10. PWM and Low Pass Filters

Table of Contents

Objective

Explore Arduino's `analogWrite()` function that produces pulse width modulation (PWM). Design and measure the performance of filters to turn PWM into clean DC without any "PWM noise", and with fast settling time. We'll see how settling time and PWM noise attenuation are inversely related and find a way to improve both by using a two stage filter. We'll also study effects of impedance and loading on circuit performance.

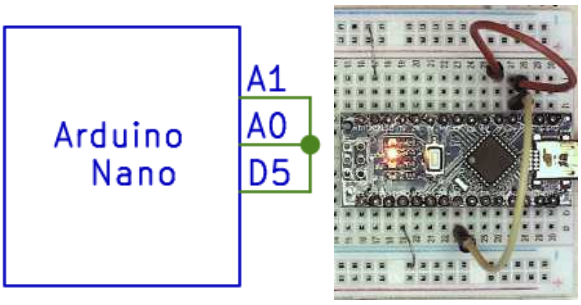


Figure 1. Schematic and photo of our simplest test circuit. Connect D5 to both A0 and A1 so neither input is a "floating antenna" for 60Hz power line noise.

Looking at the bigger picture, we're continuing our quest to automate measurements and experiments. We've already automated data acquisition into MATLAB and made a 2 channel oscilloscope useful for audio signals. Our next goal is for MATLAB to control the LM317's output voltage, so we won't need to manually turn LM317's R2 pot to change its voltage. Since the LM317 isn't capable of serial communication, the most likely path to achieving remote control will be through a remotely controlled analog voltage or current, and that suggests Arduino's `analogWrite()` is part of the solution. We haven't designed that solution yet, so this is an example of "bottom-upping" (https://en.wikipedia.org/wiki/Top-down_and_bottom-up_design), focusing on lower levels of abstraction as we work towards the goal.

One more "big picture point". This lab is our second exposure to working in the "frequency domain". We'll quantitatively simulate simple circuit characteristics in the frequency domain. However, we're not quite ready to quantify measurements. Our function `aspectrum`, introduced in the prior activity to visualize amplitude spectra is not yet quantitatively accurate in general. We'll have to address several issues with Fourier analysis, such as aliasing, reflections, windowing, etc., to accurately quantify the spectra (both its frequency and amplitude components). When we put it all together, you'll be amazed at how accurately measurements and theory will agree.

What is PWM?

In Arduino's IDE, Help –> Reference takes your browser to a useful site listing the functions in Arduino's standard library. `analogWrite()`'s help page is one click away. The first line has a link to PWM wave that tells us what we expect `analogWrite`'s output will look like. In anticipation for this activity, we programmed the Arduino in `oscope1.ino` to configure pin 5 as an output pin in `setup();` and provided a interface to `analogWrite` a value to pin 5 by sending a serial string `'p <value>'`.

Fire up `>> oscope2;` and verify communication with Arduino by clicking the run button on and off . All I see is a single horizontal line of voltage = 0. And that's what you should see ... the default value of 0 ADU. Let's change the value, e.g., execute `>> writeline(ard, 'p 135')`, (an arbitrary middle value) and toggle the run button. Christmas time (at least with my color scheme – see Appendix, `startup.m`):

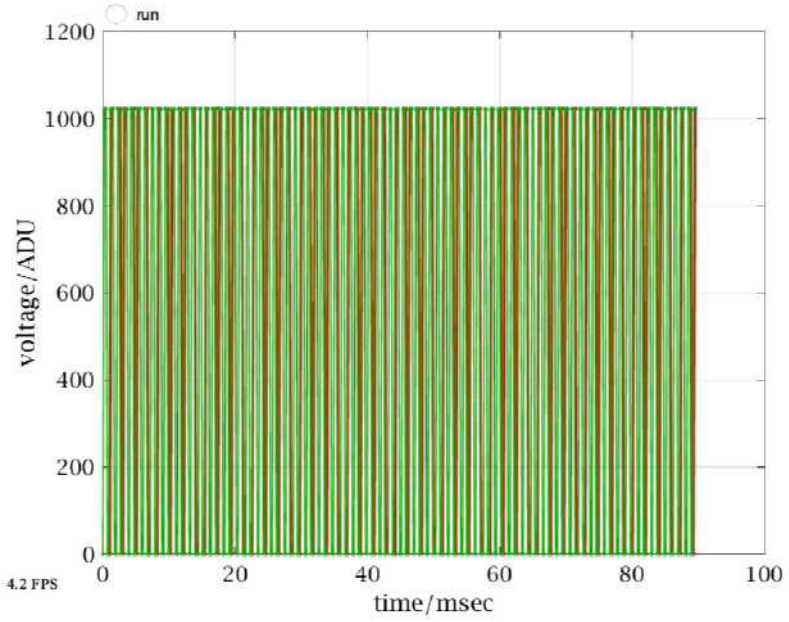


Figure 2. Oscilloscope display after sending 'p 135' command.

Surprised? If you expected a nice flat analog DC voltage, we sure didn't get that. Also since A0 and A1 are connected together, shouldn't they have identical measurements, green *should* always plot on top of red – so that we see *any* red says A0 ≠ A1. Why?? Fasten your seat belts ...

Pulse width modulation means the ouput flips between two binary states and the width or duration of the "pulse" is proportional to the encoded signal or command. Here are the first few data values:

```
data(1:10,:)
ans =
           0        1022
```

```
1023        1023
1023        1022
   0           0
   0           0
1023        1023
1023        1023
1022           0
   0           0
   0        1023
```

Arduino's (10-bit) ADC measures those 2 states around 0 and 1023. **Verify that Arduino's `analogWrite(5, 0)` invoked by MATLAB's `writeline(ard, 'p 0')` generates all zeros, and `writeline(ard,'p 255')` produces values ~1021-1023**, there's a little noise at the top in my system. The mean of the data should thus be close to 4 times the (8 bit) PWM value, let's see:

```
mean(data) / 4
ans =
  135.5025  136.1394
```

Check the means of `data` after writing other PWM values. So in spite of the messy appearance of Fig. 2, the PWM value *is* encoded as the *average*. We'll come back to why A0 $\neq$ A1 when we interface the function generator – but don't stop thinking about it.

Signal averaging

How can we implement `mean(data)` just using just electronic hardware? The answer is quite simple: a resistor and capacitor in series (Fig. 3)! One way to rationalize this is that the resistance limits the current flowing into and out the capacitor, and the capacitance determines how fast the voltage changes for a given current, i.e., $i = C\frac{dV}{dt}$. Before modeling such a circuit, let's see it work. **Build Fig. 3 using R1 = 5.1kΩ and C1 = 10$\mu F$and test it at a few PWM values**. Are the means of A0 and A1 always equal?

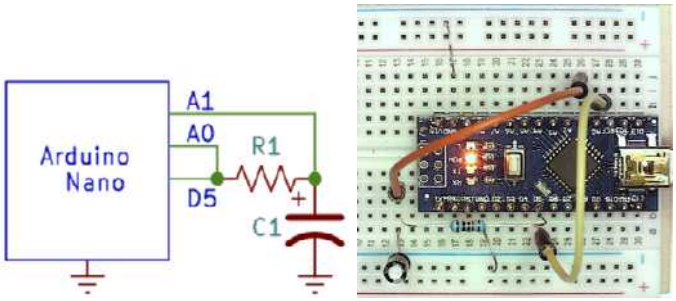


Figure 3. An R-C series PWM filter.

