Signal Processing Report for Straxen

Shuaicheng Li (Sam), Rice AstroParticle Lab

August 2020

1 Introduction

This report will cover the works conducted over Summer 2020 on straxen/XenonnT Data Processing. Corresponding demo notebooks can be found in /signal_processing on the Rice AstroParticle Lab github.

The two major tasks we need to encounter in XenonnT data processing is filtering and compressing.

2 Filtering

Raw signals are captured from the universe and amplified by our hardware. The resulting signal, in other words data, contains both the amplified signal and the channel noise. Therefore our task is to find methods that could effectively reduce the noises while keeping the signals. The signals, are also called peaks as they correspond to particle events in the real world.

2.1 Previous Solution

Prior to this summer, there are two approaches being applied.

2.1.1 Hard Thresholding

The first approach is by hard thresholding: Based on our experience, a datapoint with ADC value larger than 30 is prone to be part of a pulse. Using this property, we can locate the peak positions and keep the datapoints near it. However this method requires the user's experience – that is to determine what level of ADC value is the "cutoff" value.

2.1.2 Matched Filter

The second idea is two use a matched filter. By providing the frequency response of the "desired" signal (the pulse), the filter can look for portions in the given signal with the same frequency response, thereby identifying the peaks. With the modern scipy library, one no longer needs to provide a frequency response – the desired time domain signal is sufficient.

However, the acquisition of the "desired" signal is again, supervised. We need to identify what a peak looks like in the given signal to identify the peaks – sounds conflicted. If we use a general model for "desired" peaks, then we are increasing the chance of ma false judgement.

2.2 Approaches Proposed over the Summer

Realizing the problems in the existing solutions, we want to design a ideally unsupervised, high speed filter for our signals. By saying unsupervised, we mean designing an ideal "black box" – takes minimal instructions and find where the peaks are by itself with high accuracy. Over the summer, we propose the following filtering strategies.

2.2.1 Gaussian Filters

Assuming the channel noises from the hardware to be Gaussian is not the ideal choice, as Gaussian models are just a approximation of random events in the world. But given our current situation, Gaussian guesses are honestly the best.

With such assumption, we can then apply a Gaussian Filter using the corresponding method in scipy. The parameters of the Gaussian Filter is supervised – determined by one experiment that's purly noise (the detector was off and we are only listening the noises from the hardware). The noise experiment was hours long and therefore provides us a rather steady distribution of hardware noises. By feeding the standard deviation of this noise record (on a per channel basis), we can cancel out the their similar counterparts in the given signal with high precision. What's left are the peaks (pulse signals cancelled by std noise; but since signals are way larger than noises, still able to identify them).

This method is of high speed – 1 second processing time for 10000 samples (covers data in one second). On average it's able to detect 96%+ of the peaks (see notebook for analysis).

2.3 Wiener Filter

Wiener Filter is a complex statistical filter, that takes an input signal, and reduce the standard deviation of the trivial parts in this signal. In our case, the Wiener Filter was able "judge" the noise components as noises by itself and reduce their amplitudes.

This method is of high speed – 2 second processing time for 10000 samples (covers data in one second). On average it's able to detect 94%+ of the peaks (see notebook for analysis). Therefore we recommend this as a purely unsupervised, conservative method; and as the backup approach for the more aggressive Gaussian Filter solution above.

3 Compression

Compression is another big issue for us. Since each experiment generate hundreds of GB of data, in less than a day we can accumulate TBs of data, and PBs+ of data in long term. To effectively reduce the size of our data is very important, and since a large portion of our data is comprised of noise, identifying and removing them seems to be the correct way.

Identification of noises (at the same time peaks) is introduced above. No matter what method/filter we finally decide to use, that method is going to remove the zeros in the given signal, leaving the useful pieces in the samples. But can we do more?

Yes, in additional to that, we can apply lossless compressors to the data, obtaining an extra compression. Now let's talk about them.

3.1 Existing Solutions

In straxen the default lossless compressor is blosc. Library blosc also contains access to a few more compressors, namely lz4, bz2, zstd.

3.2 New Solutions Evaluated

Additional to the four compressors mentioned above, we decide to include these additional popular compressors: lzma, zlib, snappy, gzip.

3.3 Performance

Upon examination, we find the following results:

In terms of compression speed, the two fastest compressors are snappy (337M/s) and lz4 (300M/s). blosc (150M/s), zstd and zlib (100M/s) follows, while bz2, lzma and gzip are all around 1M/s.

In terms of compression ratio, lzma provides the best performance: 4x. Then bz2 and gzip: 3.5x; Then zstd and zlib – 3x; then snappy and lz4, 2x. Finally blosc: 1.6x.

Decompression speed are generally 10x the compression speed for the lower speed compressors, while for snappy and lz4 remains a little bit faster.

3.4 Conclusion

From the results above we can see that gzip and bz2 are both slow and low-performance compared to lzma. lz4 should be thrown away since snappy outperformed it. blosc is really bad in compression rate and should not be considered.

What's left are lzma (slow but high compressed); zlib and zstd (moderate speed and moderate compression); snappy (fast compression).

Based on the conclusion above we would propose these:

1. Use lzma for real-time compression for new incoming data (1/5 of the duration of the data).
2. Try zlib/zstd for all cases.
3. Use snappy for our on-disk data (PBs) to at least reduce 50% of current space usage.

Together with filtering, lossless compression algorithms can reduce the original data file down to 5 percent of its original size.

3.5 Ideas: Compressive Sensing

Compressive Sensing is a algorithm that reduces the size of the data if the data is sparse – In our case Yes. It can be interesting and useful in long-term for reduce the size of data (with easy reconstruction).

Works haven been experimented on this idea but the results were disappointing. Future researchers are more than welcomed to try it out. PhD students Daniel and Sina have already met Sophia and can be helpful sources for approaching this problem.

4 To-do

Based on the conclusions above, we expect future researchers to:

1. Try the new filtering methods (Gaussian/Wiener), make them numba functions and merge into straxen.
2. Possibly propose better filtering methods.
3. Meet with the computing team people and propose the addition of lzma, snappy and zlib.
4. Potentially use other languages (other than Python since it's single-threaded) to speed up lzma.
5. Apply high quality Compressive Sensing if possible.