16. Bode Plot of an R-C Series Circuit

Table of Contents

Introduction

Reflect on what we've accomplished. You've practiced for weeks having "dialogs" in MATLAB with your data and models. You've learned to build hardware, software, and data interfaces between circuits, microcontrollers, and MATLAB. We started by characterizing simple 2 terminal devices by measuring their I-V curves and Thevenin resistance, then explored 3-terminal transistor characteristics in their 3 common states. We applied transistor states towards making building blocks like switches that turned on LEDs in dim lighting or amplifying imperceptible currents through our bodies. Other transistor circuits we built were emitter followers and current mirrors, and we built an audio amplifier with an op-amp. We analyzed time-varying (AC) voltages in the frequency domain, with Fourier Analysis transforming $V(t)$ and $I(t)$ into corresponding frequency components, $V(\omega)$ and $I(\omega)$. These have the advantage of obeying the algebraic equations of DC electricity: Ohm's Law, Kirchhoff's Laws, Voltage divider equation, etc., instead of differential equations governing their time domain duals. The price for algebra instead of calculus is that we have to use complex numbers for V and I, and do the algebra with different coefficients $Z(\omega)$ at each frequency $\omega = 2\pi f$. Fortunately MATLAB does much of the complex number computation automatically.

At another level, we learned a lot about abstraction. I sometimes would ask students to tell me what a car is, and relished in the diversity of answers. Some describe its components, some its functions like the obvious "transportation", but also "a symbol of status". One of my favorite answers describes an economic function of a car, "an efficient way to transfer my hard-earned wealth to the richest corporations". So "What does a resistor do?" I think now would illicit many more answers than at the start of this course. We've used R's to transduce currents into voltages, i.e., $V = iR$; to limit currents; to divide voltages, $V_{OUT} = V_{IN}\frac{R2}{R1+R2}$; in AC circuits they determined corner frequencies, time constants, convergence rates, etc.

We've started thinking of and describing circuits and schematics in more modular, abstract ways that allow us to manage increased complexity. Our LM317 controller is hard to understand in terms of its various R's, C's, and three Q's. But it makes more sense as a PWM filter who's output impedance is lowered by an emitter follower, that programs a current mirror. Analogously, it would be difficult to understand this paragraph if one tries to read it one letter at a time.

Even more abstract, when we looked at frequency control of our function generator, we realized our "Arduino controlled current mirror" was a versatile object in its entirety. Modifying only the software component, `volts2pwm` to `freq2pwm` was enough to control an entirely different instrument. Software flexibility was key to implementing simple "transfer functions" between V, f, and PWM, which we curve fitted from empirical data. This *negative feedback* compensated for hardware linearities and nonlinearities. E.g., that one device increased output and the other decreased output vs. PWM was inconsequential. What other higher level building blocks might be added to our electronics repertoire?

Bode Plots

The "Bode Plot is a graph of the frequency response of a system" (https://en.wikipedia.org/wiki/Bode_plot). Bode drew them in the 1930's from piecewise linear asymptotic segments he learned to estimate. E.g., our PWM filters are characterized by a horizontal asymptote of gain = 1 on the left (low frequency) side, meeting a negatively sloped line at f_{Corner}. These graphical shortcuts were very useful especially before computers could easily calculate frequency responses or transfer functions, or we could measure them with automated equipment. In this final lab, we'll measure and simulate the frequency response of a circuit and compare both results.

What circuit should we study? The important point is that *the process* is the same for all circuits (with one input and one output). Over some range of frequencies, we measure and model $V_{IN}(\omega)$ and $V_{OUT}(\omega)$ and plot the gain $\equiv |V_{OUT}(\omega)/V_{IN}(\omega)|$ and phase shift = `angle`$(V_{OUT}(\omega)/V_{IN}(\omega))$ vs. ω or $f = \omega/(2\pi)$. For ease, we'll demonstrate this using one of the simplest circuits, a high pass filter, and please read Reflections on how other simple circuits can pose challenges to our hardware, e.g., requiring bias networks to protect the Arduino from negative capacitive (or inductive) voltages.

Methods

Refactoring

After running this lab, calling `oscope3`, my workspace looked like this:

```
>>who

Your variables are:

C1          ampl        bin         fd          npds        runOnce     theta
R1          amplitudes  data        freq        plotHandle  targetFreqs tt
Vtheory     ans         dt          freqs       rawdata     sr          vc
Zc          ard         fc          i           runBtn      t           vr
```

Too many variables to keep track of, too complicated, and I got lost debugging the "Comparing to theory" section, so I went back and added the next section below. How do we manage complexity? Sometimes simplification is a viable solution. We can recall where some of the complexity comes from. Raw data from Arduino, called `bin` is transformed to `rawdata` and `dt`, trimmed to `data` and `npds`. `npds` and `dt` $\rightarrow$ `freq` and Fourier transformed $\rightarrow$ `fd` from which we select `ampl` and compute `theta`. Better naming conventions might make this easier to follow – or add more verbocity and confusion? Regardless, we're doing *a lot* of math on the measurements and have a lot to keep track of. We can do a better job organizing the data by *structuring* it – using data structures to group related items. We've been using data structures already – recall `ard.NumBytesAvailable` – `ard` is a `SerialPort` structure with multiple *fields*. `runBtn` is a user interface data structure. So before proceeding with this activity (again), I'm going to put as much of the related data into a structure called `dat` (feel free to call it something else if you prefer). In MATLAB the more efficient way to make a data structure is to preallocate it: `>> dat = struct('fieldname',value, ...)`. The other way is just keep adding fields on the fly, and that's how I do it below. Note there are several kinds of "structures", of which `struct` is perhaps the oldest and simplest. Objects, cell arrays, tables, are some others.

oscope4.m

I saved `oscope3.m` as `oscope4.m` and started making changes. The formal name for this is "refactoring" (https://en.wikipedia.org/wiki/Code_refactoring). It's usually a bad idea to change too much without testing – because when the d*#$! thing no longer works, where are the bug(s)? To debug, I put breakpoints in the `catch` clauses to be sure they weren't executing all the time. Before almost every variable name, I added "`dat.`".

Once those changes were made and tested, I did one more refactor to add a visual indicator of the `arddeskew` algorithm and a GUI interface to change it. Can you find it and figure out how it works? Code is here:

```
%% oscope4.m -- 24jan24 putting variables in dat structure to reduce clutter
% using runBtn.UserData for runOnce

if ~exist('runBtn','var') || ~isvalid(runBtn)  % add the button once
    runBtn = uicontrol('style','radiobutton','string','run', ...
        'units','normalized','position',[.13 .93 .1 .04]);
    runBtn.Callback = 'oscope4';
    runBtn.UserData = true; % using this field for runOnce
    dat.skew = uicontrol('Style','popupmenu','Units','normalized',...
        'Position',[.1 .01 .18 .05],'String',{'No deskew','Spline','Fourier'});
    dat.trimThresh = 336;
    dat.runOnce = true;
end
if ~exist('ard','var') % initalize arduino
    disp('initializing arduino ...')
    ports = serialportlist;
    ard = serialport(ports{end},115200/2,'Timeout',2);
    clear ports;
    readline(ard); % waits for ard to boot
end

while runBtn.Value || dat.runOnce
    writeline(ard,'b'); % pause(.02);
    try
        dat.bin = read(ard,802,'uint16');
        dat.dt = bitshift(dat.bin(802),16)+dat.bin(801); % microseconds
        dat.raw = reshape(dat.bin(1:800),2,400)';
        [dat.data, dat.npds] = trim(dat.raw, dat.trimThresh, -1, 1); % adjust 2nd arg ~ mean(dat.raw)
        dat.t = linspace(0,dat.dt/1000,400)';
        dat.t = dat.t(1:length(dat.data));
        dat.sr = length(dat.raw)/dat.dt*1e6; % sample rate (#/sec)
        dat.freq = dat.sr/(length(dat.data)/dat.npds); % Hz, 1/sec
        dat.data = arddeskew(dat.data, dat.skew.String{dat.skew.Value});
        dat.fd = fft(dat.data); % data was trimmed to npds
        dat.ampl = dat.fd(dat.npds+1,:) / length(dat.data) * 2;
        dat.theta = rad2deg(angle(dat.fd(dat.npds+1,:)));
    catch
        flush(ard);
        fprintf('*');
        break;
    end
    plot(dat.t, dat.data, '.-');
    xlabel('msec'); ylabel('ADU'); grid on;
    title(sprintf('%.2fHz V1=%.2f V2 = %.2f deg=%.2f', ...
        dat.freq, abs(dat.ampl), diff(dat.theta)),'FontSize',12);
    displayfps;
    dat.runOnce = false;    % runOnce
end
```

Now `>>clear` the workspace and run `>> oscope4`

```
>> who
Your variables are:
     ard     dat     runBtn
>> dat
dat =
  struct with fields:

             skew: [1×1 UIControl]
    trimThreshold: 332
              bin: [574 584 576 577 572 571 569 567 565 562 ... ] (1×802 double)
               dt: 89608
              raw: [400×2 double]
             data: [358×2 double]
             npds: 13
                t: [358×1 double]
               sr: 4.4639e+03
             freq: 162.0965
               fd: [358×2 double]
             ampl: [-1.0425e+01 - 2.2964e+02i -1.0382e+01 - 2.2954e+02i]
            theta: [-92.5992 -92.5896]
```

Reiterating, the data is all still there, just organized differently. To access e.g., `freq`, we have to name the structure first, `dat.freq`. In the metaphor of the workspace as our office, we just made a folder or created a file cabinet. Frame rate is the same, behavior is the same. You might ask, why didn't we do this before? It would have saved the step of refactoring. But if we tried writing `oscope` the first time with all the `dat.`'s in the code, we'd likely have gotten hopelessly lost trying to debug it. When you're trying to figure out algorithms and get code to work, you want the syntax (and everything else) as simple as possible. Once the algorithm and basic code is behaving, you can add complexity in order to reduce complexity elsewhere (i.e., clean up the workspace). Refactoring is a smart thing to do. Documenting how you test code (in the code) increases the likelihood that refactoring doesn't add new bugs.

Hardware Interface

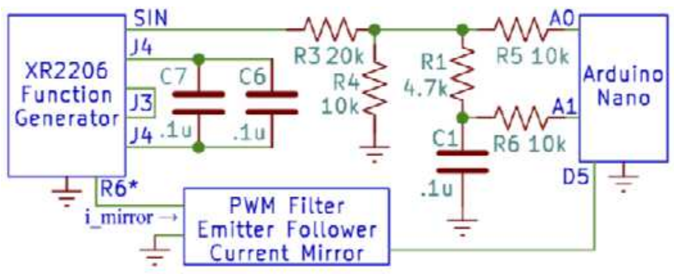


Figure 1. Schematic to measure frequency response of the R1+C1 series circuit ("device under test" or DUT).

