Fourier Methods

Johnny Pribyl

March 13, 2018

Abstract

This lab is designed to familiarize the student with thinking in the frequency domain. It requires far more data collection than the previous labs. There are four main objectives. First, we worked with a few simple input functions in order to become familiar with the spectrum analyzer. Second, we investigated the response of an LRC circuit to several different signal types. Third, we did the same thing to an acoustical cavity. And lastly, we examined a system of coupled torsional reed oscillators. Unfortunately, we ran out of time and were not able to collect all of the necessary data from the reed oscillators - so, this report will only include an informal discussion of theory and observed results.

1 Introduction

Somehow, every undergraduate physics student manages to sneak through their classes without ever fully understanding Fourier methods. Brian took it upon himself to fill this educational hole with experimental knowledge. That means this lab operates almost entirely in the frequency domain. Or, in other words, we Fourier Transform *everything*.

Although it's pretty tough to get funding these days, MSU still has some toys leftover from the 90s. Among those is the SR770 Spectrum Analyzer. It boasts a peaceful green-hued digital output and the capacity to live stream the frequency domain of a time-dependent signal. It only lacks the phrase ***DON'T PANIC.*** Ideally, this would be inscribed in large friendly letters on its cover.

We used the SR770 and TBS1052B-EDU oscilloscope extensively in this lab. Both of them are capable of saving hundreds of data points at the press of a button. So, we had quite a bit more real data to analyze for this lab than we have had for any of our previous forays.

As before, you can find all of my code and data at:

`jpribyl/cautious-palm-tree`

1.1 Database Design

The majority of this lab's data is relatively straightforward. Typically there are two columns - one for the dependent variable (say, voltage) and one for

the independent variable (typically time or frequency). We saved this data into .csv files on a USB. However, all of the measurements carry a large amount of metadata regarding input frequency, amplitude, channel, and the source of the data. Unfortunately, all of the metadata is not recorded into the files. We wrote it all down in our lab notebooks, but that makes it difficult to cross reference.

It would definitely be possible to store all the time / voltage data in excel and assign values to the metadata with python as we read it in. However, that approach does not scale very well. It's particularly ineffective for collaborative efforts. Ideally, there would be a way to tie all of the metadata to the files themselves. This ensures data integrity without requiring all collaborators to use an identical code base. It turns out, I'm not the first person in the world who has had this problem. The correct tool for the job is a "database."

Databases are created and maintained with a language called SQL, or Structured Query Language. Every measurement is given a unique identifier that allows anyone to access or "query" all of its metadata.

Our data did not come pre-packaged with unique identifiers, so I had to add them manually. Bash has several incredibly powerful text-editing commands that are perfect for this kind of task. I'm not as familiar with Awk, so I opted to use Sed. If you have a bash shell on your computer you should be able to access the man pages with

```
man sed
```

Otherwise, you can read them online at gnu.org/software/sed/manual/sed.txt. The first step was collecting all of our files into a single directory and naming them numerically so that I can sort through them with a for loop:

```
#!/bin/bash
cur_id=1;

for i in $(ls * | sort -V);
do
    echo $i
    echo $cur_id

    #double quotes allow variables
    sed -i "s/\(.*\)/$cur_id,\1/" $i

    let cur_id+s=1
done
```

I find bash incredibly hard to read, so let's break this down a bit. The very first line is known as a "shebang" and tells the computer to use bash for this script. Next I define a variable "cur_id" which holds the unique identifier that will get prepended to every line of the data files.

After that I run through every file in the current directory and sort them "naturally." For each file, I read out (or echo) the file name and the file's identifier. Then, I tack the current id and a comma in front of every line in the file and increment the cur_id by 1.

Let's say that the first file in a given directory looks like this:

```
0.0000000e+000, -1.3732910e-001,
6.2500000e+001,  9.4042969e+000,
...
1.2500000e+002, -4.9438477e-002,
1.8750000e+002, -7.1411133e-002,
2.5000000e+002, -3.2958984e-002,
3.1250000e+002, -3.2958984e-002,
```

My script will turn it into something that looks like this:

```
1,  0.0000000e+000, -1.3732910e-001,
1,  6.2500000e+001,  9.4042969e+000,
...
1,  1.2500000e+002, -4.9438477e-002,
1,  1.8750000e+002, -7.1411133e-002,
1,  2.5000000e+002, -3.2958984e-002,
1,  3.1250000e+002, -3.2958984e-002,
```

Where 1 is the unique identifier for this measurement. The second file in the directory would look very similar except that it would be prepended with a 2 instead of a 1.

After pre-processing the data, I read it into my database. If you want to play with the database at home, you can restore it using MySQL Workbench from the dump files in

`lab3/data/database/databasedump`

If you don't feel like downloading my database, I really wouldn't blame you. It's not exactly anything groundbreaking. Besides, I took the 30 seconds required to generate an EER diagram mapping the relationships between tables. I'll let you peruse it at your leisure.

