

Let's try to understand with a picture, look at this example of a good sampling:



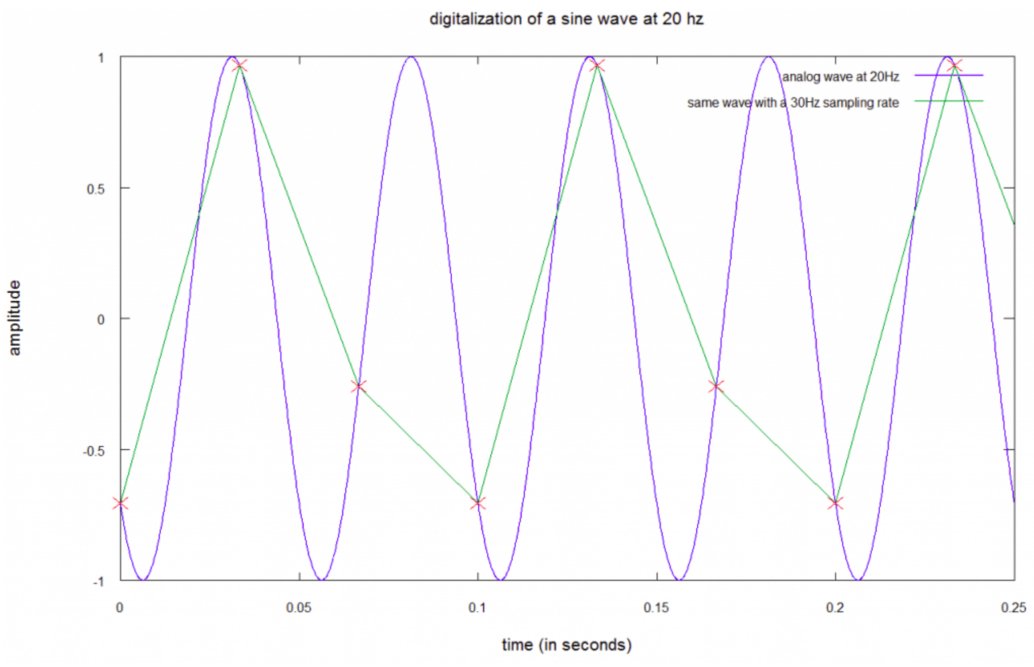


In this figure, a sound at 20Hz is digitalized using a 40Hz sampling rate:

- the blue curve represents the sound at 20 Hz,
- the red crosses represent the sampled sound, which means I marked the blue curve with a red cross every 1/40 second,
- the green line an interpolation of the sampled sound.

Though it hasn't the same shape nor the same amplitude, **the frequency of the sampled signal remains the same**.

And here is an example of bad sampling :



In this figure, a sound at 20 Hz is digitalized with a 30Hz sampling rate. This time the **frequency of the sampled signal is not the same as the original signal**: it's only 10Hz. If you look carefully, you can see that one cycle in the sampled signal represents two cycles in the original signal. This case is an under sampling.

This case also shows something else: if you want to digitalize a signal between 0Hz and 20 kHz, you need remove from the signal its frequencies over 20kHz before the sampling. Otherwise those frequencies will be transformed into frequencies between 0Hz and 20Khz and therefore add unwanted sounds (it's called **aliasing**).

To sum up, if you want a good music conversion from analogic to digital you have to record the analog music at least 40000 times per second. HIFI corporations (like Sony) chose 44,1kHz during the 80s because it was above 40000 Hz and compatible with the video norms NTSC and PAL. Other standards exist for audio like 48 kHz (Blueray), 96 kHz or 192 kHz but if you're neither a professional nor an audiophile you're likely to listen to 44.1 kHz music.

Note1: The theorem of Nyquist-Shannon is broader than what I said, you can check on Wikipedia if you want to know more about it.

Note2: The frequency of the sampling rate needs to be **strictly** superior of 2 times the frequency of the signal to digitalize because in the worst case scenario, you could end up with a constant digitalized signal.

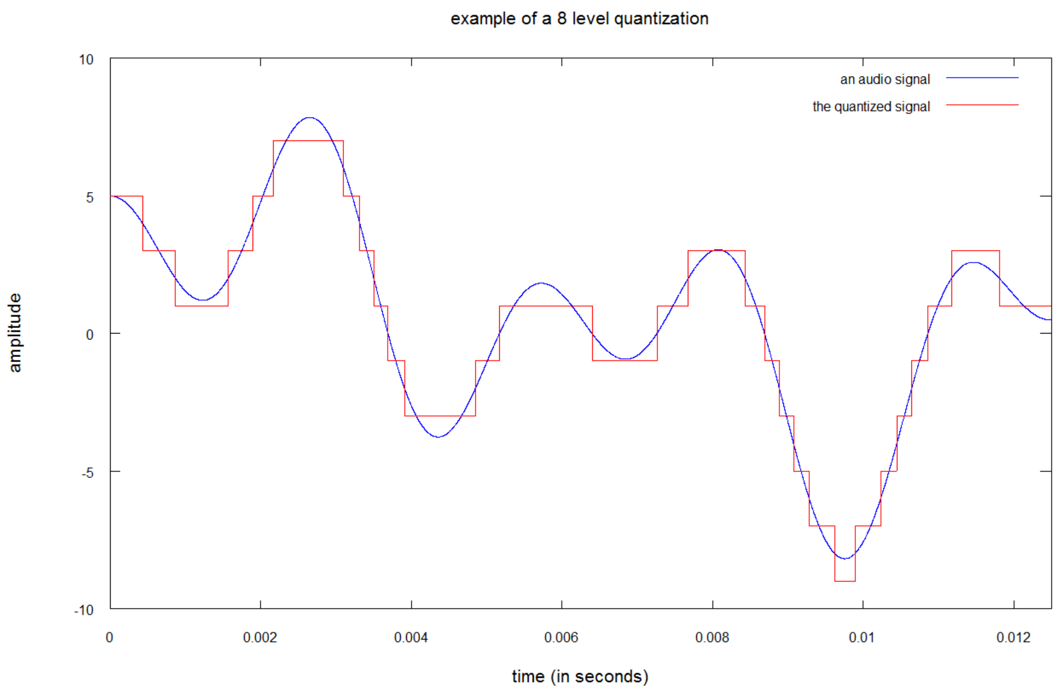
Quantization

We saw how to digitalize the frequencies of an analogic music but what about the loudness of music? The loudness is a relative measure: for the same loudness inside the signal, if you increase your speakers the sound will be higher. The loudness measures the variation between the lowest and the highest level of sound inside a song.

The same problem appears loudness: how to pass from a continuous world (with an infinite variation of volume) to a discrete one?

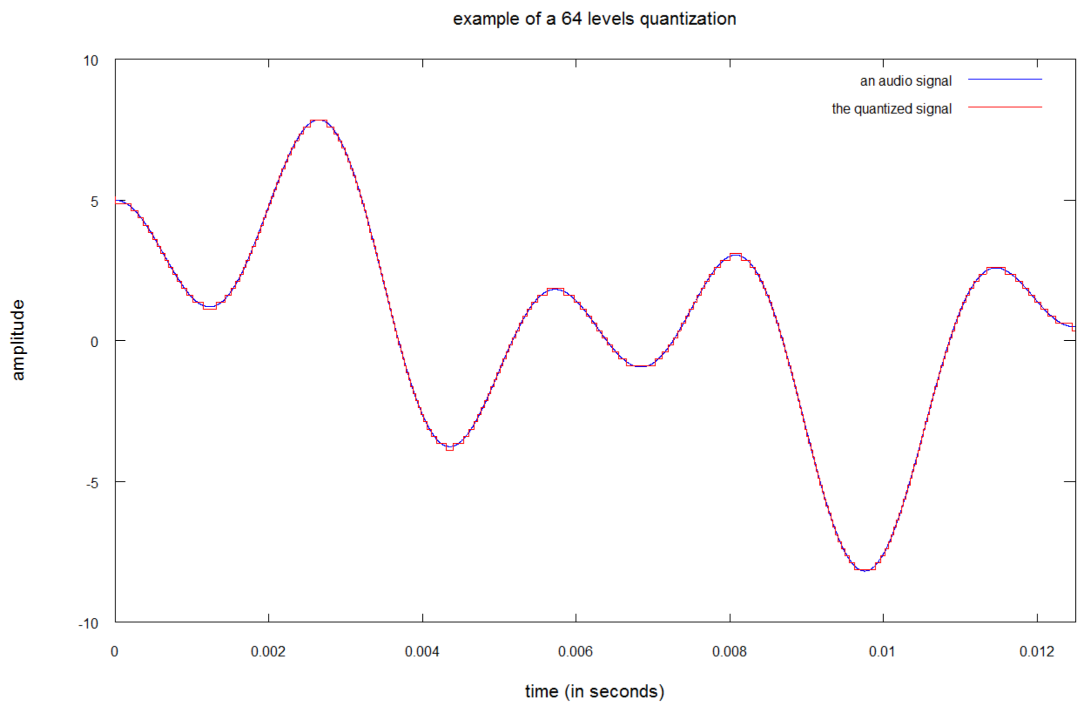
Imagine your favorite music with only 4 states of loudness: no sound, low sound, high sound and full power. Even the best song in the world becomes unbearable. What you've just imagined was a 4-level quantization.

Here is an example of a low quantization of an audio signal:



This figure presents an 8 level quantization. As you can see, the resulting sound (in red) is very altered. The difference between the real sound and the quantized one is called **quantization error** or **quantization noise**. **This 8 level quantization is also called a 3 bits quantization** because you only need 3 bits to implement the 8 different levels (8 = 2^3).

Here is the same signal with a 64 levels quantization (or 6 bits quantization)



Though the resulting sound is still altered, it looks (and sounds) more like the original sound.

Thankfully, humans don't have extra sensitive ears. **The standard quantization is coded on 16 bits**, which means 65536 levels. With a 16 bits quantization, the quantization noise is low enough for human ears.

