Dylan Falzone
10/25/23
4539-8897

EE Design 1 Technical Report
Analog Filter Design

Introduction

As we've previously discussed, digital and analog are like two sides of the same coin. Both are representations of the world, but each in different domains: the continuous and the discreet. This relationship has naturally warranted much debate over the years as to which representation is superior. While such a question is beyond the scope of this lab, we will instead focus on just one facet of this debate. Which type of system is more capable of representing a simple lowpass filter?

Methods

Analog

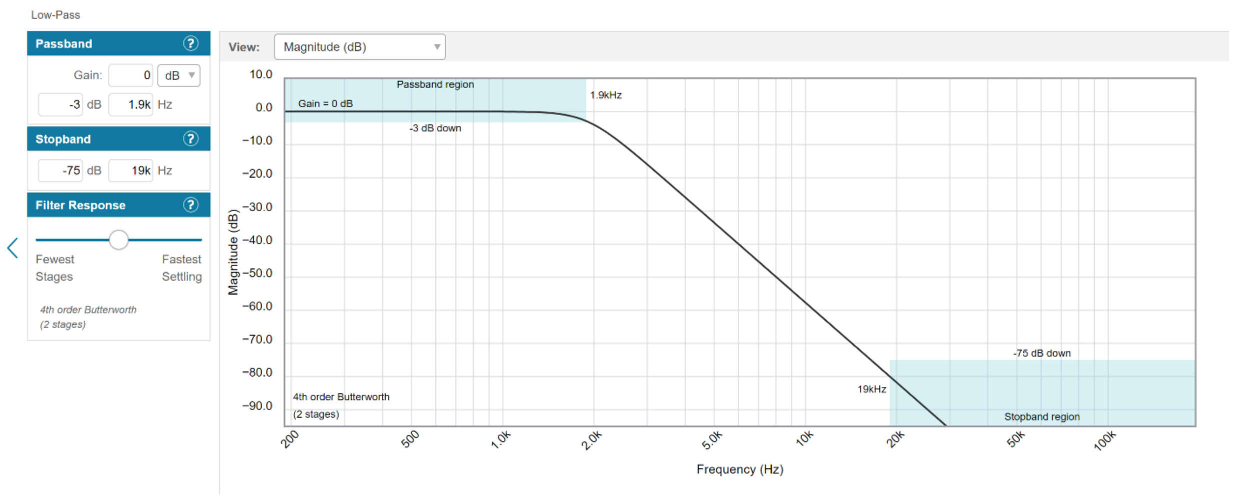
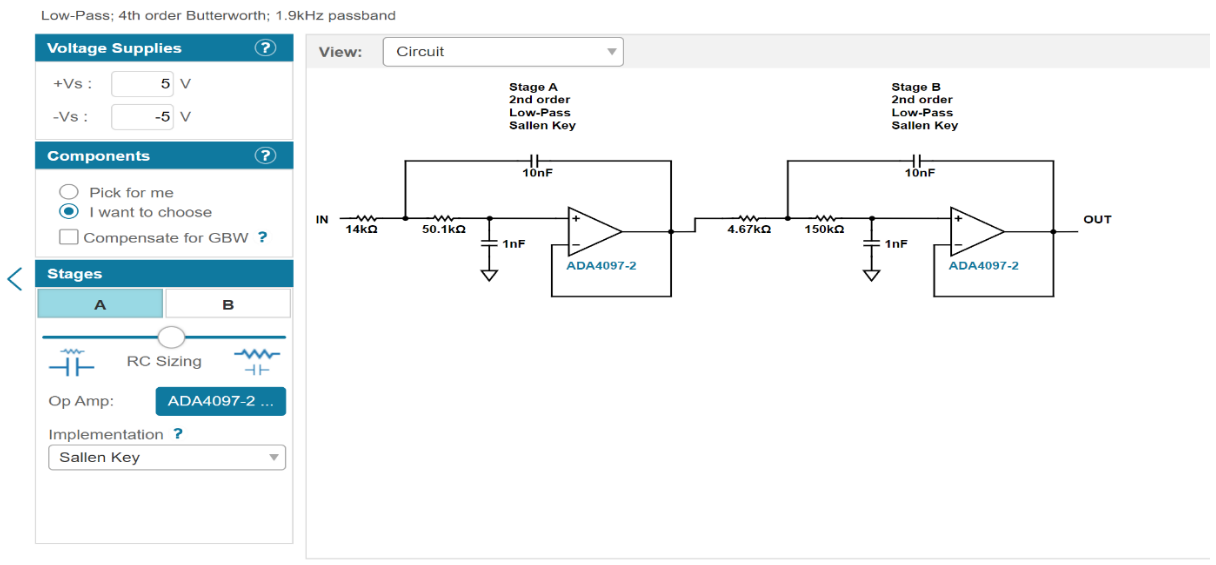

