

**Gerry Beauregard**

*Audio programming and occasionally  
more*

---

## High Accuracy Monophonic Pitch Estimation Using Normalized Autocorrelation

Posted on [July 15, 2013](#)

★★★★★ ⓘ 2 Votes

A few weeks ago, I posted a [demo Flash real-time pitch detector](#), and described a bit how it worked without showing any source code. Well, today, I am posting some actual code, C++ code for a very accurate monophonic time-domain pitch estimator.

I've used variants of this pitch estimator in various projects since the late 1990s. The version posted here is a simple, non-optimized version which I wrote for a friend. Note that it's written for clarity, not speed. Note also that it's only appropriate for monophonic (in the sense of single pitched source) signals. Polyphonic pitch detection is a harder problem, best tackled using spectral techniques.

Scroll to the bottom of this post for the code. I'm releasing it under the MIT license, which means you can do pretty much whatever you like with it as long as your source code also includes the license. Hat-tips in the form of comments, credits, and free copies of whatever products you create using it are welcome.

The first essential step in this and many other time-domain pitch estimators is the normalized autocorrelation (NAC), which in my code looks like this:

```
1  vector nac(maxP+1);
2
3  for ( int p = minP-1; p <= maxP+1; p++ )
4  {
5      double ac = 0.0;           // Standard auto-correlation
6      double sumSqBeg = 0.0;     // Sum of squares of beginning part
7      double sumSqEnd = 0.0;     // Sum of squares of ending part
8
9      for ( int i = 0; i < n-p; i++ )
10     {
11         ac += x[i]*x[i+p];
12         sumSqBeg += x[i]*x[i];
13         sumSqEnd += x[i+p]*x[i+p];
14     }
15     nac[p] = ac / sqrt( sumSqBeg * sumSqEnd );
16 }
```

In this code, x points to the input signal, which should normally have a length at least two times the maximum

period; minP/maxP are the minimum/maximum periods of interest; p is the ‘lag’ or time shift in samples (i.e. the hypothetical period); and ac is just the standard (non-normalized) autocorrelation.

Plenty of pitch estimators use autocorrelation, but the problem is that the magnitude of the autocorrelation depends on the magnitude of the signal. Another problem is that because the autocorrelation is computed based on fewer points as the ‘lag’ p increases, the autocorrelation tends to get smaller with increasing p. That makes it difficult to choose the best period.

The normalized autocorrelation (nac in the code) is computed by dividing the (non-normalized) autocorrelation ac by the square root of the product of the sums-of-squares of the two sub-sequences that were multiplied to give the autocorrelation. That’s a mouthful – it’s easier to say in math lingo, but clearest in code.

The net result of the normalization is that if p is exactly equal to the real period (or an integer multiple thereof), the NAC at that period has a value of exactly 1.0. A happy by-product of the normalization is that if you have a signal that’s periodic but has an exponentially growing or decaying envelope, the NAC will still be 1.0. It even works if there’s no energy at the fundamental frequency.

I came up with this particular normalization independently in the late 1990s, but it’s not unique. A couple of years ago, a friend of mine, Dave Fernandes (CEO of [Mint Leaf Software](#)) told me my normalization is the same as the one in Paul Boersma’s [Praat](#) speech analysis tool, which has been around from the mid-1990s. And I’ve seen the same normalization in various academic papers, for example this [2003 paper by Sumit Basu](#) from Microsoft.

What’s more interesting than the NAC algo is how one can apply some simple tricks to get musically useful results. First off, we need to improve the resolution. For typical sample rates and musical pitches, estimating the period to the nearest number of samples is nowhere near good enough. Consider the top note on a piano, C8 = 4186Hz. For a sample rate of 44.1kHz, C8 would have a period of 10.53 samples. If we could only estimate the period to the nearest sample, we’d get either 10 or 11 samples – error of 5%, nearly a semitone! To get a vastly better estimate, we can apply quadratic interpolation using the peak NAC and the points on either side of it. With this interpolation, the error is reduced to tiny fractions of a semitone, throughout the entire range of a piano at least.

The other challenge is to eliminate so-called ‘octave errors’ in which the estimated period is actually a multiple of the real period (e.g. C4 gets misrecognized as C3). The trick is to check whether the NAC has strong peaks at integer submultiples of period implied by the strongest peak. For example, if the biggest peak in the NAC is for a period of 300 samples, but there are very strong peaks at 100 and 200 as well, then assume the period is 100 samples.