Note: In studio, the quantization used by professionals is 24 bits, which means there are 2^24 (16 millions) possible variations of loudness between the lowest point of the track and the highest.

Note2: I made some approximations in my examples concerning the number of quantization levels.

Pulse Coded Modulation

PCM or Pulse Coded Modulation is a standard that represents digital signals. It is used by compact discs and most electronics devices. For example, when you listen to an mp3 file in your computer/phone/tablet, the mp3 is automatically transformed into a PCM signal and then send to your headphones.

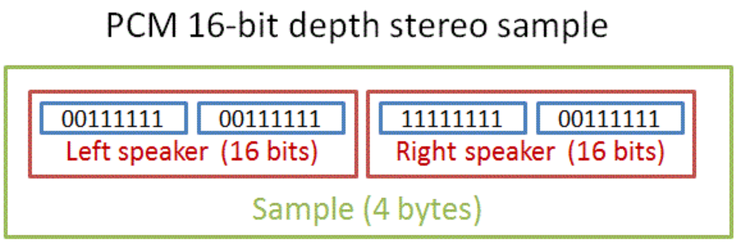
A PCM stream is a stream of organized bits. It can be composed of multiple channels. For example, a stereo music has 2 channels.

In a stream, the amplitude of the signal is divided into samples. The number of samples per second correspond to the sampling rate of the music. For instance a 44,1kHz sampled music will have 44100 samples per second. Each sample gives the (quantized) amplitude of the sound of the corresponding fraction of seconds.

There are multiple PCM formats but the most used one in audio is the (linear) PCM 44,1kHz, 16-

bit depth stereo format. This format has 44 100 samples for each second of music. Each sample takes 4 bytes:

- 2 bytes (16 bits) for the intensity (from -32,768 to 32,767) of the left speaker
- 2 bytes (16 bits) for the intensity (from -32,768 to 32,767) of the right speaker



In a PCM 44,1kHz 16-bit depth stereo format, you have 44100 samples like this one for every second of music.

From digital sound to frequencies

You now know how to pass from an analog sound to a digital one. But how can you get the frequencies inside a digital signal? This part is very important since the Shazam fingerprinting algorithm works only with frequencies.

For analog (and therefore continuous) signals, there is a transformation called the Contiguous **Fourier transform**. This function transforms a function of time into a function of frequencies. In other words, if you apply the Fourier transform on a sound, it will give you the frequencies (and their intensities) inside this sound.

But there are 2 problems:

- We are dealing with digital sounds and therefore finite (none continuous) sounds.
- To have a better knowledge of the frequencies inside a music, we need to apply the Fourier Transform on small parts of the full length audio signal, like 0.1 second parts so that we know what are the frequencies for each 0.1 second parts of an audio track).

Thankfully, there is another mathematical function, the **Discrete Fourier Transform (DFT)**, that works with some limitations.

Note: The Fourier Transform must be applied on only one channel, which means that if you have a stereo song you need to transform it into a mono song.

Discrete Fourier Transform

The DFT (Discrete Fourier Transform) applies to discrete signals and gives a discrete spectrum (the frequencies inside the signal).

Here is the magic formula to transform a digital signal into frequencies (don't run away, I'll explain it):

$$X(n) = \sum_{k=0}^{N-1} x[k] e^{-j(2\pi kn/N)}$$

In this formula:

- N is the size of the **window**: the number of samples that composed the signal (we'll talk a lot about windows in the next part).
- X(n) represents the nth **bin of frequencies**
- x(k) is kth sample of the audio signal

For example, for an audio signal with a 4096-sample window this formula must be applied 4096 times:

- 1 time for n = 0 to compute the 0^{th} bin a frequencies
- 1 time for n = 1 to compute the 1^{st} bin a frequencies
- 1 time for n = 2 to compute the 2^{nd} bin a frequencies
- …

As you might have noticed, I spoke about bin of frequencies and not frequency. The reason is that the DFT gives a **discrete spectrum**. A bin of frequencies is the smallest unit of frequency the DFT can compute. The size of the bin (called **spectral/spectrum resolution** or **frequency resolution**) equals the sampling rate of the signal divided by the size of the window (N). In our example, with a 4096-sample window and a standard audio sampling rate at 44.1kHz, the frequency resolution is 10.77 Hz (except the 0^{th} bin that is special):

- the 0th bin represents the frequencies between 0Hz to 5.38Hz
- the 1st bin represents the frequencies between 5.38Hz to 16.15Hz
- the 2nd bin represents the frequencies between 16.15Hz to 26.92Hz
- the 3rd bin represents the frequencies between 26.92Hz to 37.68Hz
- …

That means that the DFT can't dissociate 2 frequencies that are closer than 10.77Hz. For example notes at 27Hz, 32Hz and 37Hz ends up in the same bin. If the note at 37Hz is very powerful you'll just know that the 3^{rd} bin is powerful. This is problematic for dissociating notes in the lowest octaves. For example:

- a A1 (or La -1) is at 55Hz whereas a B1 (or Si -1) is at 58.27Hz and a G1 (or Sol -1) is at 49 Hz.
- the first note of a standard 88-key piano is a A0 at 27.5 Hz followed by a A#0 at 29.14Hz.

You can improve the frequency resolution by increasing the window size but that means losing fast frequency/note changes inside the music:

- An audio signal has a sampling rate of 44,1 kHz
- Increasing the window means taking more samples and therefore increasing the time taken by the window.
- With 4096 samples, the window duration is 0.1 sec and the frequency resolution is 10.7 Hz: you can detect a change every 0.1 sec.
- With 16384 samples, the window duration is 0.37 sec and the frequency resolution is 2.7 Hz: you can detect a change every 0.37 sec.

Another particularity for an audio signal is that **we only need half the bins computed by the DFT**. In the previous example, the bin definition is 10.7 Hz, which means that the 2047th bin represents the frequencies from 21902,9 Hz to 21913,6 Hz. But:

- The 2048th bin will give the same information as the 0th bin
- The 2049th bin will give the same information as the 1th bin
- The X+2048th bin will give the same information as the Xth bin
- ..

If you want to know why the bin resolution equals" the sampling rate" divided by "the size of the window" or why this formula is so bizarre, you can read a 5-part article on Fourier Transform on this very good website (especially part 4 and part 5) which is the best article for beginners that I read (and I read a lot of articles on the matter).

Window functions

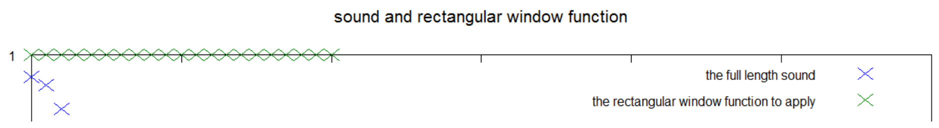
If you want to get the frequency of a one-second sound for each 0.1-second parts, you have to apply the Fourier Transform for the first 0.1-second part, apply it for the second 0.1-second part, apply it for the third 0.1-second part …

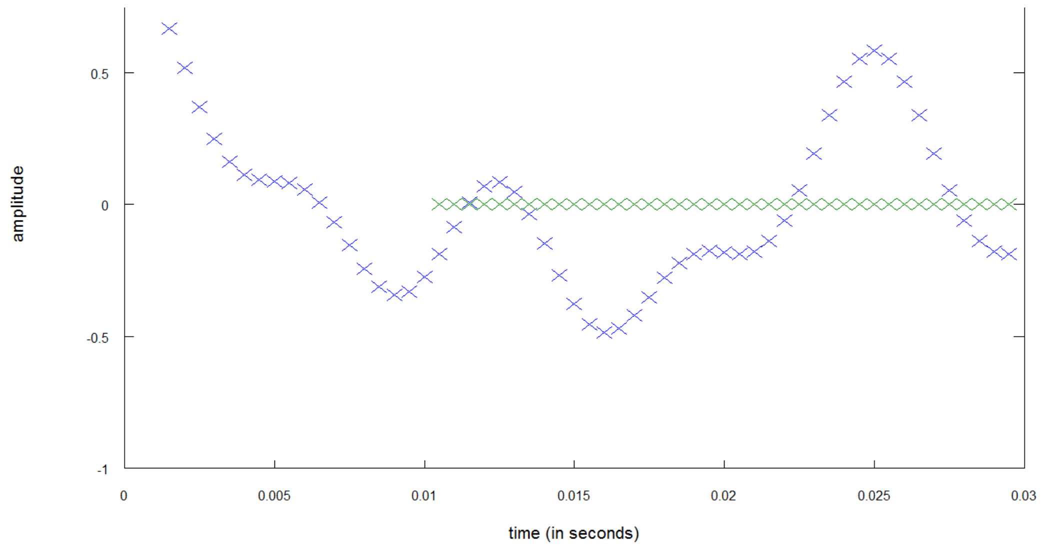
The problem

By doing so, you are implicitly applying a (rectangular) window function:

