Design and Implementation of Digital Logic Filtration on Open-Source Field-Programmable Gate Arrays

Samuel LeRose

Department of Physics and Astronomy
University of Kentucky
Lexington, KY 40506, USA

1 Abstract

Field-programmable gate arrays (FPGAs) serve as extremely powerful hardware tools used in data acquisition (DAQ) and digital signal processing (DSP) environments thanks to their reprogrammable nature and parallel computation. Unfortunately, this technology is frequently concealed from junior scientists or hobbyists due to their high complexity and/or cost, rendering invaluable research opportunities inaccessible to these individuals. Here I discuss the importance of DSP in physics applications and detail the development of a trapezoidal method for filtering exponentially decaying pulses on an entry-level open-source FPGA. I outline the design considerations for handling data and performing analysis, the breakdown and translation of a simple summation formula for proper implementation in digital logic, and the development and final testing of a complete project on the FPGA itself. The finalized product exhibited promising results, accomplishing nearly 98% accuracy in most tests of pulse peak detection. Likely design flaws were also quick to be discovered, leaving an opportunity for making small improvements to said accuracy. Furthermore, there was success in generating an automated script that allows end users to recreate the complete project from the ground up with minimal experience.

Additionally, I have provided open access to the detailed development package for unfettered use and modification or application to DSP environments, with plans to incorporate new DSP projects and perform upkeep.

2 Introduction: Why Do We Use Digital Signal Processing?

In many fields of modern physics, the results of experiments are often depicted in the form of analog (or digital) waves, such as those found in radiation experiments used to study nuclear structure. In these cases, a photon deposits energy over time as it interacts with a detector, generating an electrical signal that increases in strength as more charge is collected. The current produced by the detector is then integrated by a preamplifier (preamp) resulting in a step-like or exponentially decaying pulse which is the focus of the scientific analysis [1]. However, when it comes to this analysis, one often runs into three common issues: ballistic deficit, signal-to-noise ratio, and pulse pile up.

Ballistic deficit refers to the difference in pulse height between the exponentially decaying pulse from the preamp and the output of the pulse shaper [1, 2]. This occurs when the peaking time, the time it takes for a pulse to reach its maximum value from the moment the charge collection process begins, of the shaped pulse is too similar to that of the preamp [3]. In Figure 1 it can be seen how a long shaping time provides an opportunity for the full charge to be pro-

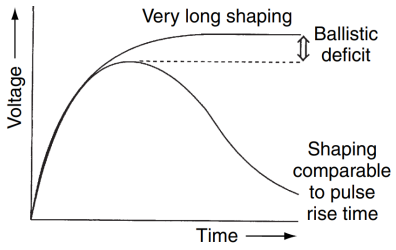

Figure 1: Ballistic deficit between sufficiently long shaping time and a shaping time too similar to that of the pulse rise time (not shown) [2].

cessed by the filter while the shorter shaping process does not.

The signal-to-noise ratio (SNR) of a system with signal amplitude S and noise N is given by $SNR = \frac{S^2}{(\sigma_N)^2}$ where σ_N is the standard deviation of the noise [4]. SNR affects the quality of a system's signal transmission and subsequent analysis, with high SNR meaning clear, easy to read signals and low SNR meaning obscured, hard to distinguish signals.

Pulse pile-up occurs when two or more pulses arrive from the preamp within the resolution time of the filter. In these situations, the system cannot measure the heights correctly and the response of the system is corrupted. When two such pulses occur very close to one another, the system will not take the pulses as separate events and will instead take the sum of the first pulse and a portion of the second pulse [1, 2, 5]. This specific type of pulse pile-up is generally referred to as peak pile-up and can be seen in Figure 2.

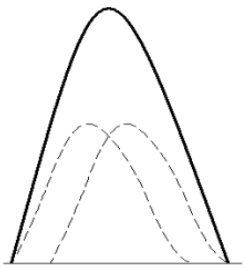

Figure 2: Peak pile-up of close-occurring pulses [5].

Digital signal processing (DSP) filters are designed to reduce the effects of these issues and allow for easier and more accurate analysis, often containing variables that allow for a trade off between optimizing one parameter while another becomes more prominent. Applying filters requires passing the signal from the preamp to an analog-to-digital converter (ADC) which sends the now digital signal to a field-programmable gate array (FPGA) for processing, the output of which is referred to as the "response" of the system (this kind of system will be referenced in the remainder of this paper).

This paper primarily details the development and testing of a simple trapezoidal filtering project on open-source data acquisition (DAQ) hardware with additional discussion of relevant background information distributed throughout. For this process of development and testing I rely on the Red Pitaya STEMlab 125-14 FPGA for its affordability, portability, expansion options, and remotely accessible user interface.

3 A Crash Course on FPGAs

In the early 1990s, FPGAs were primarily used in the telecommunications industry, but they quickly solidified their spot in DSP in the early 2000s and eventually became a staple of DAQ instrumentation [6]. At face value, the design of an FPGA is rather straightforward: a large network of logic blocks joined together via interconnects that can be configured/programmed to complete a desired computational function. The term "program" is used loosely when referring to FPGAs as the configuration of logic blocks is set using hardware definition languages (HDLs), which are functionally different from (however, structurally similar to) software programming languages (SPLs) like C. More specifically, SPLs utilize the predefined networks and functionality of CPUs to execute any code while HDLs restructure the hardware of an FPGA chip to perform a specific job repeatedly. For the STEMlab, the configuration

is performed using the Xilinx Vivado application and a combination of intellectual property (IP) cores and custom modules. Each successful configuration, often called a project, has a corresponding bitstream that can be generated and sent to the FPGA for implementation.

It is also important to understand why FPGAs are preferred when it comes to DSP and the difference in how they handle data compared to a CPU. FPGAs can establish connections and interact directly with interfaces, sensors, storage systems, displays, and other IO, drastically reducing the latency of data movement compared to CPUs which must incur heavy processing costs to maintain communication with external equipment [7]. And while modern CPUs can execute several billion processes per second across multiple cores, all executions within a CPU must be performed one-by-one sequentially, meaning any delay can result in corrupted information or missed timings [8]. By contrast, an FPGA can perform many processes and computations in parallel, streamlining data processing, storage, and distribution, and in some use cases exceeding the throughput of CPUs by nearly $150x$ [7].

4 Trapezoidal Filtering Method

The trapezoidal filtering method, aptly named for the shape of the filter's response, is designed to process an exponentially decaying pulse of the form $V(t) = V_0 e^{-t/\tau}$, where $V(t)$ is the input pulse at time t, $V_0 = V(0)$ is the initial pulse amplitude, and τ is the corresponding time constant. First introduced by Valentin T. Jordanov and Glenn F. Knoll in 1994, the trapezoidal filter operates by convolving an incoming exponential pulse and the "trapezoidal shaper": a combination of a rectangular function and two unit slope truncated ramp functions, each at different offsets and delays [9]. A simple diagram of this convolution can be seen in Figure 3.

