Pitch detection in digital audio

First steps towards a tuner

January 15, 2021

Abstract

A really good tuner shows the user how to tune by being intuitive. That's a technical requirement towards which this explanation goes. Some basics are touched and built on the way; it would't bother if you have already written „hello world“.

„[...] it's just a little coding [...]“

„[...] it's not just the fft [...]“

Part I
Fork

```
// initialize portaudio
err = Pa_Initialize();
if( err != paNoError ) goto error;
```

1 Why?

Being a guitarist for 30 years, I encountered many ways to get the Instrument in tune. When looking back I would like to know how long my guitar was out of tune with me not knowing better... tuning is daily work and not trivial for the beginner. My first tuning tool was a thing to whistle about, which had six notes corresponding to the strings. The sound of this was as far from a guitar as could be, so the difference in pitch was hard to tell. Next came the fifth fret method, as I had suffered long enough from the whistle. A difficult task to handle, using two hands crossed getting the strings to sound while you adjust the string. I found it a little easier to get a glimpse on the difference in pitch with this method, but nevertheless your hearing should be trained to this. Depending on the guitar, this can be error prone due to imperfect fretting. As playing evolves, overtones can be used for tuning. I finally managed to get the guitar in tune fairly well with overtones and a tuning fork. By this method it

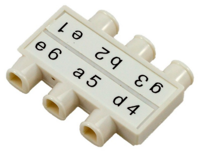


Figure 1: Tuning pipe

became clear that no guitar can ever be perfectly tuned. The interplay of string tension, string length, fret positions, nut and saddle positions is always just as good as it can be. But the compromise keeps getting better and better, and that's the road to go.

While being a guitar teacher I realised the weakness of some tuners concerning the low strings. Especially the low E-string is a fierce demon that some machines just won't see. Also many cheap tuners suffer from being chatty and displaying too much in too little time. Values jump around and can't decide where to go.

2 Evolution of digital tuners

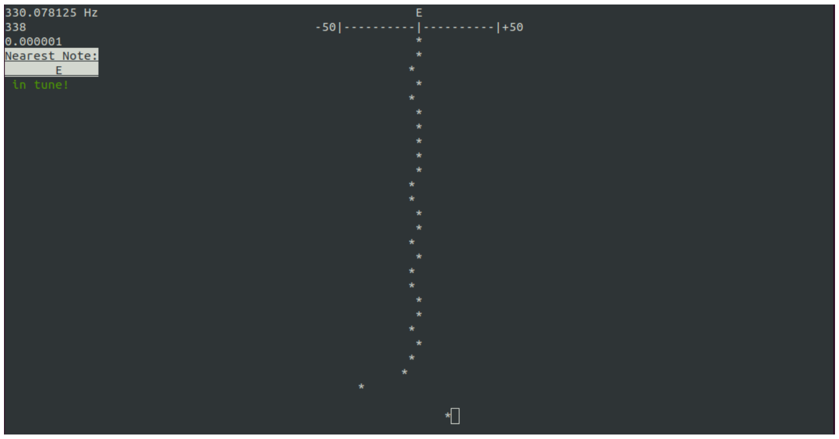
Apart from the whistle, nowadays there are floor tuners, rack tuners and clip-on tuners. These last could be a landmark in useability, if they did their job at least acceptable. Depending on the guitar, an experienced user is necessary - if the vibration of the guitar is not high enough in amplitude the piezo of the clip tuner does not get enough input. Still the difficulty of low strings remained. With the advent of smartphones in almost everybody's hands, a new age in digital tuning has arrived. Suddenly a tuner is capable of a calm display and really neat pitch detection. What's going on? How and why do they accomplish this - whereas the former generations didn't?

