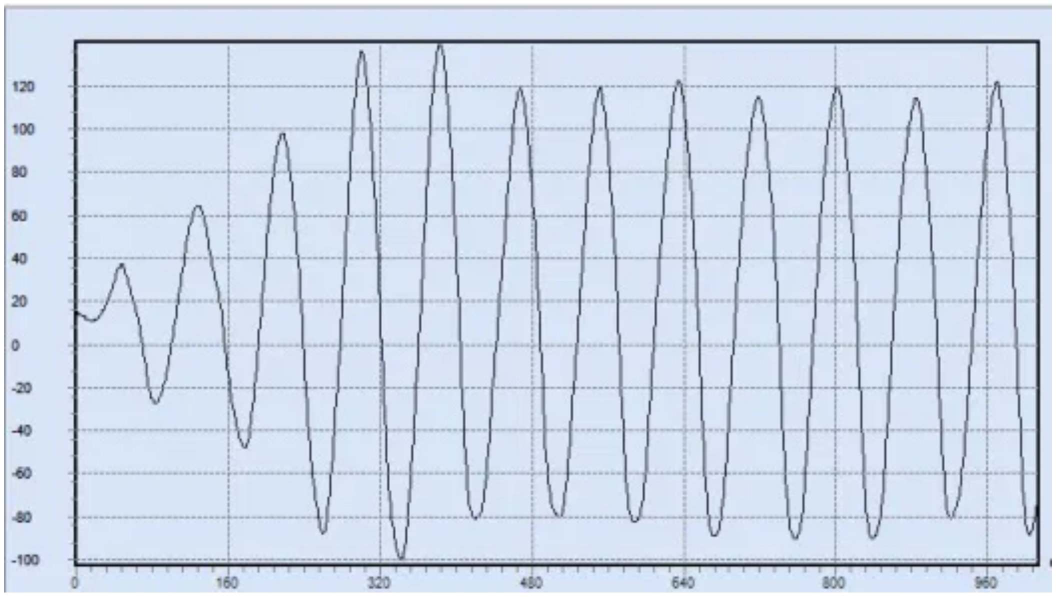
akellyirl.com

Reliable Frequency Detection Using DSP Techniques

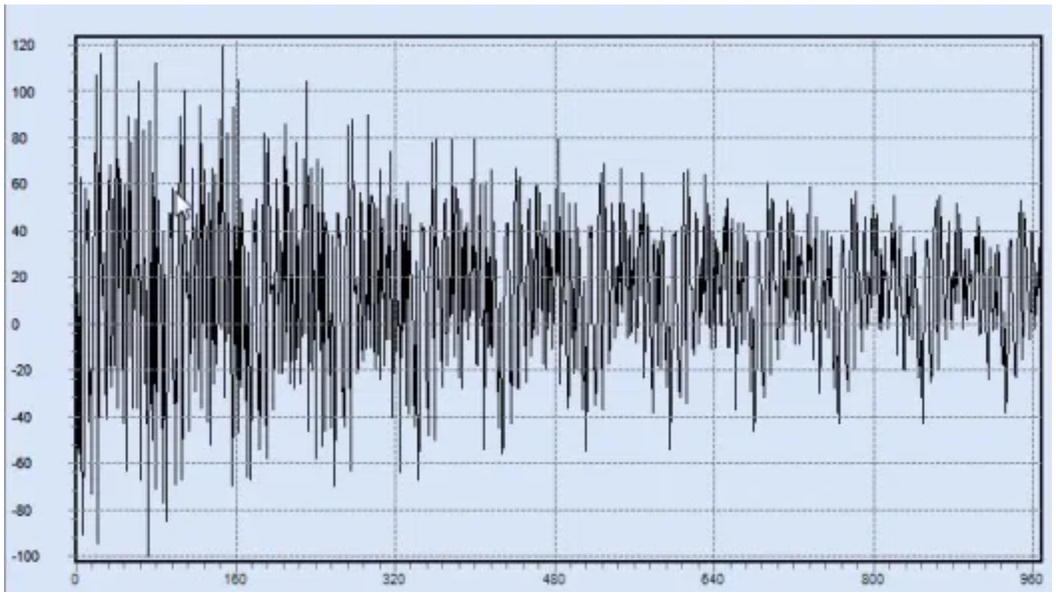
6-7 minutes

Accurate Frequency Detection is important for many projects such as Guitar/Piano Tuners, Vibration Analyzers, Heartrate Monitors, MEMs Sensor Analysis and Laboratory Instruments.