Mathematically, the output of the trapezoidal system can be determined in terms of the sampled input pulse as

$$Y[n] = \sum_{i=0}^{n} \sum_{j=0}^{i} (a^{K,L}[j] + a^{K,L}[i]M) \tag{1}$$

where $Y[n]$ is the filter response, the function $a^{K,L}[n]$ is given by

$$a^{K,L}[n] = X[n] - X[n-K] - X[n-L] + X[n-K-L], \tag{2}$$

and the multiplication factor M by

$$M = \frac{1}{\exp(T_{clk}/\tau) - 1}. \tag{3}$$

For values of $\tau/T_{clk} > 5$, M can be approximated as as $M \approx \tau/T_{clk} - 0.5$ [10].

In Eq. 2, $X[n]$ is the sampled input pulse and K and L are delays such that $X[n-K]$ is the sampled input pulse at a time K units after n [1, 9]. The lesser value of K or L is the duration of the rising and falling edges of the trapezoidal output, and the length of the flat

top is given by $m = |L - K|$ (where if $L = K$, the system results in a triangular shape) [1, 9]. For consistency, all of my project designs and tests maintain values of K and L such that $K \leq L$, and therefore, K is the duration of the rising and falling edges and L determines the length of the flat top.

A careful balance of the delays allows for the optimization of the parameters discussed in the previous section. A shorter rise/fall time allows for quicker pulse processing and a higher pulse count rate, thereby reduced pulse pile-up, while a longer rise time improves the SNR by taking a larger sample size [1]. Likewise, the effects of ballistic deficit can be mitigated by ensuring the duration of the flat top m is greater than the peaking time of the preamp [1, 3].

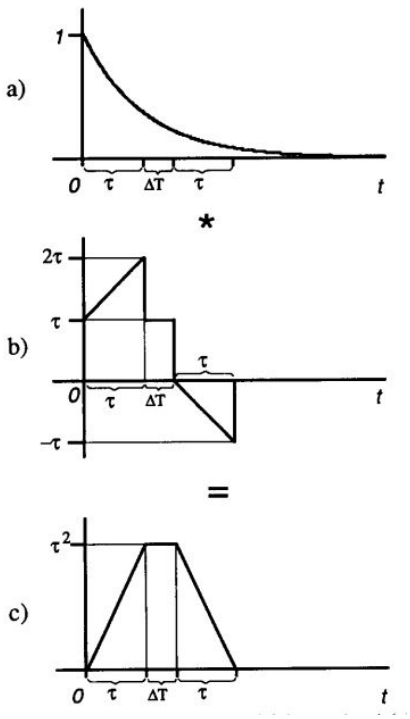

Figure 3: Convolution of (a) an exponential input pulse with (b) a trapezoidal shaper and (c) the output of the system [9].

5 Digital Implementation of the Trapezoidal Filter

Throughout this section I will frequently refer to the Python and Verilog modules I have developed. For the sake of convenience and organization I have constructed a GitHub repository that contains all the up-to-date information which can be accessed at `https://github.com/sslerose/RedPitaya-DSP`. However, to ensure this paper is not rendered useless when the repository is updated or otherwise changed, a copy of the modules can be found in the final section labeled "Code Reference".

5.1 Verilog Basics

Before discussing the construction of the module itself, it is important to understand the data types and assignments of Verilog:

`parameter` A constant value within the module. Can be changed when instantiating the module.

`input/output[b:a]` An input or output of the module with bits numbered `a` to `b` from right to left. `a` is called the least significant bit (LSB) and `b` the most significant bit (MSB).

`wire` An interconnect between two parts of a module and the default data type of inputs and outputs. Must be

driven by a continuous assignment of data (using an `input/output` or `assign` statement) which is stored for a single clock cycle.

register (`reg`) A data storage element that maintains is data value until it is reassigned using a blocking or non-blocking assignment.

blocking assignment (`=`) Used in assigning values to registers sequentially.

non-blocking assignment (`<=`) Used to schedule the assignment of values to registers without preventing succeeding statements from executing.

5.2 Design Considerations

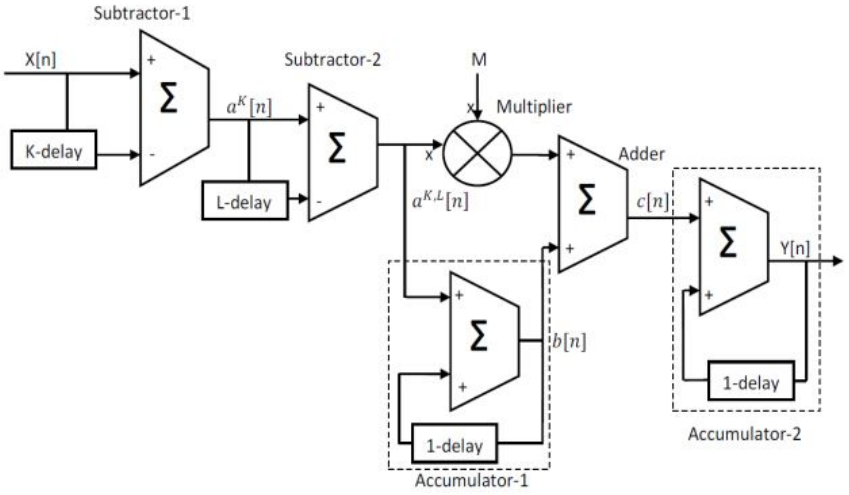

Figure 4: Simple block diagram of convolution-based trapezoidal filter [1].

The goal for applying the trapezoidal filter digitally was to develop a high-performance and resource-efficient Verilog module that could be integrated into the existing architecture of the STEMlab FPGA. As a visual aide for this process, the trapezoidal filter can be represented by the block diagram in Figure 4. In the context of DSP, the block diagram depicts the flow of data from a sampled signal where the data present before each block is processed simultaneously after a given period [1]. One can think of this procedure like a constant stream of manufacturing products being passed through machines on a conveyor belt where each machine acts on its respective product at the same time.

The first design consideration was to optimize the filter arithmetic such that it could be more easily employed on the FPGA. According to another of Jordanov and Knolls' publications, Eq. 1 can be broken down into

$$b[n] = b[n-1] + a^{K,L}[n], \tag{4}$$

$$c[n] = b[n] + Ma^{K,L}[n], \tag{5}$$

$$Y[n] = Y[n-1] + c[n] \tag{6}$$

where $b[n]$ and $Y[n]$ represent the accumulators and $c[n]$ the adder/multiplier in the block diagram above [10].