The `oscope2` display shows A1 is now constant, some of my results are:

```
>> writeline(ard, 'p 20')
>> runOnce = true; pause(.1); oscope2; mean(data), std(data)
ans =
   96.9800   82.1100

>> writeline(ard, 'p 200')
>> runOnce = true; pause(.1); oscope2; mean(data)
ans =
  797.2250  801.9625
```

```
>> writeline(ard, 'p 128')
>> runOnce = true; pause(.1); oscope2; mean(data)
ans =
  513.4075  513.5800

>> writeline(ard, 'p 50')
>> runOnce = true; pause(.1); oscope2; mean(data)
ans =
  224.6100  201.6725
```

The PWM filter output (column 2) corresponds better with 4*PWM than does the raw PWM mean, probably because the capacitor isn't just averaging 400 voltages, but is "measuring" a continuous, real-time average voltage across C1. In other words, edge effects from averaging 400 time points have a measurable effect on the sample mean, especially for extreme duty cycles (PWM values far from the middle = 128).

PWM in the frequency domain

For simplicity, let's set the PWM duty cycle to 50%. Can you predict what the data and spectra should look like?

The Fourier transform of a PWM square wave is not hard to derive, but looking up the answer is easier, e.g., https://mathworld.wolfram.com/FourierSeriesSquareWave.html. The only nonzero frequency components are discrete odd harmonics with decreasing amplitudes $\frac{4}{\pi n}$, with n = 1, 3, 5, ... at frequency nf_0, where f_0 is the square wave's frequency, ~980Hz for Arduino's D5 pin. That means we should expect non-zero harmonics around these frequencies

```
980 * (1:2:13) % Hz
```

```
ans = 1x7
         980        2940        4900        6860        8820       10780       12740
```

with amplitudes (in ADU)

```
(1023 / 2) * 4 / pi ./ (1:2:13) % ADU
```

```
ans = 1x7
  651.2620  217.0873  130.2524   93.0374   72.3624   59.2056   50.0971
```

Let's measure it:

```
writeline(ard, 'p 128');
pause(1); % extra settling time
runOnce = true; oscope2;

sr = length(data)/dt*1e6 % sample rate (#/sec)
ans =
   4.4643e+03
figure
aspectrum(data(1:end-2,:), sr); % try trimming a few data points ... will discuss why later
```

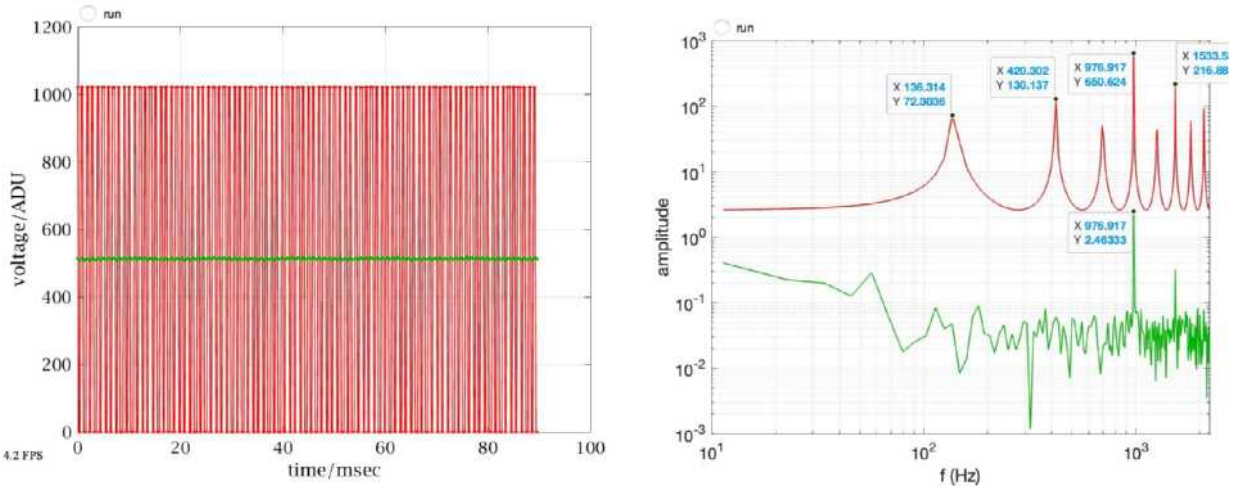


Figure 4. Time series and spectra for PWM = 128.

`aspectrum` hides the dc component by design, so you have to guess that it's 512ADU (or query `mean(data)`). The largest frequency component is 977Hz at 651ADU amplitude, consistent with Arduino's `analogWrite` documentation saying the PWM frequency is 980Hz. There's a peak with the expected 3rd harmonic amplitude, but at the *wrong* frequency: 1533Hz instead of 3*977 = 2931Hz. In fact, there are a variety of lower frequencies that don't belong – and that's because those are *all artifacts*, the result of a process called "aliasing", which we'll discuss later. It's not a coincidence that the 3rd harmonic *appears to be* at `sr - 977*3` = 1553Hz, the 5th harmonic *appears* at 977*5 - sr = 420Hz, the 7th harmonic *appears* at 2*sr - 977*7 = 2.09kHz, on the right side of the screen with amplitude 92.96, the 9th harmonic *appears* at 2*sr - 977*9 = 136Hz with amplitude 72.30. Aliasing creates all of these artifacts – *real* signals but in the *wrong* places in the frequency domain.

The RC filtered data (Fig. 4 green) has dc amplitude = 513.30 measured above, and *all* of its frequency components are <1ADU with the exception of the PWM frequency who's amplitude is 2.46ADU. So this filter exhibits gain at the PWM frequency of 2.46/651 = 3.78e-3 or an attenuation of 99.6%.

Modeling the RC filter

As explained earlier, we avoid solving differential equations by modeling circuits at distinct frequencies, treating capacitors as (complex) impedances (generalized resistance): `Zc = 1/(1j*2*pi*f*C)`, and applying the Voltage Divider Equation. Although repeating this formula over and over can help you memorize it (and you should, i.e., you might be asked it at a job interview), we wrote a simple function `zc(farads, hertz)` in the previous chapter. We solve two impedances in series using the voltage divider equation.

```
R1 = 5.00e3; % ohms
C1 = 10.0e-6; % farads
freqs = logspace(-1,3)'; % Hz
ZC1 = zc(C1, freqs); % capacitor's impedance
Vc = ZC1 ./ (R1+ZC1); % voltage divider equation
gain = Vc; % complex numbers
loglog(freqs, abs(gain)); grid
xlabel('frequency/Hz'); ylabel('Vout/Vin');
title(sprintf('R=%.2gk\\Omega , C=%.2g\\muF',R1/1e3,C1*1e6));
```