In some kits, R1 = 5.1k is the nearest value, and R3 and R4, the essential voltage divider to reduce the SINE output to <5V, are replaced with 2k and 1k resistors. NOTE: there are two "R6" referenced here. The function generator's R6*=5.1k is a resistor on the middle of the function generator board, distinct from the other R6 on the right connecting to A1. Unique numbering is a challenge when combining circuits!

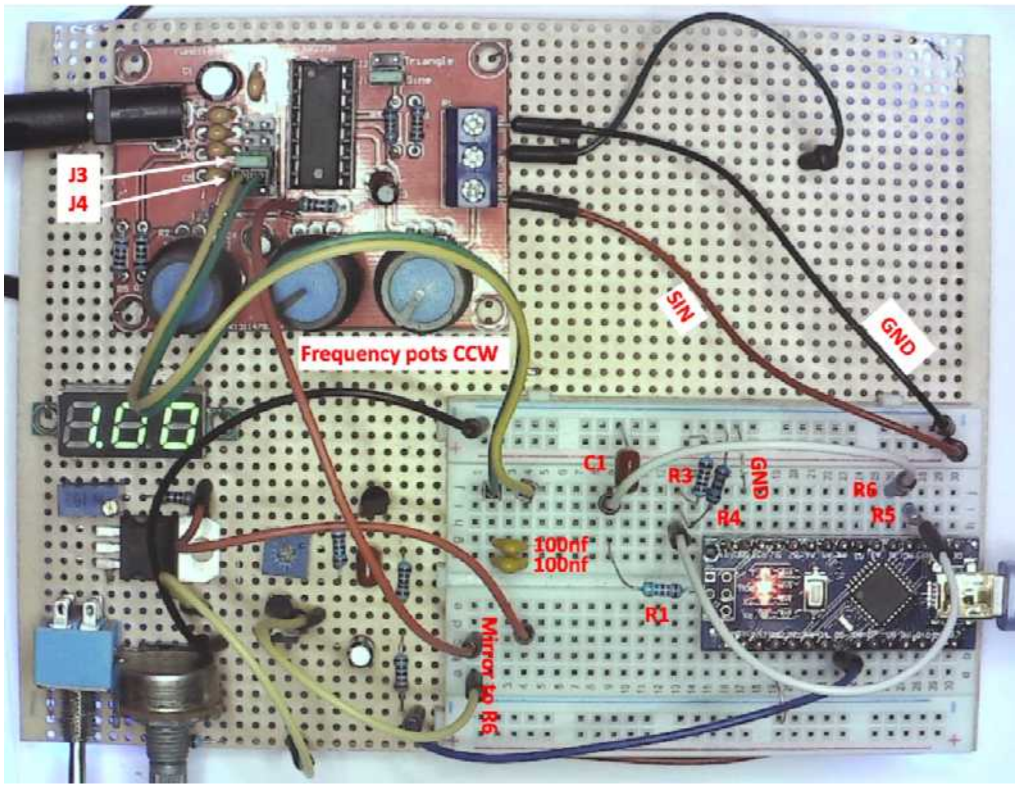


Figure 2. Implementation of Figure 1.

Frequency is controlled by the Arduino through the unmodified current mirror circuit developed to control the LM317 DC voltage. To maximize the usable range of PWM-controlled frequencies while avoiding resoldering the board, a male-to-female jumper connects two 100nf capacitors in parallel (on the solderless breadboard) to frequency jumper position J4, and J3 adds 47nf in parallel with these (and keeps you from loosing the jumper), resulting in a total timing capacitance and frequency range (computed in the previous chapter) of:

```
0.198uf -> 31.3 - 1812.8Hz
```

What's the center of that range on a logarithmic frequency scale? It's also know as the geometrical mean:

```
fprintf('logarithmic mean frequency = %.1fHz\n', sqrt(31 * 1813))
```

```
logarithmic mean frequency = 237.1Hz
```

Trial and error with kit components identified these two: a 5.1k resistor and 100nf capacitor with measured values:

```
R1 = 5080; % ohms measured with DMM
C1 = 95.1e-9; % farads with DMM
fc = 1/(2*pi*R1*C1);
fprintf('corner frequency expected at %.1fHz\n',fc);
```

```
corner frequency expected at 329.4Hz
```

Arduino Software

We continue using Arduino's `oscope1.ino` software. The Arduino responds to 2 serial commands:

```
    p <pwmValue> -- analogWrite(D5, pwmValue)
    b -- send 400x2 measurements of A0 and A1 appended with etime in microseconds
```

MATLAB Software

As discussed above, `oscope4.m` fetches blocks of time-series voltages on pins A0 and A1 and displays them like an oscilloscope would. It auto-initializes the Arduino and creates a graphical user interface (GUI) "run" button to turn on and off the data acquisition. It sends a '`b`' command to the Arduino, then reads back a block of binary data, creates a stable trigger with `trim.m`, corrects for temporal skew between A0 and A1 with `arddeskew.m`, measures the frequency and amplitude (via `fft`), displays the frame rate with `displayfps.m`, and updates the `plot` figure. Results are left in the workspace in a structure called `dat`.

`bodePlot(frequency, amplitude, fcorner)` is a new function which manages the graphics of displaying a Bode plot from the data provided in its arguments. Save the code below in your search path:

```
function bodePlot(f,a,fc)
% function bodePlot(frequency,amplitude, fcorner)
% generate a Bode plot in the current figure window
% amplitude is assumed complex and normalized
% 3rd argument draws vertical line at fcorner

clf;
subplot(2,1,1)
loglog(f,abs(a),'.-')
ylabel('amplitude');
grid; axis tight;
if nargin == 3, plotfc(fc); end

subplot(2,1,2)
semilogx(f, rad2deg(unwrap(angle(a))),'.-')
xlabel('frequency'); ylabel('degrees');
grid; axis tight
if nargin == 3, plotfc(fc); end
end

function plotfc(fc) % plot vertical line at fc
    ax = axis;
    hold on;
    plot([fc fc], [ax(3) ax(4)], '--c');
    hold off;
end
```

Initialization

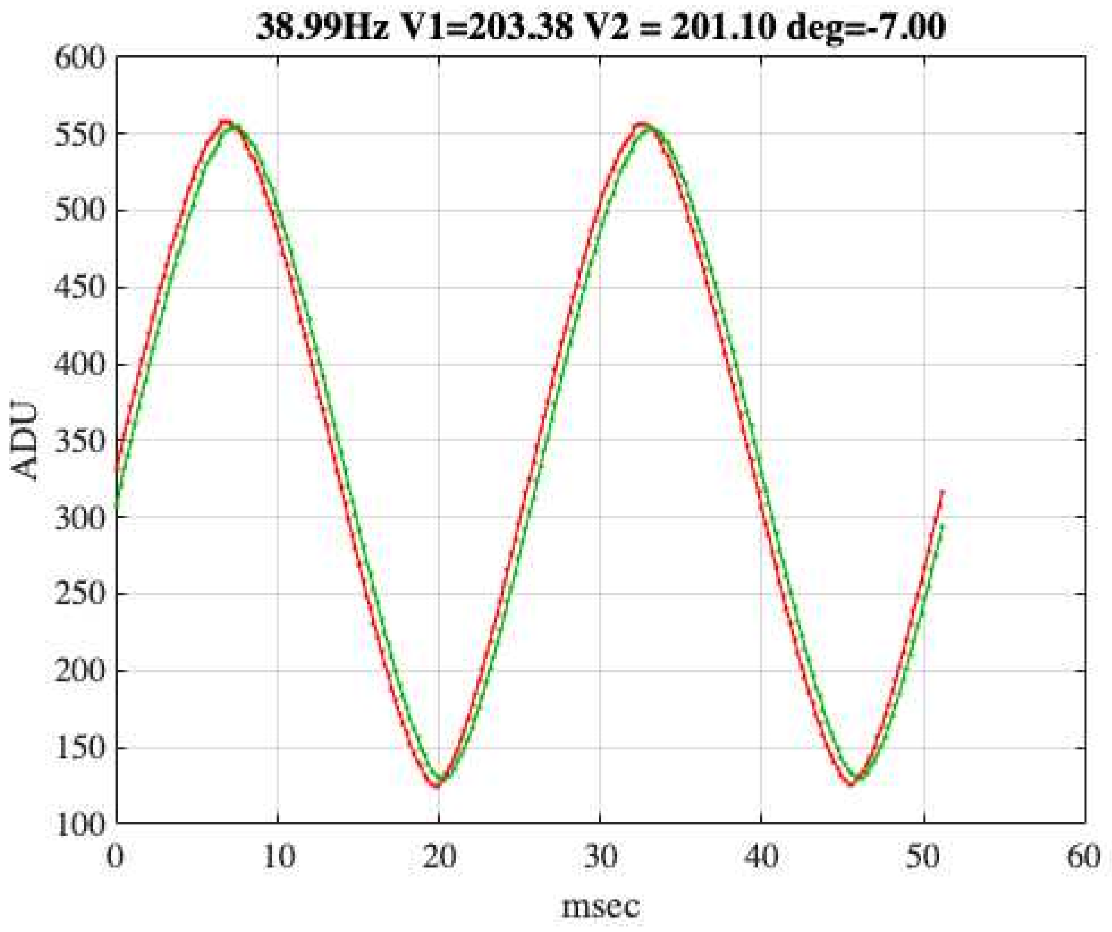
In an earlier version, I had written, "Make sure to run `oscope4` first, set deskewing to Fourier, and make sure the trim threshold is ok", but later realized that sentence could instead be written in code. Instead of telling you, the reader to do stuff, telling the computer ensures it happens automagically. Executable (and readable) code is much better than comments:

```
oscope4; % initialize ard, runBtn, ...
dat.skew.Value = 3; % set Fourier deskewing
dat.trimThresh = mean(dat.data(:,1));
```

