

# How to find the fundamental frequency of a guitar string sound?

## [closed]

Asked 13 years, 10 months ago   Modified 9 years ago   Viewed 9k times



17



**Closed.** This question is seeking recommendations for software libraries, tutorials, tools, books, or other off-site resources. It does not meet [Stack Overflow guidelines](#). It is not currently accepting answers.



We don't allow questions seeking recommendations for software libraries, tutorials, tools, books, or other off-site resources. You can edit the question so it can be answered with facts and citations.

Closed last year.

[Improve this question](#)

I want to build a guitar tuner app for Iphone. My goal is to find the fundamental frequency of sound generated by a guitar string. I have used bits of code from auriotouch sample provided by Apple to calculate frequency spectrum and I find the frequency with the highest amplitude . It works fine for pure sounds (the ones that have only one frequency) but for sounds from a guitar string it produces wrong results. I have read that this is

because of the overtones generate by the guitar string that might have higher amplitudes than the fundamental one. How can I find the fundamental frequency so it works for guitar strings? Is there an open-source library in C/C++/Obj-C for sound analyzing (or signal processing)?

audio

signal-processing

guitar

Share

Improve this question

Follow

edited May 2, 2012 at 12:22



AudioDroid

2,324 ● 2 ● 21 ● 33

asked Feb 18, 2011 at 17:08



Mircea

925 ● 6 ● 12

2 Answers

Sorted by:

Highest score (default)



47



You can use the signal's autocorrelation, which is the inverse transform of the magnitude squared of the DFT. If you're sampling at 44100 samples/s, then a 82.4 Hz fundamental is about 535 samples, whereas 1479.98 Hz is about 30 samples. Look for the peak positive lag in that range (e.g. from 28 to 560). Make sure your window is at least two periods of the longest fundamental, which would be 1070 samples here. To the next power of two that's a 2048-sample buffer. For better frequency resolution and a less biased estimate, use a longer buffer, but not so long

that the signal is no longer approximately stationary.

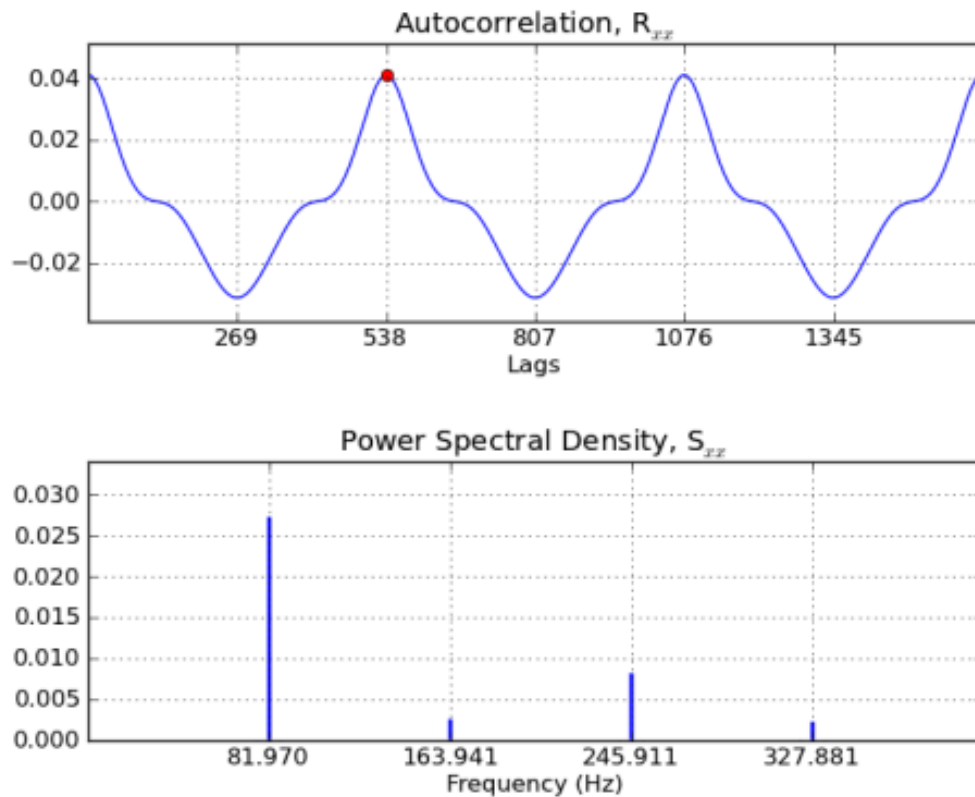
Here's an example in Python:

```
from pylab import *
import wave

fs = 44100.0    # sample rate
K = 3           # number of windows
L = 8192        # 1st pass window overlap, 50%
M = 16384       # 1st pass window length
N = 32768       # 1st pass DFT length: acyclic
correlation

# load a sample of guitar playing an open string 6
# with a fundamental frequency of 82.4 Hz (in
# theory),
# but this sample is actually at about 81.97 Hz
g =
fromstring(wave.open('dist_gtr_6.wav').readframes(-1
                dtype='int16'))
g = g / float64(max(abs(g)))    # normalize to +/-
1.0
mi = len(g) / 4                # start index

def welch(x, w, L, N):
    # Welch's method
    M = len(w)
    K = (len(x) - L) / (M - L)
    Xsq = zeros(N/2+1)          # len(N-
point rfft) = N/2+1
    for k in range(K):
        m = k * (M - L)
        xt = w * x[m:m+M]
        # use rfft for efficiency (assumes x is
        real-valued)
        Xsq = Xsq + abs(rfft(xt, N)) ** 2
    Xsq = Xsq / K
    Wsq = abs(rfft(w, N)) ** 2
    bias = irfft(Wsq)          # for
    unbiasing Rxx and Sxx
    n = dot(x, x) / len(x)    # avg
```