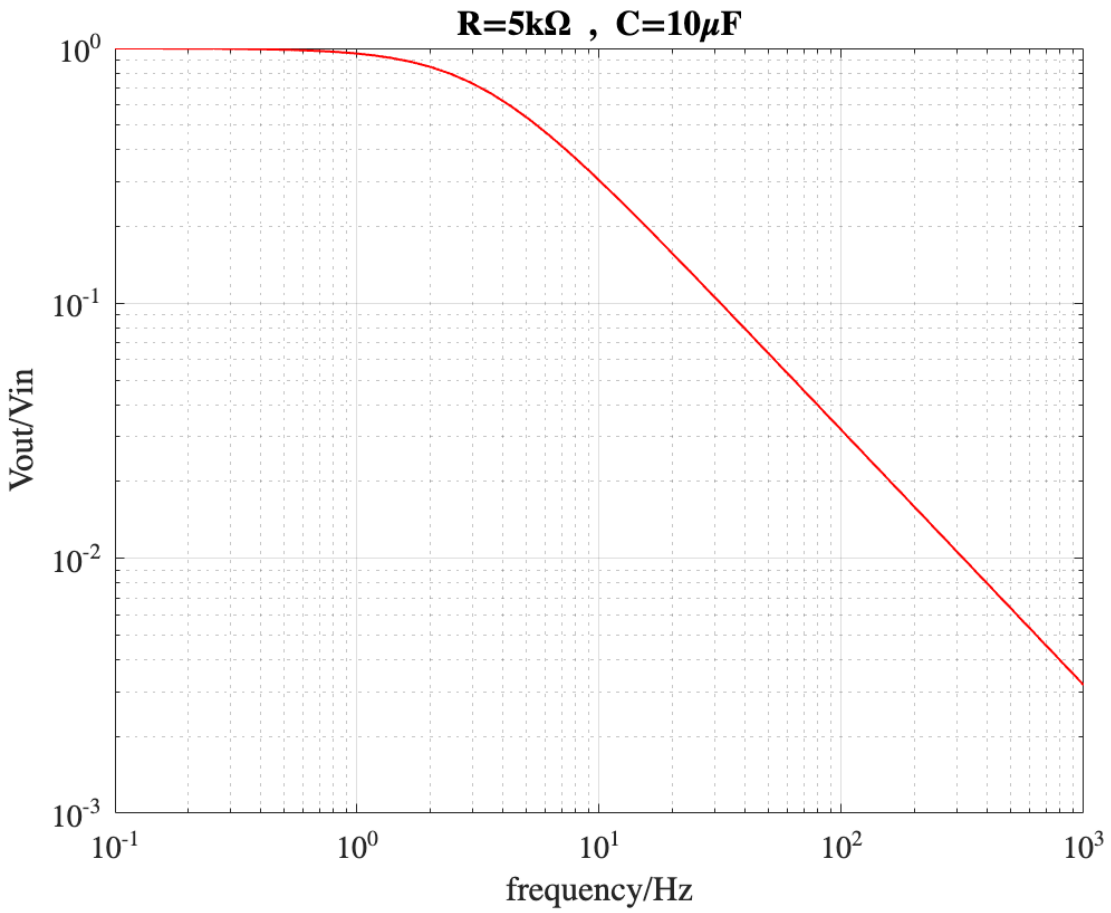


Figure 5. Modeled gain of our RC filter.

Let's explore what we just did, since there's a lot of implicit knowledge here. First, let's dissect the code:

1. `>> help logspace` tells you what that function does and expects for arguments, `>> which logspace` tells you where it is (i.e., in 310MATLABFiles or in the bowels of MATLAB's toolboxes).
2. Why the prime (') at the end of logspace? To transpose the default row vector into a column vector, which has advantages for plotting multiple curves.
3. We're solving 50 equations at 50 distinct frequencies, without any looping! `>> size(freq)` or the workspace panel of variables shows you the size of variables.
4. Why are some divisions ./ and others /? If the denominator is a vector, then ./ is necessary to do "pointwise" division, otherwise MATLAB interprets "/" as a linear algebra operation that doesn't apply to this data.
5. We're applying the voltage divider equation (algebraically) to the two series impedances R1 and ZC1. The fact that one of them is a capacitor is encapsulated in the complex value (and sign) of its impedance.
6. `gain` are complex numbers, which if plotted (in the complex plane) doesn't make much sense. We plot their magnitudes `abs(gain)` but keep the complex phase information which is necessary for all calculations (like voltage divider equations).
7. Why do I put `xlabel` and `ylabel` commands on the same line? Generally code can be more readable with one command per line. After searching for where something is and finding it off on the right side, you realize the value of putting important steps on their own line. In this case, I think of "labeling" as the task on that line, and "creating the plot" (with 2 commands) on the previous line. I value compactness and believe sometimes compactness offers more clarity, but there are other valid programming styles.

8. `title(sprintf(...)` has some wierd syntax, and why are they nested? For debugging, I run the `sprintf` on the command line alone to debug it. Since only `title` uses the string, nesting the commands saves a line and keep the workspace less cluttered. sp`rintf` uses a cryptic and powerful syntax inherited from the C language. In this case, it's printing a LaTeX string that `title` will interpret. The \\ generates a literal \ preceeding the name of a Greek letter. I divide R and C by appropriate powers of 10 anticipating more natural values and units.

Now dissect the results:

1. We can think of the curve (Fig. 5) as the "transfer function" of the RC circuit, where Vout/Vin = gain. Since it's a passive circuit, the gain is always less than 1.
2. Low frequencies "pass" to the output with low attenuation. Higher frequencies are increasingly attenuated. Hence this circuit is called a "low pass filter".
3. This curve can be characterized as having two assymptotes and a "corner" where they meet. (Note: it's linear only because it's plotted on a log-log scale – and that's *why* we are plotting it on a log-log scale – recall Chapter 2).
4. The corner frequency (https://en.wikipedia.org/wiki/Cutoff_frequency) corresponds to |ZC1| = R1, is at

```
fc = 1/(2*pi*R1*C1) % corner frequency (Hz)
```

```
fc = 3.1831
```

But we're still exploring the weeds, here's the *significance* of this model. Our RC filter model predicts gain at the PWM frequency, 976Hz ~ 3.5e-3 from the far right side of Fig. 5, *very* close to what was measured by the ratio of the two 976Hz peaks in Fig. 4 (3.78e-3).

I have to admit that at first I used a cheap 4.7uf aluminum electrolytic capacitor which gave a higher gain ~ 5e-3 at the PWM frequency, and was stumped for a while about the mismatch. Aluminum capacitors aren't ideal at kHz frequencies, they exhibit significant "effective series resistance" (ESR), which increases the voltage across them. Their common application is to attenuate low frequencies ~120Hz (rectified 60Hz). The results above are with a tantalum 4.7uf capacitor (https://en.wikipedia.org/wiki/Tantalum_capacitor), who's ingredients mostly originate from child labor in Africa (https://en.wikipedia.org/wiki/Coltan). Quality film capacitors are much larger, but have lower ESR and likely better agree with theory.