Low-Pass
Passband — Gain: 0 dB; -3 dB 1.9k Hz
Stopband — -75 dB 19k Hz
Filter Response — Fewest Stages / Fastest Settling
4th order Butterworth (2 stages)
View: Magnitude (dB)

Figure 1: Analog Filter Wizard design tool for my filter.

Per the assignment instructions, I begin with the analog filter. Using the provided filter designer, I designed the lowpass with a passband up to 1.9kHz (I would have done 1.897 to match my UFID but it unfortunately rounded), and a stopband from 19kHz so that it would be easy to achieve the -80db/decade requirement. However, in order to achieve that slope, the filter wizard forced the system to become 5th order, so I ended up reducing the slope to -75db/decade instead. This gave me a perfect 2-stage, 4th order Butterworth filter, as shown in the following figure. The website automatically chose Butterworth, so I did not question it at first but during the digital section, I will talk more about the distinction between the types.


Low-Pass; 4th order Butterworth; 1.9kHz passband

Voltage Supplies

+Vs : 5 V

-Vs : -5 V

Components

( ) Pick for me [✓ I want to choose]

☐ Compensate for GBW

Stages

A | B

RC Sizing

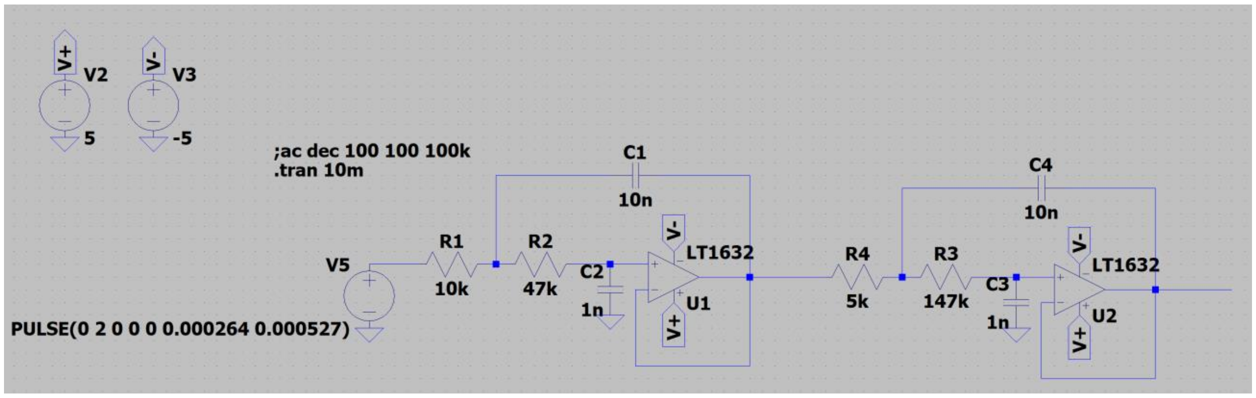
Op Amp: ADA4097-2 ...

Implementation: Sallen Key

View: Circuit

Figure 2: Schematic for the generated filter circuit.

Getting the components right did take some time. I wanted to choose capacitor values that I had in my kit, while making sure the resistors were between 5k and 300k, just to be safe. In an initial attempt at this lab, I had used 330ohm resistors, which did actually produce a decent result, but it was nowhere near as effective at making the wave as clean as the values I ended up with. I did find the relationship between the resistance and capacitance values, and their impact on the output to be very interesting. Though the designer scaled the values proportionally, learning more about the pushing and pulling of the output wave depending on these values was the highlight of this lab for me [1]. I settled on the values pictured above, as I had just enough capacitors to make it work, and the resistance values were close enough to ones that I had in my kit.



```
;ac dec 100 100 100k
.tran 10m
PULSE(0 2 0 0 0 0.000264 0.000527)
```

Figure 3: LTSPICE Schematic for the filter circuit.