Data collection

The following script sweeps the frequency across 50 values and for each frequency, measures amplitudes and phases of Vr, Vc, and the actual frequency produced by the function generator. I tweaked the frequency limits empirically to utilize the maximum range we can measure without <1 period or aliasing. Large frequency changes may require longer settling times, so the code starts with long pause, and also returns to the first frequency at the end of the loop, in anticipation of repeating this section. Don't expect this to run perfectly the first time. You'll likely have hardware bugs, parameters needing tweaking, and convincing yourself the system is stationary and working as intended will take several (dozens?) of runs. The automated run time is a great opportunity to think – what did I forget, what other tests can be made on the data integrity, where are my keys ... ;-). The loop generates two arrays, `freq`, and `amplitudes`, which are passed to `bodePlot` for visualization.

```
flush(ard); % make sure serial buffer is empty
%% measure Bode plot data for an RC series circuit
targetFreqs = linspace(1750,32,50)';
% targetFreqs = logspace(log10(32),log10(1750))'; % low and high frequency
amplitudes = zeros(length(targetFreqs), 2); % allocate storage for results
freqs = zeros(size(targetFreqs));
tic;
for i = 1:length(targetFreqs)
    freq2pwm(targetFreqs(i),ard);
    pause(.2);
    dat.runOnce = true; % runOnce
    oscope4;
    freqs(i) = dat.freq;
    amplitudes(i,:) = dat.ampl;
%     fprintf('.'); % simple progress indicator
end
```



```
toc
```

```
Elapsed time is 29.378058 seconds.
```

```
clear i
freq2pwm(targetFreqs(1),ard); % let it rest for next time
figure; % take a quick peek at raw data
plot(freqs,abs(amplitudes),'-o'); grid;
xlabel('Hz'); ylabel('amplitude/ADU')
legend('|A0|','|A1|')
```

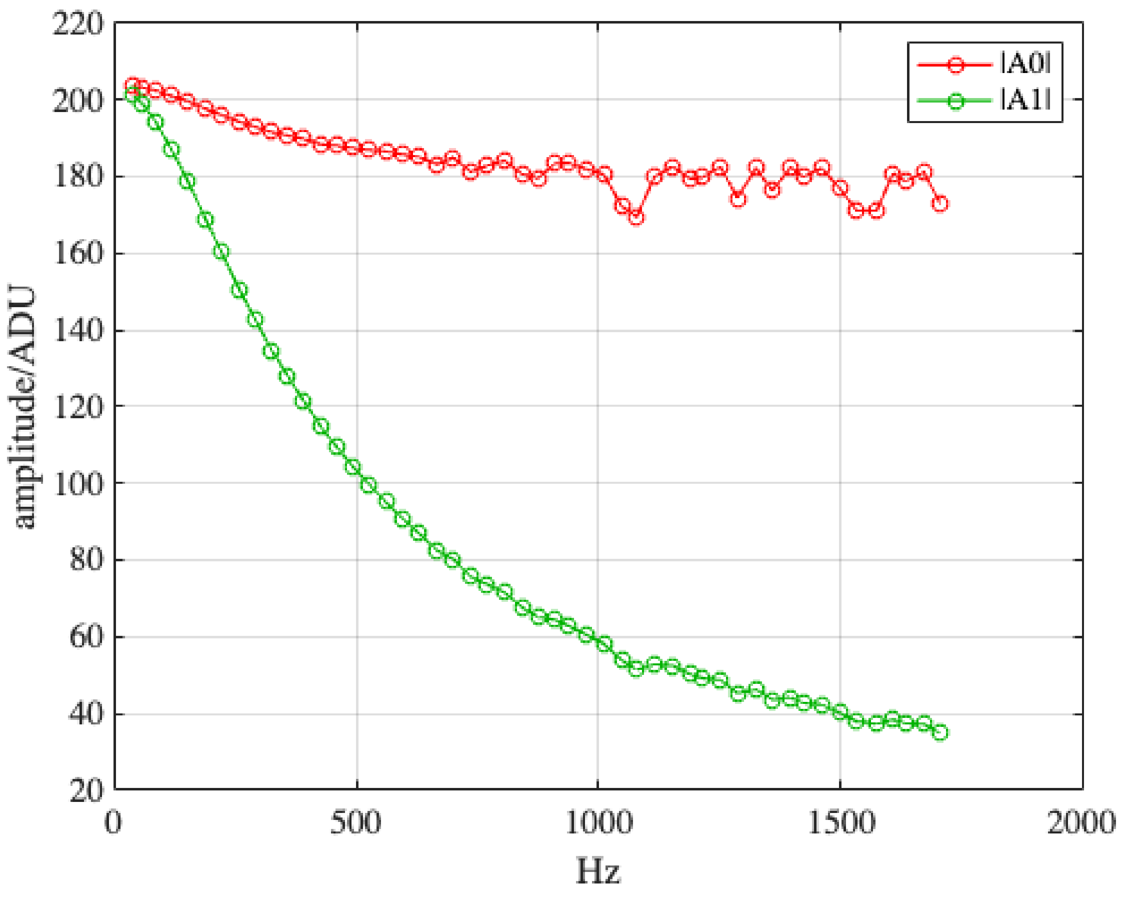


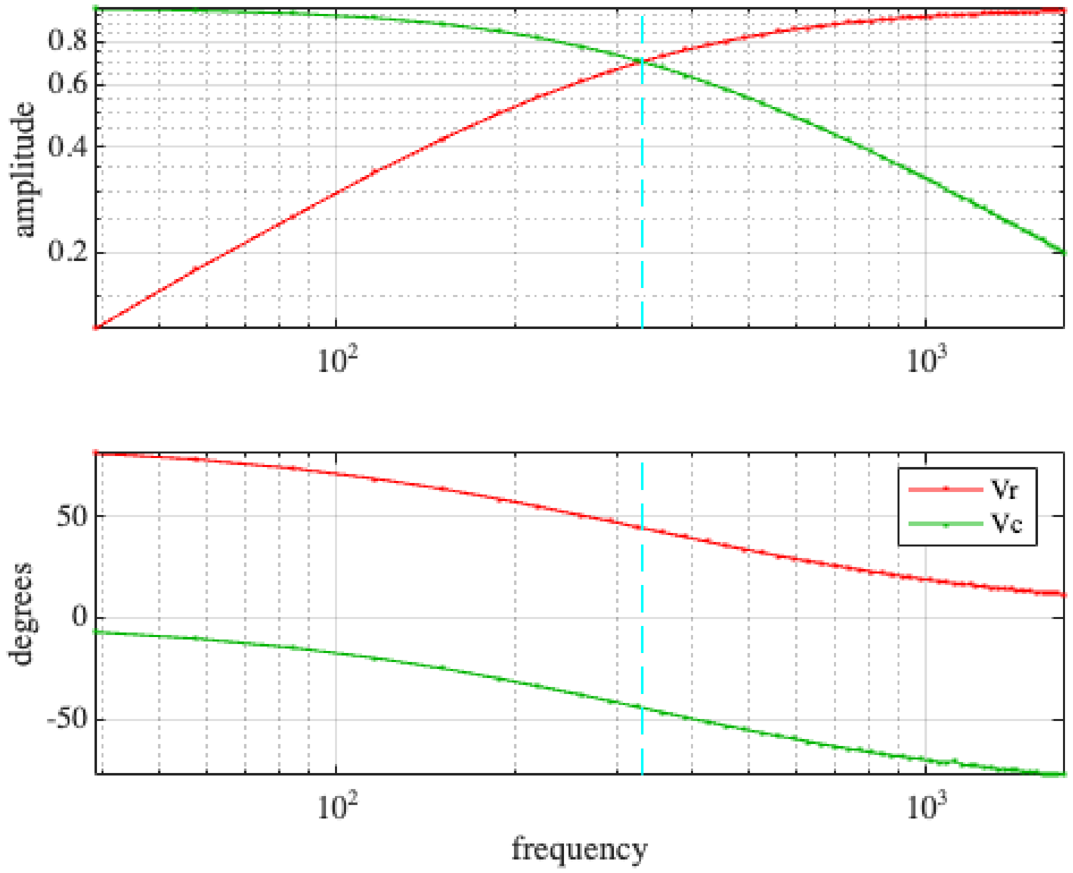
Figure 3. Plot of the "raw" Fourier amplitudes vs. 50 frequencies. I put "raw" in quotes, because we did a lot of "cooking" of ADU values to get this far. But "raw" in that we're about to put this data into the oven ...

Results

I hope you noticed as the experiment was running how the A1 (= capacitor voltage) was dropping with frequency relative to A0 (resistor + capacitor voltages). The drop in A0 perhaps wasn't so notable because `oscope` auto-scales the `plot`, but its a phenomena we will explain below. The point for now is that this RC circuit is clearly showing frequency-dependent behavior, which we'll now characterize in a "Bode plot". We can think of the R and C as voltage dividers, the voltage across R, `vr = Vin * R / (R+Zc)` and `vc = Vin * Zc / (R+Zc)`. Dividing both sides of both equations by `Vin` makes the result independent of the (arbitrary) `Vin`, and gives the ratio or "gain" (<1) of each impedance.

```
%% analysis (remember, the amplitudes are complex numbers)
vr = amplitudes(:,1)-amplitudes(:,2); % voltage across R1
vc = amplitudes(:,2); % voltage across C1
vr = vr ./amplitudes(:,1); % divide by Vo normalizes vr to a gain
vc = vc ./amplitudes(:,1); % divide by Vo normalizes vc to a gain
figure;
```

```
bodePlot(freqs,[vr vc],fc)
legend('Vr','Vc','location','best')
```



```
% print -djpeg bode1.jpg % save figure as jpg file
```

Figure 4. Bode plot of the R-C circuit. Corner frequency, f_c in cyan (computed from R1 and C1), should be where $|V_R| = |V_C|$and the phase shifts are $\pm 45°$, consistent with the measured frequency response.

Remarkable how much better the processed data looks. I hope you realize that programming computers (and microcontrollers) to measure data, turn knobs, run experiments, is way more pleasant and accurate than the old fashioned manual methods of data acquisition. We just made 50 *sets* of measurements – each consisting of 2 amplitudes and phases (each computed from 400 voltage measurements) – in <40 seconds, and plotted the results. And if the data isn't gorgeous, if you messed something, up, then debug it and run the whole experiment again with one click of the mouse.

```
fprintf('%.2f <= frequency <= %.2f\n',min(freqs), max(freqs))
```

```
38.99 <= frequency <= 1706.45
```

is perhaps a little different than expected from the calibration of `freq2pwm` last time. Not sure why it changed. Our current mirror circuit wasn't designed for long-term stability. Temperature, changes in Arduino's Vref, and likely other factors could contribute to significant drifts. But errors in `freq2pwm` between the target and resulting frequency don't affect our results because we measure the actual frequency from the data returned by `oscope4` and use our measured values, not the target ones.