The model predicts that DC and frequencies << `fc` pass with gain of 1. Also voltage `gain = 1/sqrt(2)` ≈ 71% at `fc` (or power = `V^2/R` = 1/2 = 50% at `fc`), and decreases above `fc`. If you're convinced the model reasonably describes the circuit, then go ahead and **change the values of R1 and C1 in the model** and understand their effects. Increasing the product R1*C1 shifts the fc lower and shifts the curve left, and conversely decreasing the product R1C1 shifts the corner and curve to the right. To attenuate the PWM frequency below 1ADU requires a leftward shift.

Assessment: What component values in your kit will achieve that?

Attenuation and Settling time

Why not just use a much bigger R1C1 product for greater attenuation? Because changes in PWM values will take longer to settle to their dc assymptote. Convergence time can be measured as follows:

```
writeline(ard, 'p 0');
pause(1); % let the value of 0 settle
runOnce = true;
writeline(ard, 'p 255'); oscope2;
plot(t,data);
```

I put 2 commands on one line in case you paste them on the command line so they execute as quickly in succession as possible, since we're trying to measure a transient phenomena. Here's what I got:

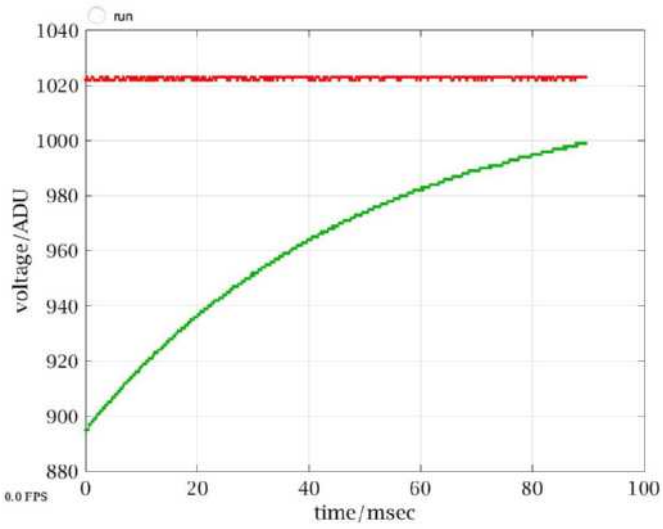


Figure 6. Transient response following a step from PWM = 0 to 255, showing apparent exponential convergence towards 1023.

You may recall the Arduino is programmed to execute `delay(200); /* msec */` after `analogWrite`, and MATLAB has finite latencies between commands, so that's why the the filter already has gotten from 0 to almost 900ADU before measurements began. I encourage you to add `pause(0.1);` and longer pauses between `writeline` and `oscope2` to explore convergence further to the right – how long should we wait for convergence?

What determines the settling time? The "time constant" of an RC circuit is $\tau = RC = \frac{1}{2\pi f_c}$, and the voltage converges as $1 - e^{-t/\tau}$. See https://en.wikipedia.org/wiki/RC_time_constant for further discussion.

But back to practicality, we have a fundamental conflict between a filter that settles quickly and one that attenuates the PWM frequency deeply. Moving Fig. 5 left improves PWM attenuation but makes settling slower, and visa versa ... what to do?

2nd order filter

What we really need to do is change the *slope* of Fig. 5 – but how? By cascading two RC low pass filters in series, as in Fig. 7.

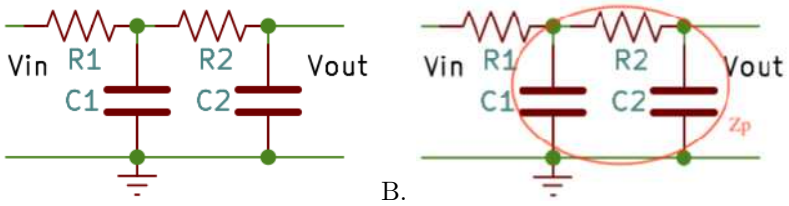


Figure 7. Two stage or 2nd order filter. If R2 >> R1, we'll show the two filters can be treated independently. But to be more accurate, we should treat the filter as R1 in series with the circled circuit called Zp.

Let's start by simulation using the simplest model, treating the two filters independently. Compute the voltage across C1 using the voltage divider equation, $V_{C1} = V_{in} \frac{Z_{C1}}{R1+Z_{C1}}$, followed by $V_{C2} = V_{C1} \frac{Z_{C2}}{R2+Z_{C2}}$. A more accurate model recognizes that R2 in series with C2 are in parallel with C1 (Fig. 7B), we'll consider that later.

So we don't make the settling time any slower, let's shift the second stage's corner frequency higher, e.g.,:

```
% 2nd order filter without loading:
R1 = 10e3; C1 = 200e-9;
ZC1 = zc(C1); % freqs is persistent from prior zc call
gain1 = ZC1 ./ (R1+ZC1); % Vo = 1
R2 = 2e3;  C2 = 10e-6;
ZC2 = zc(C2);
gain2 = ZC2 ./ (R2+ZC2);
gain2stage = gain1 .* gain2; % very different from abs(gain1) .* abs(gain2)
loglog(freqs,abs([gain1 gain2 gain2stage])); grid
ylabel('gain'); xlabel('frequency/Hz')
legend('stage1','stage2','stage1*stage2','location','best')
axis tight;
```