In the above LTSPICE schematic, I modified the values to match what I used on my breadboard. Notably, I rounded 14k to 10k, 50k to 47k, 4.67k to 5k, and 150k to 147k. This may have been the cause of the non-ideality I observed that I will discuss in the results section.

Transfer Function:

$$G(s)=\frac{212765957.44681}{s^2+12127.659574468s+212765957.44681}$$

Transfer Function:

$$G(s)=\frac{136054421.76871}{s^2+20680.272108844s+136054421.76871}$$

Figure 4: Transfer functions for stages 1 and 2 respectively.

The above transfer functions were generated by plugging my component values into a different Sallen-key designer website [2]. Since each one is 2nd order, judging by the s^2, we can figure that cascading them (multiplication) will indeed provide the 4th order system we designed.

Digital

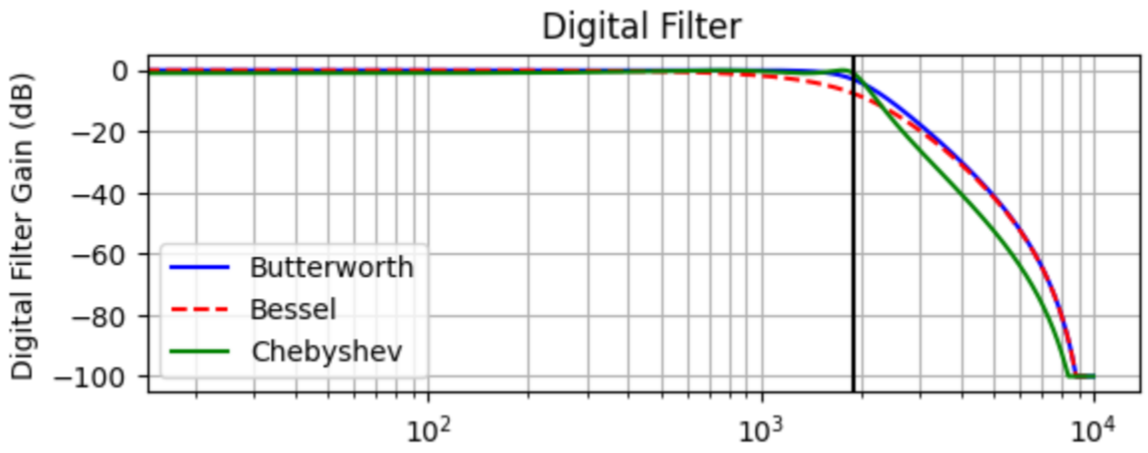


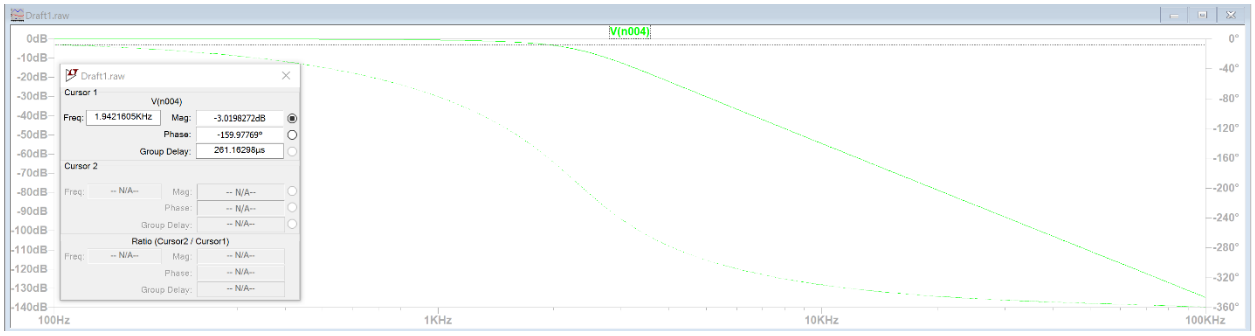
Figure 5: Frequency domain for different filter types via lowpass.py in Jupyter Notebook.

The first step for designing my digital filter was to run the provided python script in Jupyter. Per the advice of Dr. Patrick, I chose a sample rate of 20k, allowing me to reach the proper resolution needed to replicate the waveform accurately. Initially, I had tried much smaller rates, but the resolution was not enough. It was an attempt to fix this problem that led to the communication debacle in which my computer could no longer interface with my microcontroller. Removing the only sleep statement in a main file that is repeatedly writing to SPI is a mistake that I will not make again.

