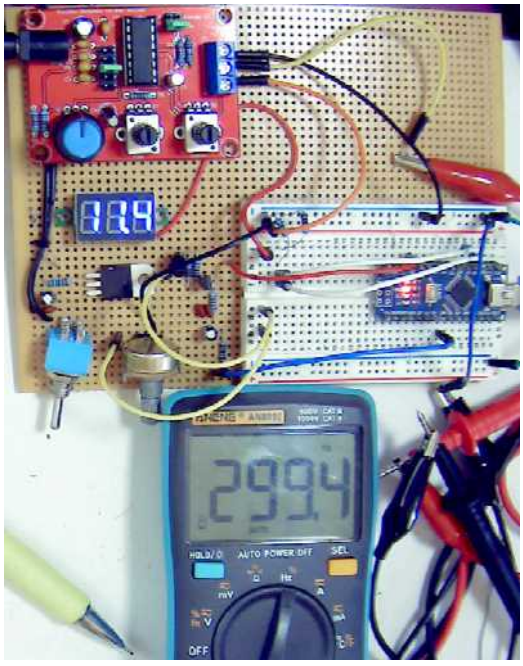
13. Function Generator

Table of Contents



Objective

Assemble and debug as necessary your function generator kit, using Arduino (running `oscope1.ino`) and MATLAB's' `oscope2.m` to display its output. We'll implement new modifications of `oscope2.m` to display a stable (triggered) waveform and measure and display voltage amplitudes, frequency *and* phase difference between the A0 and A1 inputs. We'll also address and correct the fact that A0 and A1 aren't simultaneously measured, saving the result as a new script called `oscope3.m`.

Function generator construction

The kit has some rudimentary directions. I suggest the following.

1. Make sure your soldering iron is clean and hot. Solder readily melts and sticks to the tip in a pool that can transfer heat to the parts and pads that you're soldering to.
2. Components sit on the labeled (silk screen) side of the board.
3. Install flat/short components first – start with resistors. Measure their value with your ohmmeter before soldering them.
4. Install the IC socket next – the notch at top indicates polarity. Solder two alternative corner pins, then make certain the socket is flush with the board before soldering all the pins.
5. Electrolytic capacitors are polarized, the negative side is silk screen hashed.
6. R2 is wired backwards on the board – resulting in the amplitude *decreasing* when turned clockwise. With some force and dexterity, its optional to rewire the R2 pins, as described below.
7. Instead of installing the power socket, I recommend soldering a 1M resistor through its horizontal holes
8. The value of R6 can be changed to increase the frequency range on each scale. If you're planning to use an Arduino, keep things as they are. But if you want greater frequency range, e.g., for use with a Raspberry pi Pico, then a lower value is handy. With the frequency jumper in the middle position and R6's default value of 5.1k, I get a frequency range of ~160-4500Hz. Reducing R6 to 2k increases the range to 160-11,300Hz, and 1k results in 160-22,600Hz.
9. Don't add the XR2206 IC until you complete initial testing with your DVM.

Fixing the backwards Amplitude potentiometer

The amplitude control variable resistor, R2, in the kit is wired backwards – that is if you expect amplitude to increase with clockwise rotation. This can be fixed by cutting and bridging 3 traces as follows.

1. On the back side of the board carefully cut the trace between two pins of R2 (make sure it's R2), Fig. 1A.
2. Cut the trace just above the middle pin of R2 (Fig 1B, label B), like you cut the trace in step 1.
3. Carefully scrape the red paint off the trace you just cut in Step 2 in front of the adjacent pin (Fig. 1B label A)
4. Bend the pot pin at label A so instead of going through the hole beneath it, it rests on the copper trace of step 3. This may take a few adjustments to get the angle right. Then trim the pin, solder the pot in place, and solder the flat pin onto the underlaying trace.
5. Solder a bridge across the other two pins (Fig. 1B label C).

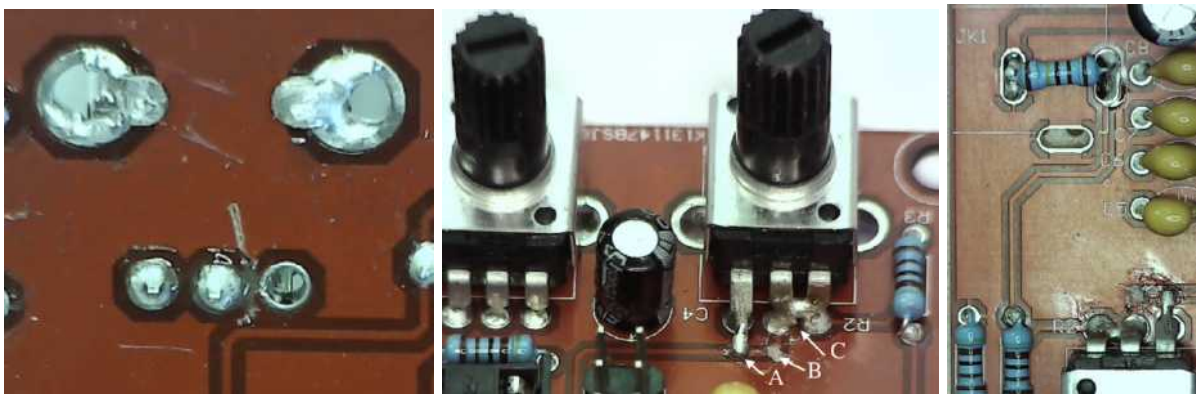


Figure 1. Modifications to the kit. A. Back side of R2 showing cut trace. B. Component side of R2 showing modifications to make the amplitude increase with clockwise rotation. C. 1MΩ resistor instead of power socket in JK1 position adds long power leads to solder to under the perforated board.

Once soldering is complete, visually scrutinize the board for any solder bridges or unsoldered pads. Before installing the XR2206 IC, verify with your voltmeter (use a clip lead to ground the black lead and probe with the red one):

1. IC pin 1 = 0V
2. IC pin 4 = 12V
3. IC pin 7 = 0V
4. IC pin 3 ~ 6V
5. SQU output = 12V

These are easy tests to make, and if any fail, they are easier to troubleshoot – just follow the traces. Later on, when the *!@$ thing doesn't work, troubleshooting functional failures is more challenging.