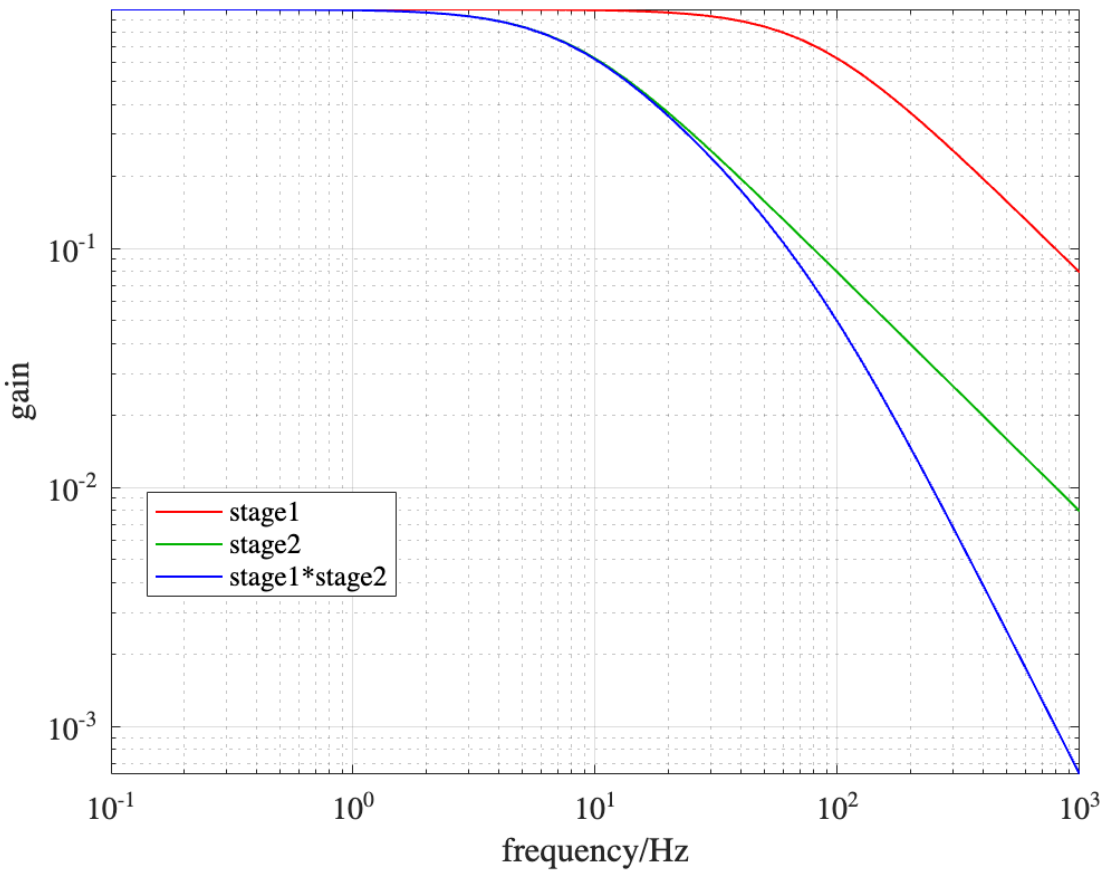


Figure 8. Cascading 2 RC filters results in faster attenuation to the right of f_c.

The two stages have corner frequencies:

```
fprintf('fc1 = %.1fHz, fc2 = %.1fHz\n', 1 ./ (2*pi * [R1*C1 R2*C2]))
```

```
fc1 = 79.6Hz, fc2 = 8.0Hz
```

Shifting the stage2 corner to the right makes its gain close to 1 at `fc1`, resulting in nearly the same corner frequency of the cascaded filter as `fc1`, but above `fc2`, the slope becomes twice as steep. The corner frequency of the 2 stage filter can be queried from the data a few different ways, e.g., using `find`, `min`, or `spline`. The last is the most accurate and compact as well.

```
help spline
 spline Cubic spline data interpolation.
    YQ = spline(X,Y,XQ) performs cubic spline interpolation using the
    values Y at sample points X to find interpolated values YQ at the query
    points XQ.
        - X must be a vector.
        - If Y is a vector, Y(j) is the value at X(j).
        - If Y is a matrix or n-D array, Y(:,...,:,j) is the value at X(j).
```

The corner frequency corresponds to a gain of 1/sqrt(2), so y = frequency and x = gain:

```
spline(abs(gain2stage), freqs, 1/sqrt(2)) % fcorner in Hz
```

```
ans = 7.8801
```

The 2 stage corner frequency shifted <5% left of `fc1`, while the attenuation at 1kHz increased by over an order of magnitude. Looks good? Will it work this way?

Impedance and Loading

Each filter stage was modeled above as a simple voltage divider, and the combined gain was the product of two independent transfer functions based on the voltage divider equation. But as emphasized in Fig. 7B, $V_{C1} = V_{in}\frac{Z_{C1}}{R1+Z_{C1}}$ is wrong. A correct voltage divider equation would replace Z_{C1} with $Zp = Z_{C1}||(R2 + Z_{C2})$. The 2nd stage *can* affect (i.e., "load") the first. The output of the first voltage divider is$V_{C1} = Zp/(R1 + Zp)$, which then gets divided by $Z_{C2}/(R2 + Z_{C2})$.

Let's work it out, but make it easier by building a helper function to compute the impedance of two parallel impedances:

```
function zparallel = zp(z1, z2)
% returns the parallel impedance, z1 || z2
zparallel = z1 .* z2 ./ (z1 + z2);
end
```

Is it worthwhile to create a function consisting of just 1 line of executable code? I do it often to save error-prone keystrokes and make the resulting code more readable. Save `zp.m` in your PATH. Make sure to have a sketch of the circuit in front of you to refer to while reading the simulation code below:

```
Zparallel = zp(ZC1, R2+ZC2); % which is in series with R1
gain1P = Zparallel ./ (R1 + Zparallel); % which is the input to the 2nd stage'
loglog(freqs, abs([gain1 gain2 gain1.*gain2 gain1P.*gain2]));
ylabel('gain'); xlabel('frequency/Hz'); grid
legend('stage1', 'stage2', 'stage1*stage2','stage1 || stage2','location','best')
```

```
axis tight;
```

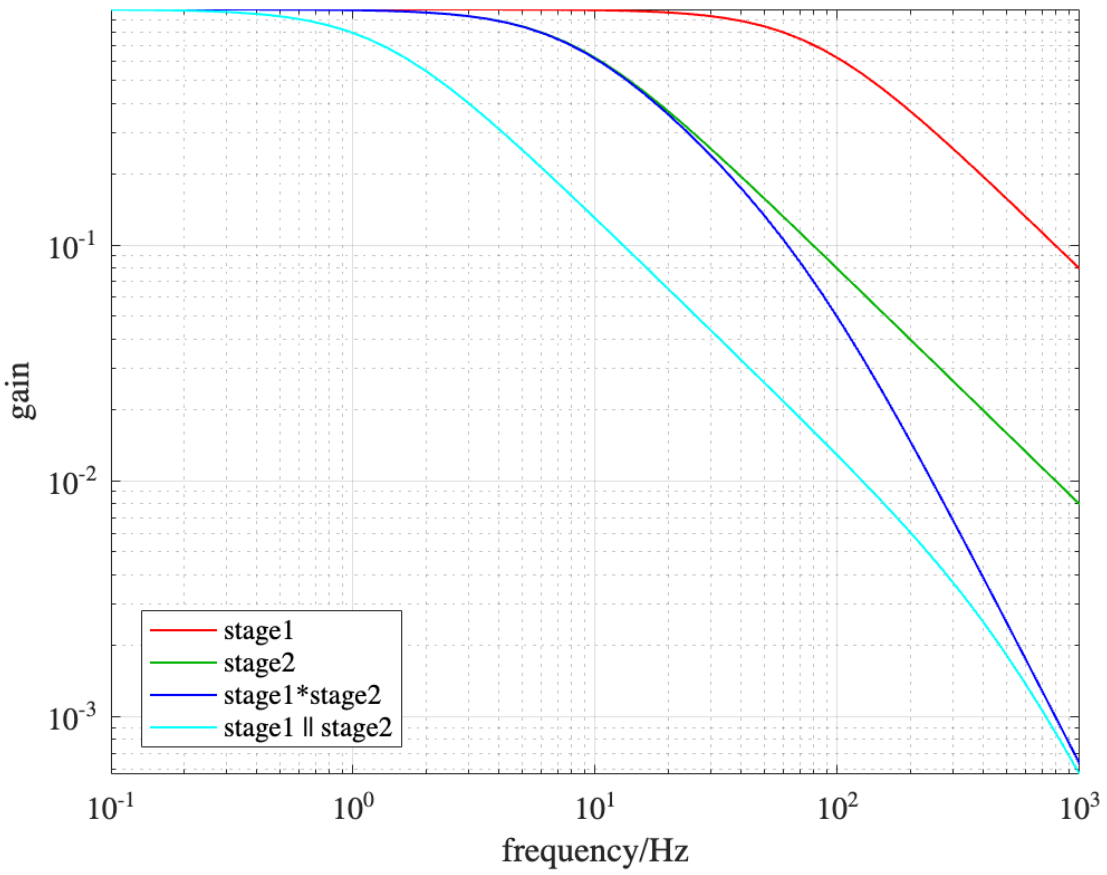


Figure 9. Stage 2 loads stage 1 of the filter reducing fc and increasing the settling time.