3 Not necessarily from scratch

There are numerous tuner algorithms out there, for various platforms. So there's definitely no need to write just another tuner. But the thing with digital audio and signal processing is, you need to start somehow from scratch to gain a basic understanding of what is going on „under the hood“. We use Bjørn Roches Code[1] to start our way into the „black magic“ of the fft. It's a tuner written in c and designed to be easy to understand and follow. The outline of processing is as follows:

- Read audio for fft

[1]http://blog.bjornroche.com/2012/07/frequency-detection-using-fft-aka-pitch.html



```
330.078125 Hz                                             E
338                                         -50|----------|----------|+50
0.000001                                                  *
Nearest Note:                                             *
      E                                                  *
 in tune!                                                 *
                                                         *
                                                          *
                                                          *
                                                          *
                                                          *
                                                          *
                                                         *
                                                         *
                                                          *
                                                          *
                                                         *
                                                          *
                                                         *
                                                         *
                                                          *
                                                          *
                                                         *
                                                          *
                                                         *
                                                        *
                                                  *

                                                              *
```

Figure 2: ccgt 0.1 in the terminal

- Low pass data
- Apply window function on data
- Apply fft
- Find peak in the data
- Determine frequency based on the peak value

Following this skeleton we will discuss the way of the audio signal.

Part II
Execution

```
applyfft( fft, data, datai, false );
```

4 Digital[2] audio signals

All sounds of the real world, that find their way into the computer, live there only as numbers. While the vibrations of air are continuously happening all the time, the computer just gets one moment of a sound at a time. And every moment of the sound the computer records is - a number.

[2]"digitus" is a word for "finger", so the term "digital" means something like "countable with the fingers" - even if today's machines have a bunch of fingers.

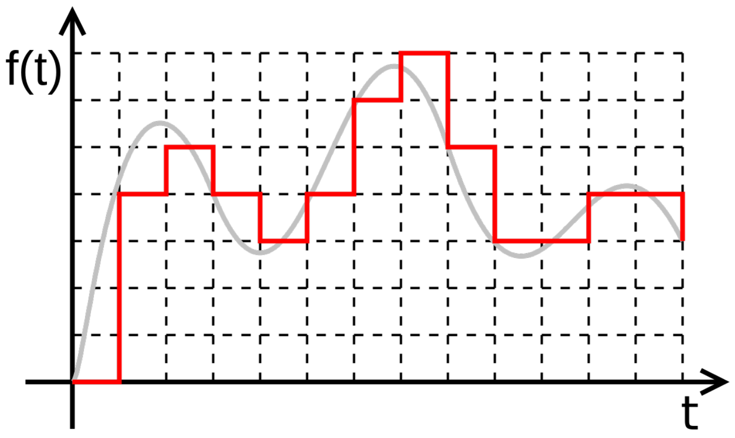


Figure 3: Digital sampling

So if our program records sounds, these vibrations of air somehow get translated into numbers. The numbers are defined by their position in time, one position is one number which is called a sample. The samplerate is how often per second the computer listens. The unit for this is Hertz.

4.1 Perception of tones, sounds and noises

Let's imagine I produce a particular sound, for example tapping the pencil against the table. The single sound of this would be a kind of „tick". When tapping faster, there would be a faster sequence of ticks. Above a certain frequency, these ticks would „collapse" in our perception - into a tone with a frequency. From 20 Hz on upwards, we would perceive what could perhaps be called a tone - at least there is a periodic vibration. I'm afraid I can't do that with my pencil. In musical instruments there are strings, membranes and lots of other things producing these periodic vibrations. If you want to listen to the phenomenon of a series of impulses getting faster and collapsing to a tone, listen to the intro of „Gamma Ray" by the krautrocker group Birthcontrol[3]... a synthesizer does the job here.

The human ear is product of a long process of evolution, which once was one of our life-saving abilities. This means, it can do lots of really fine scanning and listening. But how is the computer doing it? On which basis does he work? One prerequisite - the samplerate - was already mentioned above. There is a fixed rate of scans per second. If human beings can perceive a tone or pitch from 20 Hz on upwards ... how often should the computer sample?