Furthermore, the multi-operation delay line $a^{K,L}[n]$ (Eq. 2) can be split into two single-operation delay lines (or subtractors), the first being an intermediate equation given by

$$a^{K}[n] = X[n] - X[n-K] \tag{7}$$

and the second being a redefinition of Eq. (2) given by

$$a^{K,L}[n] = a^{K}[n] - a^{K}[n-L]. \tag{8}$$

This secondary breakdown reduces the number of data buffers and arithmetic operations from three to two, lightening the hardware utilization. Consequently, equations 4 - 8 now follow the structure of the block diagram in Figure 4.

The second consideration was to choose an efficient method of temporary data storage, called a buffer, for the delays in Eqs. 7 and 8. For this process I opted to use a ring buffer due to its low memory usage and relative simplicity. In a typical ring buffer, read and write commands are enabled as needed and the distance between the read and write locations is varied, resulting in a very flexible data storage method. However, a common flaw occurs when ring buffers have insufficient size, resulting in the potential for old data to be overwritten before it is read, causing data corruption. Fortunately, my usage of the buffer maintains a constant distance between read and write (given by the values of K and L), meaning this shortcoming was not of concern for use in the filter so long as the buffer had a greater number of addressable locations than the value of L (remembering $K \leq L$).

The final consideration was the method by which to extract the peak value as determined by the trapezoidal filter. To avoid extraction from the edges of the flattop, which are rounded when a pulse has non-zero rise time (see Figure 5), and to reduce the probability of extracting a large peak caused by noise, which would occur if one simply looked for the greatest value of the response, it is best to choose the middle-most value of the filter's flattop. Knowing the flattop has length $m = L - K$ and the length of the rising and falling edges is given by K, the middle is given by $2K + m = \frac{K+L}{2}$. Additionally, peak detection should only occur when the incoming signal has passed a given threshold, thus indicating that the incoming signal is above the standard noise levels and a pulse is being processed. Neglecting this threshold will lead to erroneous values that correspond to noise peaks rather than pulse peaks.

5.3 Python Simulation

To test the correctness of my Verilog modules, I created one Python module (`trap_filter`) that modeled the trapezoidal filter and another (`e_exp_decay`) that constructed arrays of simulated exponentially decaying data which could be fed to the filter module.

Given I was not constrained by hardware limitations or the rigidity of the Verilog language, I was able to use Eqs. 2 and 4 directly for the delay line and first accumulator, respectively, employing the "roll" function from the "Numpy" Python package for the individual delays, and then combine Eqs. 5 and 6 for the second accumulator. The respective inputs and outputs are listed at the top of these modules in the repository.

When working with the filter module, I found that the peak of filter response was scaled up compared to the peak of the exponential input, but that Jordanov and Knoll's publication did not explicitly mention a scaling factor. According to a WordPress article by "GoLuckyRyan", the trapezoidal output should be divided by a factor of KM to remove this gain [11]. When I tested this assertion using the Python modules, I observed a one-to-one linear relationship between the gain of the filter and KM, yet there existed a non-zero y-intercept within this relationship that was consistently equal to the value of K. When examining this new relationship between the gain and $K(M+1)$, the slope was linear and the y-intercept was zero. Therefore, Eq. 1 should be redefined as

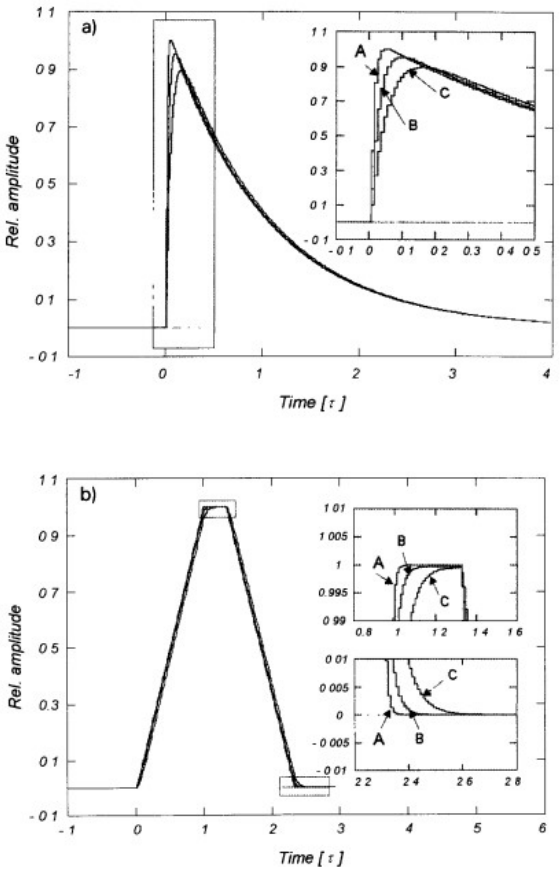

Figure 5: Exponential pulses with varied rise times (a) and their corresponding filter responses (b) [9].

$$Y[n] = \frac{1}{K(M+1)} \sum_{i=0}^{n} \sum_{j=0}^{i} (a^{K,L}[j] + a^{K,L}[i]M). \tag{9}$$

The recursive equation cannot be redefined in the same manner as the recursion depends on this gain for correctness, so Eq. 6 must stay as is. This gain was accounted for in a future Python program that interfaces with the FPGA, which is discussed later in the paper.

5.4 Verilog Modules

The bulk of the remaining development was spent creating the Verilog modules themselves, these of which have the same name as the Python modules as they perform the same job.

The design and testing was performed using Intel ModelSim rather than Vivado as I needed only make a controlling Verilog module rather than a behavioral script to assess the filter and buffer modules. In actuality, Vivado could have been used as the sole program for design and testing, but this would have necessitated knowledge of C programming and hardware behavioral script structure.

Because my goal was to port the simulated modules onto an FPGA, I chose to use the requisite FPGA components that are not necessarily required within ModelSim:

`clk` Input timing interface controlling the synchronous execution of modules and cores in Vivado.

`aresetn` Active low reset argument used to bring a core or module to its initial state.

`s_axis_tdata` AXI-Stream (AXIS) slave interface used to receive data as a sequential stream.

`m_axis_tdata` AXIS master interface used to send data as a sequential stream.

The STEMlab's ADC converts two 14-bit wide analog signals into a combined 32-bit wide AXIS signal: `m_axis_tdata = [{16-bit IN2},{16-bit IN1}]`. Because the filter is designed for a single signal, it need only take the 16 lower order bits from the ADC. The parameter for this value (`AXIS_TDATA_WIDTH`) that heads each module exists for convenience and ease of modification.