To build sample, just drop the code (see below) into a main.cpp in your favourite C++ development environment. It’s hard-coded to generate and analyze a signal with fundamental frequency corresponding to middle C (C4 ~= 261.6 Hz). When you run it, you should get this output:

```
Actual freq:      261.626
Estimated freq:  261.625
Error (cents):    -0.002
Periodicity quality:  1.000
```

‘Actual freq’ is the pitch of the test signal; ‘Estimated freq’ is the estimated pitch of the test signal as computed

by my algorithm; ‘Error (cents)’ is the error in the estimate is hundredths of semitones; and ‘Periodicity quality’ is a measure of how periodic the signal is, which 1.0 meaning perfectly periodic. Note that error: 2 millicents!

Incidentally, that ‘periodicity quality’ can be handy if you’re synthesizing speech – rather than use a boolean voiced/unvoiced decision, you can synthesize with varying amounts of noisiness based on the quality. Also good for synthesis of musical signals (which after all are never purely periodic or purely noisy).

There are loads of possible performance tweaks not shown here:

- Use an FFT-based method to compute the autocorrelation
- Compute the square of each sample only once.
- Find places in the signal where the periodicity is very strong, then scan forwards and backwards from there to track the pitch into the less certain portions. (Typically only applicable for non-real-time cases).
- Limit the search range to only a small part of the possible pitch range around the most recently identified pitch.
- Mix in some noise and/or use a level threshold to prevent spurious pitch detections when the signal level gets very small.

All things I’ve done at various times, and which people ‘skilled-in-the-art’ (as they say in patents) could figure out. Besides performance optimizations, to use this algo in a real-time context you have to know how to capture input audio, how to call the pitch estimator at appropriate times, how to display the output, etc. These are left as exercises for the reader (as they say in textbooks)... but if you’d like to hire someone to help with it, I know someone you can call. 😊

```
1 // =====
2 //  PeriodEstimator demo
3 //
4 //  Demonstrates use of period estimator algorithm based on
5 //  normalized autocorrelation. Other neat tricks include sub-sample
6 //  accuracy of the period estimate, and avoidance of octave errors.
7 //
8 //  Released under the MIT License
9 //
10 //  The MIT License (MIT)
11 //
12 //  Copyright (c) 2009 Gerald T Beauregard
13 //
14 //  Permission is hereby granted, free of charge, to any person obtaining a copy
15 //  of this software and associated documentation files (the "Software"), to deal
16 //  in the Software without restriction, including without limitation the rights
17 //  to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
18 //  copies of the Software, and to permit persons to whom the Software is
19 //  furnished to do so, subject to the following conditions:
20 //
21 //  The above copyright notice and this permission notice shall be included in
22 //  all copies or substantial portions of the Software.
23 //
24 //  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
25 //  IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
26 //  FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
27 //  AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
28 //  LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
29 //  OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
30 //  THE SOFTWARE.
31 // =====
32
33 #include <stdio.h>
34 #include <math.h>
35 #include <assert.h>
```



```

36 #include <vector>
37
38 using namespace std;
39
40 double EstimatePeriod(
41     const double *x,          // Sample data.
42     const int n,              // Number of samples. For best results, should be
43     const int minP,           // Minimum period of interest
44     const int maxP,           // Maximum period of interest
45     double& q );              // Quality (1= perfectly periodic)
46
47
48
49
50 int main (int argc, char * const argv[])
51 {
52     const double pi = 4*atan(1);
53
54     const double sr = 44100;    // Sample rate.
55     const double minF = 27.5;   // Lowest pitch of interest (27.5 = A0, lowest
56     const double maxF = 4186.0; // Highest pitch of interest(4186 = C8, highest
57
58     const int minP = int(sr/maxF-1); // Minimum period
59     const int maxP = int(sr/minF+1); // Maximum period
60
61     // Generate a test signal
62
63     const double A440 = 440.0;    // A440
64     double f = A440 * pow(2.0,-9.0/12.0); // Middle C (9 semitones below A440)
65
66     double p = sr/f;
67     double q;
68     const int n = 2*maxP;
69     double x[n];
70
71     for ( int k = 0; k < n; k++ )
72     {
73         x[k] = 0;
74         x[k] += 1.0*sin(2*pi*1*k/p); // First harmonic
75         x[k] += 0.6*sin(2*pi*2*k/p); // Second harmonic
76         x[k] += 0.3*sin(2*pi*3*k/p); // Third harmonic
77     }
78
79     // TODO: Add low-pass filter to remove very high frequency
80     // energy. Harmonics above about 1/4 of Nyquist tend to mess
81     // things up, as their periods are often nowhere close to
82     // integer numbers of samples.
83
84     // Estimate the period
85     double pEst = EstimatePeriod( x, n, minP, maxP, q );
86
87     // Compute the fundamental frequency (reciprocal of period)
88     double fEst = 0;
89     if ( pEst > 0 )
90         fEst = sr/pEst;
91
92     printf( "Actual freq:      %8.3lf\n", f );
93     printf( "Estimated freq:     %8.3lf\n", sr/pEst );
94     printf( "Error (cents):         %8.3lf\n", 100*12*log(fEst/f)/log(2) );
95     printf( "Periodicity quality: %8.3lf\n", q );
96
97     return 0;
98 }
99
100
101
102 // =====
103 // EstimatePeriod