There have been many fine examples of projects that try to solve this problem, for example: Arduino Frequency Detection by amandaghassaei and Arduino Frequency Counter Library (instructables.com).But they all use Time Domain techniques; analyzing the signal for features such as : Zero-Crossings, Peak Detection, Slope Detection etc..



Piano playing Middle-C



Synthesizer Playing Middle-C

Take a look at the Waveforms shown above. One of them is recorded from a Piano playing Middle-C (C4) . The other is from a Synthesizer Playing Middle-C (C4). Clearly any good Time Domain algorithm will work well with the Piano waveform. But the Synthesizer waveform will not be identifiable that way because its very strong harmonic content makes the fundamental frequency undiscernable . It looks impossible to Identify the Frequency of this signal.

It is possible.

Using the technique I'm going to show you it was measured to be **259.91Hz** … only 0.09Hz away from an Exact Middle C Frequency of 260Hz.

You Will Need

You Will
Need

I used an Arduino because it will make a great basis for building a Frequency Detector with Analogue Input such as a Guitar Tuner or Heartrate Monitor.But the principles apply to any platform.

To demonstrate the principles I'm going to use pre-recorded sound files captured as an array in a .h file. So we won't be needing any circuit for the Arduino this time.

Autocorrelation

The trick we use to identify the frequency of a noisy signal is well known in the Mathematical World of Digital Signal Processing (DSP), and is based on some pretty fancy maths. But the technique is not difficult to understand and better still it's super-easy to code. The core of it is just 3 lines of code.

What we need to do is to change the original signal into another one that highlights the periodicity of the original signal. So if it is indeed periodic, then that will stand out in the new signal and then we can measure that in the usual way using peak-detect or zero crossing detect.

What's the magic algorithm that does that?

It's **Autocorrelation**.

Imagine your signal is contained in a window or buffer. Now imagine you have an exact copy of that window or buffer with a time delay.

What Autocorrelation does is to measure the correlation (or similarity) between the signal and its delayed copy each time the copy is delayed by a sample period.

Autocorrelation

See the diagram. When the signal and the copy have no delay they are very similar (i.e. highly correlated) as shown in step 1, and therefore the autocorrelation value for delay = 0 is maximum.Step 2 shows that when the copy is delayed significantly it doesn't look similar to the original in the overlapping area. Therefore the autocorrelation value for this delay is small.

Step 3 shows that when the copy is delayed even more the signal in the overlapping area is very similar to the original because the signal is periodic. Therefore the autocorrelation value for this delay shows a peak.

We can see that the distance in time between the maximum peak at the beginning and the first peak afterwards must be equal to the fundamental period of the waveform.

Now that we've emphasised the periodicity of the signal by Autocorrelation we just need to perform a Peak-Detect to measure the period.

Technically the "similarity" or correlation between the signal and its delayed copy is the **sum of the product** of the two signals.

For the technically minded all the details of Autocorrelation can be found here: http://en.wikipedia.org/wiki/Autocorrelation

Autocorrelation Code

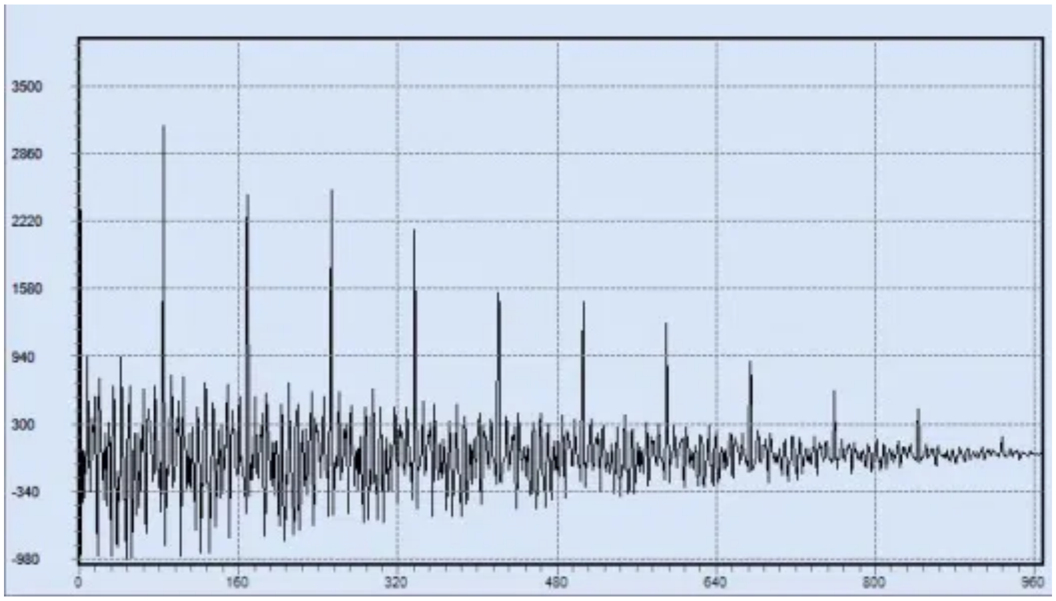
The core of the Autocorrelation Code is very short:

```
  for(i=0; i < len; i++)
  {
    sum = 0;
    for(k=0; k < len-i; k++) sum +=
(rawData[k]-128)*(rawData[k+i]-
128)/256;
  }
```

