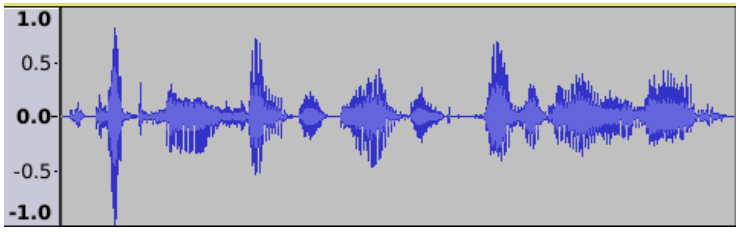
joachim heintz
workshop at TIEMF 2019
21 and 24 july

FFT in Csound

I. Basics

What is time-domain?

Sound is the change of amplitudes in time. No time, no sound. So when record a sound, it is in time. And when we draw the result, it is also in time. This is how the sentence "The quick brown fox jumps over the lazy dog" looks like:

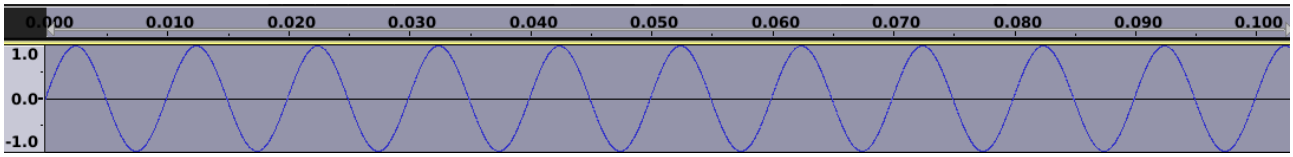


"The quick brown fox ..." in an audio editor (Audacity)

As convention, **time** is written horizontally (left to right), and the **amplitudes** are written vertically: 0 in the middle, +1 and -1 on top and bottom.
This view onto sound is called **time-domain**.

What is frequency-domain?

Let us create a sine tone with 100 Hz, and see how it looks like in time-domain:



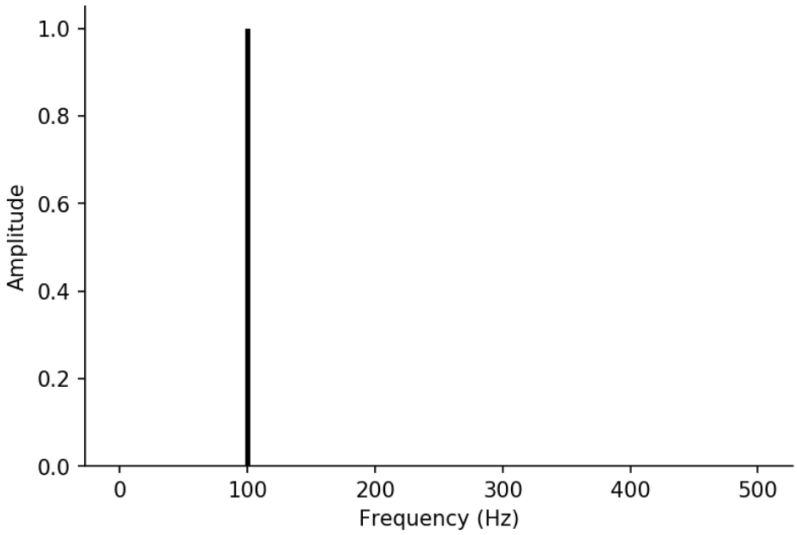
Sine wave with 100 Hz and Amplitude 1 in time-domain

Each period of the sine wave needs 1/100 of a second; ten periods need 1/10 of a second (or 0.1 seconds).

Actually, this signal only needs two information to be created: the frequency (= 100 Hz) and the amplitude (= 1). If we have this information, we can drive an oscillator, and this oscillator will produce exactly this signal.

Exercise: Create this signal in Csound with the opcode `poscil` *(= precise oscillator) and* `out` *(to hear the output). Turn down the volume control of your device: this is maximum amplitude!*

Rather than drawing period by period in time-domain, we can put the information about frequency and amplitude in a plot like this:



Sine wave with 100 Hz and Amplitude 1 in frequency-domain

This way to look at an audio signal is called **frequency-domain**: horizontally we have the **frequencies**, and vertically we have the **amplitude** as strength of a certain frquency. In our simple example, we only have the frequency 100 Hz with an amplitude 1 present.

Please note this:

1. Both views are carrying the same information!

2. There is **no time** in the **frequency domain**. The sine wave is supposed to sound "forever" ...

3. The amplitudes in the time domain are -1 to +1, but in the frequency domain they are always positive (0-1).

How can we get from time-domain to frequency-domain?