Final step is installing the IC. You likely will need to straighten the pins to fit into the socket. Pressing one side of the chip against the table can slightly bend all pins equally – flip and do the other side. If you bend too far, the long nose plier is the tool of choice. Position the IC over the socket carefully, respecting polarity, and when all pins are aligned, carefully press firmly. In the event you need to remove the IC, gently insert a flat screwdriver between the IC and socket, wiggle upward, and repeat on the other side. Pulling with fingers runs a risk of impaling your finger with the IC pins!

If your board failed any test, then troubleshooting is in order – magnified visual inspection and ohmmeter are the key tools. At this point, I would tentatively wire a couple corners of the board to the underlying perforated board so it doesn't flop around during further testing.

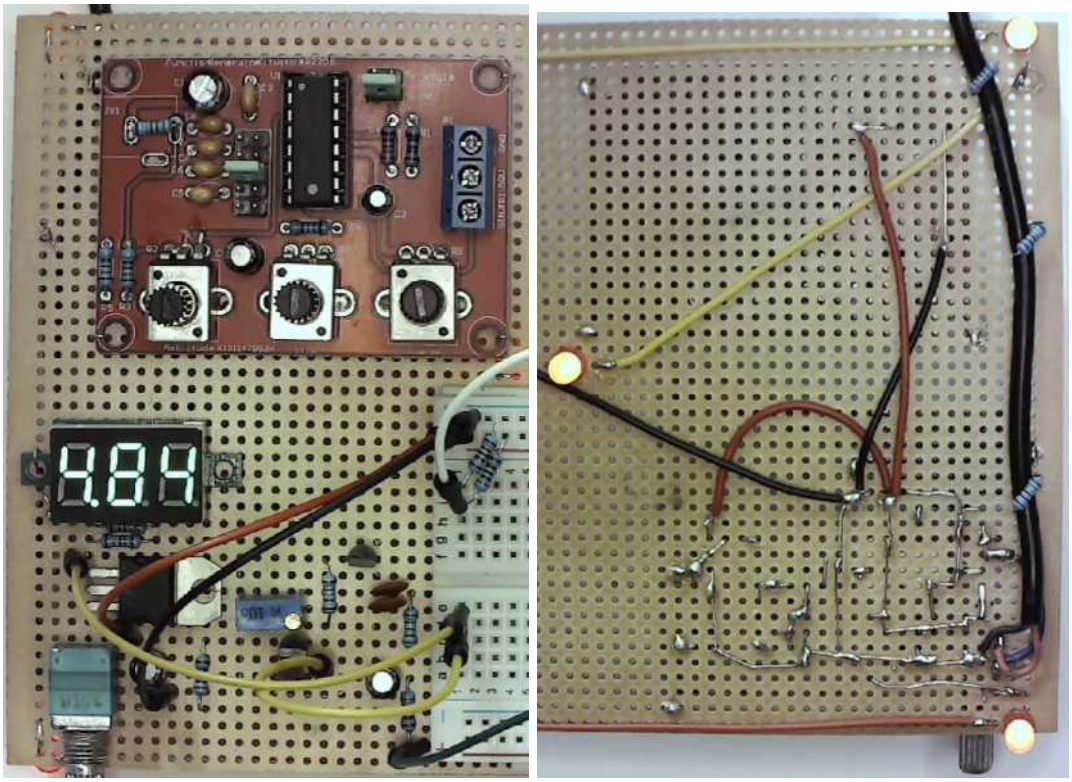


Figure 2. Front and back of perforated board. 12V power connections are crossing red and black vertical wires connecting to the 1M resistor leads.

Function generator functional testing

1. Test that the XR2206 function generator is likely working. **Set the 3 pots in the middle of their range, one jumper on sine, and the frequency jumper in the middle (3rd) position**. The DVM should read on the sine wave output (bottom; ground is top) ~6VDC, ~1.4VAC, the AC voltage should respond to the amplitude pot (note: the amplitude pot is wired backwards on my function generator, so the AC voltage increases by turning the pot unconventionally counter-clockwise). My DVM won't measure frequency on the Sine output.
2. Move the red voltmeter probe to the SQU square wave output (middle). I read ~6.8VDC, 5.5VAC, and hitting the select button again to measure Hz, I always get (a false reading of) zero. To measure frequency, switch the main dial to Hz and I see ~300Hz, and the value changes with the Fine and Course frequency pots. Move the frequency jumper up and down, and note the range of frequencies available for each jumper setting.

Arduino prerequisites

Verify that `oscope2` is working. `>> oscope2` should display reasonable traces with its `runBtn` on, and `volts2pwm(ard, 5)` should set your LM317 to around 5V. If that doesn't happen, debug ... If you're lost, work backwards (all the way toward blink from the IDE if necessary).

Arduino ground is connected to function generator and power supply ground via 2 jumpers to the blue rails (the USB powers the Arduino and could be at a different ground relative to your apparatus if they aren't connected).

Hardware Interface

1. We won't be using the LM317 for this activity, so move its output jumper from the top red row and place it in an unused column so it doesn't accidently touch another wire. Also safest to set the LM317 voltage to minimum.
2. AFTER UNPLUGGING THE LM317 from the top red row, plug into that row the function generator's SINE output. This could be between ~0-12V, so we'll need a voltage divider between it and the Arduino – reuse the pair of resistors already there (the 20k+10k voltage divider in Fig. 1 can be substituted with 2k+1k if one of those values isn't in your kit).
3. For extra Arduino protection, use a 10k series resistor to A0.
4. So A1 doesn't float, add a jumper between A0 and A1.

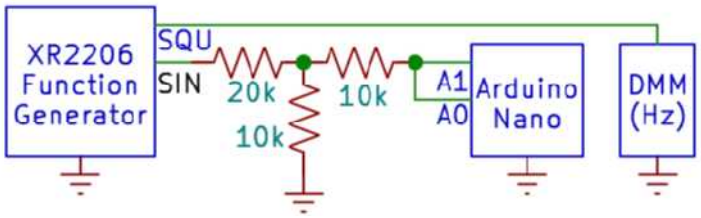


Figure 1. Make certain to disconnect the LM317 before connecting the function generator SIN output to the voltage divider and Arduino. Use clip leads to attach the digital multimeter (DMM) to the SQU output to measure frequency.