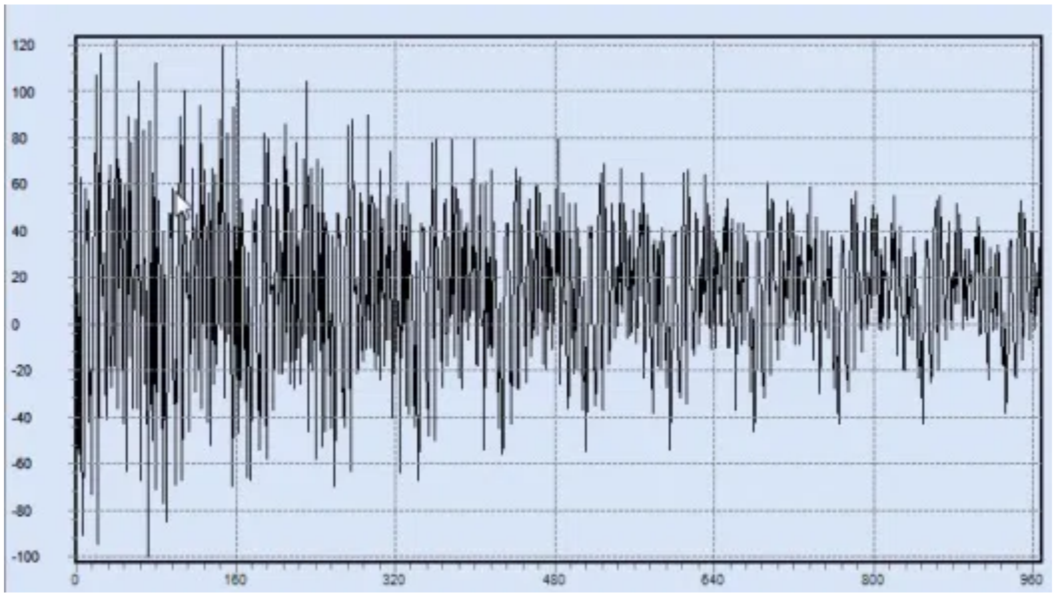
The data is in the **rawData[]** array. We subtract 128 from each value because it’s 8bit unsigned and we require signed values.

The **sum** value is the result of each autocorrelation calculation, i.e. each point of the function. In order to save memory we don’t save the output to an array. We’re going to work on the individual **sum** values to find the first peak and therefore calculate the period.

Sending the sum values out to be plotted, we get the Autocorrelation function shown below. Comparing to the original signal (also shown) it is clear that there is periodicity in the original signal and this has been clearly highlighted by the Autocorrelation function.



Autocorrelation Result



Original Signal

Peak Detect

To detect the location of the first peak after the maximum we use a simple peak detector coded as a State Machine as follows:

```
    // Peak Detect State Machine
    if (pd_state == 2 && (sum-
sum_old) <=0)
    {
      period = i;
      pd_state = 3;
    }
    if (pd_state == 1 && (sum >
thresh) && (sum-sum_old) > 0)
pd_state = 2;
    if (!i && pd_state == 0) {
      thresh = sum * 0.5;
      pd_state = 1;
    }
```

The state machine moves from one state to the next when an event occurs as follows:

STATE0 : Set **thresh** the threshold under which value we'll ignore the data : NEW STATE = 1

STATE1 : look for the signal being above the threshold AND the slope of the signal is positive : NEW STATE = 2

STATE2 : look for the slope of the signal is negative or zero. If so we've **found the PEAK**! : NEW STATE = 3

C4.h File

The C4.h File contains the buffer we're analysing.

You can fill that by reading a Block of Data from the ADC in the Arduino.

Or you can generate the data from a program such as Audacity.

The samples were taken from the extensive collection captured by the University of Iowa Electronic Music Studios.

Here’s all of the Code

You can get all the code over on GitHub