[3]https://www.youtube.com/watch?v=Vzlv7LFmLMg

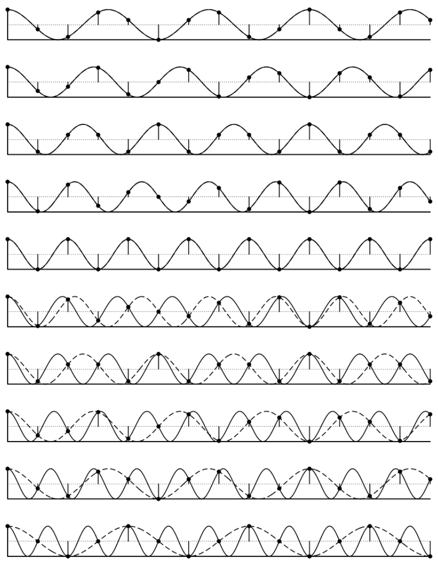


Figure 4: Nyquist theorem

4.2 How the computer listens

The Nyquist - Theorem explains, how often a signal has to be sampled to be able to represent it digitally. Figure 4[4] tries to show, how frequencies above the nyquist frequency behave. In the middle you see a wave that gets sampled, that has exactly the frequency of half the samplerate. That is an important threshold for frequencies that get into the computer. How come, that you can only represent frequencies up to half of the samplerate?

Could the computer hear my pencil tapping at 20 Hz? Yes, if he only listens frequently enough. Suppose I live directly next to a street, on which exactly one car per minute passes past my house. Supposed I listened hard all the time, car noises would become louder at first. Then - when passing by - a climax is reached. After that the noise fades away, until the next car is beginning to come nearer. The level of noise is like a wave. But unfortunately I sit in front of my computer with headphones on and therefore cannot hear what is going on around me. Nothing from the outside world gets to my ears. If I took off my headphones every 60 seconds to check what's going on, I could get the impression there are no cars on the street. That would happen if I accidentally pick the moment when the outside noise is lowest. It could as well happen I always listen exactly in the moment when a car is directly in front of my house. Then I might come to the conclusion, there's a whole lot of traffic. This misconception of what is going on has its roots in the periodic nature of my listening on the one hand, and the processes I listen to on the other hand.

[4]freundlicherweise von Peterpall - Eigenes Werk, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=8730560 zur Verfügung gestellt.

The computer is like me in the example: sampled audio signals are not continuous, but a bunch of points with nothing between them. Figure 5 shows this - the high frequencies are very fast “cars” racing through the points while not really getting noticed. In a periodic signal incredibly many “cars” are passing by with different speeds; you can easily get false impressions about the traffic. As human hearing can sense vibrations up to 20000 Hz, a samplerate of 20000 would never be enough to represent such high frequencies. Therefore you might have heard of the famous value 44100 Hz, roughly about the double of perceivable frequencies plus a little headroom.

The signal inside the computer is not only discrete in time. Also the strength of the vibration is represented as numbers. The amplitude of the signal is - after being recorded by the computer - not continuous any more. The louder a sample, the higher the number on a given scale. This value represents a step of loudness, and between the steps there are gaps again. Most audio data has 8 or 16 bit samples, which means the number for one sample can be that long or precise.

```
recorder = new AudioRecord(AUDIO_SOURCE,
SAMPLE_RATE, CHANNEL, AUDIO_FORMAT, BUFFER_SIZE);
```

If you see the processing of input data as some kind of “hearing”, so this is a discrete process. The computer cannot be a unity of perception and processing at the same time like the human hearing. It’s always going to be a finite buffer of numbers that the computer processes. After finishing the processing, the result can be viewed and the next buffer will be processed and so on. Of course this happens very fast - so fast you could think the process is continuous. The changes happen within milliseconds. Nevertheless there has to be a defined region of audio data, on which processing occurs and on which steps of processing are defined. Here the buffer comes into focus.

4.3 Buffer: One at a time, please

Buffer size determines the amount of audio data on which calculations are done. Suppose I work in the kitchen of a restaurant and have the task of peeling potatos. As the potatos are in the basement, I won’t go there for every single potato, peel it and put it into the water. Normal human beings would fetch a bag of potatoes. This quantity gets peeled and eaten, when that is done the next bag is fetched. It is unlikely that so many potatoes get eaten for one lunch, but never mind the bad example. The size of the bag is the buffersize for the processing. The computer gets data all the time, but he can only deal with it in defined portions. The audio system may need a fine sample resolution in order to be able to represent the desired frequencies - but buffer size also has its constraints and should neither be too big nor too small. Maybe you have already encountered the problem of latency while recording audio and then tried to change your buffer size. While doing this you often see powers of 2 as values.