Arduino software

Arduino is running `oscope1.ino` (Chapter 8). Recall, this code responds to two commands: `p <pwmValue = 0:255>` `analogWrite`'s the value to pin D5, and the `b` command returns a buffer of 400 consecutive `analogReads` of A0 and A1 appended with the elapsed time. However, we hardly use these commands anymore, relying more on the "higher level" commands `oscope2` and `volts2pwm` which in turn send these 'low level" messages.

MATLAB software

Connecting to the Arduino

`>> oscope2` If it doesn't work reliably, go back and troubleshoot it – do you need to slow down the baud rate or modify how you select the serial port name, ... It may be easier to debug `oscope1.m` because the `try/catch` construct obfuscates (hides) where the error occurs. You should see something like Fig. 2. from the circuit in Fig. 1.

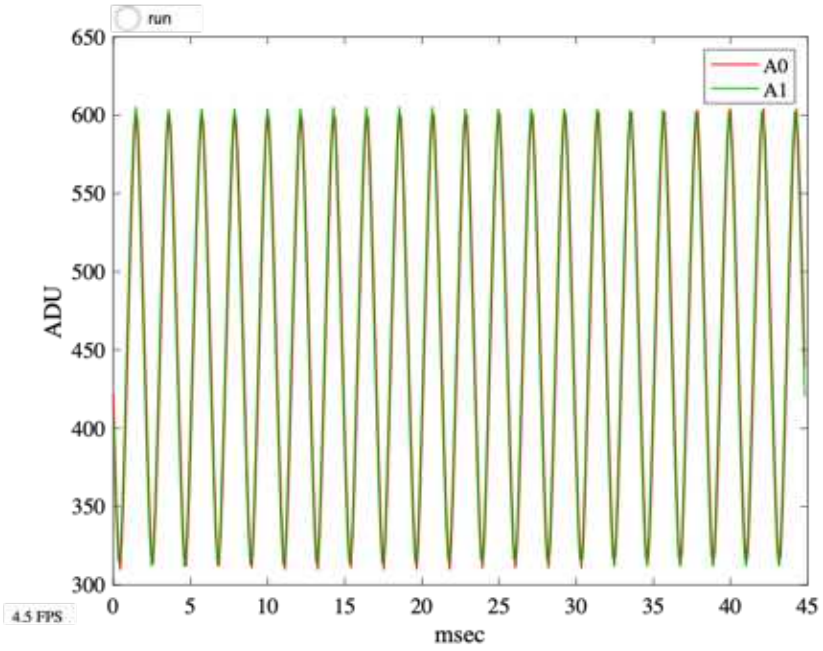


Figure 2. Run button (top left) and frames per second display (bottom left)

Note that MATLAB can be sluggish while the `runBtn` is on because it's doing a lot of work updating the figure several times per second. If you're not able to get a sine wave display like above, then go back and debug ... look at the error messages, e.g., maybe `displayfps` isn't found in your path. In the command window, query `>> ard` If the `port` is invalid then is it still connected to the IDE? If `NumBytesAvailable > 0` then `>> flush(ard)`

`>> volts2pwm(5,ard);` % does LM317 respond (approximately correctly) on the 3 digit display?

Troubleshooting is all about reductionism – divide and conquer – of all the code and hardware you've built, can you determine what parts works and what doesn't, and narrow the "doesn't" until you isolate the bug.

Triggering

One annoying feature of this oscilloscope display is that the initial phase jumps around randomly or drifts. This behavior is like a "free-running" oscilloscope. Oscilloscopes have "trigger" circuits [1] so repetitive waveforms are repeated at the same phase or initial voltage or slope. The following function trims the data to always begin with the same initial threshold crossing, and it does a bit more too, returning not just trimmed data but also the number of periods:

```
function [d, numperiods] = trim(data, thresh, npds, refchan)
% function [d, numperiods] = trim(data, thresh, npds, refchan);
% trim data to npds starting from the positive going thresh
```

```
% crossing (default: 0) on data(:,refchan).
% default refchan = data(:,end);
% npds < 1 --> return as many whole ones as possible (default)

if nargin < 4, refchan = length(data(1,:)); end
if nargin < 3, npds = -1; end
if nargin < 2, thresh = 0; end

a = data(:,refchan) > thresh; % boolean
b = [a(2:end); a(1)]; % shifted
p0x = find(b-a == 1); % positive crossing indices
if length(p0x) < 3, d = data; numperiods = 0; return; end
p0x(end) = []; % the last one is length(data), not a p0x.
if npds < 1
    d = data(p0x(1):p0x(end)-1,:);
else
    d = data(p0x(end-npds):p0x(end)-1,:); % the last one
end

if nargout == 2
    if npds == -1
        numperiods = length(p0x)-1;
    else
        numperiods = npds;
    end
end
```

This function demonstrates a software "design pattern" [2] I've found very useful:

1. Check arguments and set default values (the 3 `if` statements, `nargin` is an automatic variable containing the number of input arguments. So if `trim` is called with less than 4 arguments, we assign default values for the reference channel which is used to trigger, and in this case it is assigned to the last channel or A1.
2. The next blocks of code do the work of trimming ... here's how: `a` is a boolean vector = `true` when the reference channel is greater than threshold, and false otherwise. `b` is shifted one sample in time, i.e., the first element of `b` is the second element of a, and the last b is the first a, (to keep them the same size). `b-a` is the difference of two booleans, so each element is either -1, 0, or 1, the 1's are located where the threshold is crossed from negative to positive (you may need to draw some pictures to see why that is). The `find` function returns the indicies (locations) of the positive threshold crossings and puts them in `p0x`. The last part does the trimming into `d`.
3. The 3rd part of the design pattern is preparing the output. If the automatic variable `nargout` indicates 2 return values, then we return the number of periods, along with the trimmed data. The waveform's frequency can be readily computed from the number of periods and duration `dt`. What threshold should we use? Where the slope is steepest will minimize jitter, and that's around `mean(data)`~400 to 500ADU.

