CASCADIA Low-Pass Filter Design

MARITIME SYSTEMS OPERATION, Leidos
Mark Lockwood

May 28, 2014

Contents

1 Summary

The low-pass filtering software handles the sensor types from multiple research institutions, all using diverse sample rates from 40 Hz to 125 Hz. This document describes the properties of the filters common to all sample rates, their generation algorithm, and the tool that can filter software rapidly at sea post-recovery.

2 Filter Description

The following filter attributes are ensured to exist for all sample rates:

- Generalized Linear Phase (GLP) Finite Impulse Response (FIR) Digital Filter
- 3 Hz pass-band cutoff
- 0.25 Hz transition band width
- 140 dB stopband attenuation
- Double-precision taps

The filter length, in taps, is linearly related to stopband level, and also inversely proportional to transition width. Stopband levels are designed to suppress spectral content above 3.25 Hz.

To avoid numerical artifacts, the filter is generated and stored as a double-precision (64-bit IEEE floating point) array, and convolved at double-precision before casting time-domain samples to their original two's-complement integral representation; spectral content in the stop band after the integer cast is generally distributed as quantization noise.

The filter design (this section, Sections 2.2 and 2.3) and the filtering software (see Section 3) is sensitive to four critical details, which are mutually balanced:

- Robustly and rapidly generating filters for multiple sample rates
- Minimizing passband ripple effects to maintain proper magnitude in filtered data
- Eliminating phase distortion by compensating for lag on GLP filter, yielding no shift
- Filtering the multichannel data several orders of magnitude faster than real-time, hence using an FFT-based solution

2.1 Impulse Response

Figure 2.1 plots a 32-second 3 Hz low-pass impulse response for a 50 Hz sample rate as an example.

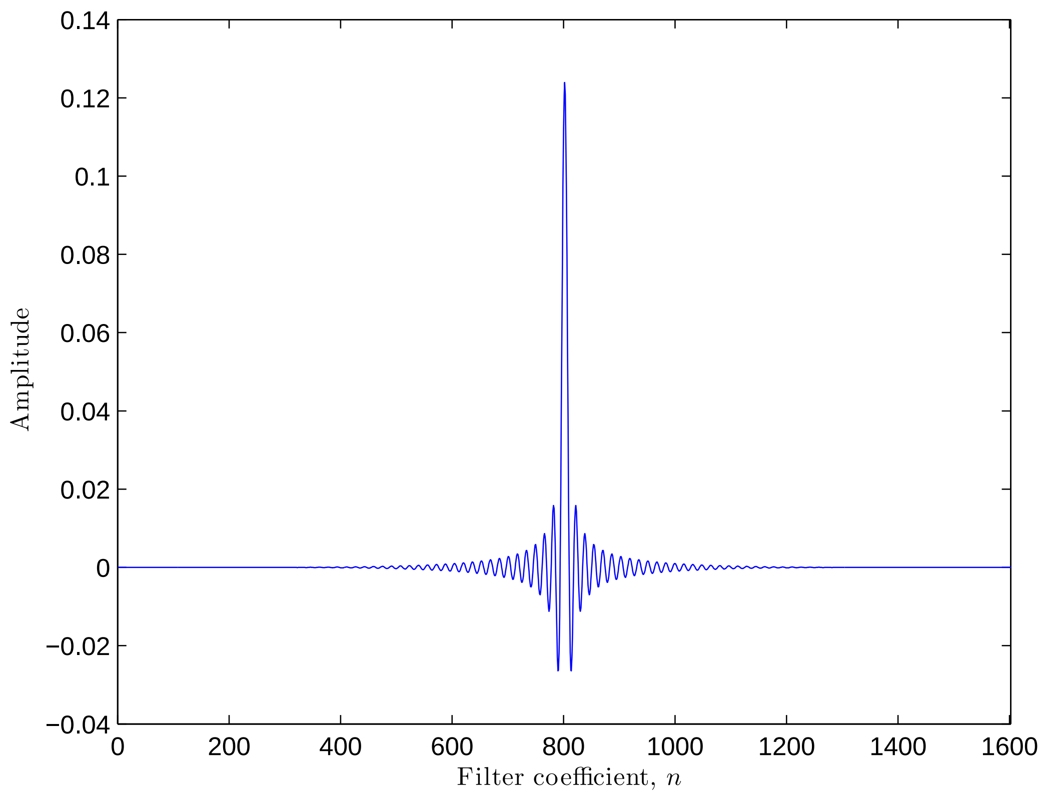

Figure 2.1: Causual Filter Impulse Response For $f_s = 50$ Hz

2.2 Frequency Response

Figure 2.2 first demonstrates the unwrapped phase response of the filter. Since the phase is linear, the group delay can be fully compensated by a pre-padding of samples at the start of the filtering process.