5.4.1 Exponential Decay Generator

The IO of the `e_exp_decay` module contains a pulse amplitude, decay factor, trigger, and AXIS master that sends the exponential pulse. For the sake of simplicity and testing, the exponentially decaying pulse contains no noise and no rise time.

When triggered, the module calculates the pulse using fixed-point*, recursive multiplication (FPM), and outputs using the AXIS interface. For the case of the STEMlab, its ADC/DAC is 14 bits wide, so to create a decay rate of $e^{-t/\tau}$†, the decay factor should be calculated as

$$\texttt{decay\_factor} = \texttt{round}(2^{14}e^{-t/\tau}) = \texttt{round}(16384e^{-t/\tau}). \tag{10}$$

At the trigger, the current value (`current_value`) of the exponential pulse is populated with the initial pulse amplitude. Afterwards, an intermediate register is used to handle the multiplication of the current value and the decay factor, where the successive value is determined by dividing the intermediate register by 2^{14} using a right bit shift. As a mathematical recursive equation, this process is given by

$$\texttt{current\_value}[n] = \texttt{current\_value}[n-1] * \texttt{decay\_factor}/2^{14}. \tag{11}$$

*A method of allowing non-integer multiplication in binary values. Because a binary register cannot hold a decimal value, the decimal value should be scaled, the multiplication be performed, and then the result be descaled.

†In the case of digital logic, the values of τ and t are given in units of T_{clk}.

It should be noted that ModelSim can utilize the `$exp` function to calculate $V[n] = V_0 e^{-t/\tau}$ and avoid FPM. However, the `$exp` function is not implementable on hardware, thus necessitating the fixed point method. Initially I tested the trapezoidal filtration project using the `$exp` value and verified the correctness of the filtering module (`trap_filter`) using the previously discussed Python program. After this verification process, I switched to the fixed point method to more easily understand the method implemented on the FPGA.

5.4.2 Pulse Generator

The IO of the pulse generator (`pulse_gen`) module contains an incoming trigger and outgoing pulse. When the pulse generator is triggered (i.e., turned on), it generates a single clock cycle high signal that forces the exponential decay generator to produce a single pulse of its exponential signal. Including this module allowed for easier handling when commanding the FPGA and prevents multiple pulses from occurring in series.

5.4.3 Ring Buffer

The IO of the ring buffer (`ring_buffer`) module contains read and write controllers, delay value, data to write to the buffer, and data to read from the buffer. The buffer itself is constructed with `BUFFER_LENGTH` addressable locations, each of which is a register of width equal to the AXIS width. I also established `clog2(BUFFER_LENGTH)`-bit* wide write and read pointer registers (`wr_ptr` and `rd_ptr`, respectively), the widths of which are selected to allow the maximum value of the pointers to index the last location of the buffer.

To traverse through the ring buffer, `wr_ptr` is iterated by one at each clock cycle. Once `wr_ptr` reaches its maximum value, the next iteration will force the pointer to experience binary overflow and the value to be set to zero. `rd_ptr` follows a similar pattern at the overflow boundary, but in the opposite direction using 2's complement†, a method of writing negative numbers in binary. Any negative number is simply the 2's complement of its positive value, so when dealing with subtraction we find $A - B = A + [2Comp.(B)]$. Figure 6 depicts an example of this process. Note that `rd_ptr` contains a +2 within the module. This is present to remove the

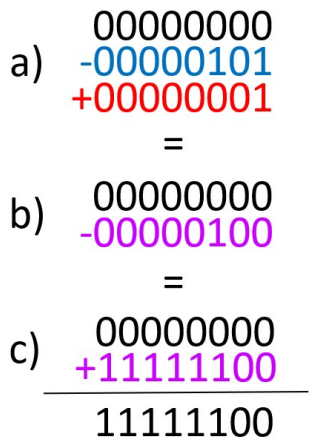

Figure 6: Bit overflow example. Here, (a) is initial state, (b) is reduction to $A - B$, and (c) is final reduction to $A + [2Comp.(B)]$.

*The function $clog_2$, or "the ceiling of log base 2", takes the log base 2 of a value and rounds the value up to the nearest integer.

†To convert a binary number to its 2's complement, swap all 1s to 0s, and vice versa, then add 1 to the result.

inherent delays caused by data transfer between each clock cycle and was primarily discovered through trial and error.

5.4.4 Trapezoidal Filter

The filtering module (`trap_filter`) contains an AXIS slave that receives an incoming signal and a master that sends the outgoing filter response. The remaining IO are for the K- and L-delays and the multiplication factor M. Two wires exist to interface with the `ring-buffer` module for the delay lines (Eqs. 7 and 8) and registers are defined for handling the data of Eqs. 4 - 8. Another set of registers exists to handle the ring buffers, and the buffer instances themselves are defined afterwards.

An `always` block manages the operations of the filter module as well as resetting all registers when prompted. The buffers for the K- and L-delay lines (Eqs. 7 and 8) have their reading and writing enabled based on the value of the buffer iterate (`buff_iterate`) register such that neither is reading from an empty memory location. Once the buffer for the L-delay line has valid data being read and transferred, the filter arithmetic begins.

The operations in lines 120 to 125 of the filter module execute Eqs. 4 - 8, where Eq. 5 is broken into two steps (lines 122 and 124). When an incoming signal is constant (or near constant for noisy data), line 120 equates to zero (or near zero for noisy data), resulting in the remaining lines also trending around zero. This property of the trapezoidal filter's first delay line allows for baseline removal, ensuring the peak of the filter response is always measured from near zero.

5.4.5 Peak Detector

Based on the considerations in Section 5.2, the peak detector (`peak_detector`) module has IO consisting of an AXIS slave to receive the filter response and inputs for the values of a threshold and the K- and L-delays. To extract the middle-most value as discussed in the aforementioned section, the module waits until the incoming filter response passes a given threshold and then waits until the middle-most point in the response to record its value.

5.5 Vivado Project and Jupyter Notebook

To program the STEMlab FPGA, I created a Vivado project designed to generate a simulated exponential pulse, perform trapezoidal filtration on the pulse, and analyze the filter's response. Within the "prj" directory of my GitHub repository, there exists an automatic script that performs the entire building process of the filter project, including producing outside port connections, adding Xilinx and Red Pitaya IP cores, adding the custom modules*, and establishing the proper constraints, properties, and wiring for the ports, cores, and

*Because the ring buffer modules are instantiated within the filter module, they do not have an associated block in the project, and are instead embedded within the filter's block.

modules. The script does not automatically generate the bitstream necessary to command the STEMlab, but this allows for an end user to view and edit the project before beginning that process.