Save this function in your toolbox as `trim.m`, then modify `oscope2.m` to call it (and Save As `oscope3.m`). Here's a sneaky way to include `trim` by only changing a few lines of `oscope2`: rename `data` –> `rawdata`, and let `trim` return `data`. `plot` will throw an error if its two arguments are different lengths, so we must trim the time vector too:

```
rawdata = reshape(bin(1:800),2,400)';
[data, npds] = trim(rawdata, 340); % adjust 2nd arg = ADU with steepest slope
t = linspace(0,dt/1000,400)';
```

```
t = t(1:length(data));
```

We need to also change runBtn's callback, then `Save As ...  oscope3.m`

```
    runBtn.Callback = 'oscope3';
```

Click the run button, and the initial phase should now be stable just below the 500 ADU threshold, and the final phase is also stable – almost have to tweak the amplitude or frequency knobs to see a change. Now when I turn the amplitude too low (high – if you haven't rewired the amplitude pot), trim doesn't change the data – because there were no threshold crossings. Changing the threshold to 460, saving, and toggling off and on the run button, now lets me reduce the amplitude to a minimum, corresponding to ~440-480ADU – your numbers may vary a bit from those.

Measuring Frequency

The function generator's square (SQU) and sine (SIN) wave outputs always have the same frequency and phase, so measuring frequency from the larger voltage SQU output is more reliable with our DMM. Arduino *could* measure the square wave period with Arduino's built-in `pulseIn()` function, or the signal could trigger an interrupt and an interrupt service routine (ISR) [3] could read the time with `micros();` However, the the square wave would have to be reduced to <5V and the Arduino reprogrammed. I prefer leaving the Arduino programmed as is and compute the frequency from `data` in MATLAB. I'm much more productive programming in MATLAB's higher level language and interpreted environment than writing C, compiling in Arduino's IDE, uploading, running, debugging, repeating ...

We can compute the frequency from the sample rate, number of samples, and number of periods, as follows. How did I come up with these equations? How might you come up with them? Dimensional analysis is a great tool – the units of frequency are Hz = 1/time, so we want number of periods / their duration. The code below follows `trim`, i.e., `data` has already been trimmed to `npds` periods.

```
sr = length(rawdata)/dt*1e6; % sample rate (#/sec)
freq = sr/(length(data)/npds); % Hz, 1/sec
sprintf('%.1fHz',freq)
```

```
ans = '312.0Hz'
```

Question: Explain why the sample rate and frequency are given by the above formulas, using conversion factors. (Conversion factors are ratios who's numerators equal denominators, i.e., they are equal to 1, but they have units, e.g., (1e6 microseconds/1 second) when multiplied by seconds results in microseconds.

Connect your DVM to the Square wave output, set the main dial to Hz. How well do the two agree?

Question: Which one do you think is more accurate – the DMM or Arduino+MATLAB? Why??

Exercise: Show the frequency in the title of of the figure.

Measuring Amplitude in the Time Domain

Without proving it, the rms amplitude of our waveforms is simply computed by:

```
function y = mrms(x)
% returns the root-mean-squared average with dc removed of
% vector x or each column of matrix x.  Also see rms().

y = sqrt(mean(x .* x) - mean(x) .^ 2);
```

```
vdvm = 2.313; % vac on dvm
vref = 4.562; % vdc on dvm
vard = mrms(data) * vref / 1023 * 3; % multiply by voltage divider (2k+1k)/1k
[vard vdvm]
```

```
ans = 1x3
    2.3703    2.3717    2.3130
```

```
err = (vard - vdvm) / vdvm
```

```
err = 1x2
    0.0248    0.0254
```

2.5% difference between Arduino and DVM is a little big but doesn't raise warning flags that something is terribly wrong. E.g., is the voltage divider ratio exactly 3? A problem with mrms is that it destroys any phase information (by summing over time), and it also doesn't do a great job of rejecting noise.

Measuring Amplitude and Phase with the Fourier Transform

The (Fast) Fourier transform (`fft`) converts time domain waveforms into (complex) amplitudes (and phases) at each frequency. The first frequency is zero Hz or DC, equal to the mean of the signal, the next amplitude corresponds to 1 period, etc. With `npds` in the time domain, we expect the amplitude to be the `npds+1` Fourier component. This has to be normalized into ADUs as follows (it took some effort to find/test these formulas):

```
fd = fft(data); % data is trimmed to integral npds
ampl = abs(fd(npds+1,:)) / length(data) * 2
```

```
ampl = 1x2
  249.3689  249.5279
```

How does that compare with mrms (the above is in ADU units)?

```
err = (ampl - mrms(data)*sqrt(2))./ampl
```

```
err = 1x2
   -0.0048   -0.0048
```

The Fourier amplitudes are .5% lower (and probably more accurate because noise at all other frequencies contribute to the `mrms` values).

Phase is given in radians with `angle`, let's convert it to degrees:

```
theta = rad2deg(angle(fd(npds+1,:))), diff(theta)
```

```
theta = 1x2
 -110.9074  -99.4641

ans = 11.4432
```

The phase of each waveform is somewhat arbitrary – related to the threshold we picked and the amplitude, but the difference in phase between the channels is significant when we're measuring waveforms within a circuit. Here we see a significant phase difference between A0 and A1 (which are the *identical* signal) – and I hope you're saying, "Ahhh! That's because .....

Before moving on, let's add the amplitudes and phases to the title:

```
sprintf('%.2fHz V1=%.2f V2 = %.2f deg=%.2f',freq, ampl, diff(theta))
```

```
ans = '284.17Hz V1=249.37 V2 = 249.53 deg=11.44'
```

Adding all this to the title requires shrinking the font:

```
title(sprintf('%.2fHz V1=%.2f V2 = %.2f deg=%.2f', ...
            freq, ampl, diff(theta)),'FontSize',12);
```

I put all these changes into `oscope2`, Saved As ... `oscope3.m`, and ... it didn't work ... until I edited `runBtn.Callback = 'oscope3';` It's so easy to forget key details ...