Three plots demonstrate structure in the magnitude of the frequency response. Figure 2.3 shows the filter's small transition width and low stopband level up to the Nyquist frequency. Figure 2.4 highlights the transition band, and Figure 2.5 zooms into the passband.

The resulting 'ripple' is still small for all sample rates: approximately 0.005 dB.

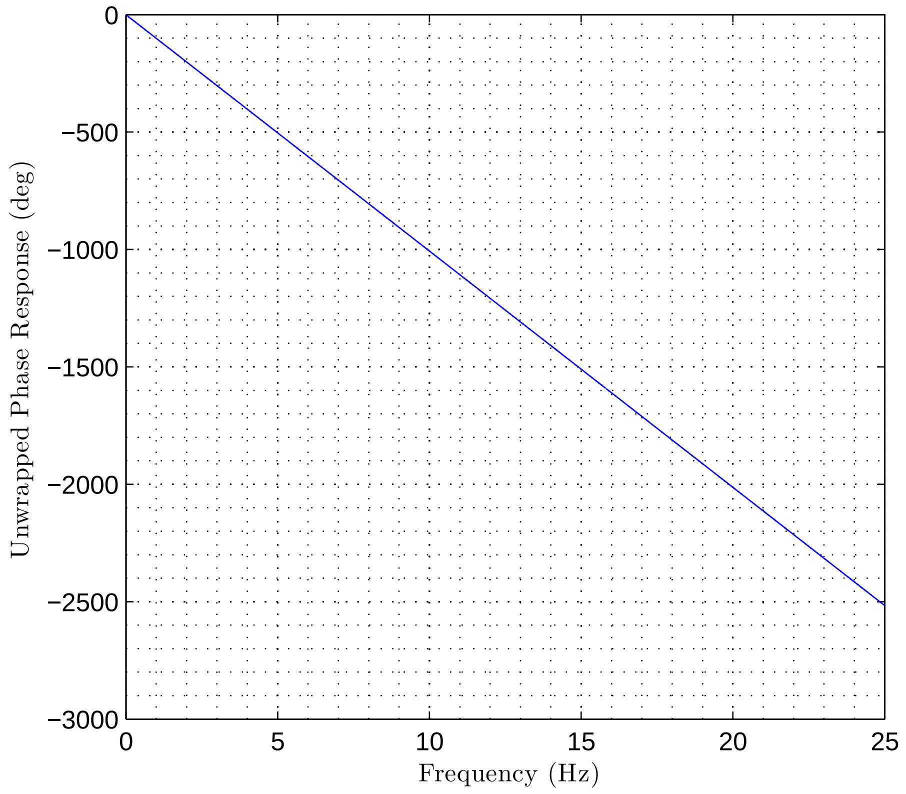

Figure 2.2: Unwrapped Phase Response for $f_s = 50$ Hz, $f_{\text{cutoff}} = 3$ Hz Filter