By analysis. What we did in the previous example: We looked at the signal in time-domain and we saw that 10 periods of the sine wave took 1/10 of a second. So we knew that the frequency is 100 Hz, because 100 periods will take 1 second. And also we saw that the amplitude is 1 (maximum).

You can easily imagine that for more complex sounds, it is not that easy ... But actually it is also in clever algorithms the same: you look at a signal in time domain, and you try to figure out, which sine waves are in the signal, and with which strengths.

This analysis is called **Fourier Analysis** or **Fourier Transform**; after the french scientist Jean Baptiste Joseph Fourier (1768-1830).

If we do such an analysis, we also call this **Spectral Analysis**. And we call the sine tones which are parts of the sound **Partials**.

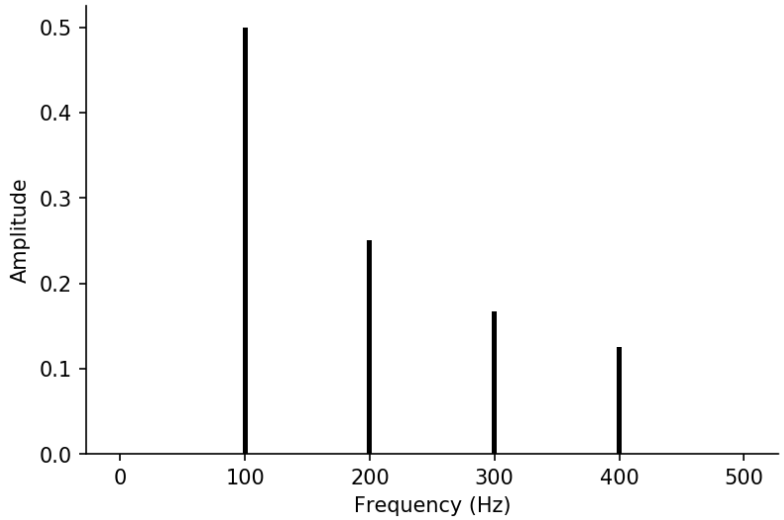
How can we get from frequency-domain to time-domain?

In the simple example, we already did: We synthesized a sound by driving a sine oscillator.

If we use multiple oscillators to create a sound, this is called **Additive Synthesis**.

If we have a sound, which we analyzed before with a Fourier Analysis, and now we synthesize it with sine oscillators, we call it **Resynthesis** (also additive resynthesis or spectral resynthesis).

Exercise: Create a signal in Csound with 4 sine oscillators (partials) with the amplitudes and frequencies shown in the frequency-domain plot below. Render one second of sound to a file, and see how it looks like in time domain (open it in Audacity or a similar audio editor).



4 Sine waves with amplitudes 1/2, 1/4, 1/6, 1/8

II. Performing Fast Fourier Transform (FFT)

A sequence of snapshots

We said that there is no time in a Fourier analysis. But music *is* in time. How can this come together?
The solution is quite similar to what a film does which actually consists of a number of single photos. The sequence of the photos gives the result of a moving image.
So we can think of the Fast Fourier Transform (FFT) used in an audio application as a **sequence of frequency-domain snapshots**. One single snapshot is called an **FFT frame** or **FFT window**. Each snapshot analyses the frequencies in it. In the sequence of the snapshots we get the changes of the spectra over time.

How much time is one FFT snapshot?

The size of the FFT window is measured in samples. At a rate of 44100 samples per second, the **most common frame sizes** are **512**, **1024** or **2048** samples. (The size must always be a power of two: 512 = 2^9 , 1024 = 2^{10} , 2048 = 2^{11} .) This corresponds to the following durations and numbers of frames per second:

- frame size 512/44100 Hz = ca. 0.012 seconds (11 milliseconds) for one FFT frame and ca. 86 frames per second
- frame size 1024/44100 Hz = ca. 0.023 seconds (23 milliseconds) for one FFT frame and ca. 43 frames per second
- frame size 2048/44100 Hz = ca. 0.046 seconds (46 milliseconds) for one FFT frame and ca. 22 frames per second

How many partials (bins) are in one FFT snapshot?

The number of partials which are analyzed by the Fourier Transform depend on the window size: It is half of the number of samples plus one. These partials are usually called **bins**. So:

- for a window size of 512 samples we get 512/2+1 = 257 bins
- for a window size of 1024 samples we get 1024/2+1 = 513 bins
- for a window size of 2048 samples we get 2048/2+1 = 1025 bins

What is the frequency distribution of the bins?