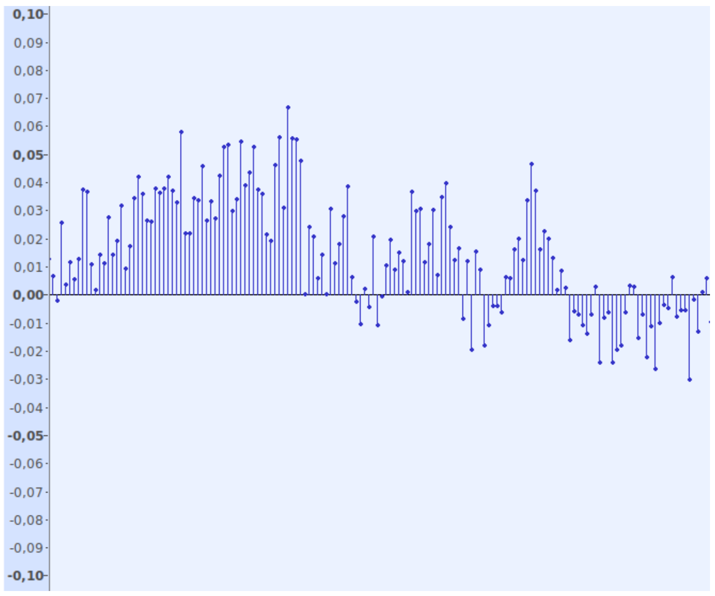


Figure 5: Discrete values on the time axis

```
err = Pa_ReadStream( stream, data, FFT_SIZE );
```

The easiest case is, your buffer size is equal to your fft size. Of course there are more refined concepts. But for first we divide the problem into reading audio data once - and then analysing around on this "screenshot" of audio.

5 Fourier transformation

5.1 Change dimensions

Having read audio data into the buffer, the numbers are in memory as time discrete values. They represent vibrations of air at a given moment in time. Data is ordered on the time axis, and each value has a certain "height" or amplitude. The imagination of a waveform might help. The array of audio data is a vector of numbers with the length of n.

$$v = [v_{0,}v_1, ..., v_{n-1}]^T \;\; v \,\epsilon\, \mathbb{R}$$

The elements of this vector are float numbers with values from -1 to 1. The cector v is transformed to a vector c of equal length by the fourier transformation:

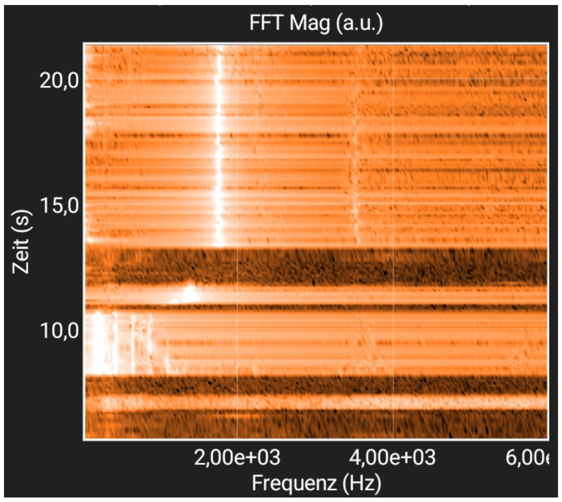


Figure 6: Waterfall-display

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{pmatrix} = ft \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ \vdots \\ v_{n-1} \end{pmatrix}$$

After that, again we have a series of numbers of equal length. But the numbers now represent frequencies, not amplitudes. After the fourier transformation, the vector c

$$c = [c_0, c_1, ..., c_{n-1}]$$

contains the frequency spectrum of the input signal. There was a cocktail in your glass before, but now you have all the ingredients separated in it - at least ideally. The cake was ready, but now it is backwards engineered and divided into the parts of its recipe again. The transformation takes amplitudes on the time axis, and gives you amplitudes on the frequency axis. See this happening in realtime on a waterfall display, and you will know what I mean.[5] Spectral components are plotted as time advances. Dominant frequencies are highlighted.