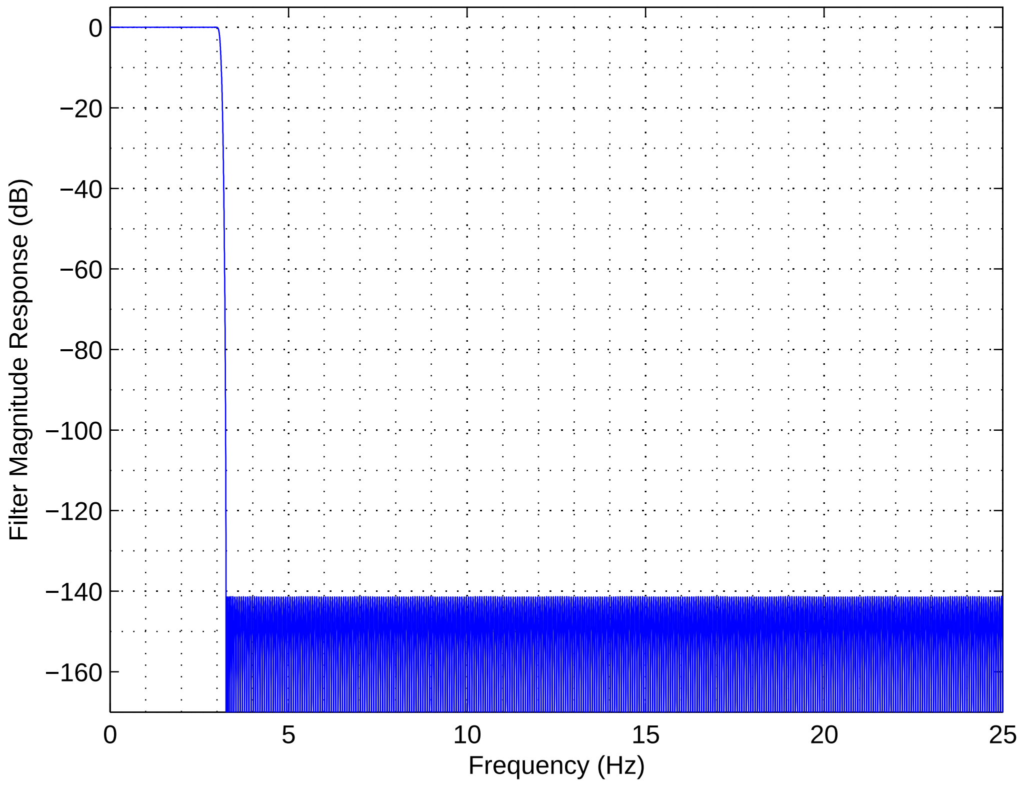

Figure 2.3: Magnitude Response for $f_s = 50$ Hz, $f_{\text{cutoff}} = 3$ Hz Filter