Comparing to theory

Compute theoretical amplitudes and phases at the measured frequencies. `vtheory` consists of 2 columns of Vr and Vc at each measured frequency modeled with the Voltage Divider Equation:

```
zc1 = zc(C1, freqs);
vtheory = [R1 ./ (R1+zc1), zc1 ./ (R1+zc1)]; % voltage divider equation
figure;
bodePlot(freqs, [vr vc vtheory])
subplot(2,1,1); % more space on top subfigure for legend
legend('Vr_{meas}','Vc_{meas}','Vr_{theory}','Vc_{theory}', ...
    'location','best')
```

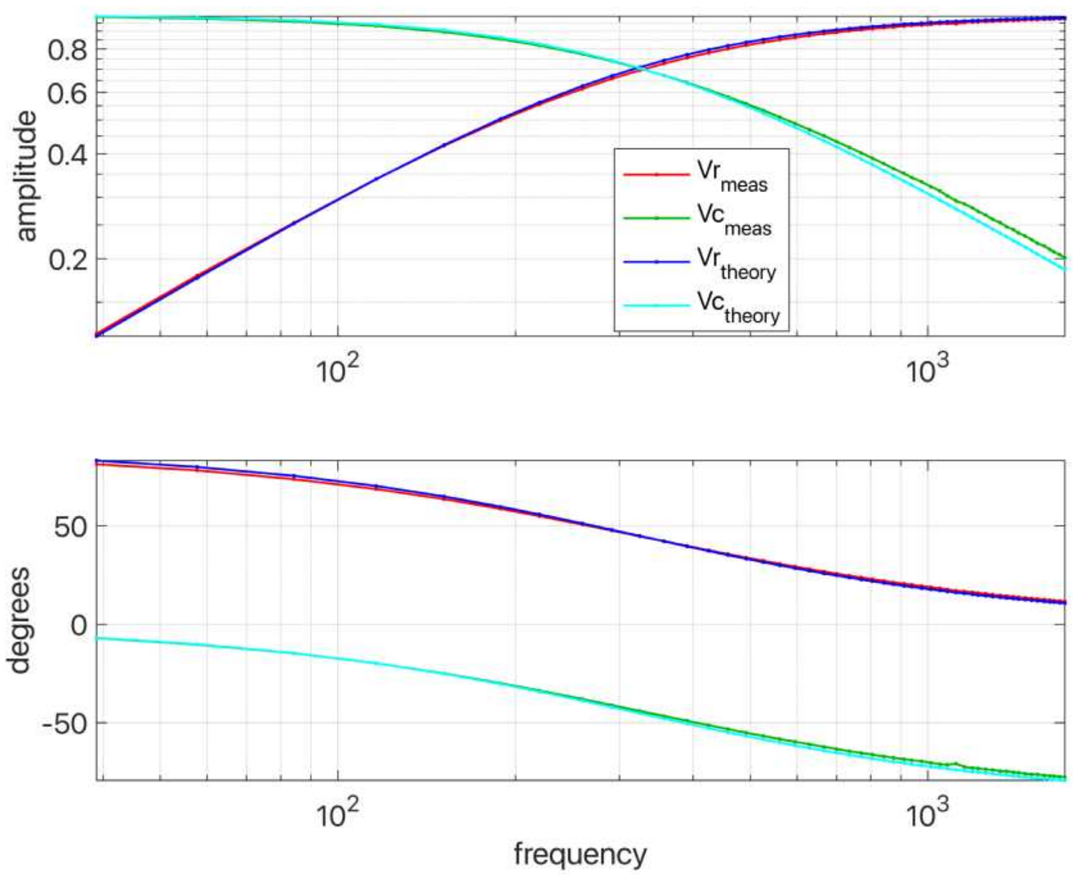


Figure 5. Theory and measured data are mostly sitting on top of each other.

The discrepancies are remarkably small and uniform – likely explainable by small errors in the values of C and/or R.

Difference plots

Quantifying errors with difference plots is generally more useful than qualitative statements like "relatively small".

```
err = abs([vr vc]) - abs(vtheory);  % subtract real magnitudes
relErr = err ./ abs(vtheory);
figure;
semilogx(freqs,relErr)
xlabel('Hz'); ylabel('relative error'); legend('|Vr|','|Vc|')
title('measured - theory'); grid
```

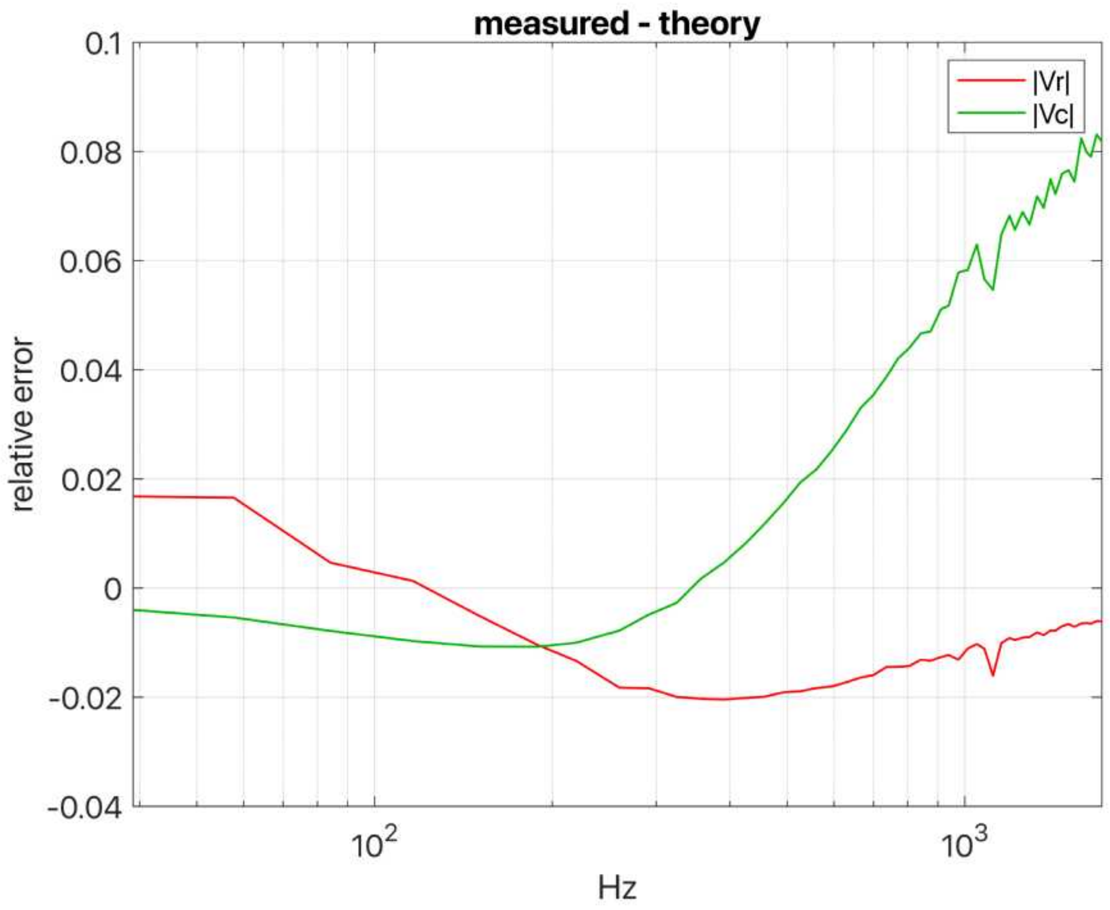


Figure 6. Difference plot shows all the errors are systematic.

The non-random nature of the errors suggests a systematic cause. Most likely is our value of C1 – $20 DMM's are notoriously imprecise in measuring small C's. Repeating the above section with various guesses for C1 ("dry-labbing") resulted in the following error plot with `zc1 = zc(103e-9, freqs)`:

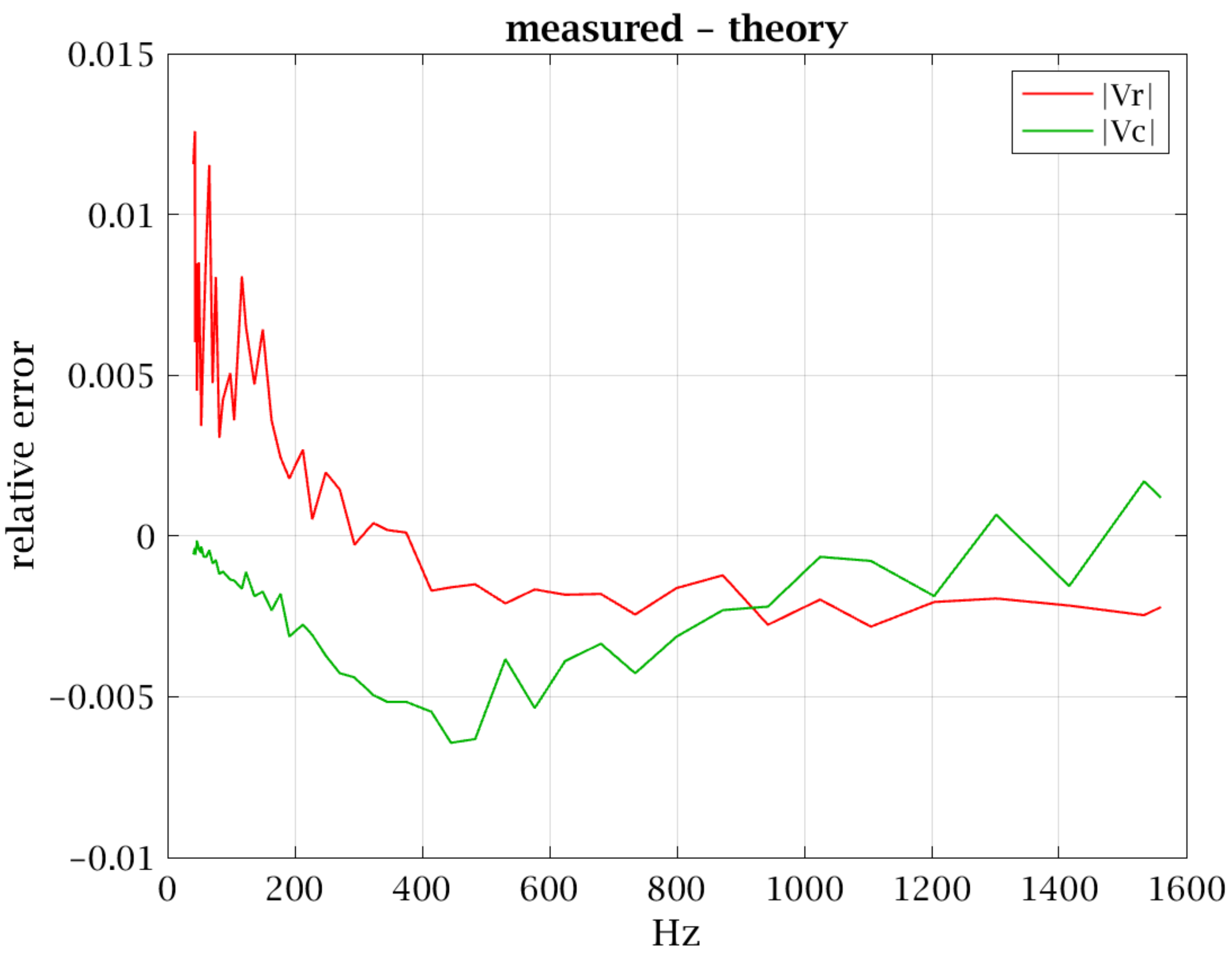


Figure 7. Relative amplitude error with C1 = 103nf is << 1%.

```
>> rms(relErr)

ans =

 0.0049 0.0029
```

Changing the value of C1 by a plausible 4% reduced the rms relative error from ~2.5% to <0.5%, a level of precision you might not expect from such inexpensive equipment.A1 = Vc has amplitudes ~30ADU at the highest frequencies (Fig. 3), so a 1ADU error is ~3%. Nevertheless, we measured errors an order of magnitude lower, i.e., ~0.1ADU. That's part of the power of frequency domain processing. A single time domain point has (at best) ~ 0.5ADU uncertainty, but each Fourier amplitude is computed from the entire 400 measurements, and thus can have lower relative error.