The python script gave me the above graph, from which I decided to continue on with the Butterworth filter since that is what my analog filter was already designed for. It would certainly have made more sense for me to use this script before designing either filter, but I did not realize that until afterward. Butterworth was seemingly the right choice though, as Bessel would have caused premature attenuation and Chebyshev would have had a ripple with a steep roll-off that would have worsened the non-ideality in my circuit that I will once again discuss in the results section. This graph is consistent with my research on filter types, as Butterworth is known as the least rippled and most flat filter, while Chebyshev is known for its ripples and steep slope, and Bessel is known for having a more gradual roll-off [3].

Results

Analog



Draft1.raw

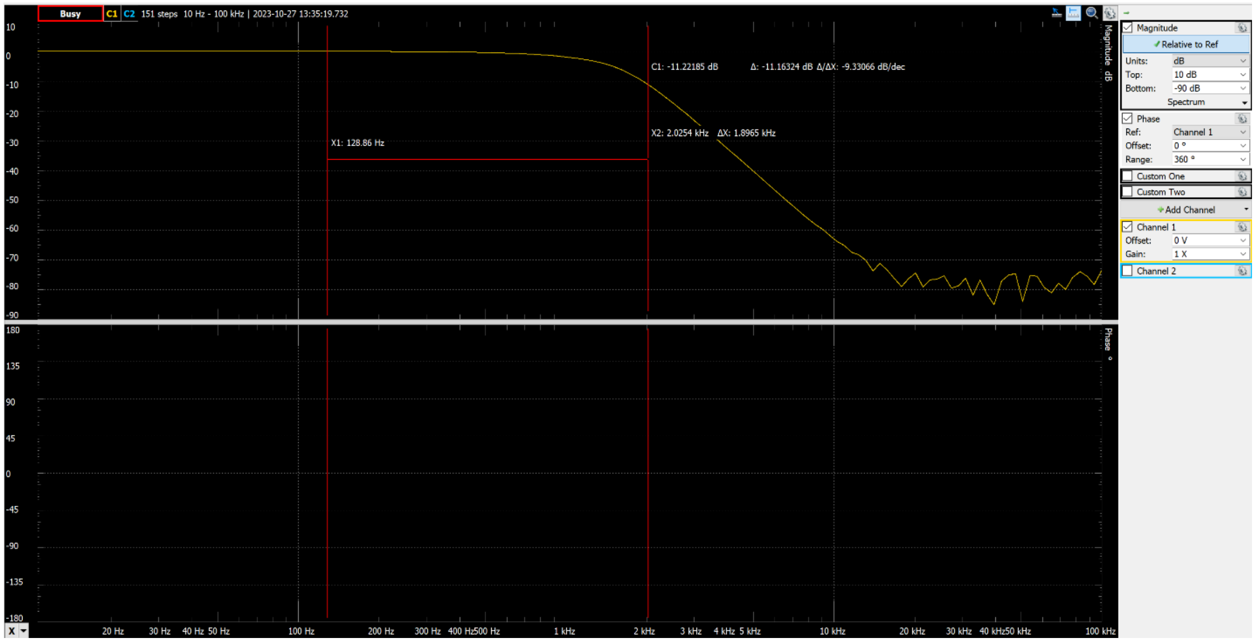
Cursor 1
V(n004)
Freq: 1.9421605KHz Mag: -3.0198272dB
Phase: -159.97769°
Group Delay: 261.16298µs

Cursor 2
Freq: -- N/A-- Mag: -- N/A--
Phase: -- N/A--
Group Delay: -- N/A--

Ratio (Cursor2 / Cursor1)
Freq: -- N/A-- Mag: -- N/A--
Phase: -- N/A--
Group Delay: -- N/A--

Figure 6: Bode plot for LTspice schematic.

The simulated circuit behaved perfectly. It’s -3db point lined up almost exactly with the 1.9kHz desired frequency, as can be seen in figure 6. Since my breadboard implementation used the same components as the simulation, I expected a similar result.