Temporal skewing and deskewing

So hopefully you've noticed that the phase angle (`deg` in the title) increases with frequency, and it *shouldn't* because V1 and V2 are the same waveforms. What's going on? Let's add a marker to the data:

```
>> dots('o')
```

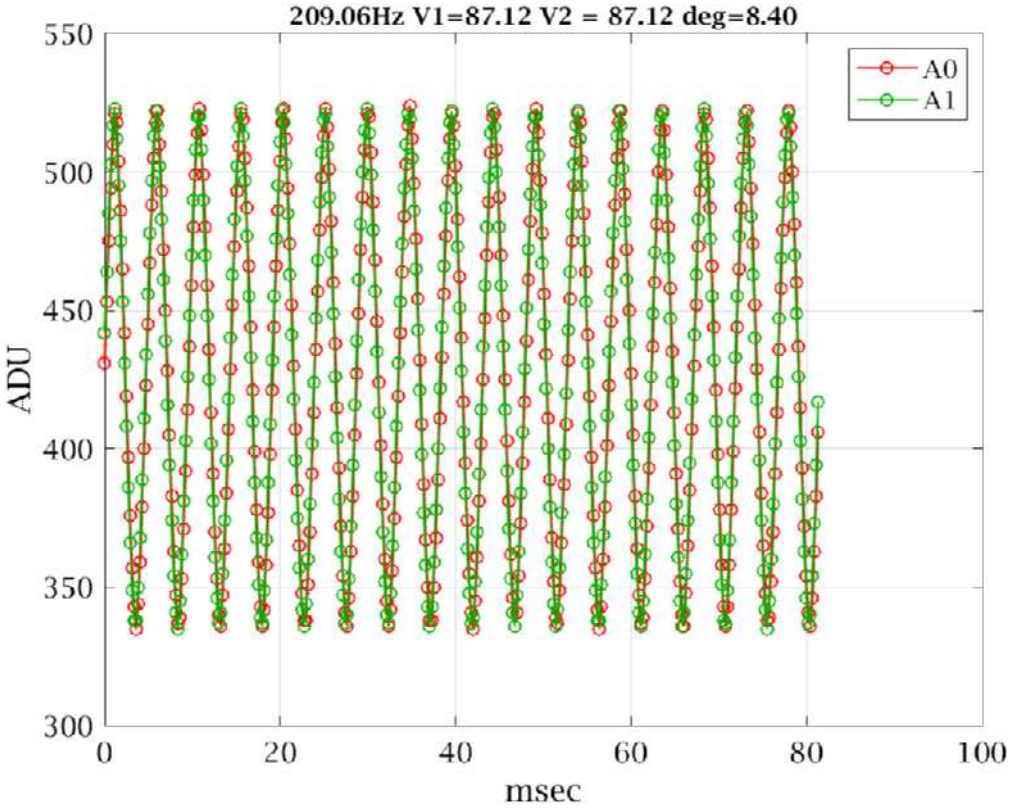


Figure 3. A0 and A1 datapoints are clearly interleaved, or said another way, A1 is *skewed* or shifted in time relative to A0.

A0 and A1 are measuring the same voltage, but at different times or phases. The Arduino software clearly reflects this: A0 is read, then A1, then A0, ... the data are interleaved in time, resulting in a temporal shift (or "skew") [4] between A0 and A1. How serious this is depends on the waveform frequency. The RMS amplitudes are the same to within 4 significant figures, but what's the rms of the difference?

```
mrms([data data(:,2)-data(:,1)])
```

```
ans = 1x3
  177.1744  177.2813   35.6689
```

Just noticed my function generator amplitude was low so these numbers are small. But we see the rms amplitude of A0 and A1 only differe below their 4th significant figure, but their difference (column 3) is HUGELY big:

```
err = [ans(1)-ans(2) ans(3)] /ans(1)
```

```
err = 1x2
   -0.0006    0.2013
```

Less than 0.1% difference between rms amplitudes of A0 and A1, but the difference in the waveforms is a huge 15% of the signal, instead of zero!

So what to do? First, recognize this is the beauty of systematic errors. They are systematic, i.e., predictable, vs. noise which is by definition unpredictable or random. When you have a systematic error, like temporal skew, you can try to build a new "system" that undoes the error. Let's try to model the error this way. Imagine a sample clock that ticks each time `analogRead` is called: 1 = `analogRead(A0)`, 2 = `analogRead(A1)`, 3 = `analogRead(A0)`, etc. This clock ticks:

```
tt = (1:numel(data))';
```

All the A0's are measured on odd t's and all the A1's are measured on even t's. Translating that into code, `data(:,2)` was measured at times `tt(2:2:end)`, and `data(:,1)` was measured at times `tt(1:2:end);`. If we were to "remeasure" `data(:,2)` at `tt(1:2:end),` that would deskew (unskew) the data ... and MATLAB has a function that does that:

```
>> 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.
```

```
deskewedA1 = spline(tt(2:2:end), data(:,2), tt(1:2:end));
plot(tt(1:2:end), data, 'o-', tt(1:2:end), deskewedA1, '.b')
xlabel('Sample #'); ylabel('ADU'); legend('A0','A1','deskewedA1')
```