How low an rms error can you achieve with a little bit of manual tweaking? (Note: MATLAB has a powerful function to find optimal parameters, called `fminsearch`).

Discussion

Systematic errors with spline deskewing

An earlier trial showed large discrepancies at the highest frequencies > ~1.2kHz where measured Vc decreased more than expected (Fig. 8).

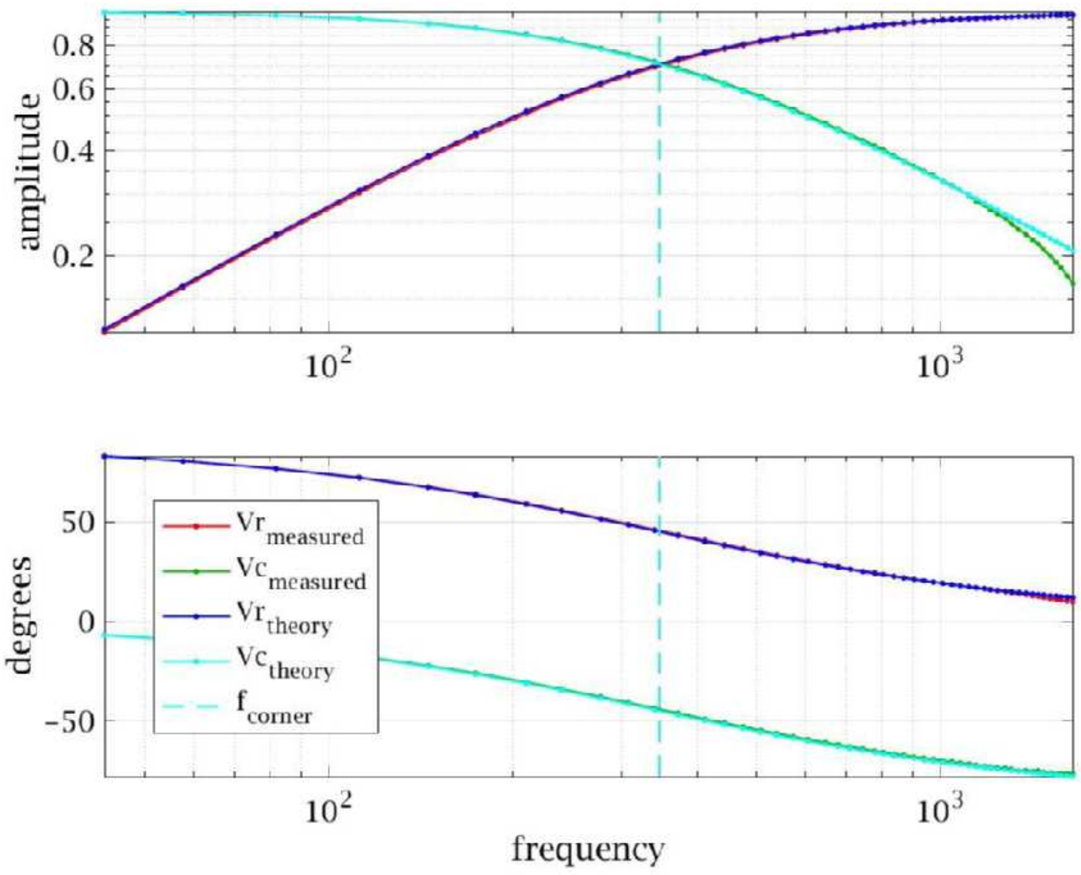


Figure 8. A prior experiment, with `arddeskew(dat.data, 'spline')` set in `oscope4`.

I immediately made a difference plot as follows:

```
err = [vr vc]-vtheory; % complex number subtraction
plot(freqs, abs(err))
legend('Vr meas-theory','Vc meas-theory','location','best');
xlabel('freq / Hz')
```

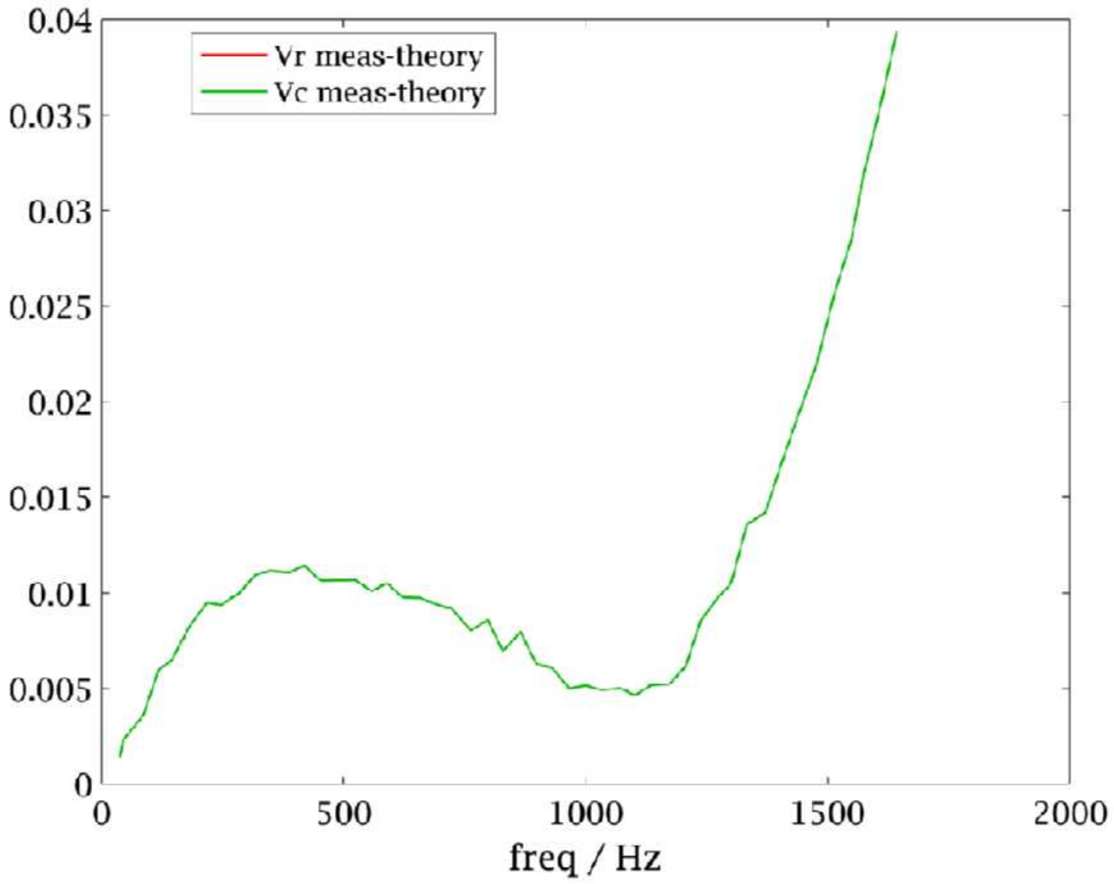


Figure 9. First attempt at difference plots didn't make sense.

Why is the error positive – Fig. 8 shows it should be negative? And why is it << 1ADU? And why do Vr and Vc seem *identical*? Which clues to pursue first? Ok, maybe they aren't identical – let's subtract them:

```
plot(freqs, abs(err(:,1))-abs(err(:,2)));
ylabel('Vr err - Vc err');
```

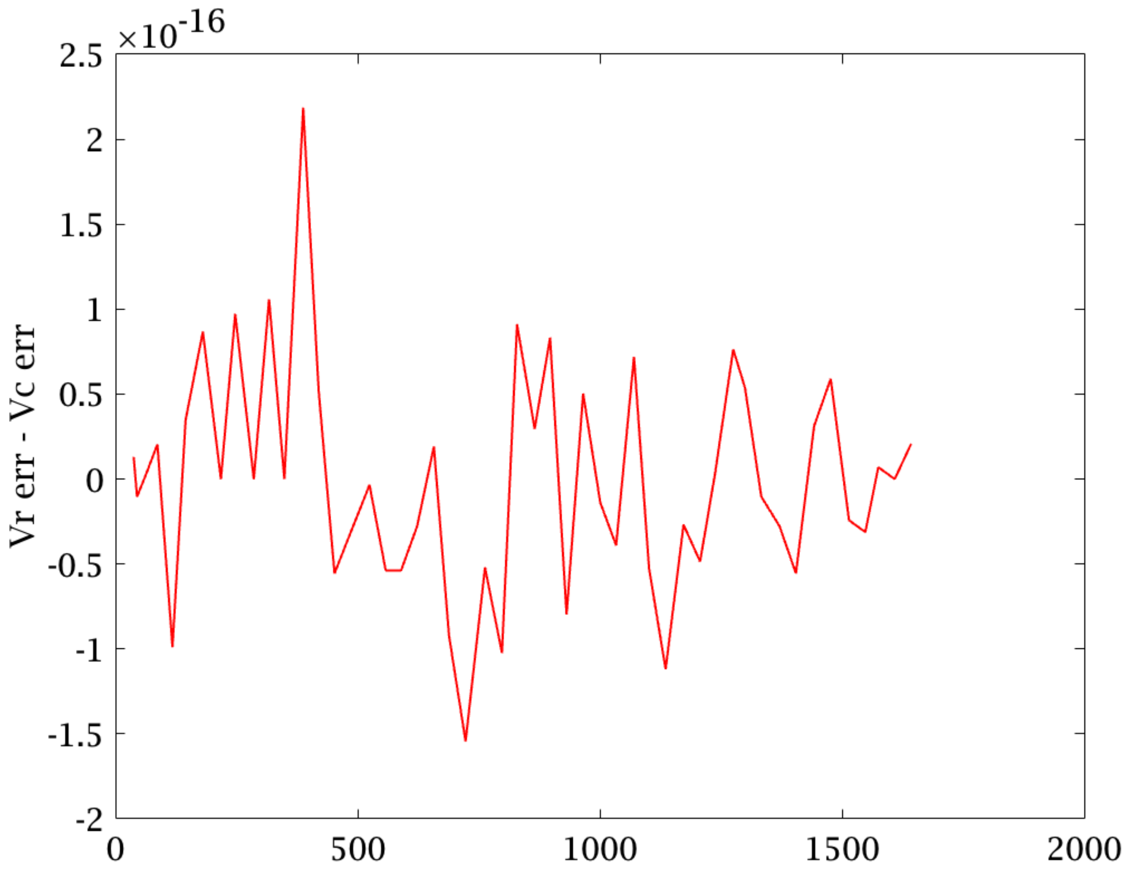


Figure 10. The difference between the 2 difference plots are zero, within round-off errors of floating point numbers.