151 steps 10 Hz - 100 kHz | 2023-10-27 13:35:19.732

C1: -11.22185 dB

Δ: -11.16324 dB Δ/ΔX: -9.33066 dB/dec

X2: 2.0254 kHz ΔX: 1.8965 kHz

X1: 128.86 Hz

Magnitude: Relative to Ref
Units: dB
Top: 10 dB
Bottom: -90 dB
Spectrum

Phase
Ref: Channel 1
Offset: 0 °
Range: 360 °

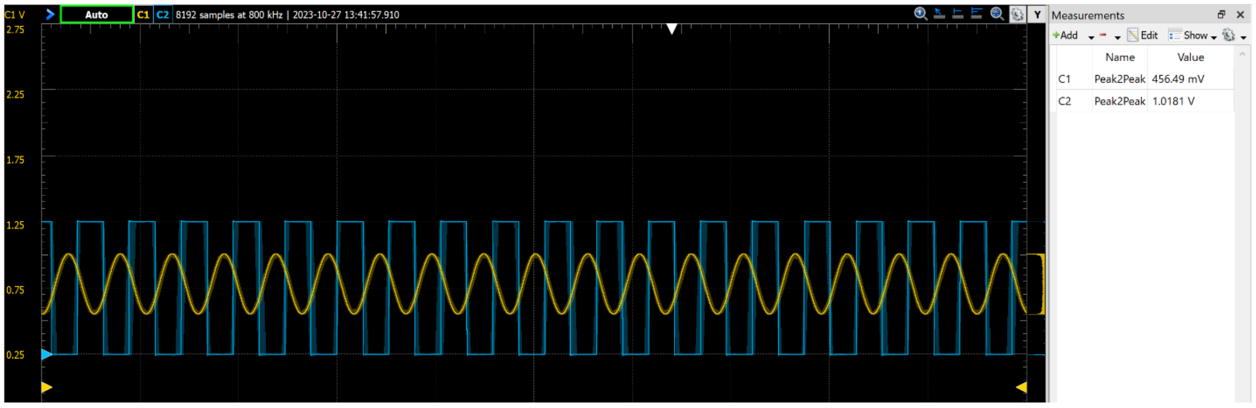
Custom One
Custom Two
Add Channel

Channel 1
Offset: 0 V
Gain: 1 X

Channel 2

Figure 7: Bode plot for the breadboard implementation of the circuit.

I was wrong to expect that. For some reason, my filter was off. Unfortunately, the traces on the graph were not properly placed when I took the screenshot in figure 7, but I can tell you from memory that the true -3db point was closer to 1.3kHz. This caused serious attenuation at the 1.9kHz range, resulting in the following waveform.



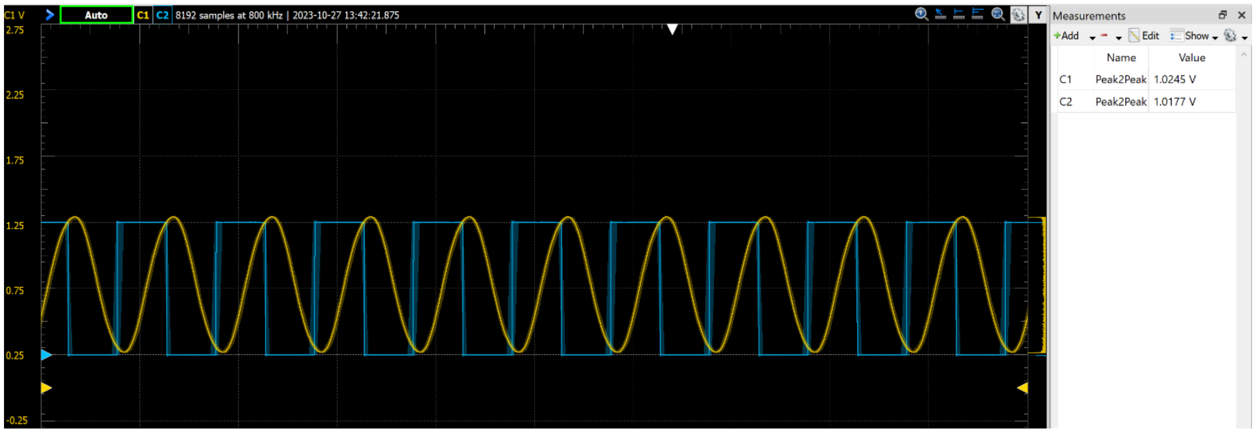
8192 samples at 800 kHz | 2023-10-27 13:41:57.910

Measurements

	Name	Value
C1	Peak2Peak	456.49 mV
C2	Peak2Peak	1.0181 V

Figure 8: Input/Output waveform for breadboard circuit at desired frequency.