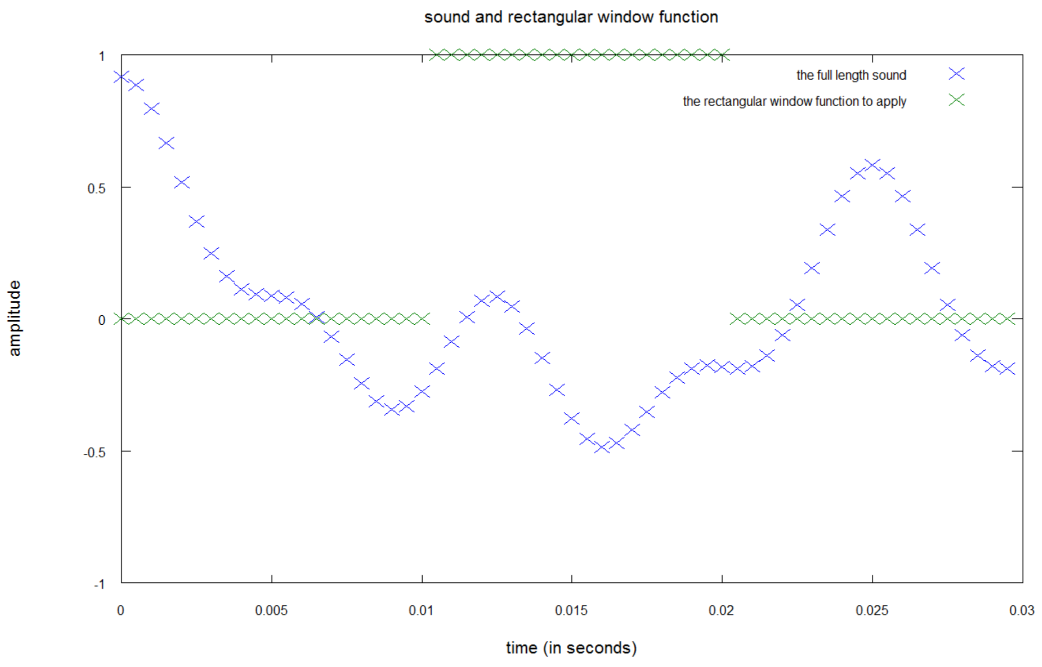
- For the first 0.1 second you are applying the Fourier transform on the full one-second signal multiplied by a function that equals 1 between 0 and 0.1second, and 0 for the rest
- For the second 0.1 second you are applying the Fourier transform on the full one-second signal multiplied by a function that equals 1 between 0.1 and 0.2 second, and 0 for the rest
- For the third 0.1 second you are applying the Fourier transform on the full one-second signal multiplied by a function that equals 1 between 0.2 and 0.3 second, and 0 for the rest
- …

Here is a visual example of the window function to apply to a digital (sampled) audio signal to get the first 0.01-second part:





In this figure, to get the frequencies for the first 0.01-second part, you need to multiply the sampled audio signal (in blue) with the window function (in green).



In this figure, to get the frequencies for the second 0.01-second part, you need to multiply the sampled audio signal (in blue) with the window function (in green).

By "windowing" the audio signal, you multiply your signal audio(t) by a window function window(t). This window function produces **spectral leakage**. Spectral leakage is the apparition of new frequencies that doesn't exist inside the audio signal. The power of the real frequencies is leaked to others frequencies.

Here is a non-formal (and very light) mathematical explanation. Let's assume you want a part of the full audio signal. You will multiply the audio signal with a window function that let pass the sound only for the part you want:

part_of_audio(t) = full_audio(t) . window (t)

When you try to get the frequencies of the part of audio, you apply the Fourier transform on the signal

Fourier(part_of_audio(t)) = Fourier(full_audio(t) . window (t))

According to the convolution theorem (* represents the convolution operator and . the multiplication operator)

Fourier(full_audio(t) . window (t)) = Fourier(full_audio(t)) * Fourier(window (t))

—>Fourier(part_of_audio(t)) = Fourier(full_audio(t)) * Fourier(window (t))

—>The frequencies of the part_of_audio(t) depend on the window() function used.

I won’t go deeper because it requires advanced mathematics. If you want to know more, look at this link on page 29, the chapter “the truncate effects” presents the mathematical effect of applying a rectangular window on a signal. What you need to keep in mind is that cutting an audio signal into small parts to analyze the frequencies of each part produces spectral leakage.

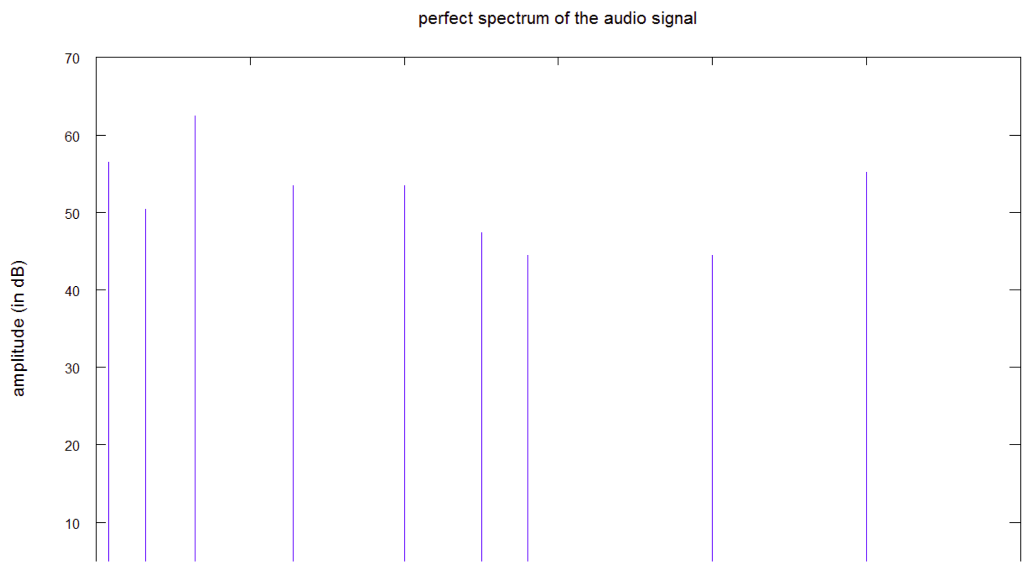
different types of windows

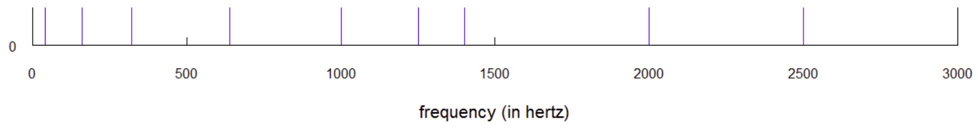
You can’t avoid spectral leakage but you can handle how the leakage will behave by choosing the right window function: instead of using a rectangular window function, you can choose a triangular widows, a Parzen window, a Blackman window, a Hamming window …

The rectangular window is the easiest window to use (because you just have to “cut” the audio signal into small parts) but for analyzing the most important frequencies in a signal, it might not be the best type of windows. Let’s have a look of 3 types of windows: rectangular, Hamming and Blackman. In order to analyse the effect of the 3 windows, we will use the following audio signal composed of:

- A frequency 40 Hz with an amplitude of 2
- A frequency 160 Hz with an amplitude of 0.5
- A frequency 320 Hz with an amplitude of 8
- A frequency 640Hz with an amplitude of 1
- A frequency 1000 Hz with an amplitude of 1
- A frequency 1225 Hz with an amplitude of0.25
- A frequency 1400 Hz with an amplitude of 0.125
- A frequency 2000 Hz with an amplitude of 0.125
- A frequency 2500Hz with an amplitude of 1.5

In a perfect world, the Fourier transform of this signal should give us the following spectrum:



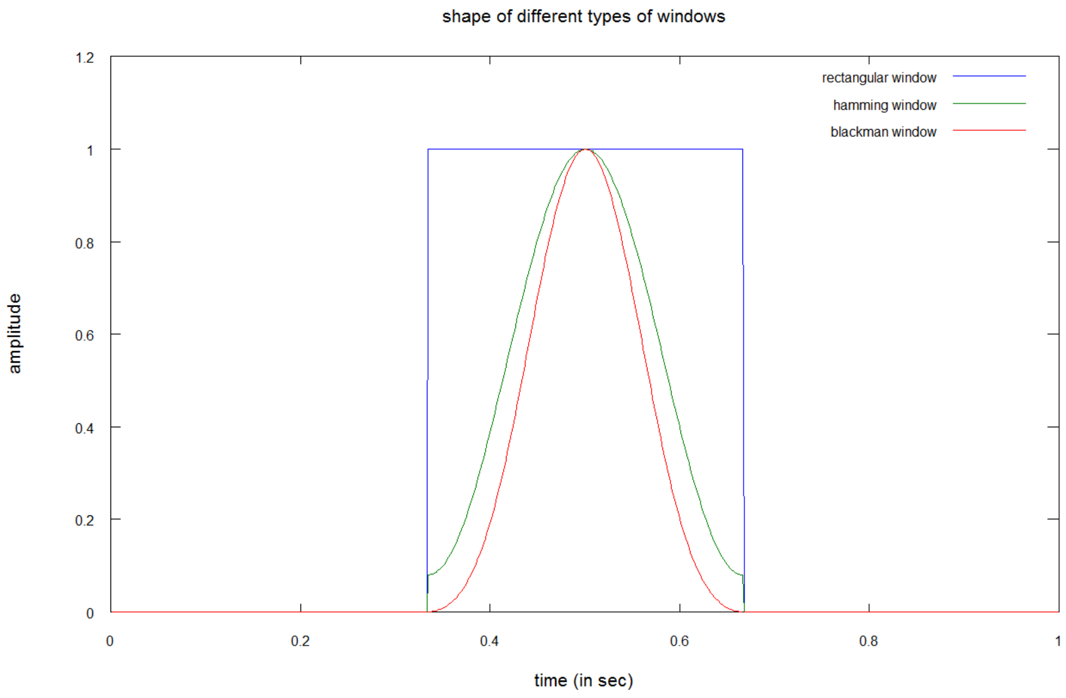


This figure shows a spectrum with only 9 vertical lines (at 40 Hz, 160 Hz, 320 Hz, 640 Hz, 1000 Hz, 1225 Hz, 1400 Hz, 2000 Hz and 2500 Hz. The y axis gives the amplitude in decibels (dB) which means the scale is logarithmic. With this scale a sound at 60 dB is 100 times more powerful than a sound at 40 dB and 10000 times more powerful than a sound at 20 dB. To give you an idea, when you speak in a quiet room, the sound you produce is 20-30 dB higher (at 1 m of you) than the sound of the room.