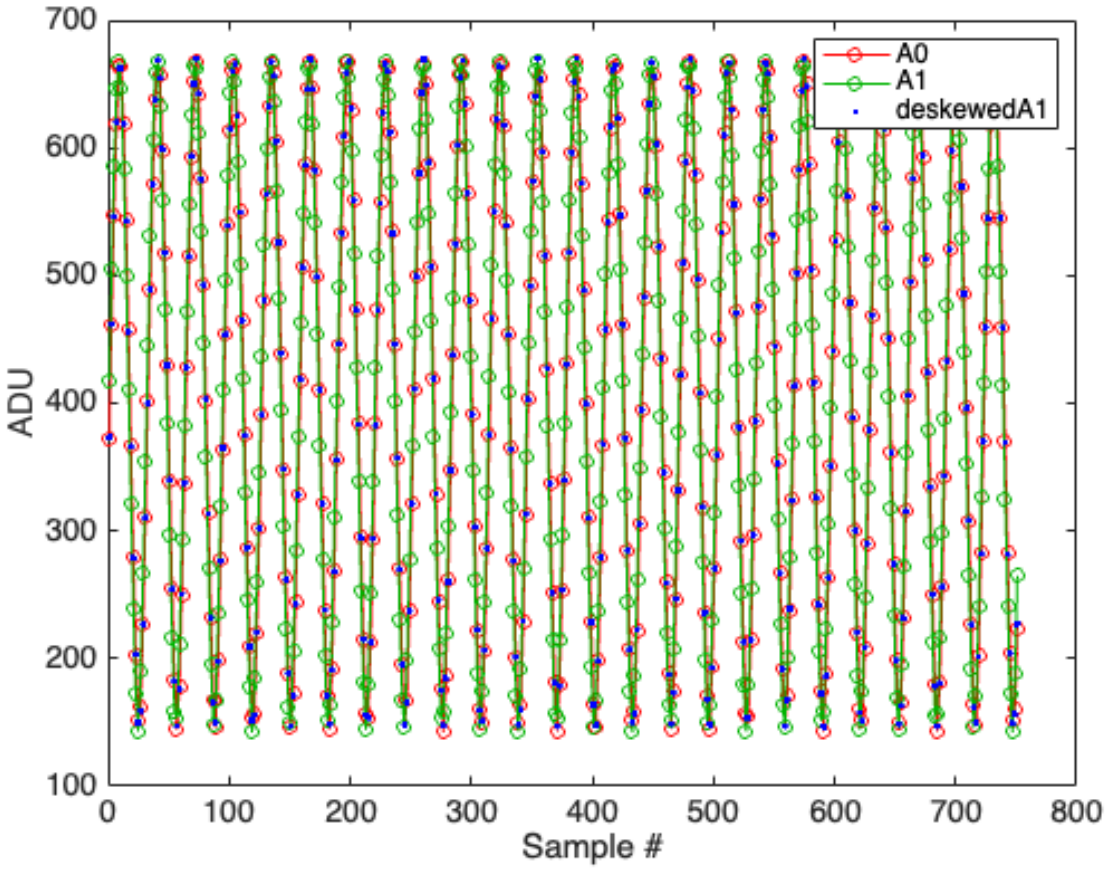


Figure 4. Deskewing A1 (blue dots) makes A1 (green circles) *appear* concurrent with A0 (red circles).

The spline moved the green data circles to the blue dots almost always located near the center of the red dots ... removing most of the systematic error. Let's quantify how much error is left, and we can use the same error calcs we did above by overwriting `data(:,2)` with its deskewed version:

```
data(:,2) = deskewedA1;
mrms([data data(:,2)-data(:,1)])
```

```
ans = 1x3
   61.6999   61.6936    0.5492
```

```
err = [ans(1)-ans(2) ans(3)] /ans(1)
```

```
err = 1x2
    0.0001    0.0089
```

That reduced the rms difference to ~1/2 ADU – can't expect to do much better (at least in the time domain, we'll see later how deskewing in the frequency domain works better at higher frequencies).

Exercise: Add the deskewing code into oscope3.

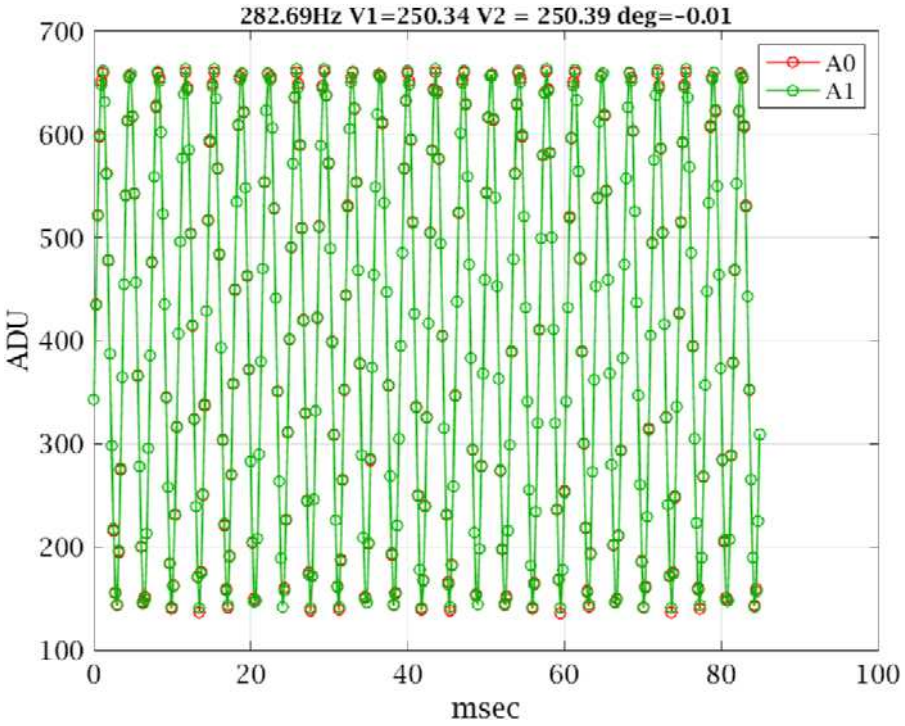


Figure 5. Display of deskewed `oscope3` after running `>> dots('o')`.

The green A1 datapoints now almost completely occlude (sit on top of, are identical to) the red A0 data – we did it!! We undid a systematic error using digital signal processing (in particular, `spline`) to remove the temporal skewing.

BTW, I've noticed that sometimes, e.g., after I wake up the computer from sleep, the graph doesn't update, and I need to cycle the run button a couple of times. MATLAB has a graphics bug when awaking with a 2nd monitor connected to a Macbook. Most of our tools have imperfections, and we make engineering tradeoffs between adapting to them, fixing them, cursing them, trying something else ...