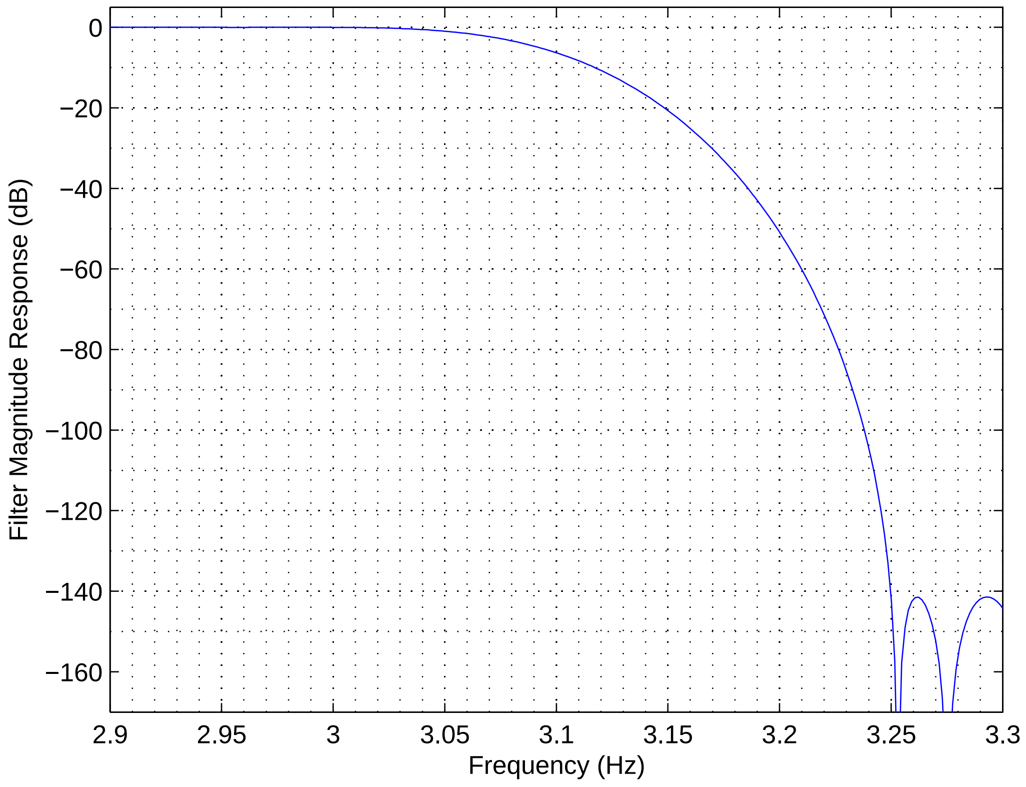

Figure 2.4: Zoomed Transition Band of Figure 2.3

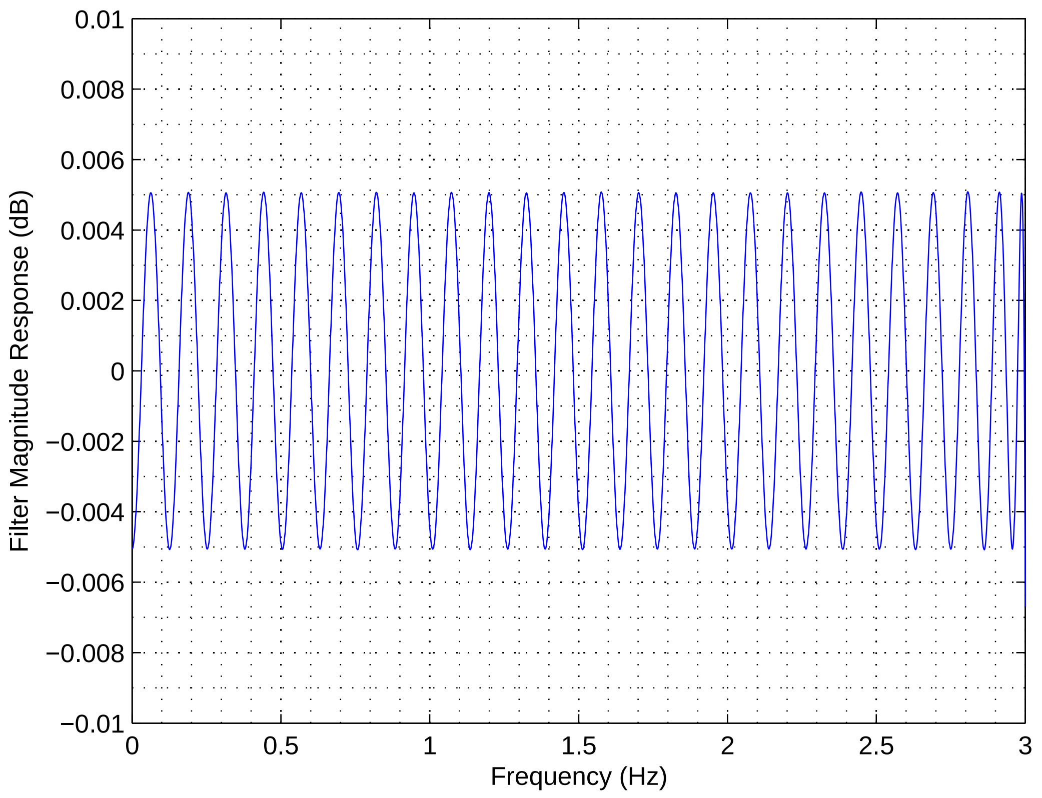

Figure 2.5: Zoomed Passband of Figure 2.3

2.3 Dynamic Generation

Filters are generated using the Parks-McClellan algorithm, a function applicable to MATLAB and Octave. After a filter is generated, it is saved to a binary file with metadata and the filter taps. For convenience, the filename contains the sample rate, which the compiled tool (Section 3.3) loads at run-time, and validates against Mini-SEED header data for every record.

Leidos field filtering laptops include a script for filter generation that runs in Octave. Should a representative encounter an undocumented sample rate, he/she can generate a new filter usable by the filtering tool to ensure that data are securely filtered in the field.