The bins cover the frequency range until half of the sample rate (called the Nyquist frequency). So depending on the number of bins we get different spacings between the bins. Again assuming a sample rate of 44100 Hz:

- for a window size of 512 samples we get 22050 Hz / 256 spacings = ca. 86 Hz distance to the next bin
- for a window size of 1024 samples we get 22050 Hz / 512 spacings = ca. 43 Hz distance to the next bin
- for a window size of 2048 samples we get 22050 Hz / 1024 spacings = ca. 21.5 Hz distance to the next bin

What is the tradeoff between time resolution and frequency resolution?

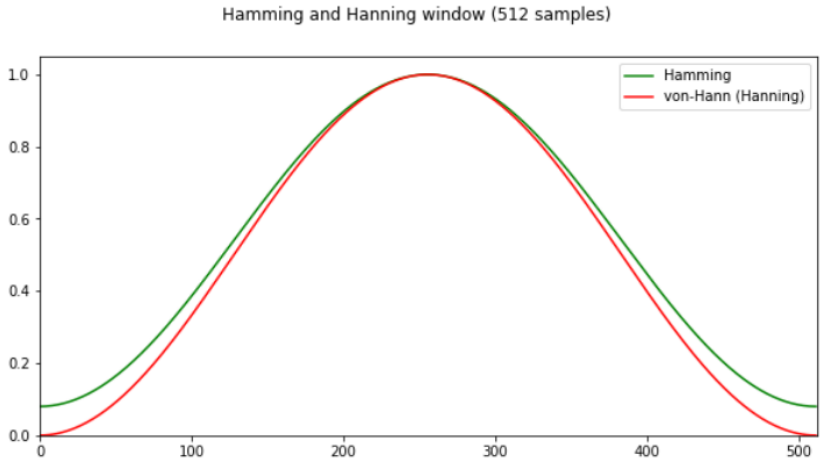
When you look at the calculations above, you can see: for the time resolution it is best to have a small FFT window size. A frame of 512 samples has a very good time resolution (every 11 milliseconds a new window), but a rather poor frequency resolution (86 Hz spacing from one bin to the next). A frame of 2048 samples has a good frequency resolution (about 20 Hz), but a rather poor time resolution (nearly 50 milliseconds).

So we have to find a tradeoff. For practical reasons, I'd say: start with a window size of 1024 samples. If you have problems with the time resolution (for fast changing sounds), try 512 as

window size. If you have problems with the frequency resolution (in particular with low frequencies), try 2048 as window size instead.

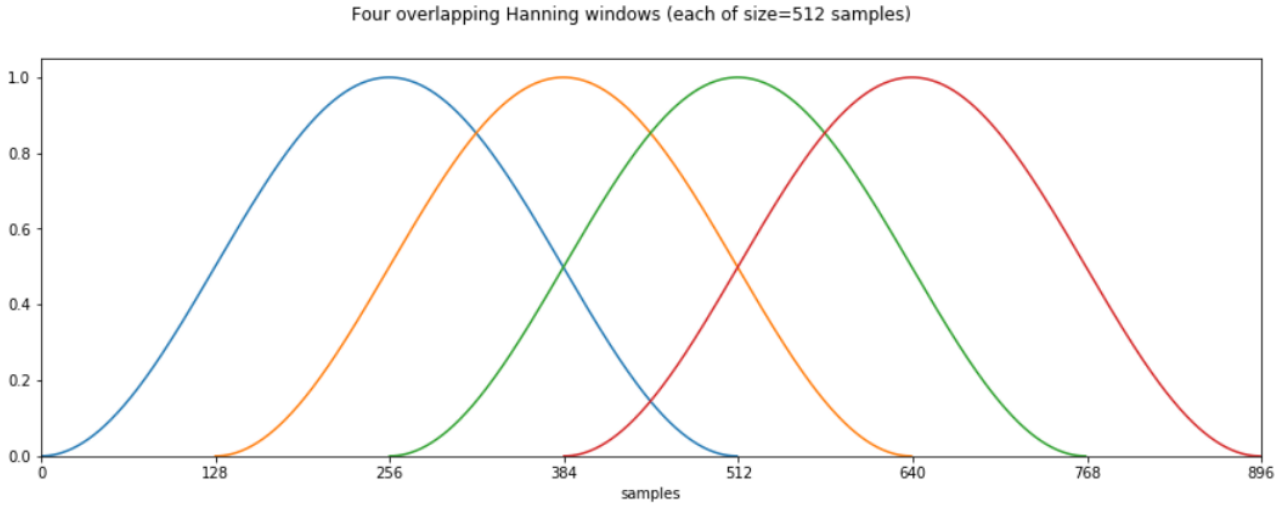
What is the window type?