But how does fourier transformation really work? What happens to our input data, that makes it so meaningful afterwards? Let's say we have a audio buffer of the length 8, and there are even some values in it:

$$v = [1, 2, 3, 4, 5, 6, 7, 8]$$

Fourier transformation would take this as input data:

[5]https://phyphox.org/de/home-de/ software for physical experiments

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{pmatrix} = ft \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{pmatrix}$$

It then computes the coefficients of sine- and cosine waves. Fourier believed, that every periodic wave could mathematically be described as the sum of sinewaves and their integer multiples.[6] This process is called fourier synthesis. For a periodic function f(t) with the period T, the definition of the fourier synthesis is as follows:

$$f(t) = \sum_{n=-\infty}^{\infty} c_n \, e^{2\pi \, i \, n \, t/T}$$

The function is built up out of infinite partial functions, every one depending on the number n. The base frequency of each partial is t/T in the exponent of e, and n as its integer factor. Ideally this should add up infinitely. The computer won't reach infinity with his calculations, and we don't want to wait until he eventually does. The frequencies of the input signal don't reach until infinity as well, as it is built of samples and therefore constrained to the nyquist frequency. So we modify the function and reduce it to the length of the buffer. Additionally, we don't want to have a synthesizer that builds soundwaves, but the exact opposite: We want to determine frequencies in the input signal. So we apply the inverse of fourier synthesis, the fourier analysis. This is often called the discrete fourier transform, because the signal we're working on is not continuous but discrete. With this said, the vector c gets computed as follows:

$$c_k = \frac{1}{n} \sum_{j=0}^{n-1} v_j \, e^{-2\pi i \frac{kj}{n}}$$

Every sample in our input data is now represented by a element v with the index j in the sum. This means: To get one single element of the transformed vector c, a operation on the whole audio buffer occurs. If you write the computation for one element of c as row, and that for every single element of c, you get the matrix version of the fourier transformation:

[6]https://www.youtube.com/watch?v=7bi5_KC08Kg

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{pmatrix} = \begin{pmatrix} \odot^0 & \odot^0 & \odot^0 & \odot^0 & \odot^0 & \odot^0 & \odot^0 & \odot^0 \\ \odot^0 & \odot^1 & \odot^2 & \odot^3 & \odot^4 & \odot^5 & \odot^6 & \odot^7 \\ \odot^0 & \odot^2 & \odot^4 & \odot^6 & \odot^8 & \odot^{10} & \odot^{12} & \odot^{14} \\ \odot^0 & \odot^3 & \odot^6 & \odot^9 & \odot^{12} & \odot^{15} & \odot^{18} & \odot^{21} \\ \odot^0 & \odot^4 & \odot^8 & \odot^{12} & \odot^{16} & \odot^{20} & \odot^{24} & \odot^{28} \\ \odot^0 & \odot^5 & \odot^{10} & \odot^{15} & \odot^{20} & \odot^{25} & \odot^{30} & \odot^{35} \\ \odot^0 & \odot^6 & \odot^{12} & \odot^{18} & \odot^{24} & \odot^{30} & \odot^{36} & \odot^{42} \\ \odot^0 & \odot^7 & \odot^{14} & \odot^{21} & \odot^{28} & \odot^{35} & \odot^{42} & \odot^{49} \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{pmatrix}$$

$$\odot = e^{-i\frac{2\pi}{n}}$$

For every row of the fourier matrix, the index k stays constant while j increases for every column. This gives a zero line as first line, integer multiples of 1 in the second line, integer multiples of 2 in the third line and so on. The first column is zero, because the index j always starts with 0. The resulting matrix is symmetric, a property that is used by the "fast fourier transform" algorithm, in short fft. As a result of this highly efficient algorithm, you find many code snippets just labeled "fft".

The more you get down into the matrix, the higher the exponent; the higher the frequencies which are calculated. You'll see even more reference to periodic functions, if we write the function with sine and cosine waves using euler's identity.

$$c_k = \frac{1}{n} \sum_{j=0}^{n-1} v_j \left[cos\left(\frac{2\pi \cdot k \cdot j}{n}\right) - i\, sin\left(\frac{2\pi \cdot k \cdot j}{n}\right)\right]$$

You find the product of k and j in the above fourier matrix. This is a sum containing a bunch of periodic waves, which are weighted depending on how the input signal looks like. Depending on how big v is at the index j, some frequencies get a greater value than others. Fourier analysis finds out, which frequencies[7] to add in order to get something like the input signal.