This parallel model is correct, and shows how adding the 2nd stage significantly changed the first stage's behavior, especially around `fc2` (but not its asymptotic behaviors – you see a hint ot the slope steepening on the bottom right). We call this general effect "**loading**", where the "load" of stage 2 on stage 1 changes the characteristics. Loading dramatically changes the gain, corner frequency, and the settling time. Loading effects are reducded when R2 >> R1, so let's **flip the order of the stages**, using the 2kΩ filter to drive the 10kΩ filter. This doesn't change the independent gain model (since `gain1 .* gain2 = gain2 .* gain1`), but it makes a big difference in the "real world" where impedance matters:

```
% Note I'm recycling variable names
R1 = 2e3; C1 = 10e-6;
R2 = 10e3;  C2 = 200e-9;
ZC1 = zc(C1); ZC2 = zc(C2);
gain1 = ZC1 ./ (R1 + ZC1); % ignoring loading by stage 2
gain2 = ZC2 ./ (R2+ZC2);
Zparallel = zp(ZC1, R2+ZC2); % which is in series with R1
gain1P = Zparallel ./ (R1 + Zparallel); % which is the input to the 2nd stage'
loglog(freqs, abs([gain1 gain2 gain1.*gain2 gain1P.*gain2]));
```

```
ylabel('gain'); xlabel('frequency/Hz'); grid
legend('stage1', 'stage2', 'stage1*stage2','stage1 || stage2','location','best')
```

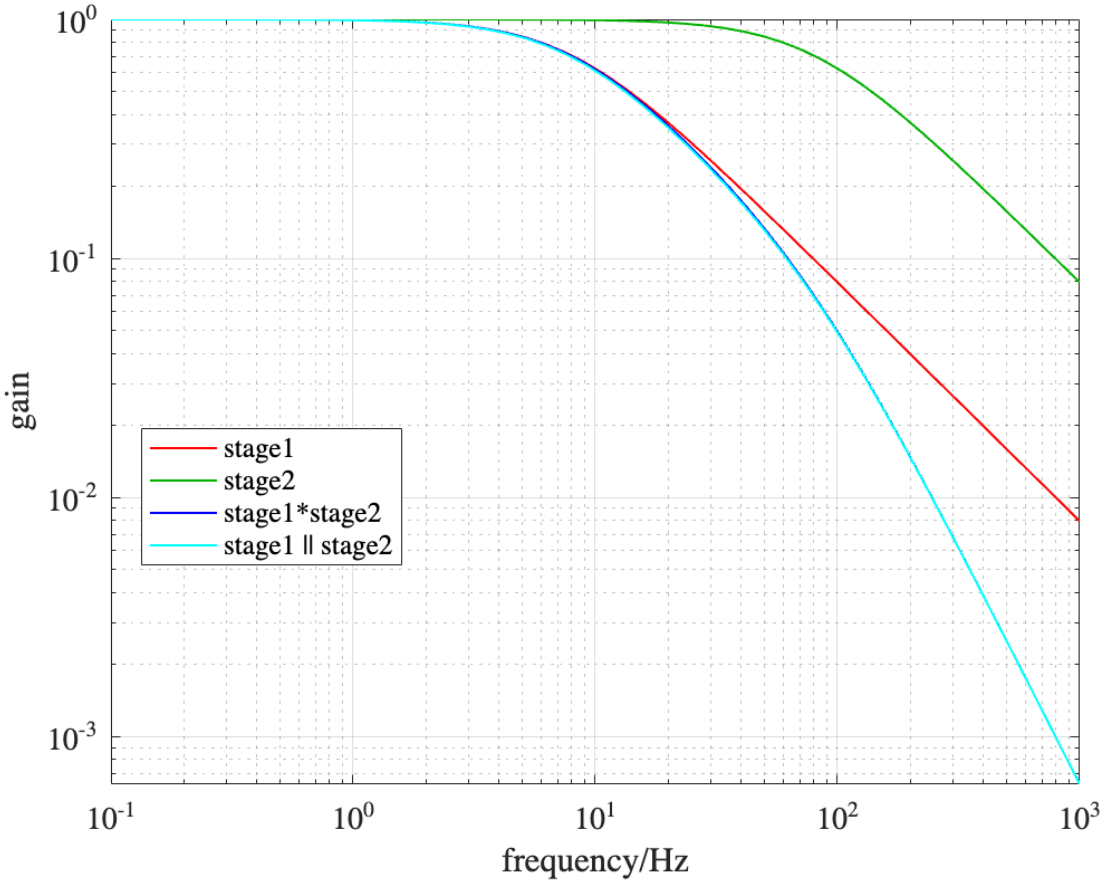


Figure 10. Reversing the stages so R2 >> R1 reduces loading on stage1 by stage2, and the stages behave independently.

A rule of thumb is that loading effects can be ignored when R2 >~ 10R1 (except for circuits requiring higher precision). In fact here, with the resistor ratio only 5, you have to zoom in (because of the log scale) to see the loading effect. Should we lower R1 or raise R2? Lowering R1 by an order of magnitude would dangerously load the Arduino's D5 output, reducing the PWM voltage and potentially burning out Arduino's voltage regulator. Raising R2 is safe, but whatever we hope to connect this filter to should have impedance Z >> R2 to avoid loading stage 2. So compromises and trade-offs must be made. That's the Art of electronic engineering.

Verifying our PWM filter

Build the circuit in Fig. 11A and B.