Differing by 2e-16 is *really* identical – roundoff error of real numbers, not a measurement error. Took me a while to figure it out, and part of doing so was serendipity:

```
plot([vr vc vtheory]) % forgot to take absolute value -- these are all complex
axis equal
```

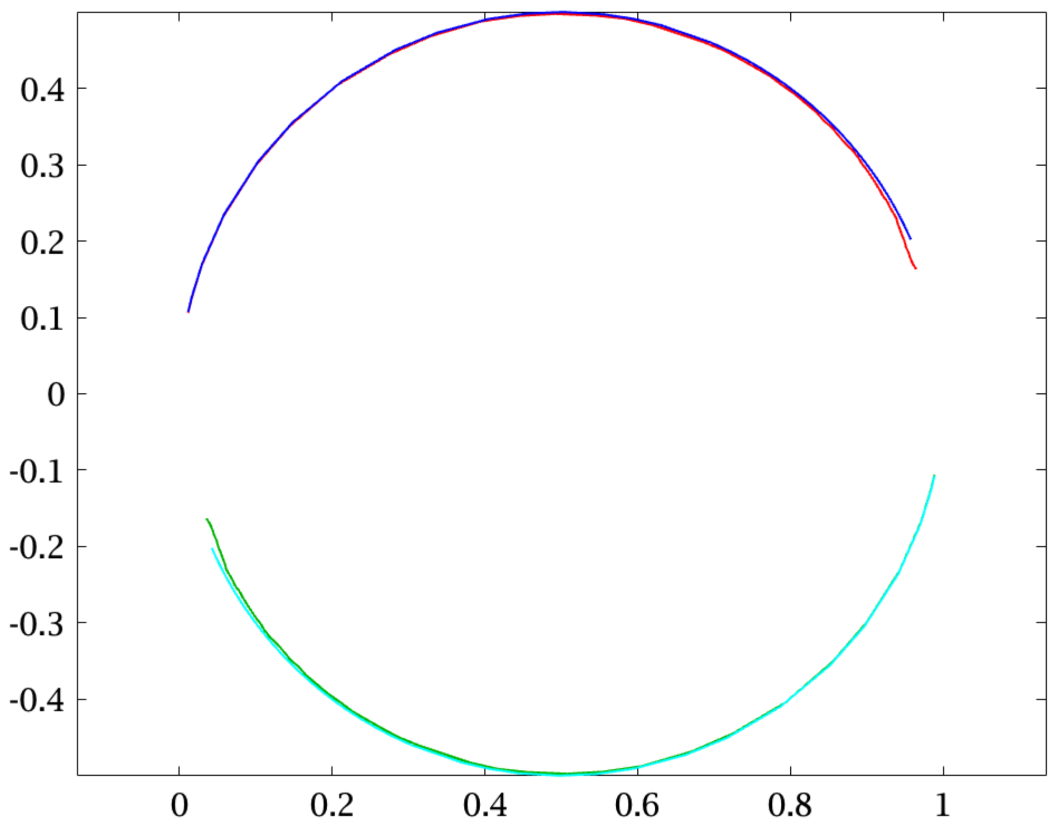


Figure 11. Accidently plotted complex numbers instead of their amplitudes.

Aha!!!! `vr` and `vc` are normalized phasors, and so are `vtheory`, so their absolute values all agree within roundoff errors. To compute the real errors:

```
err = abs([vr vc]) - abs(vtheory);  % subtract real magnitudes
relErr = err ./ abs(vtheory);
plot(freqs,relErr)
xlabel('Hz'); ylabel('relative error'); legend('Vr','Vc')
title('measured - theory'); grid
```

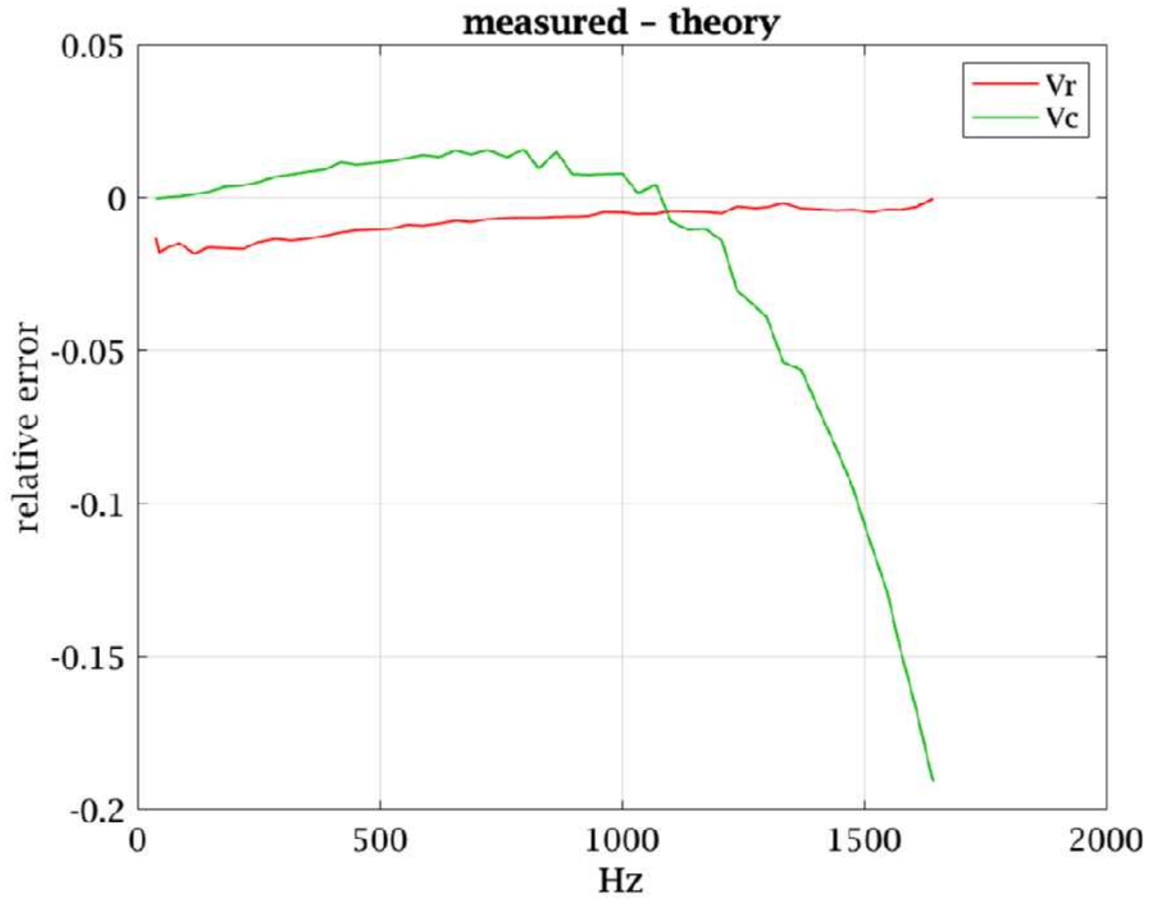


Figure 12. Difference plot done correctly. The -18% error on Vc suggests a serious problem in measurement and/or theory.

So measured and modeled voltage magnitudes on R1 and C1 agree within ~ ±2% below 1.2kHz, and then measured Vc drops much faster than model (Fig. 12's signs are consistent with Fig. 8). Here's what some of the highest frequency data looks like (zoomed in the time domain):

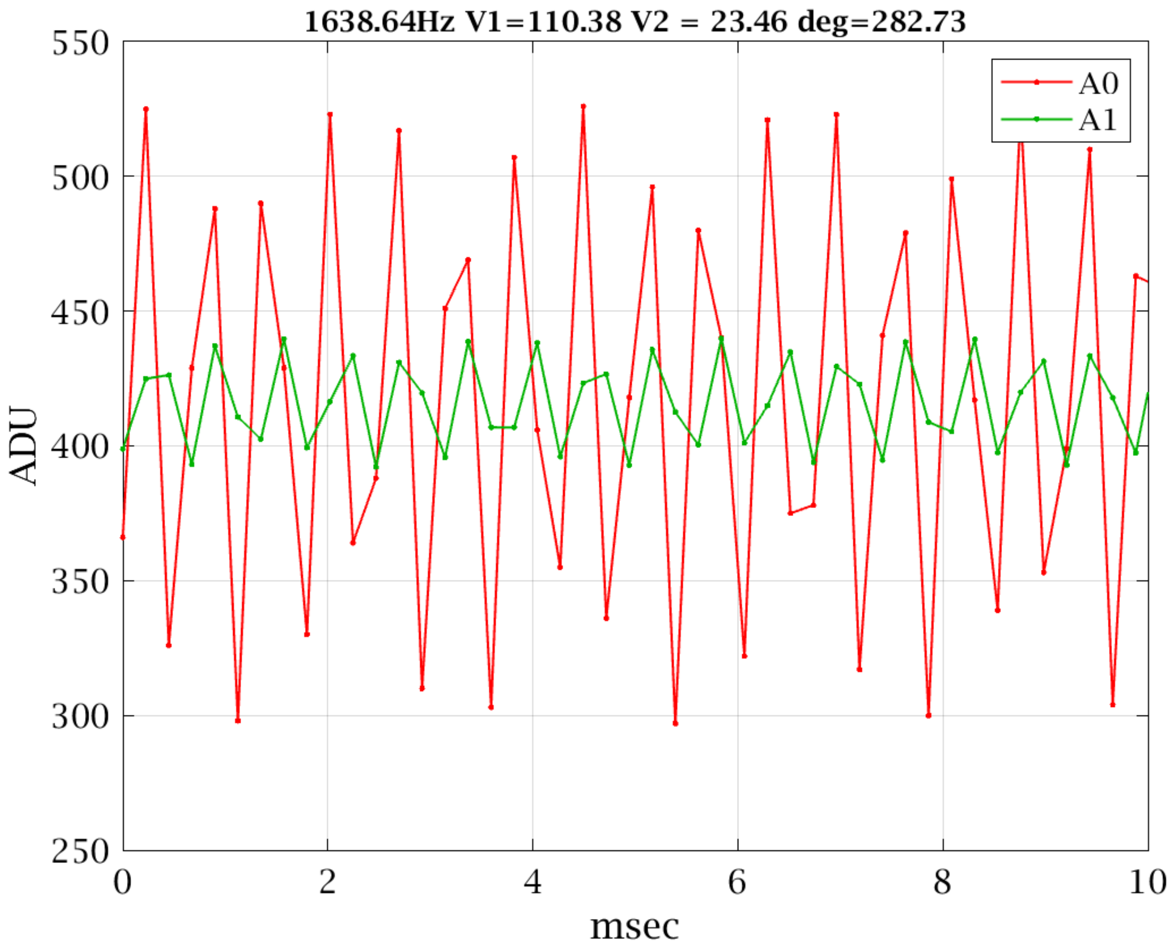


Figure 14. Zooming in on the data at the highest measured frequency reveals barely more than 2 samples per period are measured.