Mysteriously, this issue did not manifest until near the end of my time working on this lab. I suspect that some, if not all, of the variation that I observed (but unfortunately did not document) was a result of changing op-amps in the middle of the lab. I spent most of the lab using the 14 series chip, but once I realized that it could not realize the lowpass filter correctly, I switched to the 1632. This lab was chaos, so it's also very possible that any other number of issues caused this. At the end of the day, my analog filter did not quite reach the correct frequency, and if I could have spent even more time on this lab to fix it, I would have.



	Name	Value
C1	Peak2Peak	1.0245 V
C2	Peak2Peak	1.0177 V

Figure 9: Input/Output waveform for breadboard circuit at frequency of 1.3kHz.

The above figure shows that the filter did at least work, just not at the target frequency.

Digital

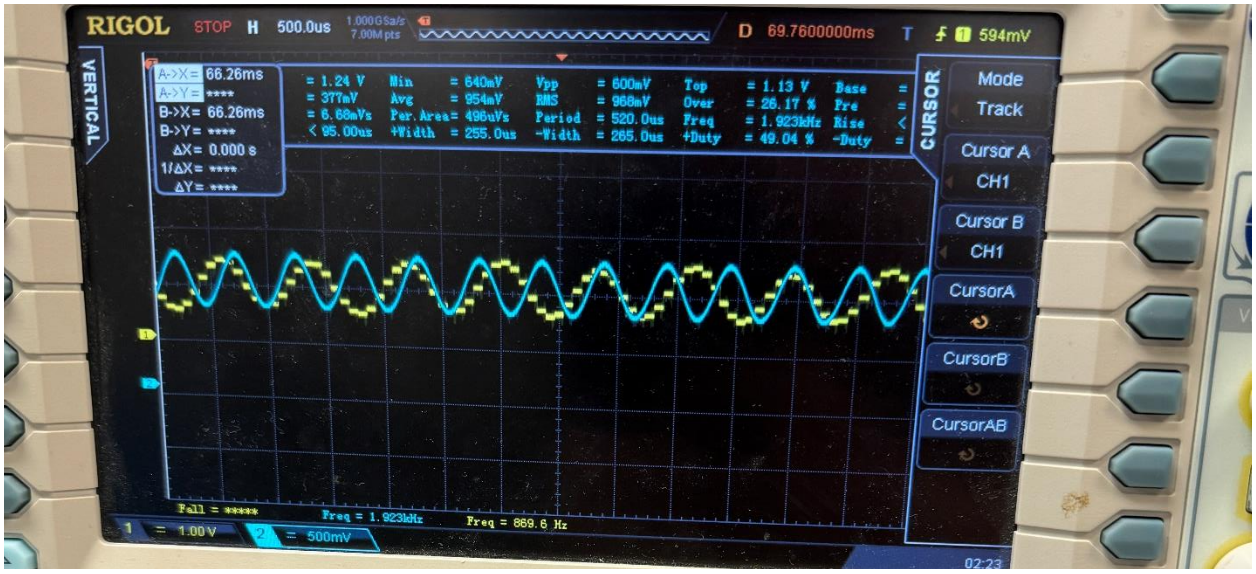


Figure 10: Digital output waveform (yellow) and analog output waveform (blue).

Things looked significantly better on the digital side of things. The amplitude was spot on, meaning that the filter was in fact properly implemented. The only issue here is that the frequency of the output ranged from 800-1.3kHz instead of the intended 1.9kHz. I attempted to increase this by adjusting the sleep statement, but it seemed that the system was already running as fast as it could, and I was told that overclocking wasn't necessary, so I left it at that.

Conclusion

The biggest takeaway from this lab is that designing filters is not fun. It is a task littered with non-idealities, hardware problems, and software problems that make for a nightmarish couple of weeks. But beyond that, there is a clear winner here as far as the battle between digital and analog. Despite my slightly less than accurate end result, the analog filter is by far and away the superior one. If designed and adjusted properly, it is faster, more efficient, and less complex than its digital counterpart. The digital filter is much slower, and has far fewer data points, creating a significantly less smooth result, as was demonstrated in figure 10. There are a lot of fantastic use cases for digital systems, but filter design is not one of them.