The two predominant cores and connections of the project are the general purpose input/output (GPIO) cores. Each GPIO core was given an address and connected to the ZYNQ processing system, allowing communication between the filter project running on the FPGA and a Jupyter Notebook running on the CPU, the later of which is managed through a local browser connection to the STEMlab.

The Python program within the Notebook (`trap_nb`) is responsible for setting the parameters of the signal generator, filter, and detector modules, and reading back the peak value from the latter. Although each IO of the GPIO cores are a maximum of 32-bits wide, the parameters can be strung together using shifts and sliced before being sent to their respective destinations. For example, the `trap_gpio` core has its first 30-bit output assigned as `gpio1_data=[{14-bit Kdelay},{14-bit Ldelay},{2-bit reset}]`, and slicing the lower 16 bits issolates the value of the K-delay.

The 2-bit reset control within the `gpio1_data` output is used to reset the custom modules, which is imperative to proper function if any parameters of the modules are changed. In its initial state, the 2-bit reset is zero and the modules are in a state of reset (their resets are active-low), and adding one to `gpio1_data` takes these modules out of reset. This same process is performed for the `peak_gpio` core to trigger the signal generator, except the trigger is active-high.

6 Results

Using the Jupyter Notebook program, I started by assigning the exponentially decaying pulse with $sig\_amp = 1000$ and $\tau = 10$. The filter was then given values of $K = 10$, $L = 15$, and $M = 10$ and the peak detector was given a threshold value of 1000. After triggering the signal generation, the STEMlab produced a peak filter response of 1019.229, meaning the trapezoidal filter produced an error of approximately 1.92%.

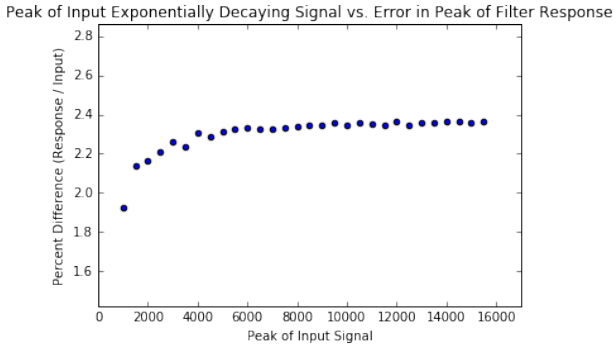

Figure 7: Peak of generated exponentially decaying pulse versus the percent error in its respective filter response.

To test whether this error was consistent for multiple different exponential pulse amplitudes, I created an array of amplitude values from 1000 to 16,000 in steps of 500 and performed the same test again with each value using a loop. I then plotted the peak of the generated exponential pulses versus their respective filter response percent errors, as can be seen in Figure 7. Based on the graph it can be observed that the error initially increases with respect to the value of the pulse amplitude, but then roughly levels

off around 2.38%.

7 Conclusion and Next Steps

The results from the previous section are generally acceptable and indicate the filtering project running on the STEMlab FPGA is operating with relative accuracy. However, there clearly exist some issues that are resulting in the discrepancy between the heights of the pulse peak and the filter response peak.

After comparing the results of the Python simulation to those of the STEMlab filtering project, it seems that the bulk, if not all of the discrepancy stems from inherent rounding error. As was discussed in Section 5.4.1, an FPGA's binary system is restricted to integer values, meaning any arithmetic containing decimals will always be rounded, such as the multiplication step of the filter arithmetic and the FPM of the exponential pulse generation. In fact, rounding actually occurs twice within the FPM of the exponential pulse, first when determining the decay factor (Eq. 10) and again when performing the bit shift (Eq. 11). This technically leaves the project with three points of rounding error, although the that within the FPM will only occur when a simulated signal is processed.

Building on these results, time should be spent determining ways, if any, to reduce the severity of rounding where possible and ensuring that all arithmetic is being properly performed on the FPGA. In conjunction, developing a method of displaying the complete pulse and filter response as processed by the FPGA is imperative. Without the ability to compare the peak detector output to a waveform, there is no system of checks that can be taken to ensure proper data acquisition.

8 Code Reference

8.1 Python Simulation

```
# -*- coding: utf-8 -*-
"""
Python script file used to test simulated outcome of Verilog trapezoidal data filter.
"""

import numpy as np
import matplotlib.pyplot as plt

# Call to generate an exponentially decaying signal of base e with zero rise time
#   Inputs:
#       sig_amp: integer value of peak amplitude of exponentially decaying signal
#       tau: real valued time constant for exponential decay
#   Outputs:
#       signal: array of integer values representing exponentially decaying signal
def e_exp_decay(sig_amp, tau):

    signal = np.zeros(15, dtype = int)
    current_value = sig_amp
    i = 0

    while current_value > 0 and i < 1000:
        current_value = round(sig_amp * np.exp(-i / tau))
        signal = np.append(signal, current_value)
        i += 1

    return signal


################################################################################


# Call to perform trapezoidal filtration of exponentially decaying signal based on the following paper:
# Jord94 - Digital synthesis of pulse shapes in real time for high resolution radiation spectroscopy (Pg. 341)
#   Inputs:
#       tdata: exponentially decaying signal produced by e_exp_decay
#       kdelay: integer value for rise/fall time of trapezoid
#       ldelay: integer value for length of trapezoid flat-top (ft = l - k), ldelay >= kdelay
#       mult_factor: integer value equal to (tau - 0.5)
#   Output:
#       Print: signal peak, filter peak, gain, and corrected filter peak
#       Plot: signal and gain-corrected filter
def trap_filter(tdata, kdelay: int, ldelay: int, mult_factor):

    # Define array of zeros to be populated (increased length to account for rollover from delay pipeline)
    s_axis_tdata = np.zeros(len(tdata) + kdelay + ldelay + 2, dtype = int)
    data_length = len(s_axis_tdata)
    loop_index = range(data_length)


    # Define divisor for filter gain correction
    divisor = kdelay * (mult_factor + 0.5)


    # Assign MSIs of s_axis_tdata the values of tdata, then keep remaining indices as zeros
    for i in range(len(tdata)):
        s_axis_tdata[i] = tdata[i]


    # Assign delay pipeline array: d^{k,l}(j) = v(j) - v(j - k) - v(j - l) + v(j - k - l)
    d_kl = s_axis_tdata - np.roll(s_axis_tdata, kdelay) - np.roll(s_axis_tdata, ldelay) + np.roll(s_axis_tdata, kdelay + ld


    # Use for loop to assign accumulator array p_prime: p'(n) = p'(n - 1) + d^{k,l}(n)
    p_prime = np.zeros(data_length, dtype = int)
    for i in loop_index:
        p_prime[i] = p_prime[i - 1] + d_kl[i]


    # Use for loop to assign final results (accumulator) array: s(n) = s(n - 1) + p'(n) + M * d^{k,l}(n)
    results = np.zeros(data_length, dtype = int)
    for i in loop_index:
        results[i] = results[i - 1] + p_prime[i] + (mult_factor - 0.5) * d_kl[i]
    print(results)

    # Print components of signal and filter
    print("\nExponential Signal Peak = " + str(max(tdata)))
    print("Trapezoidal Filter Peak = " + str(max(results)))
    print("Gain (Filter / Signal) = " + str(max(results) / max(tdata)))
    print("Corrected Filter Peak = " + str(max(results) / divisor))
```

```
    # Create plots for data before and after filter
    '''
    plt.plot(s_axis_tdata)
    plt.xlabel('Index')
    plt.ylabel('Amplitude')
    plt.title('Filter Input')
    plt.show()

    plt.plot(results)
    plt.xlabel('Index')
    plt.ylabel('Amplitude')
    plt.title('Filter Output')
    plt.show()
    '''


    # Create plot for signal and gain-corrected filter
    plt.plot(s_axis_tdata, label = 'Signal Input')
    plt.plot(results / divisor, label = 'Filter Output', color = 'r')
    plt.xlabel('Index')
    plt.ylabel('Amplitude')
    plt.title('Signal and Gain-Corrected Trapezoidal Filter')
    plt.xlim(right=80)
    plt.legend()
    plt.show()
```