Output:

```
Average Power
  p: 0.0410611012542
 Rxx: 0.0410611012542
 Sxx: 0.0410611012542
```

The peak lag is 538, which is  $44100/538 = 81.97$  Hz. The first-pass acyclic DFT shows the fundamental at bin 61, which is  $82.10 \pm 0.67$  Hz. The 2nd pass uses a window length of  $538 \times 15 = 8070$ , so the DFT frequencies include the fundamental period and harmonics of the string. This enables an unbiased cyclic autocorrelation for an improved PSD estimate with less harmonic spreading (i.e. the correlation can wrap around the window periodically).

Edit: Updated to use Welch's method to estimate the autocorrelation. Overlapping the windows compensates

for the Hamming window. I also calculate the tapered bias of the hamming window to unbiased the autocorrelation.

Edit: Added a 2nd pass with cyclic correlation to clean up the power spectral density. This pass uses 3 non-overlapping, rectangular windows length  $538 \times 15 = 8070$  (short enough to be nearly stationary). The bias for cyclic correlation is a constant, instead of the Hamming window's tapered bias.

Share Improve this answer

edited Feb 20, 2011 at 8:25

Follow

community wiki


12 revs

eryksun

---

1 Does this technique work when the fundamental is smaller than the harmonics (or missing altogether)? – [mtrw](#) Feb 18, 2011 at 22:32

---

1 @mtrw: If you have a 2 Hz overtone and a 3 Hz overtone, the triangular looking resulting waveform will still repeat every 1 second, which autocorrelation over a long enough window will find, even with zero 1 Hz sinusoidal content. – [hotpaw2](#) Feb 18, 2011 at 22:52 

---



4

Finding the musical pitches in a chord is far more difficult than estimating the pitch of one single string or note played at a time. The overtones for the multiple notes in a chord might all be overlapping and interleaving. And all the notes in common chords may themselves be at



overtone frequencies for one or more non-existent lower pitched notes.



For single notes, autocorrelation is a common technique used by some guitar tuners. But with autocorrelation, you have to be aware of some potential octave uncertainty, as guitars may produce inharmonic and decaying overtones which thus don't exactly match from pitch period to pitch period. Cepstrum and Harmonic Product Spectrum are two other pitch estimation methods which may or may not have different problems, depending on the guitar and the note.

[RAPT](#) appears to be one published algorithm for more robust pitch estimation. YIN is another.

Also Objective C is a superset of ANSI C. So you can use any C DSP routines you find for pitch estimation within an Objective C app.

Share Improve this answer

edited Dec 18, 2015 at 2:18

Follow

answered Feb 18, 2011 at 22:20



[hotpaw2](#)

70.6k ● 15 ● 92 ● 155