Results

Here's my final version of `oscope3`. I hope you don't blindly copy/paste it, but take time to understand/appreciate what each line of code does. A difficulty with learning software, and electronics, is that code and schematics don't reveal their history. How they were built piece by piece is lost. When you see the whole thing, it can be overwhelming. But if you could unwrap how it was built, little by little, tweak by tweak, it would be much easier to understand. And if you think this is too hard, consider the challenge biologists face. Evolution doesn't leave any comments or documentation explicitly for scientists – although smart experts have learned how to decipher bits of history in the genome and fossil records.

```
%% oscope3.m -- added trim, title with freq, ampls, phase diff, and spline deskew
% 16oct22 BR, Arduino running oscope1, command set = b, p
%% initialization
if ~exist('runBtn','var') || ~isvalid(runBtn)  % add the button once
    runBtn = uicontrol('style','radiobutton','string','run','units','normalized',...
        'position',[.13 .93 .1 .04], 'Callback','oscope3');
end
```

```
if ~exist('ard','var') % initalize arduino
    disp('initializing arduino ...')
    ports = serialportlist; % on my computer, arduino is always the last one
    ard = serialport(ports{end}, 57600, 'Timeout', 2);
    clear ports;
    readline(ard); % wait for Arduino to reboot
end
if ~exist('runOnce','var'), runOnce = true; end
%% loop
while runBtn.Value || runOnce
    writeline(ard,'b');
    try
        bin = read(ard,802,'uint16');
        dt = bitshift(bin(802),16)+bin(801); % microseconds
        rawdata = reshape(bin(1:800),2,400)';
        [data, npds] = trim(rawdata, 450); % adjust 2nd arg = ADU with steepest slope
        t = linspace(0,dt/1000,400)'; % msec
        sr = length(rawdata)/dt*1e6; % sample rate (#/sec)
        freq = sr/(length(data)/npds); % Hz, 1/sec
        tt = (1:length(data))'; % sample clock
        data(:,2) = spline(tt+.5, data(:,2), tt); % deskew
        fd = fft(data);
        ampl = abs(fd(npds+1,:)) / length(data) * 2;
        theta = rad2deg(angle(fd(npds+1,:)));
        %% display
        plot(t(1:length(data)), data, '.-');
        xlabel('msec'); ylabel('ADU'); grid on;
        title(sprintf('%.2fHz V1=%.2f V2 = %.2f deg=%.2f', ...
            freq, ampl, diff(theta)),'FontSize',10);
    catch
        flush(ard);
        fprintf('*');
    end
    displayfps;
    runOnce = false;
end
```

Exercise: Oscope3 above is a bit long (for Rasnow's aesthetics). How might you shorten it? What are the pros and cons?

Verify that turning the frequency and amplitude pots results in expected changes in the data and values in the title. The frequencies agree with my DMM meter to within < 1Hz.

Let's quantitatively compare

For example, the above data read 277.2Hz on the DMM. The DVM can also read the RMS amplitude and Vref, enabling independent comparisons:

```
Vac = 2.073; % volts on DMM VAC
Vdc = 4.669; % volts on DMM VDC
Vref = 4.750; % VDC on DMM
V1 = abs(ampl(1)) * Vref / 1023 / sqrt(2)
```

```
V1 = 0.6944
```

```
Vac/V1
```

```
ans = 2.9853
```

The last calculation shows the DMM voltage is 2.98 times bigger ... precisely the ratio of the 2k+1k voltage divider in front of the Arduino, which I forgot about. (Note: I also forgot to remove the 100nF capacitor between A0 and ground from Chapter 12 – that led to another factor ~2 attenuation at 277Hz!) Multiplying V1 by 3, the two measurements agree by:

```
V1 = 3 * abs(ampl(1)) * Vref / 1023 / sqrt(2), % 3 from voltage divider, sqrt(2) from amplitude -> rm
```

```
V1 = 2.0832
```

```
percentErr = (V1-Vac)/Vac * 100
```

```
percentErr = 0.4939
```

Looking at extremes of frequency, around and above 800Hz, aliasing becomes evident and deviations between the 2 channels start to show patterns, e.g.,

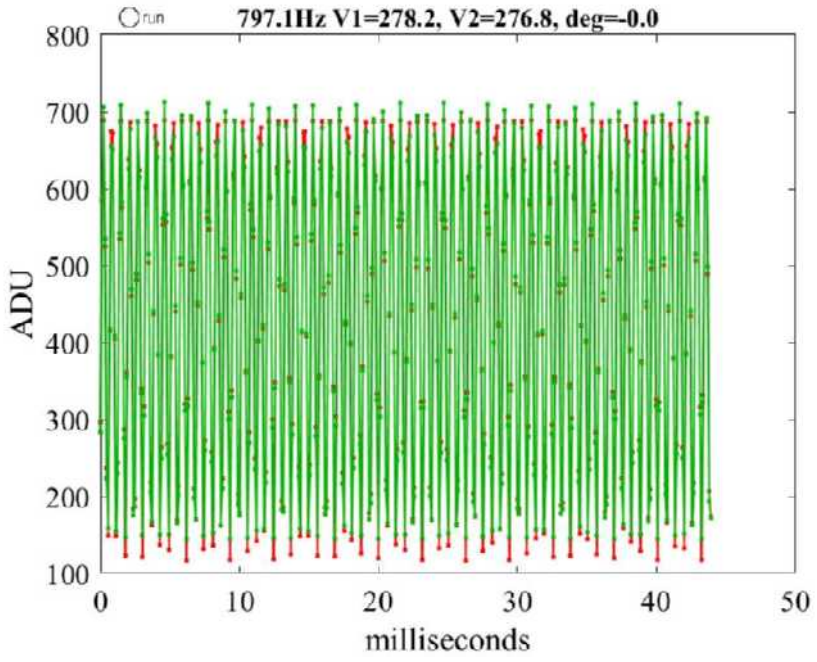


Figure 6. Higher frequency waveforms are visually aliased.

And the low end:

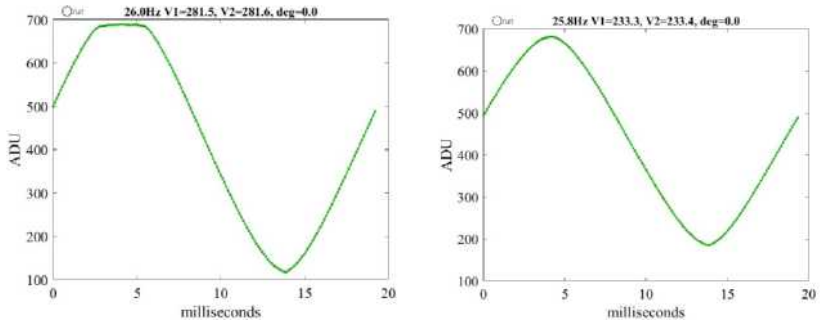


Figure 7. Lowest frequencies are those that reliably contain at least one full period after threshold.