In order to plot this "perfect" spectrum, I applied the Fourier Transform with a very long window: a 10-second window. Using a very long window reduces the spectrum leakage but 10 seconds is too long because in a real song the sound changes much faster. To give you an idea of how fast the music changes:

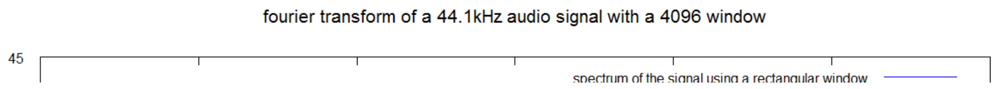
- here is a video with 1 change (or beat) per second, it sounds slow but it's a common rhythm for classical music.
- here is a video with 2.7 changes per second, it sounds much faster but this rhythm is common for electro music
- here is a video with 8.3 changes per second, it's a very (very) fast rhythm but possible for small parts of songs.

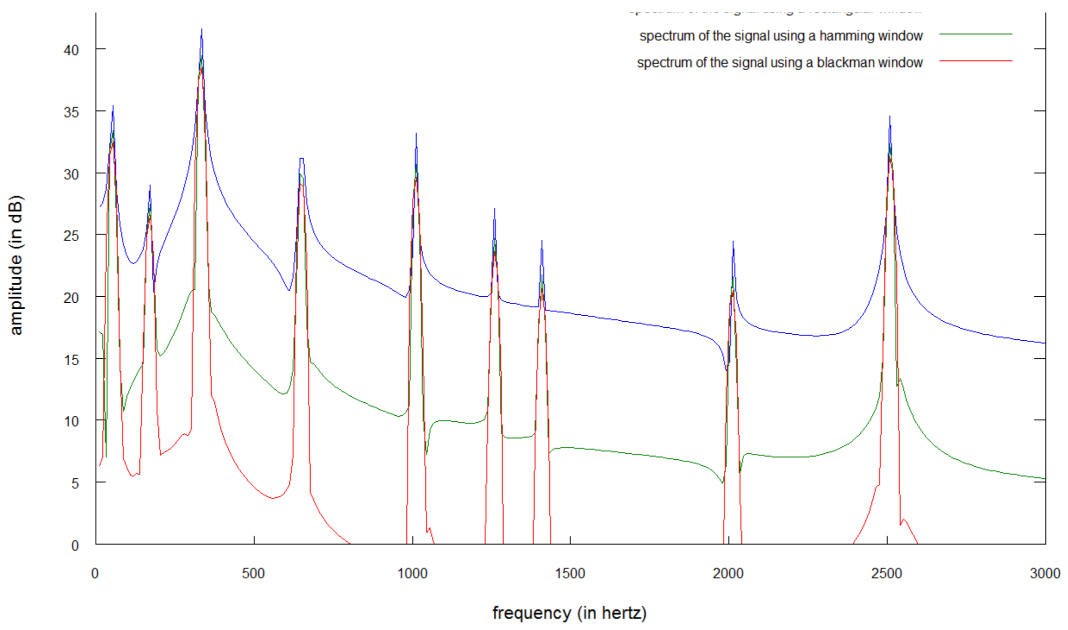
In order to capture those fast changes, you need to "cut" the sound into very small parts using window functions. Imagine you want to analyze the frequencies of a sound every 1/3 second.



In this figure, you can multiply the audio signal with one of the 3 window types to get the part of the signal between 0.333sec and 0.666 sec. As I said, using a rectangular window is like cutting the signal between 0.333sec and 0.666sec whereas with the Hamming or the Blackman windows you need to multiply the signal with the window signal.

Now, here is the spectrum of the previous audio signal with a 4096-sample window:

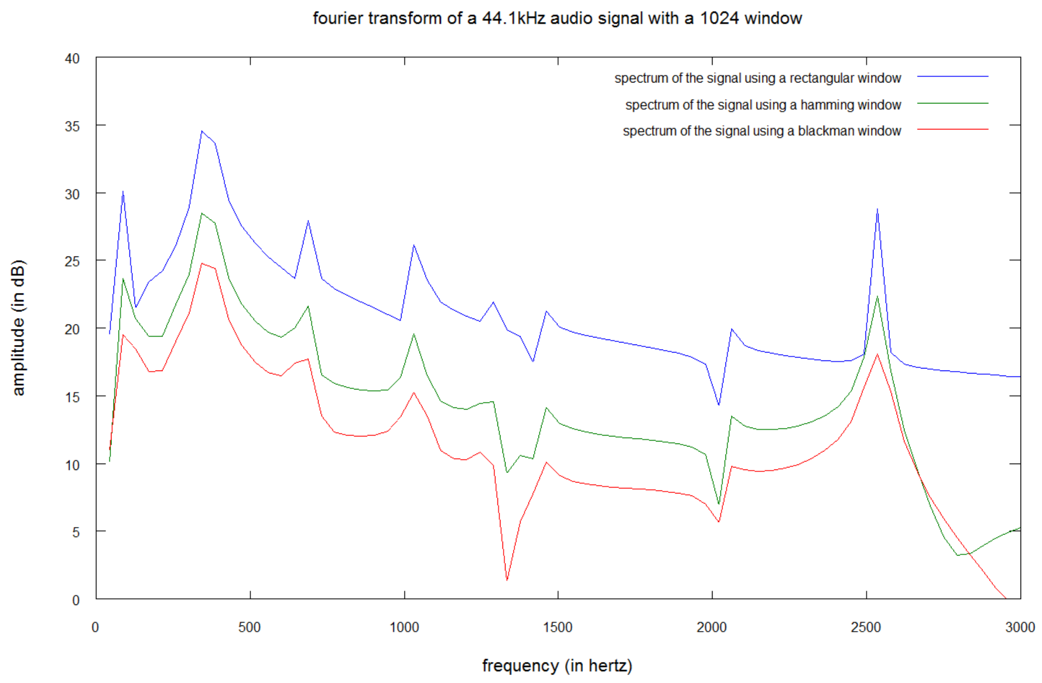




The signal is sampled at 44100Hz so a 4096-sample window represents a 93-millisecond part (4096/44100) and a frequency resolution of 10.7 Hz.

This figure shows that all windows modify the real spectrum of the sound. We clearly see that a part of **the power of the real frequencies is spread to their neighbours**. The spectrum from the rectangular window is the worst since the spectrum leakage is much higher than the 2 others. It's especially true between 40 and 160 Hz. The Blackman window gives the closest spectrum from the real spectrum.

Here is the same example with a Fourier Transform of a 1024 window:



The signal is sampled at 44100Hz so a 1024-sample window represents a 23-millisecond part (1024/44100) and a frequency resolution of 43 Hz.

This time the rectangular window gives the best spectrum. With the 3 windows the 160 Hz frequency is hidden by the spectrum leakage produced by the 40 Hz and 320 Hz frequencies. The Blackman window gives the worst result with a 1225 Hz frequency close to invisible.

Comparing both figures shows that the spectrum leakage increases (for all the window function) as the frequency resolution increases. The fingerprint algorithm used by Shazam look for the loudest frequencies inside an audio track. Because of spectrum leakage, we can't just take the X highest frequencies. In the last example, the 3 loudest frequencies are approximately 320 Hz, 277 Hz (320-43) and 363 Hz (320+43) whereas only the 320 Hz frequency exists.

Which window is the best?

There are no "best" or "worst" windows. Each window has its specificities and depending on the problem you might want to use a certain type.

A rectangular window has excellent resolution characteristics for sinusoids of comparable strength, but it is a poor choice for sinusoids of disparate amplitudes (which is the case inside a song because the musical notes don't have the same loudness).

Windows like Blackman are better to prevent from the case where spectrum leakage of strong frequencies hides weak frequencies. But, these windows deal badly with noise since a noise will hide more frequencies than rectangular window. This is problematic for an algorithm like Shazam that needs to handle noise (for instance when you Shazam a music in a bar or outdoor there are a lot of noise).

A Hamming window is between these two extremes and is (in my opinion) a better choice for an algorithm like shazam.

Here are some useful links to go deeper on window functions and spectrum leakage:

http://en.wikipedia.org/wiki/Spectral_leakage

http://en.wikipedia.org/wiki/Window_function

http://web.mit.edu/xiphmont/Public/windows.pdf

Fast Fourier Transform and time complexity

the problem

$$X(n) = \sum_{k=0}^{N-1} x[k] e^{-j(2\pi kn/N)}$$

If you look again at the DFT formula (don't worry, it's the last time you see it), you can see that to compute one bin you need to do N additions and N multiplications (where N is the size of the window). Getting the N bins requires 2 *N^2 operations which is a lot.

For example, let's assume you have a three-minute song at 44,1 kHz and you compute the spectrogram of the song with a 4096-sample window. You'll have to compute 10.7 (44100/4096)

DFT per second so 1938 DFTs for the full song. Each DFT needs 3.35*10^7 operations (2* 4096^2). To get the spectrogram of the song you need to do 6,5*10^10 operations.

Let's assume you have a music collection of 1000 three-minutes-long songs, you'll need 6,5*10^13 operations to get the spectrograms of your songs. Even with a good processor, **it would take days/months to get the result**.

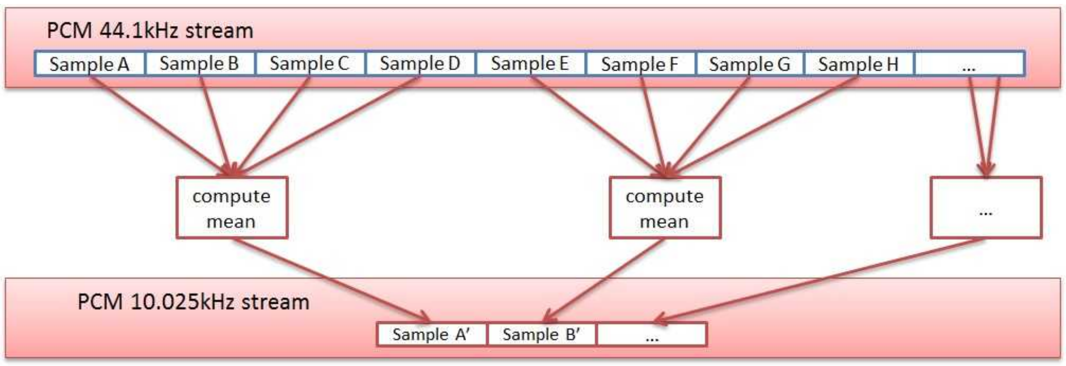
Thankfully, there are faster implementations of the DFT called FFT (Fast Fourier Transforms). Some implementations require just 1.5*N * log(N) operations. For the same music collection, **using the FFT instead of the DFT** requires 340 times less additions (1.43*10^11) and **it would take minutes/hours to get the result**.