References

Put the references in any standard citation form here. Websites are allowed. (This is your chance to learn more about filters).

[1] W. Storr, “Sallen and Key Filter Design for second order RC Filters,” Basic Electronics Tutorials, https://www.electronics-tutorials.ws/filter/sallen-key-filter.html (accessed Oct. 29, 2023).

[2] Transfer Function generator: http://sim.okawa-denshi.jp/en/OPstool.php

[3] Electrical4U, “Butterworth Filter: What is it? (Design &amp; Applications),” Electrical4U, https://www.electrical4u.com/butterworth-filter/ (accessed Oct. 29, 2023).

Code:

```
import machine

import math

import utime

from ulab import numpy as np

from ulab import scipy

import time, array, uctypes, rp_devices as devs
```

```
spi = machine.SPI(1, baudrate=1000000, polarity=0, phase=0, sck=machine.Pin(10), mosi=machine.Pin(11))

cs = machine.Pin(13, mode=machine.Pin.OUT, value=1)

ref = machine.Pin(12, mode=machine.Pin.OUT, value=1)


ADC_CHAN = 1

ADC_PIN  = 26 + ADC_CHAN


adc = devs.ADC_DEVICE

pin = devs.GPIO_PINS[ADC_PIN]

pad = devs.PAD_PINS[ADC_PIN]

pin.GPIO_CTRL_REG = devs.GPIO_FUNC_NULL

pad.PAD_REG = 0


adc.CS_REG = adc.FCS_REG = 0

adc.CS.EN = 1

adc.CS.AINSEL = ADC_CHAN




#butter1 = [0.0549706, 0.10994119, 0.0549706, 1, -0.25733954, 0.05487108]

#butter2 = [1, 2, 1, 1, -0.35584529, 0.45866006]


butter1 = [0.00402738, 0.00805475, 0.00402738, 1, -1.08998094, 0.31702251]

butter2 = [1, 2, 1, 1, -1.36254351, 0.64635938]


#butter1 =

#butter2 =




while True:

    # Multiple ADC samples using DMA

    DMA_CHAN = 0
```

```
NSAMPLES = 2000

RATE = 20000

dma_chan = devs.DMA_CHANS[DMA_CHAN]

dma = devs.DMA_DEVICE


adc.FCS.EN = adc.FCS.DREQ_EN = 1

adc_buff = array.array('H', (0 for _ in range(NSAMPLES)))

adc.DIV_REG = (48000000 // RATE - 1) << 8

adc.FCS.THRESH = adc.FCS.OVER = adc.FCS.UNDER = 1


dma_chan.READ_ADDR_REG = devs.ADC_FIFO_ADDR

dma_chan.WRITE_ADDR_REG = uctypes.addressof(adc_buff)

dma_chan.TRANS_COUNT_REG = NSAMPLES

dma_chan.CTRL_TRIG_REG = 0

dma_chan.CTRL_TRIG.CHAIN_TO = DMA_CHAN

dma_chan.CTRL_TRIG.INCR_WRITE = dma_chan.CTRL_TRIG.IRQ_QUIET = 1

dma_chan.CTRL_TRIG.TREQ_SEL = devs.DREQ_ADC

dma_chan.CTRL_TRIG.DATA_SIZE = 1

dma_chan.CTRL_TRIG.EN = 1


# Start DMA transfer and ADC sampling

adc.CS.START_MANY = 1


# Wait for the DMA transfer to complete

while dma_chan.CTRL_TRIG.BUSY:

    time.sleep_us(10)


# Stop ADC sampling

adc.CS.START_MANY = 0

dma_chan.CTRL_TRIG.EN = 0


# Process ADC samples

vals = [(val*3.3/4096) for val in adc_buff]
```

```
# Define digital filter coefficients (copy and paste form Python filter

sos = [butter1, butter2]

output = scipy.signal.sosfilt(sos, vals) #apply software filter to data

#scale the data back to values for the 10-bit DAC and define data type

out_scale =np.array(output*(1023/5), dtype=np.uint16)

#define the rate in us to be used with SPI

RATEus = int(1/RATE*1000000)



for val in out_scale:

   buf = (0xF000 | (val << 2)).to_bytes(2,1)

   cs(0)

   spi.write(buf)

   cs(1)

   utime.sleep_us(RATEus//10)
```