When we do granular synthesis, we must use a window for soft fade-in and fade-out. Otherwise we would get clicks. This is a bit similar for FFT: We need a smoothing window function to avoid analysis clicks, so to say. The most common window functions are called *Hamming* and *von-Hann* (or *Hanning*). It is not necessary for us to know all the details. Similar to what I said about the tradeoff between time resolution and frequency resolution: if you are not happy with the FFT results, remember this parameter, change it and hear whether it improved the result.



What is overlapping? What means hop size?

For improving the results of the FFT, it is good not to set one window next to the other, but to have overlapping window. A good measurement is to start the next FFT window after 1/4 of the size of one window. So for a window size of 1024 it would mean to start the next window after 256 samples. This is called the **hop size**. Sometimes it is expressed as an integer; in our case we can say: the hop size is 4. Or it is expressed in the number of samples (like in Csound).



III. Some Applications of FFT

Real-time transposition (pitch-shifting)

The rationale here is this:

- We analyse the incoming audio signal on the fly with the opcode **pvsanal**. The result is an **f-signal** in which the frequency information are stored.
- We multiply the frequencies by a factor. Multiplying by 2 would result in transposing an octave higher; multiplying by 1/2 (0.5) would result in transposing an octave lower. As we cannot multiply the f-signal directly, we use the opcode **pvscale**.
- At the end we must go back to time-domain, which means to transform the f-signal back to an audio signal. This is done by the opcode **pvsynth** which does **inverse fourier transform (IFFT)**.

Exercise: Implement this for live input and for any other input (for instance sample playback). The transposition is best given in semitones or cent (use the opcodes `semitone()` *or* `cent()` *to transform the input to a ratio). Use the CsoundQt widgets for the transposition, and if you have a midi device, try to connect a knob or a slider.*

Time stretching

Time stretching (or compressing) is only possible for prerecorded sound. The most simple way is to use the opcode **pvstanal**. All you must do is to load your sound in a function table with GEN routine number 1 in a statement like this (for a sound called "fox.wav" which is found in the same folder)(please look in the manual page for GEN01 to understand the parameters):

```
giFox ftgen 0, 0, 0, -1, "fox.wav", 0, 0, 1
```

Then you can use this function table in `pvstanal` to produce an f-signal with arbitrary time scaling. This would stretch the sound by a factor of 10 (because reading speed is 0.1):

```
fStretch pvstanal 0.1, 1, 1, giFox
```

Again you will use pvsynth to transform this f-signal to an audio signal.

Exercise: Try this for any of your sounds (wav or aif, no mp3 or wmf). Try to control the reading speed in real-time. After you are done, compare with the opcode **paulstretch***, which is based on FFT but uses some sophisticated algorithms. Also compare with the Csound opcode* **mincer***.*

Spectral shift

Image you have a simple sound consisting of five partials: 100, 200, 300, 400 and 500 Hz. If you add now 50 Hz to each of them, you have not only raised the sound by 50 Hz, you have also changed the relations between the partials. At first it was 1:2:3:4:5, now it is 1.5:2.5:3.5:4.5:5.5, or 3:5:7:9:11. This will result in another sound (often similar to the results of ring modulation).

The implementation in Csound is quite simple. The opcode **pvshift** takes the amount of spectral shift (in Hz), and offers also an additional parameter which keeps the frequencies below a certain frequency untouched. By this, we can keep the perception of the pitch (which is based on the lowest partials) but can alter the timbre of the sound.

Cross synthesis

Cross synthesis is a term for many different way to combine to sounds. The classical phase vocoder in one of them, implemented in Csound via the opcode **pvsvoc**. You need two f-signals, deriving from the two sounds you want to cross. In previous examples, we used `pvsanal` (for real-time input or sample playback) and `pvstanal` (for prerecorded sounds) to get these signals. Other opcodes for cross synthesis in Csound are **pvscross**, **pvsfilter** and **pvsmorph**.
Exercise: Read the manual page for `pvsvoc` *and try to implement a simple example. Same for the opcodes* `pvscross`, `pvsfilter` *and* `pvsmorph`. *Try to understand why they sound different.*

Pitch estimation

It is not at all easy to estimate a pitch, not even for a solo instrument or voice. But FFT is the way to go. You can think of it like this: 'Let's see if we have harmonic partials, if yes, the lowest from them should be the pitch we recognize (the others give the timbre).' Csound has a handful of different opcodes. In my experience, **ptrack** and **pvspitch** offer fairly good results.
Exercise: Build an instrument for live input. You sing (or play an instrument), the instrument analyses the frequency and plays the estimated pitch back with an oscillator. Do it with both, **ptrack** *and* **pvspitch**, *and compare the result (either by swapping between both or by driving two oscillators).*