This example shows another tradeoff: though increasing the size of the window improves the frequency resolution, it also increases the computation time. For the same music collection, if you compute the spectrogram using a 512 sample window (frequency resolution of 86 Hz), you get the result with the FFT in 1.07*10^11 operations, approximately 1/4 time faster than with a 4096 sample window (frequency resolution of 10.77 Hz).

This time complexity is important since when you shazam a sound, your phone needs to compute the spectrogram of the recorded audio and a mobile processor is less powerful than a desktop processor.

downsampling

Thankfully, there is a trick to keep the frequency resolution and reduce the window size at the same time, it's called downsampling. Let's take a standard song at 44100 Hz, if you resample it at 11025 Hz (44100/4) you will get the same frequency resolution whether you do a FFT on the 44.1kHz song with a 4096 window or you do a FFT on the 11kHz resampled song with a 1024 window. The only difference is that the resampled song will only have frequencies from 0 to 5 kHz. But the most important part of a song is between 0 and 5kHz. In fact most of you won't hear a big difference between a music at 11kHz and a music at 44.1kHz. So, the most important frequencies are still in the resampled song which is what matters for an algorithm like Shazam.



Downsampling a 44.1 kHz song to a 11.025 kHz one is not very difficult: A simple way to do it is to take the samples by group of 4 and to transform this group into just one sample by taking the average of the 4 samples. The only tricky part is that before downsampling a signal, you need to filter the higher frequencies in the sound to avoid aliasing (remember the Nyquist-Shannon theorem). This can be done by using a digital low pass filter.

FFT

But let’s go back to the FFT. The simplest implementation of the FFT is the radix 2 Cooley–Tukey algorithm which is a divide a conquer algorithm. The idea is that instead of directly computing the Fourier Transform on the N-sample window, the algorithm:

- divides the N-sample window into 2 N/2-sample windows
- computes (recursively) the FFT for the 2 N/2-sample windows
- computes efficiently the FFT for the N-sample windows from the 2 previous FFT

The last part only costs N operations using a mathematical trick on the roots of unity (the exponential terms).

Here is a readable version of the FFT (written in python) that I found on Wikipedia

For more information on the FFT, you can check this article on Wikipedia.

Shazam

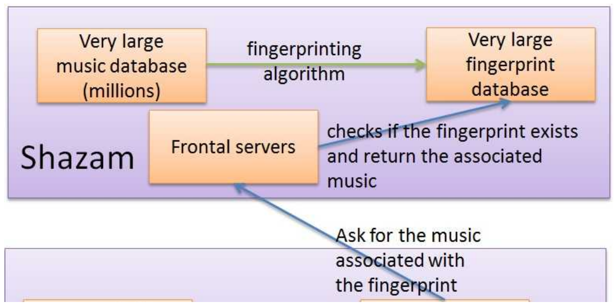
We’ve seen a lot of stuff during the previous parts. Now, we’ll put everything together to explain how Shazam quickly identifies songs (at last!). I’ll first give you a global overview of Shazam, then I’ll focus on the generation of the fingerprints and I’ll finish with the efficient audio search mechanism.

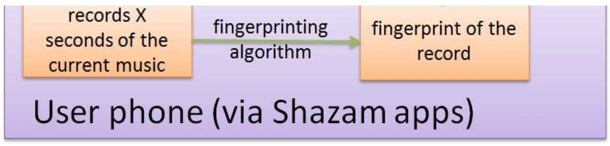
Note: From now on, I assume that you read the parts on musical notes, FFT and window functions. I’ll sometimes use the words “frequency”, “bin”, ”note” or the full expression “bin of frequencies” but it’s the same concept since we’re dealing with digital audio signals.

Global overview

An **audio fingerprint** is a digital summary that can be used to identify an audio sample or quickly locate similar items in an audio database. For example, when you’re humming a song to someone, you’re creating a fingerprint because you’re extracting from the music what you think is essential (and if you’re a good singer, the person will recognize the song).

Before going deeper, here is a simplified architecture of what Shazam might be. I don’t work at Shazam so it’s only a guess (from the 2003 paper of the co-founder of Shazam):


Very large music database (millions)
fingerprinting algorithm
Very large fingerprint database
Shazam
Frontal servers
checks if the fingerprint exists and return the associated music
Ask for the music associated with the fingerprint


records X seconds of the current music
fingerprinting algorithm
fingerprint of the record
User phone (via Shazam apps)

On the server side:

- Shazam precomputes fingerprints from a very big database of music tracks.
- All those fingerprints are put in a fingerprint database which is updated whenever a new song is added in the song database

On the client side:

- when a user uses the Shazam app, the app first records the current music with the phone microphone
- the phone applies the same fingerprinting algorithm as Shazam on the record
- the phone sends the fingerprint to Shazam
- Shazam checks if this fingerprint matches with one of its fingerprints
  - If no it informs the user that the music can't be found
  - If yes, it looks for the metadata associated with the fingerprints (name of the song, ITunes url, Amazon url …) and gives it back to the user.

The key points of Shazam are:

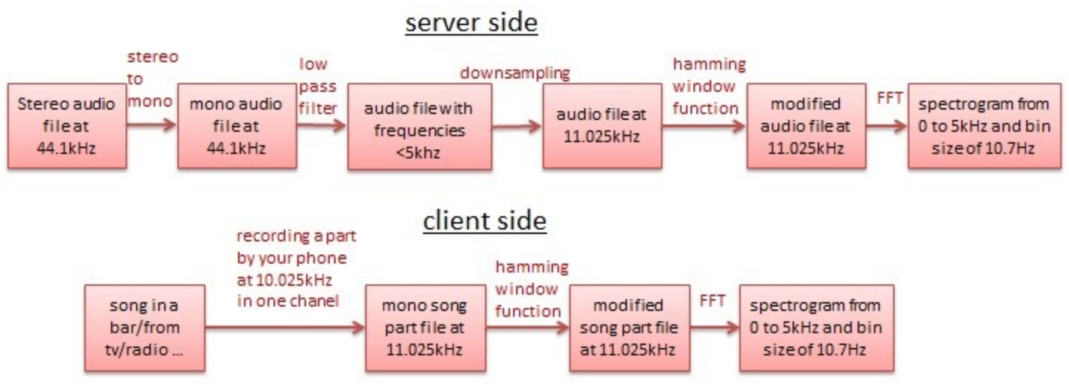
- being Noise/Fault tolerant:
  - because the music recorded by a phone in a bar/outdoor has a bad quality,
  - because of the artifact due to window functions,
  - because of the cheap microphone inside a phone that produces noise/distortion
  - because of many physical stuff I'm not aware of
- fingerprints needs to be time invariant: the fingerprint of a full song must be able to match with just a 10-second record of the song
- fingerprint matching need to be fast: who wants to wait minutes/hours to get an answer from Shazam?
- having few false positives: who wants to get an answer that doesn't correspond to the right song?

Spectrogram filtering

Audio fingerprints differ from standard computer fingerprints like SSHA or MD5 because two different files (in terms of bits) that contain the same music must have the same audio fingerprint. For example a song in a 256kbit ACC format (ITunes) must give the same fingerprint as the same song in a 256kbit MP3 format (Amazon) or in a 128kbit WMA format (Microsoft). To solve this problem, **audio fingerprinting algorithms uses the spectrogram** of audio signals to extract fingerprints.

Getting our spectrogram

I told you before that to get the spectrogram of a digital sound you need to apply a FFT. For a fingerprinting algorithm we need a good frequency resolution (like 10.7Hz) to reduce spectrum leakage and have a good idea of the most important notes played inside the song. At the same time, we need to reduce the computation time as far as possible and therefore use the lowest possible window size. In the research paper from Shazam, they don't explain how they get the spectrogram but here is a possible solution:


server side
Stereo audio file at 44.1kHz
stereo to mono
mono audio file at 44.1kHz
low pass filter
audio file with frequencies <5khz
downsampling
audio file at 11.025kHz
hamming window function
modified audio file at 11.025kHz
FFT
spectrogram from 0 to 5kHz and bin size of 10.7Hz
client side
song in a bar/from tv/radio ...
recording a part by your phone at 10.025kHz in one chanel
mono song part file at 11.025kHz
hamming window function
modified song part file at 11.025kHz
FFT
spectrogram from 0 to 5kHz and bin size of 10.7Hz

On the server side (Shazam), the 44.1khz sampled sound (from CD, MP3 or whatever sound format) needs to pass from stereo to mono. We can do that by taking the average of the left speaker and the right one. Before downsampling, we need to filter the frequencies above 5kHz to avoid aliasing. Then, the sound can be downsampled at 11.025kHz.

On the client side (phone), the sampling rate of the microphone that records the sound needs to be at 11.025 kHz.

Then, in both cases we need to apply a window function to the signal (like a hamming 1024-sample window, read the chapter on window function to see why) and apply the FFT for every 1024 samples. By doing so, each FFT analyses 0.1 second of music. This gives us a spectrogram:

- from 0 Hz to 5000Hz
- with a bin size of 10.7Hz,
- 512 possible frequencies
- and a unit of time of 0.1 second.