3 Filtering Software

The filtering software is designed to handle arbitrarily large single channel Mini-SEED files. Filtering operates on a snapshot basis, reading in records of data and storing samples in a double-ended queue. When samples have been filtered in a snapshot, they are cast back to their original type into a separate queue. When samples are written on a record-basis, they are discarded from the queue, minimizing memory consumption.

While the delivered software was compiled to natively support 32-bit integer samples stored in Mini-SEED records, the C++ class is templated to support any numerical type supported by C and C++, and has been successfully tested on Mini-SEED files containing floating-point data without compression, in addition to the standard STEIM-compressed and uncompressed integer samples.

3.1 Mini-SEED Library

To avoid distorting any record metadata, the filtering software is implemented with IRIS' *libmseed*. The C LGPL-licensed library was validated to contain no memory leaks for the symbols used, and wrapped within the C++ tool described in Section 3.3.

Mini-SEED Records written to a separate file maintain all original metadata and the same compression algorithm contained in the unfiltered file. Filtered samples are time-aligned with the same sample rate, allowing tools to operate equivalently on filtered sensor data as they would on unfiltered data. The stopband attenuation is also low enough to minimize aliasing artifacts should an end-user desire to downsample the dataset.

For simplicity across all organizations, support is limited to single-channel files. Multiplexed files must be split to files with a descriptive channel-type embedded in the filename.

3.2 Convolution Implementation

The convolution is implemented with an overlap-save frequency-domain convolution. Fast Fourier Transform lengths are preferred to be factorable to very small powers of primes (e.g., 2 and 3), and can be zero-padded separately from the snapshot length.

On startup, the filter is imported and validated. For future circular convolutions, a single symmetric real-to-complex FFT is executed for the filter:

$$H[k] = \frac{\mathrm{FFT}(h[n])}{L},$$

where dividing by L accounts for the inverse FFT factor for every sample, accomplished simultaneously with the circular convolution. The unaliased samples from the circular convolution

$$\hat{x}[n] = \mathrm{FFT}^{-1}\left(\mathrm{FFT}(x[n]) \cdot H[k]\right)$$

are appended to the output double-ended queue. The $\cdot$ operator is the complex Hadamard product, FFT, FFT^{-1} are real-to-complex and complex-to-real (respectively) transforms, and $\hat{x}[n]$ is the circularly-convolved time-domain sample sequence to be pruned for valid samples in overlap-save.

The implementation yields a vast $O(N \log N)$ improvement over normal time-domain $O(N^2)$ convolutions. While a time-domain solution is tolerable for short sample rates, e.g. 40 Hz, it would run slower than real-time at 125 Hz because of the approximately 32 second filter. Benchmarks demonstrate a modern processor can filter a year's worth of data as fast as SATA read performance, taking mere minutes regardless of sample rate.

Whenever a gap is encountered between records, the data restarts the filtration process to prevent any gaps from distorting the filtered output. The resulting filtered data preserves the gaps as originally recorded.

3.3 Compiled Executable

The compiled executable is written in C++ with C libmseed dependencies as described in Section 3.1. It requires a filter filename, an output path, and a sorted sequential list of files to be filtered. The buffer length for the Fourier Transform can be overridden, but in general is tuned for maximum filtering performance.

The list of files are managed as a stack, allowing the long filter to not affect transitions between files. The only filter transient visible in output data should be present at the end of a long dataset, not a file boundaries.

At startup, the input sample queue is populated with a mirror of the first $L/2$ samples. This prevents a zero-padded filter transient at file startup, while still allowing the tool to maintain time-alignment.

3.4 Script

A UNIX *bash* script allows a representative to use the same batch script for any recovery operation. An organization is defined, which then has an associative array, mapping channel types to their sample rates. The following code snippet demonstrates this notion, where `ORG` could be any string-description of the organization:

```
define_ORG ()
{
# channel type 3
    fs["CH3"]=50
# channel type 2
    fs["CH2"]=40
# channel type 1
    fs["CH1"]=40
# channel type 0
```

```
    fs["CH0"]=100
}
```

This example is synthetic, but it demonstrates that the operator enumerates sample rate expectations as documented in the Shipboard Procedures. The lower-level filtering tool will throw an exception should a channel type file contain records with an inconsistent sample rate. This ensures sequences are definitely filtered with the expected low-pass cutoff.