8.2 Exponential Decay Generator

```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////
// Engineer: Samuel LeRose
//
// Create Date: June 2023
// Design Name:
// Module Name: e_exp_decay
// Project Name: Trapezoidal Filter
// Target Devices: Red Pitaya STEMlab 125-14
// Tool Versions:
// Description: Provides an exponentially decaying signal using fixed-point multiplication
// Dependencies: N/A
//
// Revision: v1.0
//
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module e_exp_decay #
(
    parameter DATA_WIDTH = 16,
    parameter DAC_WIDTH = 14
)
(
    input clk,
    input aresetn,
    input [DAC_WIDTH-1:0] sig_amp,
    input [DAC_WIDTH-1:0] decay_factor,
    output signed [DATA_WIDTH-1:0] m_axis_tdata,
    output m_axis_tvalid,
    input trigger
);
    reg [DAC_WIDTH-1:0] current_value;
    reg [2*DAC_WIDTH-1:0] mult;

    assign m_axis_tdata = current_value;
    assign m_axis_tvalid = 1;

    always @ (posedge clk) begin
        if (~aresetn) begin
            current_value <= 0;

        end else begin
            if (trigger) begin
               current_value <= sig_amp;    // Set initial peak for exponential signal

            end else if (current_value != 0) begin
               mult = current_value * decay_factor; // Multiply by decay factor
               current_value <= (mult) >> 14;       // Shift to implement fixed-point multiplication

            end
        end
```

```
    end
endmodule
```

8.3 Pulse Generator

```
`timescale 1ns / 1ps
//Generates 1 clock cycle pulse on positive edge of trigger
module pulse_gen #
(
    parameter integer OUTPUT_WIDTH = 1
)
(
    input wire clk,
    input wire trigger,
    output wire pulse
);
    reg [OUTPUT_WIDTH - 1 : 0] last;
    reg [OUTPUT_WIDTH - 1 : 0] pulse_reg;

    always @ (posedge clk) begin
        last <= trigger;
        if (trigger & ~last)
            pulse_reg <= 1;
        else
            pulse_reg <= 0;
    end

    assign pulse = pulse_reg;

endmodule
```

8.4 Ring Buffer

```
`timescale 1 ns / 1 ps

//////////////////////////////////////////////////////////////////////////////////
// Engineer: Samuel LeRose
//
// Create Date: June 2023
// Design Name:
// Module Name: ring_buffer
// Project Name: Trapezoidal Filter
// Target Devices: Red Pitaya STEMlab 125-14
// Tool Versions:
// Description: Provides a read/write memory storage mechanism utilizing bit overflow.
//              Also known as a ring buffer: https://en.wikipedia.org/wiki/Circular_buffer
// Dependencies: N/A
//
// Revision: v1.0
//
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module ring_buffer #
(
  parameter integer AXIS_TDATA_WIDTH = 16,
  parameter integer BUFFER_LENGTH = 256      // Length of ring buffer, preferably a power of 2
                                             // Ldelay may not exceed BUFFER_LENGTH - 1
)
(
  input clk,
  input enrd,
  input enwr,
  input [13:0] delay,
  input signed [AXIS_TDATA_WIDTH-1:0] wr_data,
  output signed [AXIS_TDATA_WIDTH-1:0] rd_data
);

  // Ring buffer with BUFFER_LENGTH addressable locations, each location being a register of width AXIS_TDATA_WIDTH
  reg [AXIS_TDATA_WIDTH-1:0] ring_buff [0:BUFFER_LENGTH - 1];
  reg [AXIS_TDATA_WIDTH-1:0] rd_reg;                      // Register to hold read data for output assignment

  reg [$clog2(BUFFER_LENGTH)-1:0] wr_ptr = 0;             // Pointer to specify writing location in ring_buff
  reg [$clog2(BUFFER_LENGTH)-1:0] rd_ptr;                 // Pointer to specify reading location in ring_buff

  assign rd_data = rd_reg;

  always @ (posedge clk) begin
      if (enwr) begin
        // Write handling
```

```
        ring_buff[wr_ptr] <= wr_data;    // Write data at ptr
        wr_ptr <= wr_ptr + 1;            // Iterate ptr
      end

      if (enrd) begin
        // Read handling
        rd_ptr <= wr_ptr - delay + 2;    // Set read pointer, add 2 to remove inherient delays with cycles
        rd_reg <= ring_buff[rd_ptr];     // Read data at delay_ptr
      end
  end

endmodule
```