Filtering

At this stage we have the spectrogram of the song. Since Shazam needs to be noise tolerant, **only the loudest notes are kept**. But you can't just keep the X more powerful frequencies every 0.1 second. Here are some reasons:

- In the beginning of the article I spoke about psychoacoustic models. Human ears have more difficulties to hear a low sound (<500Hz) than a mid-sound (500Hz-2000Hz) or a high sound (>2000Hz). As a result low sounds of many "raw" songs are artificially increased before being released. If you only take the most powerful frequencies you'll end up with only the low ones and If 2 songs have the same drum partition, they might have a very close filtered spectrogram whereas there are flutes in the first song and guitars in the second.
- We saw on the chapter on window functions that if you have a very powerful frequency other powerful frequencies close to this one will appeared on the spectrum whereas they doesn't exist (because of spectrum leakage). You must be able to only take the real one.

Here is a simple way to keep only strong frequencies while reducing the previous problems:

step1 – For each FFT result, you put the 512 bins you inside 6 logarithmic bands:

- the very low sound band (from bin 0 to 10)
- the low sound band (from bin 10 to 20)
- the low-mid sound band (from bin 20 to 40)
- the mid sound band (from bin 40 to 80)
- the mid-high sound band (from bin 80 to 160)
- the high sound band (from bin 160 to 511)

step2 – For each band you keep the strongest bin of frequencies.

step3 – You then compute the average value of these 6 powerful bins.

step4 – You keep the bins (from the 6 ones) that are above this mean (multiplied by a coefficient).

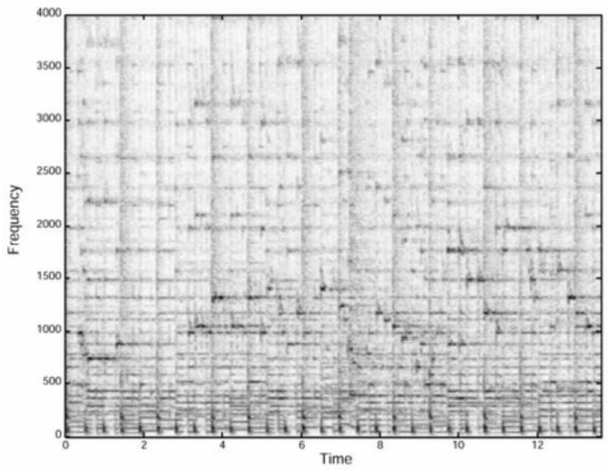
The step4 is very important because you might have:

- an a cappella music involving soprano singers with only mid or mid-high frequencies
- a jazz/rap music with only low and low-mid frequencies
- ..

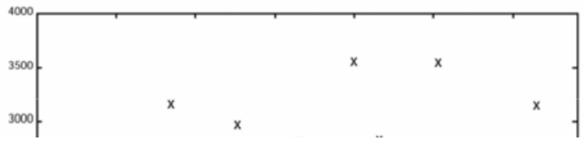
And you don’t want to keep a weak frequency in a band just because this frequency is the strongest of its band.

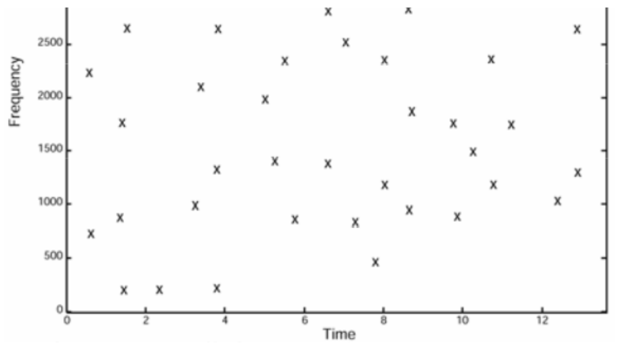
But this algorithm has a limitation. In most songs some parts are very weak (like the beginning or the end of a song). If you analyze these parts you’ll end up with false strong frequencies because the mean value (computed at step 3) of these parts is very low. To avoid that, instead of taking the mean of the 6 powerful beans of the current FFT (that represents only 0.1sec of the song) you could take the mean of the most powerful bins of the full song.

To summarize, by applying this algorithm we’re filtering the spectrogram of the song to keep the peaks of energy in the spectrum that represent the loudest notes. To give you a visual idea of what this filtering is, here is a real spectrogram of a 14-second song.



This figure is from the Shazam research article. In this spectrogram, you can see that some frequencies are more powerful than others. If you apply the previous algorithm on the spectrogram here is what you’ll get:





This figure (still from the Shazam research article) is a filtered spectrogram. Only the strongest frequencies from the previous figure are kept. Some parts of the song have no frequency (for example between 4 and 4.5 seconds).

The number of frequencies in the filtered spectrogram depends on the coefficient used with the mean during step4. It also depends on the number of bands you use (we used 6 bands but we could have used another number).

At this stage, the intensity of the frequencies is useless. Therefore, this spectrogram can modeled as a 2-column table where

- the first column represents the frequency inside the spectrogram (the Y axis)
- the second column represents the time when the frequency occurred during the song (the X axis)

This filtered spectrogram is not the final fingerprint but it's a huge part of it. Read the next chapter to know more.

Note: I gave you a simple algorithm to filter the spectrogram. A better approach could be to use a logarithmic sliding window and to keep only the most powerful frequencies above the mean + the standard deviation (multiplied by a coefficient) of a moving part of the song. I used this approach when I did my own Shazam prototype but it's more difficult to explain (and I'm not even sure that what I did was correct …).

Storing Fingerprints

We've just ended up with a filtered spectrogram of a song. How can we store and use it in an efficient way? This part is where the power of Shazam lies. To understand the problem, I'll present a simple approach where I search for a song by using directly the filtered spectrograms.

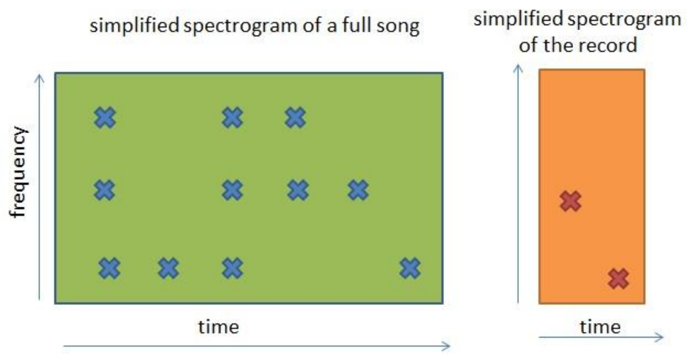
Simple search approach

Pre-step: I precompute a database of filtered spectrograms for all the songs in my computer

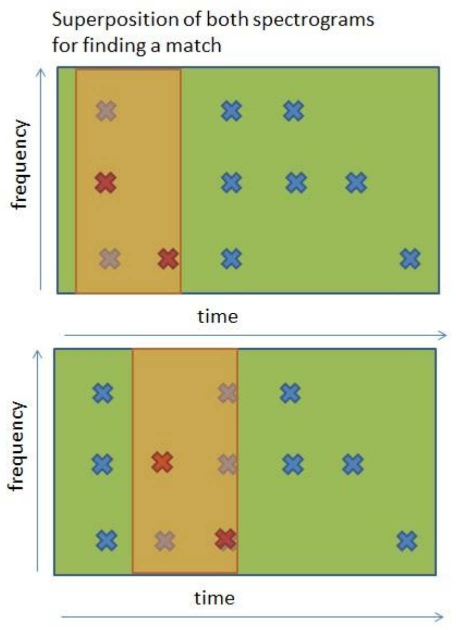
Step 1: I record a 10-second part of a song from TV in my computer

Step 2: I compute the filtered spectrogram of this record

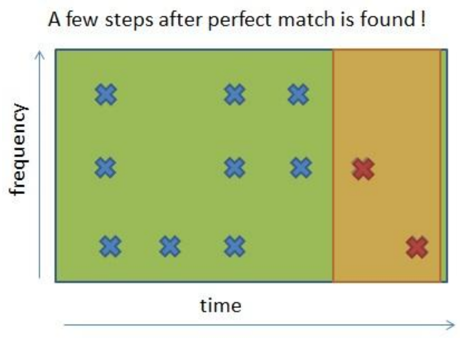
Step 3: I compare this “small” spectrogram with the “full” spectrogram of each songs. How can I compare a 10-second spectrogram with a spectrogram of a 180-second song? Instead of losing myself in a bad explanation, here is a visual explanation of what I need to do.


simplified spectrogram of a full song
simplified spectrogram of the record
frequency
time
time

Visually speaking, I need to superpose the small spectrogram everywhere inside the spectrogram of the full song to check if the small spectrogram matches with a part of the full one.



And I need to do this for each song until I find a perfect match.



In this example, there is a perfect match between the record and the end of the song. If it’s not the case, I have to compare the record with another song and so on until I find a perfect match. If I don’t find a perfect match I can choose the closest match I found (in all the songs) if the matching rate is above a threshold. For instance, if the best match I found gives me a 90% similarity between the record and a part of a song, I can assume it’s the right song because the 10% of none similarity are certainly due to external noise.

Though it works well, this simple approach requires a lot of computation time. It needs to compute all the possibilities of matching between the 10-second record and each song in the collection. Let's assume on average music contains 3 peak frequencies per 0.1 seconds. Therefore, the filtered spectrogram of the 10-second record has 300 time-frequency points. In the worst case scenario, you'll need 300 * 300 * 30* S operations to find the right song where S is the number of second of music in your collection. If like me you have 30k songs (7 * 10^6 seconds of music) it might take a long time and it's harder for Shazam with its 40 million songs collection (it's a guess I couldn't find the current size of Shazam).

So, how Shazam does it efficiently?