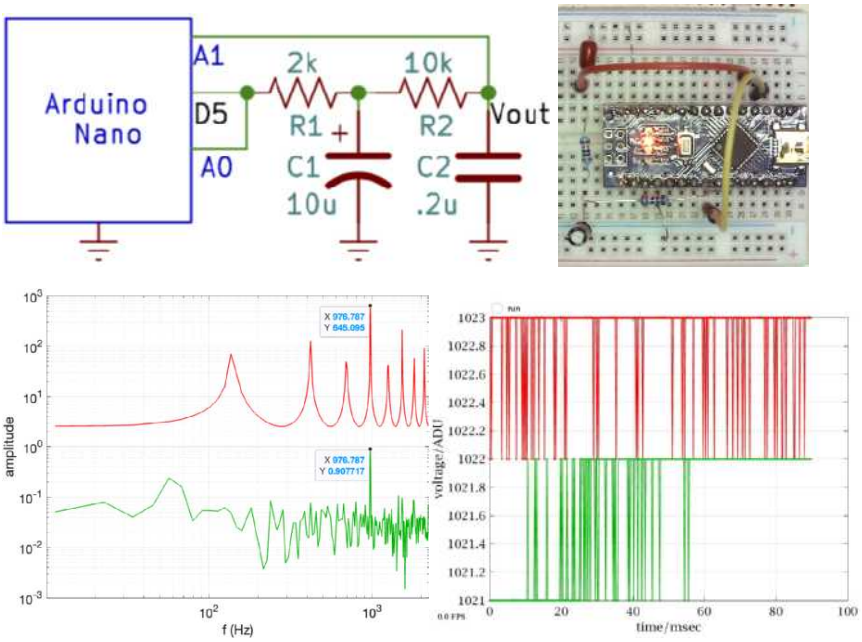


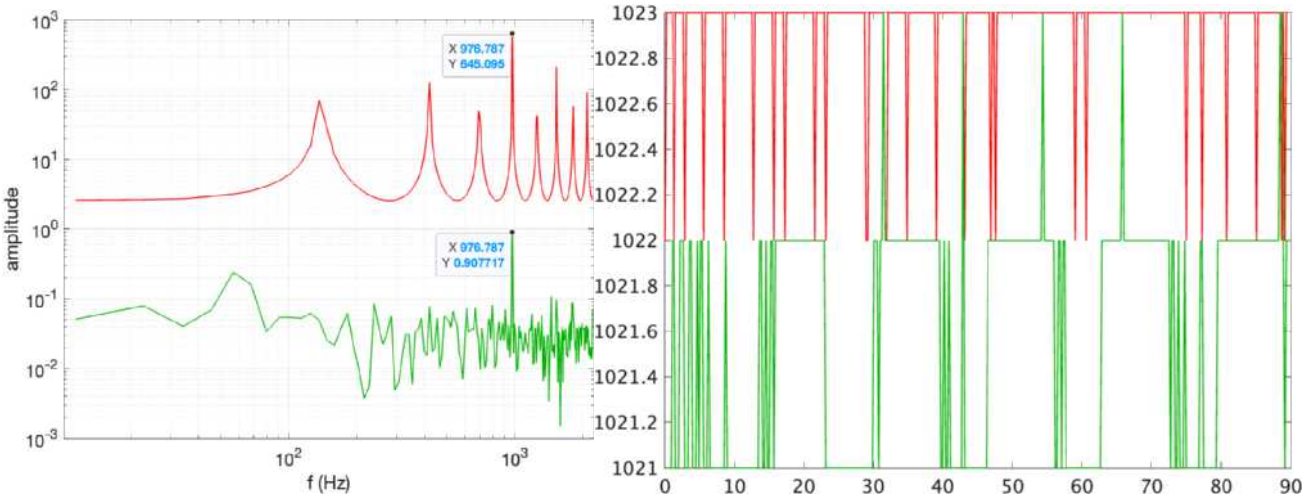
Figure 11. A. Schematic and B. photo of a 2 stage PWM filter under test results in spectra C for `analogWrite(5,128)`, and D, settling time < 200msec for a step from `pwmVal` = 0 to 255.

The 2nd order filter attenuates the 977Hz frequency component to <1ADU. More precisely, the measured gain is .908/645 = 1.407e-3 in Fig. 11C . The model, from the right side of Fig. 10C, predicts gain between .001-.002, we can get a more accurate estimate with:

```
fprintf('predicted gain at 977Hz = %f\n', spline(freqs, abs(gain1P.*gain2), 977))
```

```
predicted gain at 977Hz = 0.000664
```

```
R1 = 1.98e3; % measured with dmm
C1 = 4.7e-6;
R2 = 9.94e3; % measured with dmm
C2 = 200e-9;
```



```
[mean(data) std(data)]
ans =
  510.0550  514.3650  508.5912    0.5408
```

Those measured and modeled results agree within <1%, well within tolerances of the components – but honestly, that wasn't the case initially. I had to replace the 100nf capacitor that came in my kit (a blue one with markings μ1J100, which measured 101.3nF with my DMM) with a yellow monolythic 100nF capacitor. The blue one resulted in ~15% *greater* attenuation than predicted. Capacitors can deviate significantly from their ideal model (especially cheap ones that may have ended up in electronics kits for failing to meet specifications). Engineers can avoid problems due to non-ideal or out-of-specification behaviors by being cautious and designing safety margins into circuits. If a specific value of C is required, e.g., to set a particular resonant frequency or timing, then more attention must be paid to the components. If their function is to filter (and you can afford to define the corner frequencies with ample safety margins, then the particular value isn't critical. In this case, the blue capacitor offers *better* attenuation of the PWM frequency than the model predicts – so why not use it?

The 2 stage filter already is within 2ADU of convergence to the largest possible transient step (Fig. 11D) without adding any additional delay (beyond Arduino's 200msec + MATLAB's latency). If you wondered why there was a 200msec delay parameter in the Arduino code, now you know. Our 2 stage filter appears to work as its model predicts, providing DC voltages with several hundred msec faster rise time (Fig. 11D vs Fig. 6) AND less than half the PWM modulation noise of a single stage filter (Fig. 11C).

More about Aliasing

You've hopefully been wondering about the funky shapes of the spectra of the raw PWM signal (red curves in Figs. 4 and 13) – where do all the secondary bumps and valleys come from? The answer is that square waves are sums of odd harmonics, that become misplaced or "aliased" due to the discrete measurement process. Wikipedia's page (https://en.wikipedia.org/wiki/Aliasing) has several animations showing how sampling sine waves with frequencies above the sample frequency/2 (called the Nyquist Frequency, https://en.wikipedia.org/wiki/Nyquist_frequency) result in "reflections" of the sine wave frequency appearing (incorrectly) at a lower frequency. Fig. 12 show a 50% duty cycle PWM square wave produced by and measured with a Raspberry pi Pico2 running the circuit in Fig. 3. It's PWM frequency is programmable and was set to 1kHz, and its sample rate was 41kHz, hence aliasing reflects frequencies above `sr/2` = 20.5kHz. The Pico also has much more memory, hence 1000 samples/channel were measured (vs. 400 on the Arduino), which has the effect of more than doubling the frequency *resolution* of the spectra.

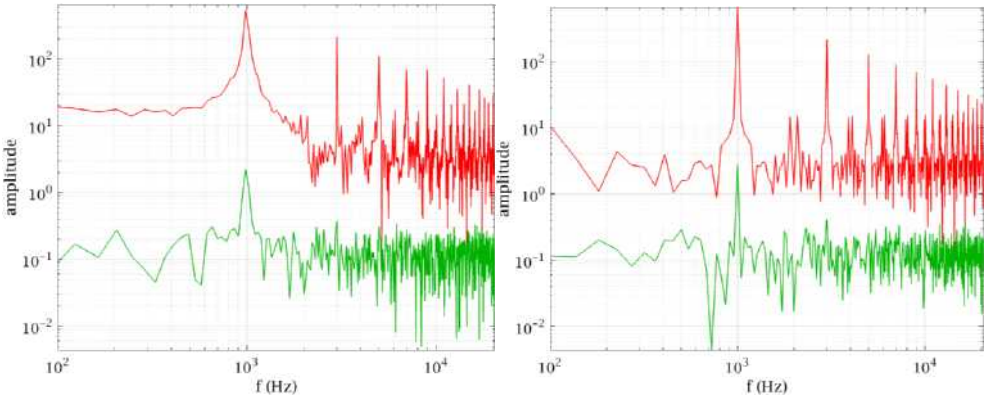


Figure 12. 1kHz PWM square wave generated and sampled on a $4 Raspberry pi Pico2 at `sr =` 41kHz. Left: spectra of 1000 samples. Right: spectra trimmed to an integral number (22) of periods sharpens the peaks.