8.5 Trapezoidal Filter

```
`timescale 1 ns / 1 ps

//////////////////////////////////////////////////////////////////////////////////
// Engineer: Samuel LeRose
//
// Create Date: June 2023
// Design Name:
// Module Name: trap_filter
// Project Name: Trapezoidal Filter
// Target Devices: Red Pitaya STEMlab 125-14
// Tool Versions:
// Description: Digital implementation of the trapezoidal filtering method created by
//      V. T. Jordanov and G. F. Knoll: https://doi.org/10.1016/0168-9002(94)91011-1
// Dependencies: ring_buffer
//
// Revision: v1.0
//
// Additional Comments: Filter arithmetic adapted from paper by Kavita Pathak and
//      Dr. Sudhir Agrawal: https://www.ijsrp.org/research-paper-0815/ijsrp-p4475.pdf
//
//////////////////////////////////////////////////////////////////////////////////

module trap_filter #
(
    parameter integer AXIS_TDATA_WIDTH = 16
)
(
    input clk,
    input aresetn,
    input signed [AXIS_TDATA_WIDTH-1:0] s_axis_tdata,
    input s_axis_tvalid,
    output m_axis_tvalid,
    output signed [2*AXIS_TDATA_WIDTH-1:0] m_axis_tdata,

    input signed [15:0] mult_factor,     // mult_factor = decay time constant of signal
    input [13:0] Kdelay,                 // Length of rising/falling edge of trapezoid
    input [13:0] Ldelay                  // Ldelay - Kdelay = length of flat-top (L >= K)
);

    wire signed [AXIS_TDATA_WIDTH-1:0] K_delay_tdata;    // Data after K time delay for Sub1
    wire signed [AXIS_TDATA_WIDTH-1:0] L_delay_tdata;    // Data after L time delay for Sub2
    reg signed [AXIS_TDATA_WIDTH-1:0] K_diff_tdata;      // Diff between incoming data and data after K delay
    reg signed [AXIS_TDATA_WIDTH-1:0] L_diff_tdata;      // Diff between Sub1 data and Sub1 data after L delay
    reg signed [2*AXIS_TDATA_WIDTH-1:0] mult_tdata;      // Product of Sub2 data and multiplication factor
    reg signed [2*AXIS_TDATA_WIDTH-1:0] accu_tdata;      // Accum of data by adding new Sub2 data at each cycle
    reg signed [2*AXIS_TDATA_WIDTH-1:0] sum_tdata;       // Sum of multiplier and accumulator values
    reg signed [2*AXIS_TDATA_WIDTH-1:0] result_tdata;    // Accum of data by adding new summed data at each cycle

    reg [28:0] buff_iterate;             // Counter for handling ring buffers
    reg enwrK, enrdK, enrdL, enwrL;      // Enable read and write for ring buffers

    assign m_axis_tvalid = s_axis_tvalid;
    assign m_axis_tdata = result_tdata;

    ring_buffer # () K_buff
    (
        .clk(clk),
        .enrd(enrdK),
        .enwr(enwrK),
        .delay(Kdelay),
        .wr_data(s_axis_tdata),
        .rd_data(K_delay_tdata)
    );

    ring_buffer # () L_buff
    (
        .clk(clk),
        .enrd(enrdL),
        .enwr(enwrL),
```

```
        .delay(Ldelay),
        .wr_data(K_diff_tdata),
        .rd_data(L_delay_tdata)
    );

    always @ (posedge clk) begin
        // Reset handling
        if (~aresetn) begin
            K_diff_tdata <= 0;
            L_diff_tdata <= 0;
            mult_tdata <= 0;
            accu_tdata <= 0;
            sum_tdata <= 0;
            result_tdata <= 0;
            buff_iterate <= 0;
            enrdK <= 0;
            enwrK <= 0;
            enrdL <= 0;
            enwrL <= 0;

        // Buffer handling
        end else if (buff_iterate < Kdelay + Ldelay + 2) begin
            // Iterate buffer controller
            buff_iterate <= buff_iterate + 1;

            // Enable K_buff write (if it is off)
            if (~enwrK) begin
                enwrK <= 1;
            end

            // Enable K_buff read (if it is off) after K_buff has written to Kdelay - 2 number of registers
            if (buff_iterate > Kdelay - 3 && ~enrdK) begin
               enrdK <= 1;
            end

            // Enable L_buff write (if it is off) and start pre-filter arithmetic for writting to L_buff
            if (buff_iterate > Kdelay) begin    // Allow K_buff reading to catch up
               if (~enwrL) begin
                   enwrL <= 1;
               end
               K_diff_tdata <= s_axis_tdata - K_delay_tdata;    // Capture baseline
            end

            // Enable L_buff read (if it is off) after L_buff has written to Ldelay - 2 number of registers
            if (buff_iterate > Kdelay + Ldelay - 2 && ~enrdL) begin
                enrdL <= 1;
            end

        end else begin
            //Filter arithmetic
            K_diff_tdata <= s_axis_tdata - K_delay_tdata;       // Subtractor 1
            L_diff_tdata <= K_diff_tdata - L_delay_tdata;       // Subtractor 2
                mult_tdata <= L_diff_tdata * (mult_factor - 0.5);       // Multiplier
            accu_tdata <= accu_tdata + L_diff_tdata;            // Accumulator 1
            sum_tdata <= mult_tdata + accu_tdata;               // Adder
            result_tdata <= result_tdata + sum_tdata;           // Accumulator 2
        end
    end
endmodule
```

8.6 Peak Detector

```
`timescale 1 ns / 1 ps

//////////////////////////////////////////////////////////////////////////////////
// Engineer: Samuel LeRose
//
// Create Date: June 2023
// Design Name:
// Module Name: peak_detector
// Project Name: Trapezoidal Filter
// Target Devices: Red Pitaya STEMlab 125-14
// Tool Versions:
// Description: Provides method of peak detection specifically for trapezoidal
//      filtration where the "peak" is the middle of the flat top.
// Dependencies: N/A
//
// Revision: v1.0
//
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module peak_detector #
```

```
(
    parameter integer AXIS_TDATA_WIDTH = 16,
    parameter integer BUFFER_LENGTH = 14
)
(
    input clk,
    input aresetn,
    input signed [2*AXIS_TDATA_WIDTH-1:0] s_axis_tdata,
    input s_axis_tvalid,
    output signed [2*AXIS_TDATA_WIDTH-1:0] peak,

    input [15:0] threshold,
    input [13:0] Kdelay,                    // Length of rising/falling edge of trapezoid
    input [13:0] Ldelay                     // Ldelay - Kdelay = length of flat-top (L >= K)
);

    reg signed [2*AXIS_TDATA_WIDTH-1:0] peak_value;
    reg [7:0] counter;

    assign peak = peak_value;

    always @ (posedge clk) begin
        if (~aresetn) begin
                peak_value <= 0;
                counter <= 0;

        end else begin
                if (s_axis_tdata > threshold)     begin   // If data is above threshold, begin peak detection
                    if (counter == 0 && peak_value == 0) begin
                        counter <= (Kdelay + Ldelay) / 2 - 1;     // Set counter to wait for middle of flat top
                    end
                end

                if (counter > 0) begin
                    if (counter == 1) begin
                        peak_value <= s_axis_tdata;       // At middle of flat top, record data value as peak
                    end
                    counter <= counter - 1;
                end
/*
                if (s_axis_tdata < threshold) begin
                    peak_value <= 0;      // Reset peak and counter when signal is under threshold
                    counter <= 0;
                end

                if (current_peak_value > peak_value) begin
                    peak_value <= current_peak_value;
                end
*/

        end
    end
endmodule
```