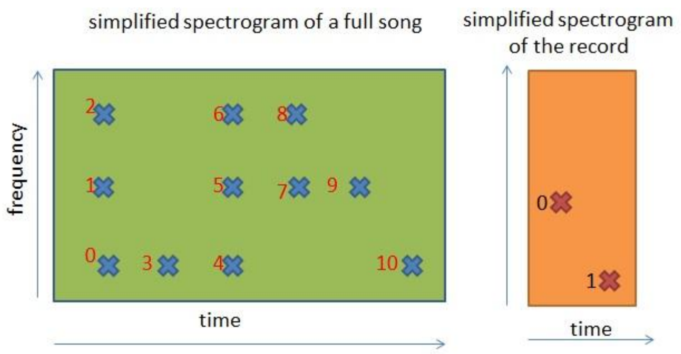
Target zones

Instead of comparing each point one by one, the idea is to look for multiple points at the same time. In the Shazam paper, this group of point is called a **target zone.** The paper from Shazam doesn't explain how to generate these target zones but here is a possibility. For the sake of comprehension I'll fix the size of the target zone at 5 frequency-time points.

In order to be sure that both the record and the full song will generate the same target zones, you need an order relation between the time-frequency points in a filtered spectrogram. Here is one:

- If two time-frequency points have the same time, the time-frequency point with the lowest frequency is before the other one.
- If a time time-frequency point has a lower time than another point one then it is before.

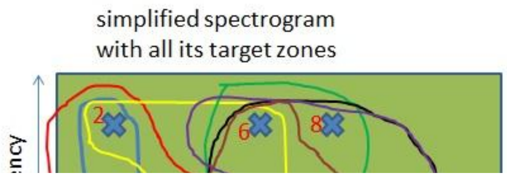
Here is what you get if you apply this order on the simplified spectrogram we saw before:

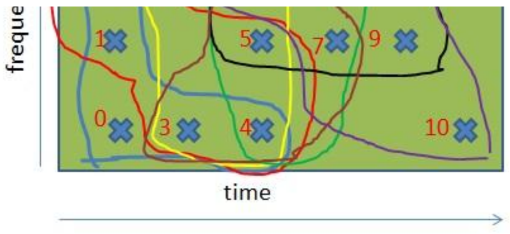

simplified spectrogram of a full song
simplified spectrogram of the record
frequency
time
time
0
1
2
3
4
5
6
7
8
9
10
0
1

In this figure I labeled all the time-frequency points using this order relation. For example:

- The point 0 is before any other points in the spectrogram.
- The point 2 is after point 0 and 1 but before all the others.

Now that the spectrograms can be inner-ordered, we can create the same target zones on different spectrogram with the following rule: "To generate target zones in a spectrogram, you need for each time-frequency point to create a group composed of this point and the 4 points after it". We'll end up with approximately the same amount of target zones as the number of points. This generation is the same for the songs or the record



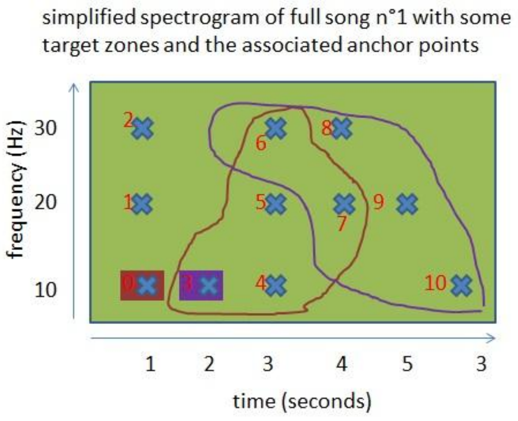


In this simplified spectrogram, you can see the different target zones generated by the previous algorithm. Since the target size is 5, most of the points belong to 5 target zones (except the points at the beginning and the end of the spectrogram).

Note: I didn't understand at first why for the record we needed to compute that much target zones. We could generate target zones with a rule like "for each point whose label is a multiple of 5 you need to create a group composed of this frequency and the 4 frequencies after it". With this rule, the number of target zones would be reduced by 5 and so the search time (explained in the next part). The only reason I found is that computing all the possible zones on both the record and the song increases a lot the noise robustness.

Address generation

We now have multiple target zones, what do we do next? We create for each point an **address** based on those target zones. In order to create those addresses, we also need an **anchor point** per target zone. Again, the paper doesn't explain how to do it. I propose this anchor point to be the 3rd point before the target zone. The anchor can be anywhere as long as the way it is generated is reproducible (which it is thanks to our order relation).


simplified spectrogram of full song n°1 with some target zones and the associated anchor points
frequency (Hz)
30
20
10
time (seconds)
1
2
3
4
5
3
0
1
2
3
4
5
6
7
8
9
10

In this picture I plotted 2 target zones with their anchor points. Let's focus on the purple target zone. The address formula proposed by shazam is following one:

["frequency of the anchor";" frequency of the point";"delta time between the anchor and the point"].

For the purple target zone:

- the address of point 6 is ["frequency of 3";" frequency of point 6";"delta_time between point 3 & point 6"] so concretely [10;30;1],
- the address of point 7 is [10;20;2].

Both points appeared also in the brown target zone, their addresses with this target zone are [10;30;2] for point 6 and [10;20;3] for point 7.

I spoke about addresses, right? That means that those addresses are linked to something. In the case of the full songs (so only in the server side), those addresses are linked to the following couple [“absolute time of the anchor in the song”;”Id of the song”]. In our simple example with the 2 previous points we have the following result:

[10;30;1] –>[2;1]

[10;30;2]–>[2;1]

[10;30;2] –>[1;1]

[10;30;3] –>[1;1]

If you apply the same logic for all the points of all the target zones of all the song spectrograms, you’ll end up with a very big table with 2 columns:

- the addresses
- the couples (“time of anchor” ; “song Id”).

This table is the fingerprint database of Shazam. If on average a song contains 30 peak frequencies per second and the size of the target zone is 5, the size of this table is 5 * 30 *S where S is the number of seconds of the music collection.

If you remember, we used an FFT with 1024 samples which means that there are only 512 possible frequency values. Those frequencies can be coded in 9 bits (2^9 = 512). Assuming that the delta time is in milliseconds, it will never be over 16 seconds because it would imply a song with a 16-second part without music (or very low sound). So, the delta time can be coded in 14 bits (2^14 = 16384). **The address can be coded in a 32-bit integer**:

- 9 bits for the “frequency of the anchor”
- 9 bits for the ” frequency of the point”
- 14 bits for the ”delta time between the anchor and the point”

Using the same logic, **the couple (“time of anchor” ; “song Id”) can be coded in a 64-bit integer** (32 bit for each part).

The fingerprint table can be implemented as a simple array of list of 64-bit integers where:

- the index of the array is the 32-bit integer address
- the list of 64-bits integers is all the couples for this address .

In other words, we transformed the **fingerprint table** into an **inverted look-up** that allows search operation in O(1) (ie. very effective search time).

Note: You may have noticed that I didn’t choose the anchor point inside the target zone (I could have chosen the first point of the target Zone for example). If I did it would have generated a lot of addresses like [frequency anchor;frequency anchor;0] and therefore too many couples(“time of anchor” ; “song Id”) would have an address like [Y,Y,0] where Y is the frequency (between 0 and 511). In other words, the look-up would have been skewed.

Searching And Scoring the fingerprints

We now have a great data structure on the server side, how can we use it? It's my last question, I promise!

Search

To perform a search, the fingerprinting step is performed on the recorded sound file to generate an address/value structure slightly different on the value side:

["frequency of the anchor";" frequency of the point";"delta time between the anchor and the point"] -> ["absolute time of the anchor in the record"].

This data is then sent to the server side (Shazam). Let's take the same assumption than before (300 time-frequency points in the filtered spectrogram of the 10-second record and the size of the target zone of 5 points), it means there are approximately 1500 data sent to Shazam.

Each address from the record is used to search in the fingerprint database for the associated couples ["absolute time of the anchor in the song";"Id of the song"]. In terms of time complexity, assuming that the fingerprint database is in-memory, the cost is the search is proportional to the number of address sent to Shazam (1500 in our case). This search returns a big amount of couples, let's say for the rest of the article it returns M couples.

Though M is huge, it's way lower than the number of notes (time-frequency points) of all the songs. **The real power of this search is that instead of looking if a one note exists in a song, we're looking if 2 notes separated from delta_time seconds exist in the song**. At the end of this part we'll talk more about time complexity.

Result filtering

Though it is not mentioned in the Shazam paper, I think the next thing to do is to filter the M results of the search by keeping only the couples of the songs that have a minimum number of target zones in common with the record.

For example, let's suppose our search has returned:

- 100 couples from song 1 which has 0 target zone in common with the record
- 10 couples from song 2 which has 0 target zone in common with the record
- 50 couples from song 5 which has 0 target zone in common with the record
- 70 couples from song 8 which has 0 target zone in common with the record
- 83 couples from song 10 which has 30 target zones in common with the record
- 210 couples from song 17 which has 100 target zones in common with the record
- 4400 couples from song 13 which has 280 target zones in common with the record
- 3500 couples from song 25 which has 400 target zones in common with the record

Our 10-second record has (approximately) 300 targets zone. In the best case scenario:

- song 1 and the record will have a 0% matching ratio
- song 2 and the record will have a 0% matching ratio
- song 5 and the record will have a 0% matching ratio
- song 8 and the record will have a 0% matching ratio
- song 10 and the record will have a 10% matching ratio
- song 17 and the record will have a 33% matching ratio
- song 13 and the record will have a 91.7% matching ratio
- song 25 and the record will have a 100% matching ratio

We'll only keep the couples of song 13 and 25 from the result. Although songs 1 2,5 and 8 have multiples couples in common with the record, none of them form at least a target zone (of 5 points) in common with the record. This step can filter a lot of false results because the fingerprint database of Shazam has a lot of couples for the same address and you can easily end up with couples at the same address that don't belong to the same target zone. If you don't understand why, look the last picture of the previous part: the [10;30;2] address is used by 2 time-frequency points that doesn't belong to the same target zone. If the record also have an [10;30;2], (at least) one of the 2 couples in the result will be filtered in this step.