```

```

104 //
105 // Returns best estimate of period.
106 // =====
107 double EstimatePeriod(
108     const double    *x,          // Sample data.
109     const int       n,          // Number of samples. Should be at least 2 x maxP
110     const int       minP,       // Minimum period of interest
111     const int       maxP,       // Maximum period
112     double&         q )         // Quality (1= perfectly periodic)
113 {
114     assert( minP > 1 );
115     assert( maxP > minP );
116     assert( n >= 2*maxP );
117     assert( x != NULL );
118
119     q = 0;
120
121     // -----
122     // Compute the normalized autocorrelation (NAC). The normalization is such th
123     // if the signal is perfectly periodic with (integer) period p, the NAC will b
124     // exactly 1.0. (Bonus: NAC is also exactly 1.0 for periodic signal
125     // with exponential decay or increase in magnitude).
126
127     vector<double> nac(maxP+1);
128
129     for ( int p = minP-1; p <= maxP+1; p++ )
130     {
131         double ac = 0.0;          // Standard auto-correlation
132         double sumSqBeg = 0.0;    // Sum of squares of beginning part
133         double sumSqEnd = 0.0;    // Sum of squares of ending part
134
135         for ( int i = 0; i < n-p; i++ )
136         {
137             ac += x[i]*x[i+p];
138             sumSqBeg += x[i]*x[i];
139             sumSqEnd += x[i+p]*x[i+p];
140         }
141         nac[p] = ac / sqrt( sumSqBeg * sumSqEnd );
142     }
143
144     // -----
145     // Find the highest peak in the range of interest.
146
147     // Get the highest value
148     int bestP = minP;
149     for ( int p = minP; p <= maxP; p++ )
150         if ( nac[p] > nac[bestP] )
151             bestP = p;
152
153     // Give up if it's highest value, but not actually a peak.
154     // This can happen if the period is outside the range [minP, maxP]
155     if ( nac[bestP] < nac[bestP-1]
156         && nac[bestP] < nac[bestP+1] )
157     {
158         return 0.0;
159     }
160
161     // "Quality" of periodicity is the normalized autocorrelation
162     // at the best period (which may be a multiple of the actual
163     // period).
164     q = nac[bestP];
165
166     // -----
167     // Interpolate based on neighboring values
168     // E.g. if value to right is bigger than value to the left,
169     // real peak is a bit to the right of discretized peak.
170     // if left == right, real peak = mid;