This is close to the Nyquist frequency where there are precisely 2 data points per period:

```
fprintf('Nyquist frequency = %.1f Hz\n', dat.sr/2);
```

```
Nyquist frequency = 2231.9 Hz
```

Imagine fitting a cubic polynomial (what `spline` does) to just a few consecutive points to interpolate data values between the points in Fig. 14 – it certainly can't be accurate in this frequency regime. Fourier deskewing on the other hand combines all the data, not just a small neighborhood. Changing to Fourier deskewing in `osocpe4` eliminated this error:

```
% dat.data = arddeskew(dat.data, 'spline');
dat.data = arddeskew(dat.data, 'fourier');
```

You're encouraged to explore further what the phase error look like. What other strange phenomena might be uncovered by exploring that rabbit hole?

A0 and A1 Amplitudes

The Bode plot explained the relative amplitudes and phases of Vr and Vc. Now lets return to Fig. 3 and understand why A0 lost more than half its amplitude over this frequency sweep (Fig. 3). When first testing the function generator, I recall the DMM showed virtually no change in *amplitude* (VAC) with frequency.

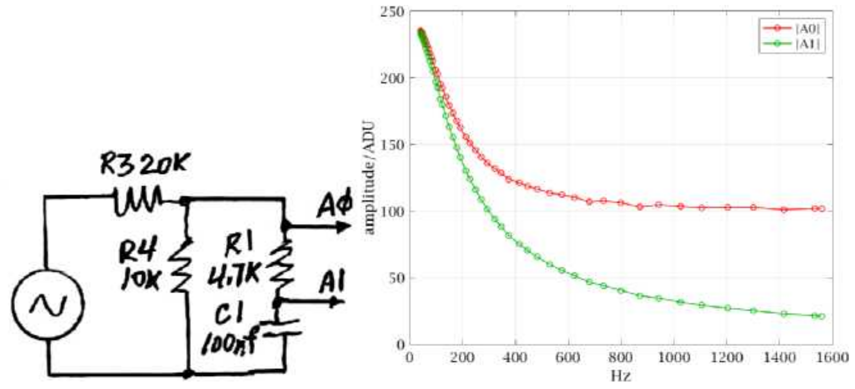


Figure 15. Simplified Fig. 1 showing the voltage divider and RC circuit, and Fig. 3, showing measured "raw" amplitudes from the frequency sweep.

I verified this again by connecting my DVM on VAC to the function generator output, and executing `freq2pwm(500,ard)` with several different frequencies (or turn the frequency pots, I almost forgot them). The DVM read 2.28-2.33VAC, with lower values ~30Hz and >1.2kHz – which could be due to the AC bandwidth of the cheap DVM. The small variability can't explain Figs. 3 and 15. However, the voltage divider is no longer simply R3 and R4, but R4 is "loaded" by the test circuit in parallel (Fig. 15). We can solve for A0 by computing the parallel impedance, R4 || (R1+ZC1). Then solve for A1 with a simple voltage divider dividing A0. Let's quantify that in code:

```
Vin = 1.937*1.03; % VRMS (VAC) measured with DVM around 60Hz
Vref = 4.745; % VDC measured with DVM
R3 = 1999; % ohms measured with dmm
R4 = 1000; % ohms
% R1, C1, zc1, freqs previously computed and measured
z4 = zp(R4, R1+zc1);
Vin = Vin * sqrt(2) * 1023 / Vref; % convert to ADU amplitude
vtheory(:,1) = Vin * z4 ./ (R3 + z4); % A0 from voltage divider eqn
vtheory(:,2) = vtheory(:,1) .* zc1 ./(R1 + zc1); % A1 from voltage divider eqn
clear Vin Vref R3 R4 z4 % clean up
figure;
plot(freqs, abs([amplitudes vtheory]),'.-');
xlabel('Hz'); ylabel('amplutide (ADU)'); grid
legend('A0m','A1m','A0t','A1t');
title('m=measured, t=theory');
```

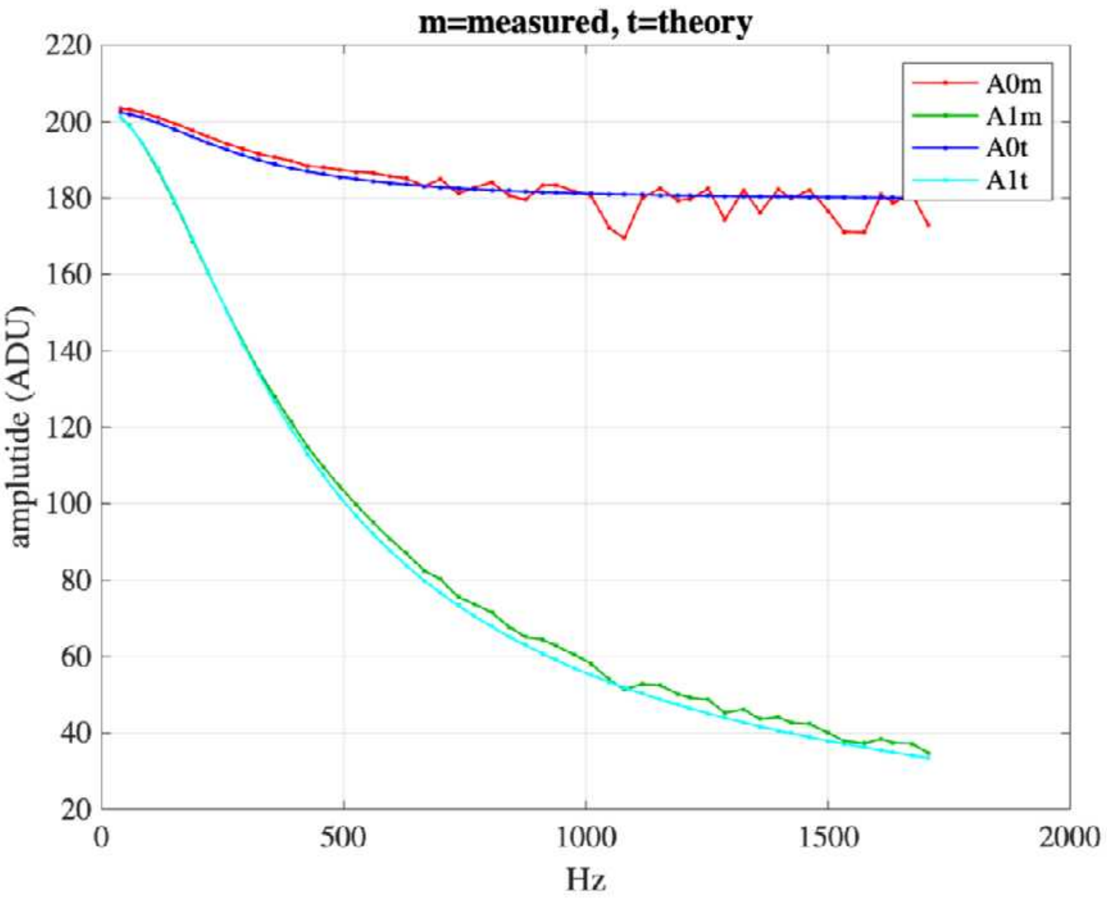


Figure 16. Measured and modeled amplitudes

Can't argue with that!

Reflections

1) Let me explain some things we *didn't* do here – some mistakes we could have made (and I did make in prior attempts). The first time I did a Bode plot experiment, I relied on a Human Robot, i.e., I turned the frequency knobs slowly by hand while MATLAB called `oscope3` and extracted data – similar to how we did our first (semi-) automated I-V plots. I renamed `freqs` and `amplitudes`, moved the frequency jumper, and repeated. Then I massaged the two data sets together. There was a lot of manual labor combining data from the two jumper positions, sorting to deal with overlapping frequencies, ... so repeating the experiment was a daunting proposition (i.e., ~15 minutes vs. 40 seconds). It's cost me many (many^n) hours to reduce the 15 minute experiments to 40 seconds, but the value is more than just saving 14 minutes each time. I've eliminated a huge space for potential errors – imagine trying to debug which of a dozen or more manual steps, renaming variables, joining, sorting, ... caused the nonsensical results of Fig. 17. That time and intellectual effort must be repeated each time ... The up-front work of automation can often be justified even if you don't intend to repeat measurements many times. It's so nice when your boss asks (Friday afternoon), "would you mind repeating that experiment that you did a year ago ..." and you can respond affirmatively – no problem because the protocol and documentation are executable.

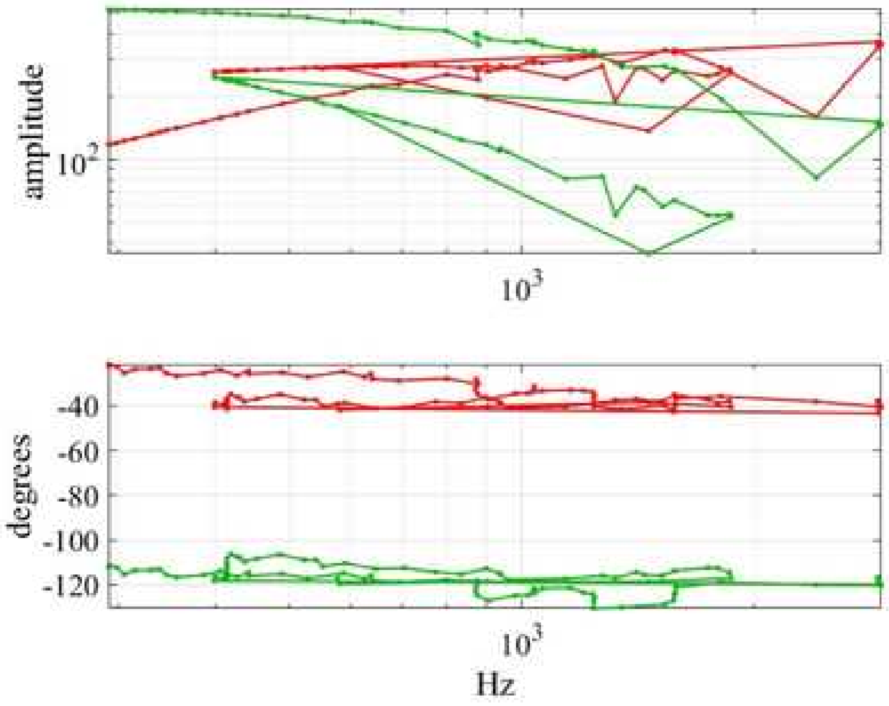


Figure 17. I shouldn't publish my trashcan, but this was artistic?! Nothing this complicated works the first time. Persevere!!

