**Low-Powered Frequency Finder**

The following outlines a method for reliably finding frequencies in the 0.5 – 3 Hz range with an accuracy of +/- .03 Hz, while using very low electrical and processing power. Originally for finding heart rate, the same technique may be diversifiable to other parts of the frequency spectrum. The device in its original configuration uses a signal in the 0-5V range and a program running on an Arduino Uno (8-bit 16MHz ATmega328 CPU, 32 Kb Flash memory and 2 Kb of RAM), a low-resource general computing environment. The same algorithm could be deployed on a customized system to achieve reduction in physical size and power requirements, e.g. for a biosensing device. It has applicability in any case where collecting oscillating data with a low power sensor is of interest, including wearable sensors (the use case that led to the creation of the algorithm).

**Hardware Component**

In its original application the frequency finder acts on a signal that has already gone through some basic hardware filtering. This improves accuracy significantly, as the high- and low-pass filters effectively amplify the desirable part of the signal. This signal (seen as a voltage in the range of 0-5V) gets sampled continuously through an analog-to-digital converter. The frequency finder could hypothetically work on a signal in any voltage range, so long as the signal can be discretized into samples with a reasonable resolution. For our purposes the 0-5V range worked nicely with the Arduino Uno we used for processing, and the filtering/ amplification was necessary to clean up a particularly noisy signal.

**Signal Processing**

The original raw voltage signal gets passed through an analog-to-digital converter and discretized into values in the range 0-1000. Sampling must occur at a known, constant rate and the samples should be grouped into arrays of size 2n (*n* is an integer) for processing. High precession at this stage isn’t necessary, so data storage can be kept as 16-bit integers to reduce the strain on memory. (Our original platform only has about 2 kilobytes of program-writeable memory.) In the case that sampling and processing both happen in real time, we found it useful to have a circular array (or queue, or other similar data structure) store the raw samples, and then copy “snapshots” of 2n samples into another array for processing. This seemed to be the most effective way to continuously sample the input while avoiding the danger of running out of memory or overwriting data prematurely. As soon as the program has enough samples to constitute a new snapshot of the signal, the sampling array copies all of its information into the processing array for the algorithm to operate on.

The basic mathematical idea behind this algorithm is the multilevel discrete wavelet transform. In theory this transform is similar to a Fast Fourier Transform (FFT), but localizes its results in time as well as in frequency (FFT provides frequency information with no time localization). In practice, we use it because it is easier to calculate than the FFT while still giving information about signal strength over the range of frequencies (The effective frequency range is between 0 Hz and the Nyquist frequency, given by the sampling rate divided by 2). This information is represented as a series of frequency “bins”, each of which represents a range of frequencies and has an absolute mean value related to their strength. The multilevel part of the transform means that it acts recursively over its results to provide higher resolution (smaller bins) at the low end of the frequency spectrum, where the information we want is located. Although any of several transform types would work for this application, we chose the Haar wavelet because it’s by far the easiest to calculate. The transform acts by recursively taking the sum and the difference of adjacent values from the signal. The differences are stored as one level (one bin) and the procedure repeats on the sums until one final sum remains. Therefore for a window of 2n samples the algorithm can perform at most an n-level decomposition. A larger frequency snapshot, though more accurate, is more time and memory expensive than a smaller one. For our purposes we found that a window size of 256 = 28 samples taken at a rate of 64 Hz worked very well. Sample code (C++) for the Haar wavelet is shown below:

///////////////////////////////////////////////////////////////

/\* transformArray and temp are both integer arrays of length //

WINDOWLENGTH = 2^n = 256 (n=8) //

LEVELS = 7 (must be <= n) \*/ //

//

void wavedec() { //

//

int length = WINDOWLENGTH >> 1; //

for(int i=0; i<LEVELS; i++) { //

for(int j=0; j<length; j++) { //

int sum = transformArray[j \* 2] + transformArray[j \* 2 + 1]; //

int diff = transformArray[j \* 2] - transformArray[j \* 2 + 1]; //

//

/\* store the sums and differences of adjacent values in temp, then //

copy them back into transformArray and recurse on the //

sums \*/ //

//

temp[j] = sum; //

temp[j + length] = diff; //

} //

//

for(int j=0; j < (length << 1); j++) { //

transformArray[j] = temp[j]; //

} //

length >>= 1; //

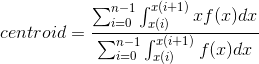
} //

} //

///////////////////////////////////////////////////////////////

\*Note on above code: The Arduino platform uses a slightly modified version of C++ for its sketches. All of the examples in this document use standard C++ commands, including the right and left bit-shifting operators (<< and >>).

Decomposing the signal with this procedure results in up to n levels of frequency information. We take the absolute mean value of each bin to be the weight or “power” of that range of frequencies. The power can (and probably should) be limited based on the target frequency. For instance, if searching for a frequency in the range of 0.5 – 3 Hz it helps the algorithm to set power values to zero for frequencies over 8 Hz. Each power value is paired with the midrange frequency value for its bin. This data can then be visualized as a line plot of power vs. frequency comprised of the points (0,0), (f1,p1), (f2,p2), … (fn,pn). By a slight stretch of the imagination, this plot can also be viewed as a solid, 2-dimensional shape composed of trapezoids. With this perspective in mind, we find the horizontal position of the geometric center of the plot and take it to be the core frequency. The relevant equation is shown below:



This approach may at first seem to be needlessly complicated for the task at hand. However, it yields much more reliable results than taking a weighted average of the frequency spectrum since it automatically accounts for the varying widths of frequency bins.

This entire algorithm makes use of very simple mathematical operations. As noted above it is currently being run on an Arduino Uno, but it could also run easily on an Atmega chip as part of an integrated circuit. There are no constraints on the signal itself, so long as it has a high enough resolution and sampling rate to yield significant frequency information. The final output of the system outlined above is a floating point number giving the calculated core frequency of a window in Hz. Depending on the output medium, it might also be useful to keep a plot of the power vs. frequency plot, to see how the frequency components of a signal shift over time. The disadvantage of finding the average value of a data set in this way is that it gives no indication of the range or standard deviation of frequency intensity, so that a data set with a lot of high and a lot of low values looks the same as one with a lot of medium values. For this reason it is especially important to manually filter out irrelevant parts of the signal, as a spike due to noise in the very high or very low frequency region can throw off the calculated value significantly. Whenever possible we also found it helpful to visualize the signal and resulting transform using more advanced computational tools before passing it through this algorithm. This provides a second safety check against strangely shaped transforms that could result in bad data.