The script iterates over all defined channel types in the associative sample rate map, and recursively searches subdirectories for filenames with matching channel strings. It sorts filenames for a given channel sequentially by time, and passes these to the tool as described in Section 3.3. If no channels are discovered, a warning message says that no channels are found for operator awareness.

4 Example Filtered File

Three time-domain examples of data filtration are presented here to demonstrate filtration continuity, relative sample level preservation, and time alignment. All examples were from a STEIM-I compressed Mini-SEED 125 Hz test file.

Figures 4.1 and 4.2 show two different 8 second (1000 sample) time periods of filtered and unfiltered data. Figure 4.3 zooms on a 1 second region of Figure 4.2 to exhibit a short-time loss of high-frequency data against the original sequence. Figure 4.4 shows the beginning of the sequence, and how the impact of the filter transient is minimized at the start of a filtration process.

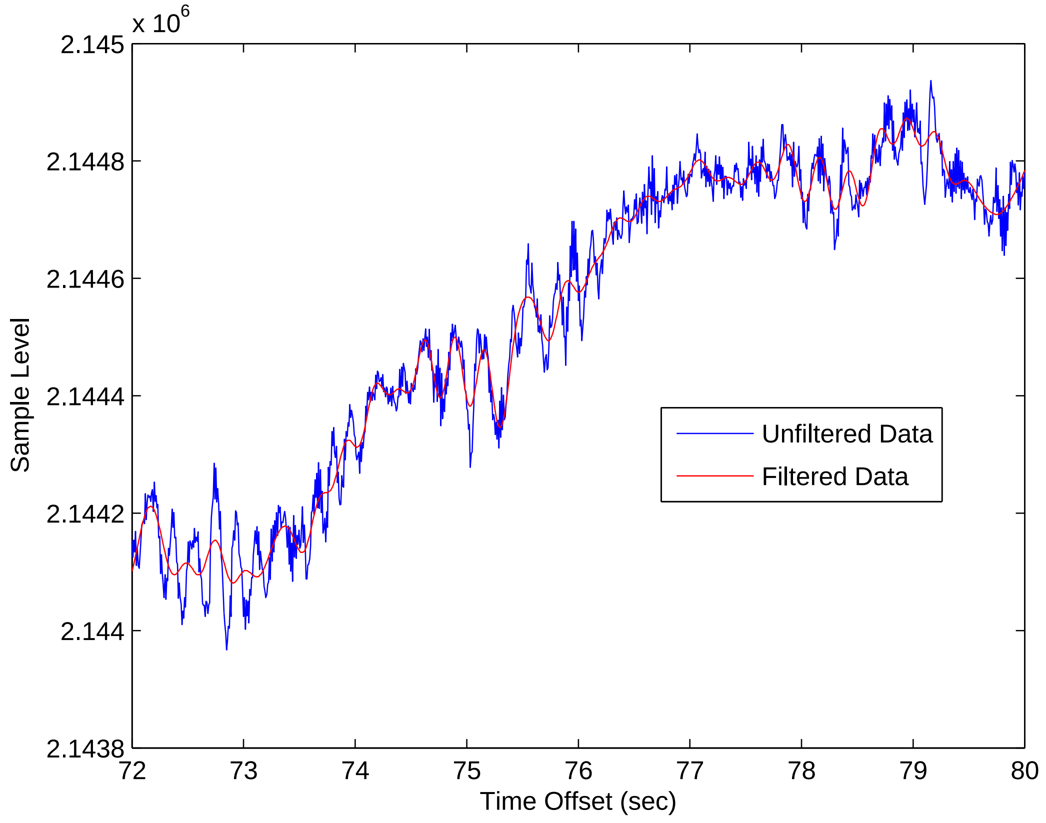

Figure 4.1: Filtered Data Comparison A For $f_s = 125$ Hz

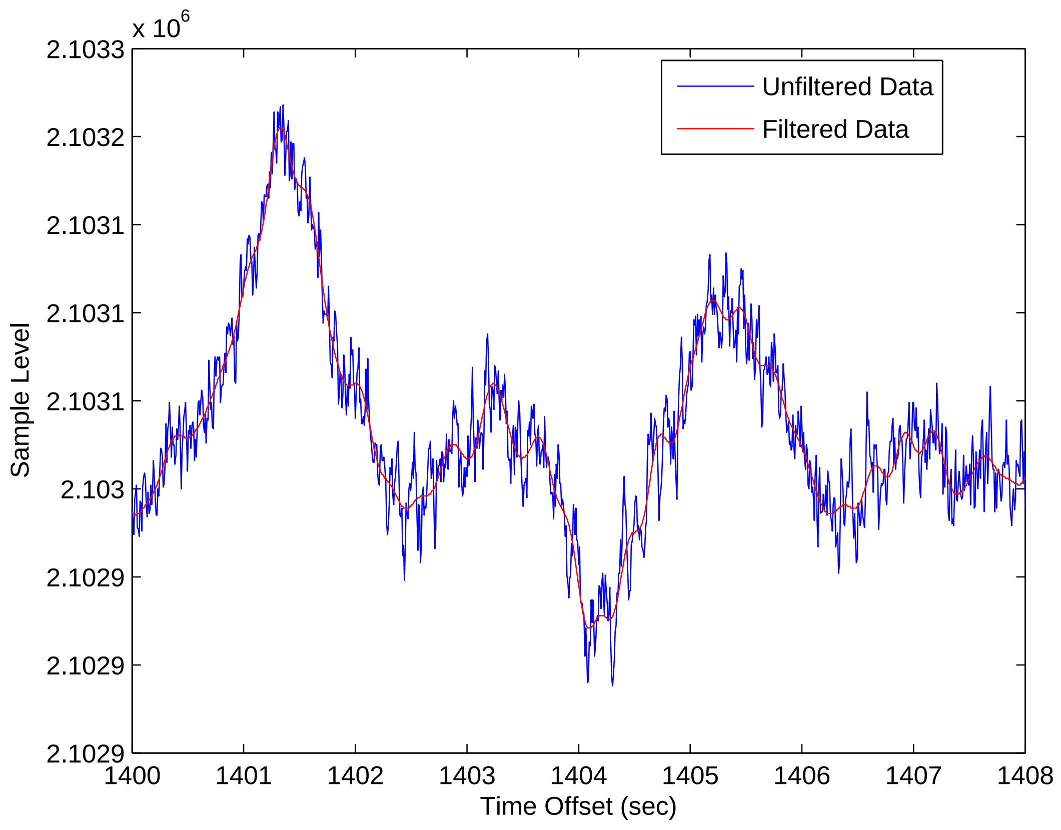

Figure 4.2: Filtered Data Comparison B For $f_s = 125$ Hz

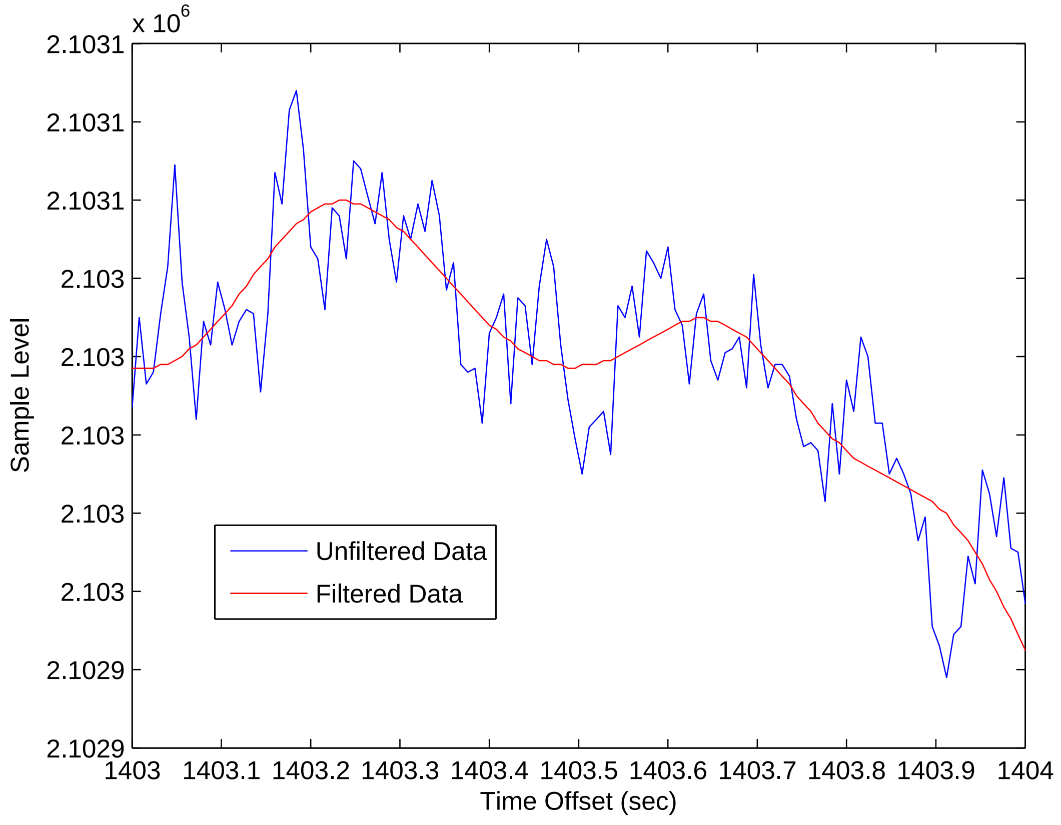

Figure 4.3: Filtered Data Comparison, Zoomed Figure 4.2

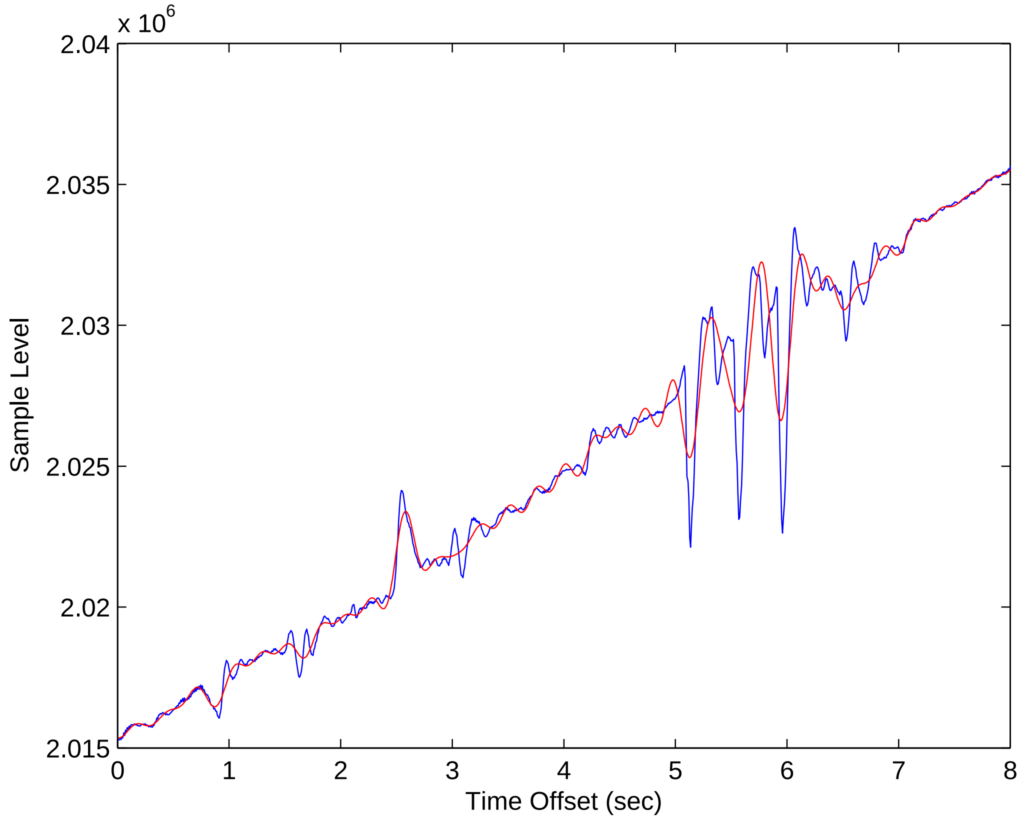

Figure 4.4: Filtered Data Comparison (Startup) For $f_s = 125$ Hz