Significant distortion ("harmonic distortion") is evident in the waveform, lowering the amplitude a little is the solution (Fig. 7).

Discussion

Over a wide range of frequencies (~26-800Hz), the amplitudes of the channels yield the same amplitudes, which are within 1% agreement with the DMM frequency and VAC readings. Frequencies agree between Arduino and DMM, and the phases of the same waveforms are always zero. Suspicious that there was some error with phase always zero, I commented out the spline line (% `data(:,2) = spline(tt+.5, data(:,2), tt);`) and ran the code. At (arbitrary) 444Hz, the phase shift was 17.9deg. Is this what we'd predict?

```
T = 1/444*1e6; % period in microseconds = 360 degrees
dPhase = 112/T*360
```

```
dPhase = 17.9021
```

Agreement doesn't get closer. This level of accuracy and precision suggests this apparatus is ready to measure amplitudes and phases in circuits.

We covered a lot of intellectual ground in this activity, and I encourage you to reflect on what you've learned, and jot down some additional questions and notes.

Our Arduino and MATLAB running this code perform the main function of a 2-channel oscilloscope, plotting voltage vs. time. Commercial oscilloscopes offer multiple voltage gains and protect the input from higher voltages. They offer easily adjustable time/division control on the horizontal axis – which we could slow down with `delay`, but unfortunately we can't speed up very much with an Arduino (there actually are much faster 8 bit alternatives to `analogRead`, but the code in assembly language). The $4 Raspberry pi Pico microcontroller is ~13x faster, and Teensy microcontrollers [5] have much faster 16 bit ADCs. Modern oscilloscopes sample voltages billions of times per second, vs. our meager 8.9 thousand times per second, which still blows away human capabilities. So it behoves us to learn how to utilize automated data acquisition and reduction, because manual data acquisition and reduction has become largely obsolete – computers have beaten us, just as calculators beat us for pencil and paper arithmetic. Instead of dwelling on our oscilloscope's shortcomings, consider positive aspects of its performance. Notice how stable and similar the mrms amplitudes of A0 and A1 are, and also how stable the frequency is. Let's quantify this a little by computing the coeficient of variance (= standard deviation / mean) of the amplitudes and frequency with 100 measurements. Turning the amplitude to 12'oclock, and frequency to ~300Hz,

```
freqs = zeros(100,1); ampls = zeros(100,2);
tic;
for k=1:100
    runOnce = true;
    oscope3;
    freqs(k) = sr/(length(td)/npds);
    ampls(k,:) = mrms(data);
end
toc
```

```
Elapsed time is 28.494501 seconds.
```

```
fprintf('CV of A0=%.4f, A1=%.4f frequency = %.4f\n',  ...
    std(ampls) ./ mean(ampls), std(freqs)/mean(freqs))
```

```
CV of A0=0.0019, A1=0.0019 frequency = 0.0207
```

```
fprintf('interchannel difference = %g\n',diff(mean(ampls))/mean(mean(ampls)))
```

```
interchannel difference = -0.000118067
```

So the short term (over 25 seconds) amplitude stability is better than 0.2 percent (2 parts per thousand), and the mean amplitudes of A0 and A1 agree with each other to 100 parts per million (ppm). This is better than most osclloscopes which use 8 bit ADCs (ours is 10 bit).

Some notable lessons I think are:

1. We saw harmonic distortion at full amplitude, where the waveform clearly deviated from sinusoidal.
2. We implemented a simple algorithm in `trim.m` for finding positive-sloped zero crossings in (low-noise) data and the number of periods in the trimmed data.
3. We used `nargout` to return more than one variable (many computer languages don't permit functions to return multiple values, although structures and objects provide a work-around.)
4. We learned about temporal skewing that happens when 2 analog inputs are multiplexed to one analog-to-digital converter (ADC), and used a spline to deskew this systematic error, virtually eliminating it. This is the great thing about systematic errors – that you can build a new "system" that corrects or negates them. Such correction is impossible with random errors because predicting noise is by definition an oxymoron.

One final caution. Just as `oscope2.m` was harder to debug than `oscope1.m` because `try/catch` hides errors, `oscope3.m` has additional hidden potential failure modes. As the system's complexity increases, you'll have to rely on prior experiences debugging the simpler systems, and sometimes revert back to them. When `oscope3` keeps failing, then try running `oscope2` – you lose trigger and amplitudes, frequency and phase difference, but it should show the waveform (or lack of it). You may have to "open the hood" to see if the threshold and channel for `trim` are inappropriate.

Conclusions

We assembled our function generator kit and meticulously validated an apparatus capable of precise measurements of frequency, amplitude, and phase of sine waves from ~26-1000Hz. We could extend the low end of the frequences by reducing the sample rate, i.e., adding `delay();` in the Arduino acquisition loop. Since Arduino memory limits us to ~400 samples/channel, we can't measure a longer buffer but we could spread out the data points. The high frequency end is limited by the ADC sample rate. This might also be tweaked higher using assembly language programming, or by swapping the Arduino for a faster device like the Raspberry pi Pico.

The *processes* we've used to validate would be similar for any system – even expensive commerical ones. Sure, if an instrument costs thousands of dollars, you might be more likely to *assume* it works accurately and precisely, but cost should never supercede empirical evidence in science. And (in my world), the reason I would buy expensive apparatus is to push it against *its* limits. So meticulous validation is *always* an important process for me to perform, and at the very least it increases familiarity with my apparatus.

What else did you find surprising about this activity?

References

[1] https://en.wikipedia.org/wiki/Oscilloscope#Triggered_sweep

[2] https://en.wikipedia.org/wiki/Design_pattern

[3] https://en.wikipedia.org/wiki/Interrupt_handler

[4] https://en.wikipedia.org/wiki/Clock_skew

[5] https://www.pjrc.com/teensy/