5.2 Frequency distribution in the result

How can I tell the user about the frequency of the tone that was played, dependent on buffer and fft size? There is this output vector which gives me some data the fft has computed. What do these values mean? To find out, we should give our program some parameters:

```
#define FFT_SIZE (8192)
#define SAMPLERATE (8000)
```

[7]You might say circular motions.

The signal gets sampled 8000 times per second, frequencies up to approximately 4000 Hz can be detected. The audio buffer and therefore the vector for the input data has a length of 8192, and that's also the amount of complex numbers we have now as output vector of the fft. The frequency f for a value from c is defined as follows:

$$N = 8192$$

$$sr = 8000$$

$$f\left(c_k\right) = k * \tfrac{sr}{N}$$

This apparently goes from zero to approximately 8000, so the values of the frequency vector above 4000 are not meaningful. Noteworthy here - every value in c is a central frequency. For every k, the frequency value changes by

$$\Delta f = \tfrac{sr}{N} = \tfrac{8000}{8192} \approx 1Hz$$

given our configuration. As a comparison check out these values:

$$N = 1024$$

$$sr = 48000$$

$$\Delta f = \tfrac{48000}{1024} \approx 48Hz$$

A low samplerate combined with a large buffer minimizes the frequency width, whereas a finer sampling combined with a small buffer gives you a really coarse frequency grid.[8]

In the code mentioned above, there are arrays of the length of the fft for frequency, note name and pitch. While this is implementation specific, it should still be mentioned that the frequency array is initialised over its whole length with the corresponding frequency value

$$f_i = \tfrac{sr \cdot i}{N}.$$

[8]These frequency bands are often called „fft bins".

The samplerate divided by size of the fft window gives you the width of one frequency bin, its central frequency determined by the index i.

```
for ( int i = 0; i < FFT_SIZE; ++i ) {
freqTable[i] =
(SAMPLE_RATE * i) / (double)(FFT_SIZE);
}
```

5.3 Pitches and names

The program needs an internal representation of how the frequencies should be called as notes. Given a detected fundamental frequency the code should know, which pitch corresponds to the frequency. The twelve pitches can be mapped to frequencies via an index. But of what frequencies are we speaking? The a4 at 440 Hz is taken as a reference in most cases. Knowing that one octave doubles frequency[9], I can identify 880 Hz, 220 or 110 Hz as notes with the pitch "a". But what happens in between? Maybe the guitar is really not in tune, and the string is almost one semitone sharp or flat - or the strings were completely changed out and have no "initial value".

As frequency exactly doubles while crossing one octave, the frequency distance between semitones grows when ascending or diminishes when descending. There is never the same numerical distance between the frequency of semitones. That's why a logarithmic scale is used to obtain a scale of equidistant semitones. Now the question is, which frequency does the pitch one semitone above a4 have? The octave - here the factor p which means "twice of what was there before" - gets equally spaced in twelve steps:

$$p = 2$$

$$n = 12$$

$$r = \sqrt[n]{p} = \sqrt[12]{2} = 2^{\frac{1}{12}}$$

If I multiply any frequency by 2, out comes the octave. If I want the tone to be just a little higher - for example the smallest possible diatonic step of one semitone - the factor should be less big. How much "less big" can be calculated, if one reference frequency and the distance is known.

[9]Exponential increase, by the way.

Note	r	Hz
A	$2^{\frac{0}{12}}$	440.00
A#/Bb	$2^{\frac{1}{12}}$	466.16
B/H	$2^{\frac{2}{12}}$	493.88
C	$2^{\frac{3}{12}}$	523.25
C#/Db	$2^{\frac{4}{12}}$	554.37
D	$2^{\frac{5}{12}}$	587.33
D#/Eb	$2^{\frac{6}{12}}$	622.25
E	$2^{\frac{7}{12}}$	659.26
F	$2^{\frac{8}{12}}$	698.46
F#/Gb	$2^{\frac{9}{12}}$	739.99
G	$2^{\frac{10}{12}}$	783.99
G#/Ab	$2^{\frac{11}{12}}$	830.61
A	$2^{\frac{12}{12}}$	880.00

With this information, a program can generate note names from frequencies - in what form or data structure you want to do it is up to you as a decision of the implementation.