```

```

172 // if left == mid,    real peak = mid-0.5
173 // if right == mid,   real peak = mid+0.5
174
175 double mid    = nac[bestP];
176 double left   = nac[bestP-1];
177 double right  = nac[bestP+1];
178
179 assert( 2*mid - left - right > 0.0 );
180
181 double shift = 0.5*(right-left) / ( 2*mid - left - right );
182
183 double pEst = bestP + shift;
184
185 // -----
186 // If the range of pitches being searched is greater
187 // than one octave, the basic algo above may make "octave"
188 // errors, in which the period identified is actually some
189 // integer multiple of the real period.  (Makes sense, as
190 // a signal that's periodic with period p is technically
191 // also period with period 2p).
192 //
193 // Algorithm is pretty simple: we hypothesize that the real
194 // period is some "submultiple" of the "bestP" above.  To
195 // check it, we see whether the NAC is strong at each of the
196 // hypothetical subpeak positions.  E.g. if we think the real
197 // period is at 1/3 our initial estimate, we check whether the
198 // NAC is strong at 1/3 and 2/3 of the original period estimate.
199
200 const double k_subMulThreshold = 0.90; // If strength at all submultiple of p
201                                         // this strong relative to the peak, p
202                                         // submultiple is the real period.
203
204 // For each possible multiple error (starting with the biggest)
205 int maxMul = bestP / minP;
206 bool found = false;
207 for ( int mul = maxMul; !found && mul >= 1; mul-- )
208 {
209     // Check whether all "submultiples" of original
210     // peak are nearly as strong.
211     bool subsAllStrong = true;
212
213     // For each submultiple
214     for ( int k = 1; k < mul; k++ )
215     {
216         int subMulP = int(k*pEst/mul+0.5);
217         // If it's not strong relative to the peak NAC, then
218         // not all submultiples are strong, so we haven't found
219         // the correct submultiple.
220         if ( nac[subMulP] < k_subMulThreshold * nac[bestP] )
221             subsAllStrong = false;
222
223         // TODO: Use spline interpolation to get better estimates of nac
224         // magnitudes for non-integer periods in the above comparison
225     }
226
227     // If yes, then we're done.  New estimate of
228     // period is "submultiple" of original period.
229     if ( subsAllStrong == true )
230     {
231         found = true;
232         pEst = pEst / mul;
233     }
234 }
235
236 return pEst;
237 }

```

Share this:

Twitter 1 1 1

Print

StumbleUpon


Facebook

Email

Digg

Reddit

★ Like



One blogger likes this.

Related

Pitch Detection in Flash  
With 2 comments

AudioStretch - real-time audio time-stretching and pitch-shifting in ActionScript  
With 11 comments

An FFT in AS3  
In "Audio"

Pitch Detection in Flash  
With 2 comments

AudioStretch - real-time audio time-stretching and pitch-shifting in ActionScript  
With 11 comments


An FFT in AS3  
In "Audio"



**About Gerry Beauregard**  
I'm a Singapore-based Canadian software engineer, inventor, musician, and occasional triathlete. My current work and projects mainly involve audio technology for the web and iOS. I'm the author of AudioStretch, an audio time-stretching/pitch-shifting app for musicians. Past jobs have included writing speech recognition software for Apple, creating automatic video editing software for muvee, and designing ASICs for Nortel. I hold a Bachelor of Applied Science (Electrical Engineering) from Queen's University and a Master of Arts in Electroacoustic Music from Dartmouth College.  
[View all posts by Gerry Beauregard →](#)

This entry was posted in [Uncategorized](#). Bookmark the [permalink](#).

6 Responses to *High Accuracy Monophonic Pitch Estimation Using Normalized Autocorrelation*

 [radu](#) says:  
July 16, 2013 at 9:55 pm

if only we could have it run in the browser

[Reply](#)



**Gerry Beauregard** says:  
July 16, 2013 at 9:58 pm

I do have an optimized Flash version of the same algorithm running [here](#).

[Reply](#)



**[Tyler](#)** *says:*

August 16, 2013 at 11:04 am

Thanks very much for sharing this. I've spent a fair amount of time playing around with autocorrelation pitch detection for band instruments. You've given me some more ideas to play around with.

A lot of the recordings I have to process have varying amounts of background music bleeding into the monophonic instrument recordings. It seems like having a bias towards shorter periods (higher frequencies) often helps to guard against octave errors caused by this. Thus far I've had the most luck with simply not normalizing, but my guess is that there's probably some middle ground that would improve things.

[Reply](#)



**[Gerry Beauregard](#)** *says:*

August 16, 2013 at 11:15 am

Thanks for the comment! Autocorrelation (whether normalized or not) tends not to work so well unless you've got a truly monophonic source. Even reverb can throw it off. That said, if there's one monophonic source that's much louder than any other instruments, it'll probably work OK, especially if you tweak k\_subMulThreshold. These days, I tend to lean more towards frequency domain methods.

[Reply](#)



**[Tyler](#)** *says:*

August 16, 2013 at 11:35 am

Yes, it seems most solutions go that route nowadays. Unfortunately my knowledge holds me back there. You really have to know what you're doing to manipulate the data into something usable. If you ever write a book on that, I'll buy it!



**[Antoine Rinié](#)** *says:*

November 9, 2013 at 12:44 am

Hi, I'm writing a guitar tuner and had tested various methods , FFT and AutoCorrelation wich gave not enough precision for the guitar frequency range. Your implementation is awesome and works so well ! I did not fully understand it right now but I got the logic thanks to your explanations and comments. Thank you !!

[Reply](#)