2) If we hadn't written the de-skewing code, then we would have seen phase shifts between A0 and A1 that would be more difficult to make sense of. The phase shifts shown above would have been confounded with shifts proportional to frequency, $\Delta\theta = 2\pi f \Delta t$ as described in Chapter 15. Preemptive de-skewing was helpful. Preemptively developing Fourier deskewing saved an iteration of troubleshooting here. In general, one maybe tempted to "go for broke" instead of taking systematic "baby-steps" towards your goal. You might get lucky with the former, but often you'll find yourself stuck with a mess that can be too complicated to debug, and the baby steps maybe necessary to uncover and correct systematic errors individually.

3) What if we swapped the location of R1 and C1 in the circuit? Look at the schematic, Fig. 1 or Fig 15. You might think "no problem, they're in series so it doesn't matter. Just swap variable names, `vr` and `vc` in the equations from above ...", copied just below (notice why Rasnow doesn't comment too much – I swapped "vr" with "vc", but forgot to change the comments. When the code doesn't work, imagine how much time can be wasted trying to make sense of the inconsistencies – how should you interpret the first line below that suggests "vc is the voltage across R1"?

```
vc = (amplitudes(:,1)-amplitudes(:,2)) ./ amplitudes(:,1); % voltage across R1
vr = amplitudes(:,2) ./ amplitudes(:,1); % voltage across C1
```

More importantly, **this won't work**, instead, you'd possibly burn out your Arduino!! Why??

Imagine C1 on top and the bottom of R1 is grounded, $V_{bottom} = 0$, as in Fig. 18. C1 alternately charges and discharges with the sine wave, i.e., current flows into and out of its plates. $V_{R1} = iR_1$ is thus alternatively positive and negative, so half the time the voltage on the top of R1 is negative, and that's lethal to the Arduino if it were connected directly there. R6=10k in series with A1 (Fig. 1) would likely save the Arduino from damage, but the data would be "clipped" and highly distorted at best. So why does Fig. 1 work?

Because our function generator has a DC offset ~5.5VDC, which is added to iR_1. But placing C1 between R1 and V_{SIN} blocks the DC offset (bias). We can replace the (blocked) DC offset with another bias network, just like we used for our microphone amplifier, shown in Fig. 18.

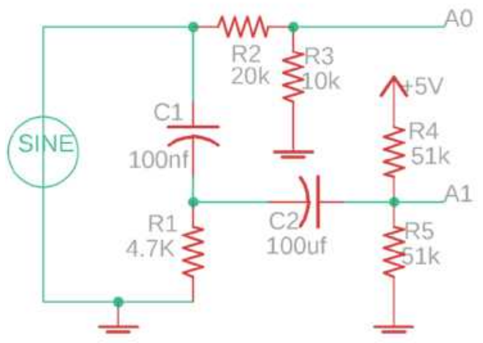


Figure 18. Reversing R1 and C1 require a bias network (C2, R4, R5) to measure Vr.

This works when $|iR_1| < 2.5$V (more precisely, Vref/2). C1 blocks the function generator's DC offset from R1, so the DC voltage at the top of R1 = 0V – R1's voltage oscillates above and *below* ground. R4 and R5 are voltage dividers setting up a 2.5VDC bias on the right side of C2, that adds to the AC signal on the left side of C2. C2 is large enough to pass frequencies above *its* $f_{C2} = 1/(2\pi * C2 * R4||R5)$ – you might ponder why are R4 and R5 in *parallel* in this equation, when they appear to be in series? Because from C2's perspective, current can flow through either, hence *parallel*.

The bias network in Fig. 18 also adds a finite parallel load to R1 which shifts the measured corner frequency to $f_{C1} = 1/(2\pi C1R1||(Z_{C2} + R4||R5))$. At frequencies of interest, Z_{C2} can be ignored and $R \approx 4.7k||51k/2 \approx 4.0k$, about 15% lower than R1, so the loading effect would be readily apparent. What a *huge* difference the order of 2 components *in series* can make!! It's also the (huge) difference between a low pass and high pass filter.

4) Here's my workspace at the end of this activity:

```
>> who
Your variables are:

C1          ans         err         relErr      vc          zc1
R1          ard         fc          runBtn      vr
amplitudes  dat         freqs       targetFreqs vtheory
```

That's a heck of a lot simpler than the start. `R1, C1, zc1,` are the parameters, `fc` and `vtheory` are theoretical variables. `freqs, amplitudes, vr, vc` are from the measurements (intermediate variables are contained inside `dat`), `targetFreqs` were the target frequencies we tried to set, and `ard` and `runBtn` are the Arduino and GUI interfaces. To make the desktop simpler required adding complexity under the hood, most of it inside `oscope4.m`, and in the data acquisition loop. Not simple, but simpler with 13 variables inside of `dat`.

5) Let's review all that we did here.

- We built a function generator capable of generating sine waves at a wide range of frequencies, tweaked the hardware to control it with MATLAB and Arduino, and excited an R-C series circuit at 50 different frequencies from ~30-1700Hz in ~44 seconds.
- The Arduino measured two V(t) waveforms as the frequency varied. MATLAB computed frequencies, amplitudes, and phases from the measured (and trimmed ...) V(t) and elapsed time (`dt`) data and corrected for systematic temporal skewing errors. We then computed voltage amplitudes and phases across R1 and C1 at each (measured) frequency with the `fft`.

- From DVM measurements of the values of R1, C1, we modeled the circuit using the voltage divider equation and complex impedance, zc1 at each frequency, then tweaked C1's value by a few percent. These two independent complex paths arrived at nearly identical voltages (amplitudes and phases) for V_{R1} and V_{C1}. That's extraordinary!!!
- Finally, `arddeskew` – deskewing in Fourier space instead of the time domain, called in `oscope4`, made high frequency errors >15% disappear. Placing the deskewing code in a function is another example of encapsulating complexity – `arddeskew` made `oscope4` shorter and simpler, while complexity increased elsewhere, in the function library folder. It also exemplified how automation enabled testing hypotheses, with a little copy/paste editing and another <1 minute of run time.

6) It's really powerful when there's more than one way to find an answer. If one is messier, try the other – model or measure. In this way, piece by piece you can construct more complex circuits with appropriate confidence. More complex circuits follow the same algebraic rules, using Ohm's Law, KVL, voltage divider equation, Thevenin's theorem, ... There are nearly infinite ways to mess up complex systems – through miswiring a circuit, using defective jumper wires and other components, or making logical mistakes in source code (e.g., Figs. 9-11). But getting two complex, independent approaches to precisely agree (e.g., Figs. 5 and 16) gives one great confidence that both are correct, and one can reasonably proceed to the next goals.

7) What else did <u>you</u> learn?

Appendix: data

```
t = table(freqs, vr*1000, vc*1000) % mV
```

```
t =
  50×3 table
    freqs        Vr*1000           Vc*1000
    ______    ______________    ______________
    1706.4    956.64+196.46i    43.358-196.46i
    1672.5    955.12+200.41i    44.877-200.41i
    1634.7    953.07+203.83i    46.928-203.83i
    1608.9    951.93+207.03i    48.072-207.03i
    1574.8    949.98+211.64i    50.021-211.64i
      1533    947.44+215.19i    52.557-215.19i
    1499.5    945.86+220.14i    54.138-220.14i
    1461.4    943.14+225.13i    56.859-225.13i
    1423.3    939.98+229.57i    60.023-229.57i
      1397    937.99+234.06i    62.015-234.06i
    1359.6    934.61+238.42i    65.388-238.42i
    1325.5    932.16+244.5i     67.842-244.5i
    1286.9    928.27+249.54i    71.729-249.54i
    1252.6    924.74+256.07i    75.257-256.07i
    1215.3    920.72+261.97i    79.281-261.97i
    1190.4    918.11+267.48i    81.892-267.48i
      1153    913.08+273.76i    86.918-273.76i
      1116     903.7+276.94i    96.297-276.94i
    1079.1    902.85+286.99i    97.149-286.99i
    1047.7    898.26+296.17i    101.74-296.17i
    1010.9    891.82+303.37i    108.18-303.37i
    975.36    883.57+311.5i     116.43-311.5i
     938.6     877.6+319.59i     122.4-319.59i
    910.96    871.33+326.83i    128.67-326.83i
```

```
874.53    862.56+335.7i     137.44-335.7i
842.03    854.24+345.36i    145.76-345.36i
 805.3    842.99+355.5i     157.01-355.5i
770.83    832.06+366.03i    167.94-366.03i
736.28    820.21+376.54i    179.79-376.54i
699.99    805.12+387.56i    194.88-387.56i
666.11    790.09+398.85i    209.91-398.85i
629.47    771.94+410.61i    228.06-410.61i
595.18     752.9+421.95i     247.1-421.95i
560.88    731.87+433.71i    268.13-433.71i
526.46    707.59+445.71i    292.41-445.71i
 492.1    680.99+456.87i    319.01-456.87i
458.14    650.89+466.93i    349.11-466.93i
425.13    618.38+475.77i    381.62-475.77i
390.72    580.16+483.21i    419.84-483.21i
357.11    538.23+488.14i    461.77-488.14i
324.87    494.31+488.57i    505.69-488.57i
291.36    443.18+485.74i    556.82-485.74i
259.79    390.43+475.59i    609.57-475.59i
219.53    318.62+453.99i    681.38-453.99i
188.61    259.63+426.18i    740.37-426.18i
151.33    187.81+377.97i    812.19-377.97i
116.11    123.15+315.57i    876.85-315.57i
84.217    71.275+243.26i    928.73-243.26i
57.725    36.672+175.05i    963.33-175.05i
38.986    18.539+120.43i    981.46-120.43i
```