The Pico2 uses a different PWM "engine" but its square wave is likely not significantly different. The big difference is the sampling (Nyquist) frequency is higher. Fourier analysis shows that a square wave consists of an infinite series of odd harmonics, $V(t) = \frac{4}{\pi}\Sigma_{n=1}^{\infty}\frac{1}{(2n-1)}\sin(2\pi(2n-1)f_0t)$, (e.g., https://www.mathworks.com/help/MATLAB/math/square-wave-from-sine-waves.html), with amplitudes decreasing as $1/(2n-1)$. The spectra in Fig. 12 show the 1/n decreasing harmonic amplitudes, with interleaved aliased harmonics "reflected" off the right side (and left side for the double peaks) of the graph. Trimming the sample to an integral number of periods narrows the peaks (which are ideally "delta functions") and reduces end artifacts that are also problematic with Fourier analysis. That the *reflected* amplitudes don't *appear* to fall with 1/n is *another* artifact – of the loglog axes.

Aliasing is avoided by 1) sampling data faster – at least twice the highest frequency present in the data; 2) using "anti-aliasing filters" – low pass filters that attenuate signals above the Nyquist frequency; and 3) using delta-sigma ADCs (https://en.wikipedia.org/wiki/Delta-sigma_modulation) which perform some clever signal processing tricks that substantially increase the aliasing frequency.

Conclusions

1. We've filtered `analogWrite`'s PWM output into a clean DC voltage with <1ADU of PWM noise and fast settling time within 1ADU in ~ 200msec.
2. We've demonstrated a method to numerically model AC circuits in MATLAB, with discrete frequency vectors (made with `logspace`), complex impedances for capacitors (computed with our `zc` function), and using algabraic formula like the voltage divider equation, Ohm's Law, and KVL, but with MATLAB doing point-wise complex algebra on each frequency component. These models accurately reproduce circuit behaviors, including gain and transient responses, but large errors may result from capacitors deviating from their ideal values and models.
3. We used the Arduino to generate PWM signals and measure them. With spectral analysis we compared modeled and measured amplitudes in the "frequency domain". In the "time domain" we measured settling times to step voltage changes. These are sophisticated measurements and analyses – typically requiring expensive, sophisticated apparatus – that were done with an inexpensive microcontroller and MATLAB!
4. We've seen that if f_{corner} is too low, settling time will be high. If f_{corner} is high, then PWM noise will leak or "bleed" through the filter. A second order filter offers fast settling time and low leakage, but at a cost of complexity and higher output impedance. Which filter is best? These are engineering trade-offs we have to decide.
5. We wrote a generic `zp` function to facilite computing parallel impedances (real or complex).
6. We explored how circuits in series can interact in complex ways by loading. Loading effects are minimized when the downstream circuit has impedance >> the upstream circuit. This is a generalization of what we learned with $R_{Thevenin}$ in Chapter 2 – that when $R_{Load} >> R_{Thevenin}$, then $V_{Load} \approx V_{Thevenin}$, and $R_{Thevenin}$ can be ignored.
7. We don't (yet) have a function generator to measure frequency response curves. We will construct one soon, automate its frequency control, and measure curves like Fig. 10 in seconds. But at this point we have confidence in our parallel impedance model because it matches theory at the two frequencies that we measured, 0 = DC and 980Hz. (Note: a trick to measure frequency response without a function generator is with a square wave, since they consist of many harmonics (Fig. 12). The SNR of the harmonics generally decreases with 1/n, so "sweeping" a sine wave can be a more accurate, but slower, method.
8. We haven't exhaustively tested our system – e.g., how linear is it across the PWM range, might there be "missing values" or other pathologies, etc. But we're also not done – making a clean analog DC voltage with the Arduino isn't the end goal. When we use this voltage to control the LM317, we'll do more thorough testing and characterization of the entire system.
9. We've seen how aliasing and non-integer number of periods (non-periodic boundary conditions) can create confusing artifacts in the Fourier spectrum.

10. I hope you are feeling that this relatively inexpensive home-brew apparatus has provided extraordinary insights into some rich electronic circuit behavior. And we're still just getting started. Our next task is to figure out how to *use* this clean DC voltage to control the LM317's output voltage. That should let us change the voltage and make measurements programmatically many times per second.
11. What other concepts have you learned from this activity??

Open questions

1. Why are A0 and A1 not identical in Fig. 2 and the data below it. How will that be a problem? How might we fix it?
2. How is the Fourier transform able to "measure" amplitudes << 1ADU, even < 0.1ADU and 0.01ADU? If a voltage measurement has fundamental uncertainty >0.5ADU, what determines the corresponding noise "floor" for a spectral amplitude?
3. Figure 11D shows the filter hasn't *quite* converged to 1023 after the Arduino's `delay(200)`. The voltage continues to change for ~60msec more. So why isn't `delay(260)` better? What trade-offs (pros and cons) are we balancing?
4. How would you most easily *prove* that your PWM filter (and circuit) is working? (I.e., can you write an automated MATLAB script to prove it?)

Appendix: startup.m

The file `startup.m` in your PATH gets executed when MATLAB launches. Here's what's in mine:

```
c = [... % red green blue cyan magenta yellow gray black
    1    0     0
    0    0.7   0
    0    0     1
    0    1     1
    1    0     1
    1    0.95  0
    .5    .5   .5
    0    0     0];
set(0,'DefaultAxesColorOrder', c);
clear c
set(0,'DefaultAxesColor', [1 1 1])
set(0,'DefaultAxesLineStyleOrder','-|--|:|-.')
set(0,'DefaultUIControlBackgroundColor','white');
set(0,'DefaultFigureColor',[1 1 1]);
set(0,'defaultsurfaceedgecolor',[0 0 0])
set(0,'DefaultFigureColormap',jet)
format compact
more on;
set(0,'defaultLineLineWidth',1); % to make grids dotted (why??)
```

References

1. https://en.wikipedia.org/wiki/Top-down_and_bottom-up_design
2. https://docs.arduino.cc/learn/microcontrollers/analog-output
3. https://docs.arduino.cc/learn/microcontrollers/analog-output
4. https://en.wikipedia.org/wiki/Cutoff_frequency

5. https://en.wikipedia.org/wiki/Tantalum_capacitor
6. https://en.wikipedia.org/wiki/Coltan
7. https://en.wikipedia.org/wiki/RC_time_constant
8. https://en.wikipedia.org/wiki/Time_constant
9. https://en.wikipedia.org/wiki/Aliasing
10. https://en.wikipedia.org/wiki/Nyquist_frequency
11. https://www.mathworks.com/help/MATLAB/math/square-wave-from-sine-waves.html
12. https://en.wikipedia.org/wiki/Delta-sigma_modulation