This step can be done in O(M) with the help of a hash table whose key is the couple (songID;absolute time of the anchor in the song) and value the number of time it appears in the result:

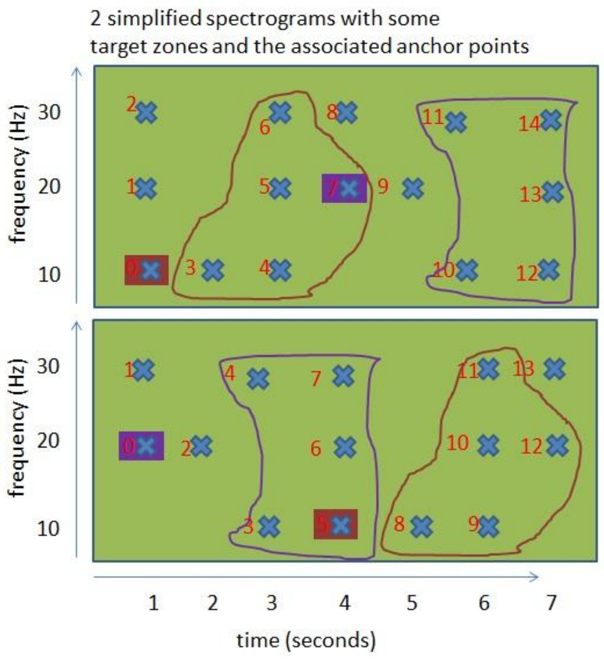
- We iterate through the M results and count (in the hash table) the number of time a couple is present
- We remove all the couples (i.e. the key of the hash table) that appear less than 4 times (in other words we remove all the points that doesn't form a target zone)*
- We count the number X of times the song ID is part of a key in the hash table (i.e we count the number of complete target zones in the song. Since the couple come from the search, those target zones are also in the record)
- We only keep the result whose song number is above 300*coeff (300 is the number of target zone of the record and we reduce this number with a coeff because of the noise).
- We put the remaining results in a new hash table whose index is the songId (this hashmap will be useful for the next step)

* The idea is to look for the target zone created by an anchor point in a song. This anchor point can be defined by the id of the song it belongs and the absolute time it occurs. I made an approximation because in a song, you can have multiple anchor points at the same time. Since we're dealing with a filtered spectrogram, you won't have a lot of anchor points at the same time. But the key [songID;absolute time of the anchor in the song] will gather all the target zones created by these target points.

Note: I used 2 hash tables in this algorithm. If you don't know how it works, just see it as a very efficient way to store and get data. If you wan't to know more, you can read my article on the HashMap in Java which is just an efficient hash table.

<u>Time coherency</u>

At this stage we only have songs that are really close to the record. But we still need to verify the time coherency between the notes of the record and these songs. Let's see why:


2 simplified spectrograms with some
target zones and the associated anchor points
frequency (Hz)
30
20
10
time (seconds)
1
2
3
4
5
6
7

In this figure, we have 2 target zones that belong to 2 different songs. If we didn't look for time coherency, those target zones would increase the matching score between the 2 songs whereas they don't sound alike since the notes in those target zones are not played in the same order.

This last step is about time ordering. The idea is:

- to compute for each remaining song the notes and their absolute time position in the song.
- to do the same for the record, which gives us the notes and their absolute time position in the record.
- if the notes in the song and those in the record are time coherent, we should find a relation like this one: "absolute time of the note in the song = absolute time of the note in record + delta", were delta is the starting time of the part of the song that matches with the record.
- for each song, we need to find the delta that maximizes the number of notes that respect this time relation
- Then we choose the song that has the maximum number of time coherent notes with the record

Now that you get the idea let's see how to do it technically. At this stage we have for the record a list of address/value:

["frequency of the anchor";" frequency of the point";"delta time between the anchor and the point"] -> ["absolute time of the anchor in the record"].

And we have for each song a list address/value (stored in the hash table of the previous step):

[“frequency of the anchor”;” frequency of the point”;”delta time between the anchor and the point”] -> [“absolute time of the anchor in the song”;”Id of the song”].

The following process needs to be done for all the remaining songs:

- For each address in the record, we get the associated value of the song and we compute delta = “absolute time of the anchor in the record” – “absolute time of the anchor in the song” and put the delta in a “list of delta”.
- It is possible that the address in the record is associated with multiples values in the song (i.e. multiple points in different target zones of the song), in this case we compute the delta for each associated values and we put the deltas in the “list of delta”
- For each different value of delta in the “list of delta” we count its number of occurrence (in other words, we count for each delta the number of notes that respect the rule “absolute time of the note in the song = absolute time of the note in record + delta”)
- We keep the greatest value (which gives us the maximum number of notes that are time coherent between the record and the song)

From all the songs, we keep the song with the maximum time coherent notes. If this coherency is above “the number of note in the record” * “a coefficient” then this song is the right one.

We just have to look for the metadata of the song (“artist name”,”song name, “Itunes URL”,”Amazon URL”, …) with the Song ID and gives the result back to the user.

Let’s talk about complexity!

This search is really more complicated that the simple one we first saw, let’s see if this is worth it. The enhanced search is a step by step approach that reduces the complexity at each step.

For the sake of comprehension, I’ll recall all the assumptions (or choices) I made and make new ones to simplify the problem:

- We have 512 possible frequencies
- on average a song contains 30 peak frequencies per second
- Therefore the 10-sec record contains 300 time-frequency points
- S is the number of seconds of music off all the songs
- The size of the target zone is 5 notes
- (new) I assume that the delta time between a point and its anchor is ether 0 or 10 msec
- (new) I assume the generation of addresses is uniformly distributed, which means there is the same amount of couple for any address [X,Y,T] where X and Y are one of the 512 frequencies and T is either 0 or 10 msec

The first step, the search, only requires 5 * 300 unitary searches.

The size of the result M is the sum of the result of the 5 * 300 unitary searches

M =(5 * 300) *(S *30* 5 * 300) / (512 *512 * 2)

The second step, the result filtering can be done in M operations. At the end of this step there are N notes distributed in Z songs. Without a statistical analysis of the music collection, it’s impossible to get the value of N and Z. I feel N is really lower than M and Z represent only a few songs, even for a 40-million song database like Shazam.

The last step is the analysis of the time coherency of the Z songs. We'll assume that each song as approximately the same amount of notes: N/Z. In the worst case scenario (a record that comes from a song that contains only one note played continuously), the complexity of one analysis is (5*300) * (N/Z).

The cost of the Z songs is 5 * 300 * N.

Since N<<M, the real cost of this search is M = (300 * 300 * 30* S) * (5 * 5)/ (512 *512 * 2)

If you remember, the cost of the simple search was: 300 * 300 * 30* S.

This new search is 20 000 times faster

Note: The real complexity depends on distribution of frequencies inside the songs of the collection but this simple calculus gives us a good idea of the real one.

Improvements

The Shazam paper is from 2003 which means the associated research is even older. In 2003, 64-bits processors were released to the mainstream market. Instead of using one anchor point per target zone like the paper proposes (because of the limited size of a 32-bit integer), you could use 3 anchor points (for example the 3 points just before the target zone) and **store the address** of a point in the target zone **in a 64-bit integer**. **This would dramatically improve the search time.** Indeed, the search would be to find 4 notes in a song separated from detla_time1, detla_time2 and detla_time3 seconds which means the number of results M would be very (very) lower than the one we just computed.

A great advantage of this fingerprint search is its **high scalability**:

- **Instead of having 1 fingerprint database you can have D databases**, each of them containing 1/D of the full song collection
- You can search at the same time for the closest song of the record in the D databases
- Then you choose the closest song from the D songs
- **The whole process is D times faster**.

Tradeoffs

Another good discussion is the noise robustness of this algorithm. I could easily add 2k words just for this subject but after 11k words I think it's better not to speak about it … or just a few words.

If you read carefully, you noticed that I used a lot of thresholds, coefficients and fixed values (like the sampling rate, the duration of a record, …). I also chose/made many algorithms (to filter a spectrogram, to generate a spectrogram,…). They all have an impact on the noise resistance and the time complexity. The real challenge is to find the right values and algorithms that maximize:

- The noises resistance
- The time complexity
- The precision (reducing the number of false positive results)

Conclusion

I hope you now understand how Shazam works. It took me a lot of time to understand the different subjects of this article and I still don't master them. This article won't make you an expert but I hope you have a very good picture of the processes behind Shazam. Keep in mind that Shazam is just one possible audio fingerprinting implementation.

You should be able to code your own Shazam. You can look at this very good article that focuses more on how to code a simplified Shazam in Java than the concepts behind it. The same author made a presentation at a Java conference and the slides are available here. You can also check this link for a MatLab/Octave implementation of Shazam. And of course, you can read by yourself the paper from Shazam co-founder Avery Li-Chun Wang by clicking right here.

The world of music computing is a very interesting field with touchy algorithms that you use every day without knowing it. Though Shazam is not easy to understand it's easier than:

- query by humming: for example SoundHound ,a concurrent of Shazam, allows you to hum/sing the song you're looking for
- speech recognition and speech synthesis: implemented by Skype, Apple "Siri" and Android "Ok Google"
- music similarity: which is the ability to find that 2 song are similar. It's used by Echonest a start-up recently acquired by Spotify
- …

If you're interested, there is an annual contest between researchers on those topics and the algorithms of each participant are available. Here is the link to the MIREX contest.

I spent approximately 200 hours during the last 3 years to understand the signal processing concepts, the mathematics behind them, to make my own Shazam prototype, to fully understand Wang's paper and imagine the processes the paper doesn't explain. I wrote this article because I have never found an article that really explains Shazam and I wished I could have found one when I began this side project in 2012. I hope I didn't write too many technical mistakes. The only thing I'm sure of is that despite my efforts there are many grammar and spelling mistakes (alas!). Tell me what you think of this article on the comments.