8.7 Jupyter Notebook Program

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import mmap
import os
import time
import numpy as np
import matplotlib.pyplot as plt

# Change name of .bit file to name of your file
os.system('cat /root/trap_test.bit >/dev/xdevcfg')

axi_gpio_regset = np.dtype([
    ('gpio1_data'   , 'uint32'),
    ('gpio1_control', 'uint32'),
    ('gpio2_data'   , 'uint32'),
    ('gpio2_control', 'uint32')
])

memory_file_handle = os.open('/dev/mem', os.O_RDWR)
axi_mmap_trap = mmap.mmap(fileno=memory_file_handle, length=mmap.PAGESIZE, offset=0x41200000)
axi_numpy_array_trap = np.recarray(1, axi_gpio_regset, buf=axi_mmap_trap)
axi_array_contents_trap = axi_numpy_array_trap[0]

axi_mmap_peak = mmap.mmap(fileno=memory_file_handle, length=mmap.PAGESIZE, offset=0x41210000)
```

```
axi_numpy_array_peak = np.recarray(1, axi_gpio_regset, buf=axi_mmap_peak)
axi_array_contents_peak = axi_numpy_array_peak[0]


# In[ ]:


Kdelay = 10
Ldelay = 15
mult = 10 # Equal to tau of exp. decay signal

sig_amp = 1000
decay = 14825 # 2^14 * e^(-1 / tau)

threshold = 1000

# Establish constants within modules and allow modules to reset
axi_array_contents_trap.gpio1_data = (Kdelay << 16) + (Ldelay << 2)
axi_array_contents_trap.gpio2_data = (sig_amp << 18) + (mult << 2)
axi_array_contents_peak.gpio1_data = (decay << 16) + threshold
time.sleep(0.01)

# Release reset
axi_array_contents_trap.gpio1_data += 1
time.sleep(0.01)

# Trigger signal and allow filter to process signal
axi_array_contents_trap.gpio2_data += 1
time.sleep(1)

# Calculate peak
print("Input Signal Peak: ", sig_amp)
peak = axi_array_contents_peak.gpio2_data / (Kdelay * (mult + 0.5))

print("Filter Response Peak: ", peak)

print("Error: " + str(((peak-sig_amp)/sig_amp)*100) + "%")


# In[ ]:


# Uncomment for interactive plots, very slow
# %matplotlib notebook
# %matplotlib qt

Kdelay = 10
Ldelay = 15
mult = 10 # Equal to tau of exp. decay signal

decay = 14825 # 2^14 * e^(-1 / tau)

threshold = 1000

sim_amps = np.arange(1000,16000,500)

calc_amps = np.array([])

errors = np.array([])

for amp in sim_amps:
    # Establish constants within modules and allow modules to reset
    axi_array_contents_trap.gpio1_data = (Kdelay << 16) + (Ldelay << 2)
    axi_array_contents_trap.gpio2_data = (amp << 18) + (mult << 2)
    axi_array_contents_peak.gpio1_data = (decay << 16) + threshold
    time.sleep(0.01)

    # Release reset
    axi_array_contents_trap.gpio1_data += 1
    time.sleep(0.01)

    # Trigger signal and allow filter to process signal
    axi_array_contents_trap.gpio2_data += 1
    time.sleep(0.5)

    # Calculate peak, add peak and error to arrays
    peak = axi_array_contents_peak.gpio2_data / (Kdelay * (mult + 0.5))
    calc_amps  = np.append(calc_amps, peak)
    errors = np.append(errors, ((peak/amp) - 1) * 100)

index = range(len(sim_amps))

plt.scatter(index, sim_amps, label="Input Signal Peaks")
plt.scatter(index, calc_amps, c='r', label="Filter Response Peaks")
plt.ylabel("Signal Amplitude")
plt.legend(loc="upper left")
plt.title("Peak of Input Exponentially Decaying Signal vs. Peak of Filter Response")
```

```
plt.ylim([0,17000])
plt.xticks([])
plt.show()

plt.scatter(sim_amps, errors)
plt.xlabel("Peak of Input Signal")
plt.ylabel("Percent Difference (Response / Input)")
plt.title("Peak of Input Exponentially Decaying Signal vs. Error in Peak of Filter Response")
plt.xlim([0,17000])
plt.ylim([min(errors)-0.5, max(errors)+0.5])
plt.show()
```

References

[1]K. Pathak and D. S. Agrawal, "VHDL Simulation of Trapezoidal Filter for Digital Nuclear Spectroscopy systems", International Journal of Scientific and Research Publications **5** (2015).

[2]G. R. Gilmore, *Practical Gamma-ray Spectrometry*, 2nd ed. (2008).

[3]A. Buttacavoli, F. Principato, G. Gerardi, M. Bettelli, A. Zappettini, P. Seller, M. C. Veale, S. Zanettini, and L. Abbene, "Ballistic Deficit Pulse Processing in Cadmium–Zinc–Telluride Pixel Detectors for High-Flux X-ray Measurements", Sensors **22**, `https://doi.org/10.3390/s22093409` (2022).

[4]G. Czanner, S. V. Sarma, D. Ba, U. Eden, W. Wu, E. Eskandar, H. Lim, S. Temereanca, W. Suzuki, and E. Brown, "Measuring the signal-to-noise ratio of a neuron", Proceedings of the National Academy of Sciences of the United States of America, 7141–7146 (2015).

[5]A. Patil, "Dead time and count loss determination for radiation detection systems in high count rate applications" (Missouri University of Science and Technology, 2010).

[6]J. Eyre, "FPGA/DSP Blend Tackles Telecom Apps", Electronic Engineering Times (2002).

[7]A. Sanaullah, C. Yang, Y. Alexeev, K. Yoshii, and M. C. Herbordt, "Real-time data analysis for medical diagnosis using FPGA-accelerated neural networks", BMC Bioinformatics **19**, `https://doi.org/10.1186/s12859-018-2505-7` (2018).

[8]S. Sirois, *What is Processor Speed and Why Does It Matter?*, `https://www.hp.com/us-en/shop/tech-takes/what-is-processor-speed` (visited on 05/26/2023).

[9]V. T. Jordanov and G. F. Knoll, "Digital synthesis of pulse shapes in real time for high resolution radiation spectroscopy", Nuclear Instruments and Methods in Physics Research **345**, 337–345 (1994).

[10]V. T. Jordanov, G. F. Knoll, A. C. Huber, and J. A. PantaziS, "Digital techniques for real-time pulse shaping in radiation measurements", Nuclear Instruments and Methods in Physics Research **353**, 261–264 (1994).

[11]GoLuckyRyan, *Trapezoid filter*, `https://nukephysik101.wordpress.com/2020/03/20/trapezoid-filter/` (visited on 07/28/2023).