5.4 Peak

```
// distance formula
v = this.ar[j]*this.ar[j] + this.ai[j]*this.ai[j];
```

How does the program find the fundamental frequency? The given code determines the energy at certain fft bins[10] by interpreting the result of the fft as a distance. In the easiest case, an iteration over the array of results is done to get the maximum value; that means we search the frequency band with the greatest amplitude.

The low e-string of the guitar has a fundamental frequency of 82.5 Hz. But unfortunately, this frequency is lower in amplitude than the first harmonic in the majority of cases. So what happens with our simple pitch detection? It detects the highest amplitude, which is in fact the harmonic and not the desired fundamental that we hear. At this point, the fft is not good enough for a reliable

[10]Auf stackoverflow „fft magnitude“

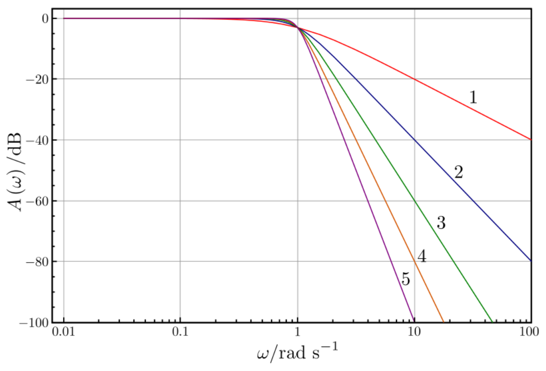


Figure 7: Lowpass filter

pitch detection. The fundamental which we somehow hear but which does not get detected points to the fact, that pitch perception is a psychoacoustic phenomenon. What we actually hear does not seem to be determined solely by the spectrum that a fft reveals. Whether fft is reliable, seems to depend on its working area. If you want to get your guitar in tune, and fft is the only tool - you will soon want to optimize things. In order to do this it is necessary to dive deeper into signal processing and pitch detection algorithms.

6 Filter

In order to minimize the problem of missing fundamentals, a low pass filter is used before doing the fft. Only frequencies up to a specific threshold get into the buffer. Most high frequencies would not be detected anyway. And in a tuning app, we do not need high quality audio as we don't want to play it back after processing it. Filtering out high frequencies and unmusical noises reduces errors while processing the data. Additionally a windowing function is used to smooth the audio screenshot at the edges. These soft edges also reduce errors caused by rough transitions, as the fourier transformation pretends the signal to be periodic.

Part III Synthesis

While not yet dealing with aspects of user interaction, the implementation of a basic functionality was discussed. Simply doing a fft gets you in the right direction, but unfortunately only half the way. Nevertheless this is the first step towards a solution of the realworld requirements. The stage is set, on

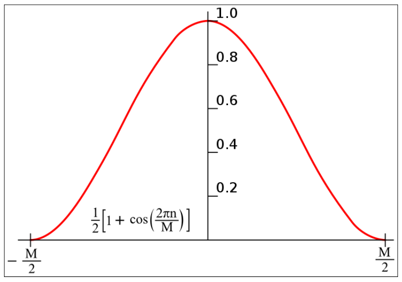


Figure 8: Hanning - window

which further refinement can take place.

Tuning a guitar is a special case in the more general field of "pitch detection". Since electronic equipment became small enough, it was employed in tuning. With smartphones in our hands, a little revolution of computing power took place. Fourier analysis can be done on most if not all of today's handheld devices. But that's not state of the art anymore. There are numerous algorithms for pitch detection out there. Fft is just one of the tools applied in these algorithms, dealing with the frequency domain of signals. If you feel you understand the numbercrunching that is done in fourier transformation, go on and take a look at the McLeod pitch method[11] which has a really good paper online. The YIN[12] algorithm is another more recent approach which is very well documented online.

Have fun coding audio, and don't forget to play your guitar!

[11]This could be sort of a follow up: http://www.cs.otago.ac.nz/tartini/papers/Visualization_of_Musical_Pitch.pdf

[12]A bit more difficult to read: http://audition.ens.fr/adc/pdf/2002_JASA_YIN.pdf