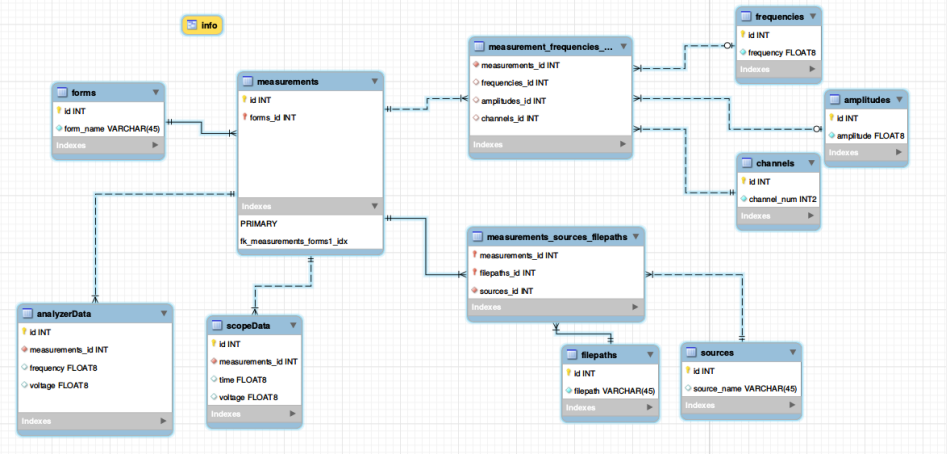

Figure 1: EER diagram mapping the database

The little yellow box entitled "Info" corresponds to a view, or saved query that allows for easier access to the desired data without sacrificing the structure of an ACID database. If you were to run a simple query on the view:

```
select * from info
```

You would get something back that looks like:

id	filepath	source_name	form_name	channel_num	frequency	amplitude
1	data/scope-440121518	scope	square	1	5097	5
1	data/440121518	analyzer	square	1	5097	5
2	data/450121518	analyzer	triangle	1	5097	5
2	data/scope-450121518	scope	triangle	1	5097	5
3	data/scope-455121518	scope	sine	1	11097	5
3	data/455121518	analyzer	sine	1	11097	5
4	data/scope-506121518	scope	sine	2	11097	1
4	data/506121518	analyzer	sine	1	11097	5
4	data/506121518	analyzer	sine	2	11097	1
4	data/scope-506121518	scope	sine	1	11097	5
5	data/511121518	analyzer	sine	1	11097	5
5	data/511121518	analyzer	sine	2	11597	1
5	data/scope-511121518	scope	sine	1	11097	5
5	data/scope-511121518	scope	sine	2	11597	1
6	data/513121518	analyzer	sine	1	11097	5
6	data/513121518	analyzer	sine	2	11697	1
6	data/scope-513121518	scope	sine	1	11097	5

Figure 2: A simple query on the database

Hopefully all of that made sense. If not, don't worry about it – it's not really too relevant to the actual data analysis.

2 Objectives

All of these labs have objectives instead of procedures. They are intended to guide our experiments without spoon feeding us the results. Sometimes I really miss eating bananas out of a jar. Fortunately for me, most grocery stores still stock Gerber snacks.

If you're not familiar with the methods and procedures of this lab, then I would suggest reviewing the manual. It is quite extensive and details all the theory far more eloquently than I ever could. It lives in:

```
lab3/lab_descrip/Fourier_Methods_manual.pdf
```

2.1 Introduction to Spectrum Analyzers

The first thing we did in lab after putting our heads on straight was run a few single frequency waves through the oscilloscope and spectrum analyzer. The goal of this section was to get comfortable with the conversion between frequency and time domains. All of the oscilloscope data lives in the time domain. This means that time is the independent variable and our plots consider $Voltage(time)$. However, it is often useful and always didactic to convert this data into $V(\omega)$ by Fourier Transforming it:

$$X(\omega) = \int_{-\infty}^{\infty} x(t)e^{-i\omega t}dt \tag{1}$$

Technically, we have a discreet data set that isn't quite infinite, so it might be more accurate to say that we are using a Fourier Series – but the theory is extremely similar and a full discourse on Fourier Analysis is outside of the scope of this report.

2.1.1 Sine, Square, and Triangle Waves

If we use the function generator to produce a Sine wave of 11,097 Hz, we get a familiar looking graph of $V(t)$ on the scope:

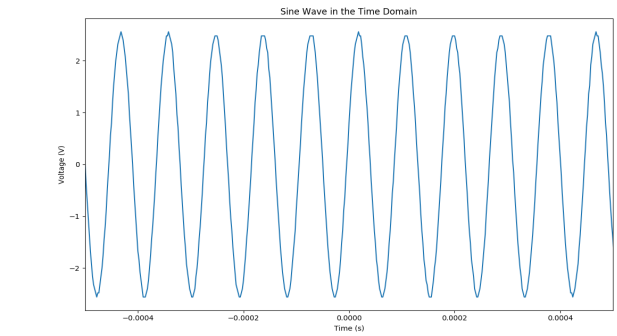

Figure 3: Sine wave of 11.097 kHz collected from the oscilloscope

Now, everyone knows that the Fourier Transform of a sine wave is a delta function so let's transform the data that we collected from the scope and see how it looks:

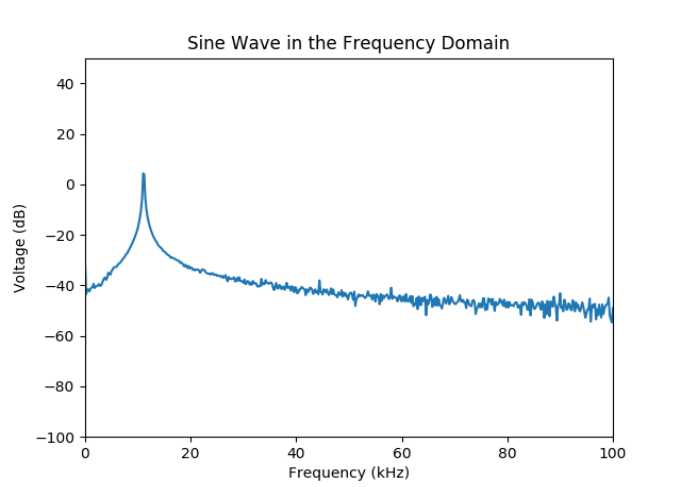

Figure 4: FFT of data presented in Figure 3

As you can see, there is a single peak at 11.097 kHz. So far, it seems like all is well in the world, but let's overlay the results with the data collected from the SR770 and see whether things actually match up.

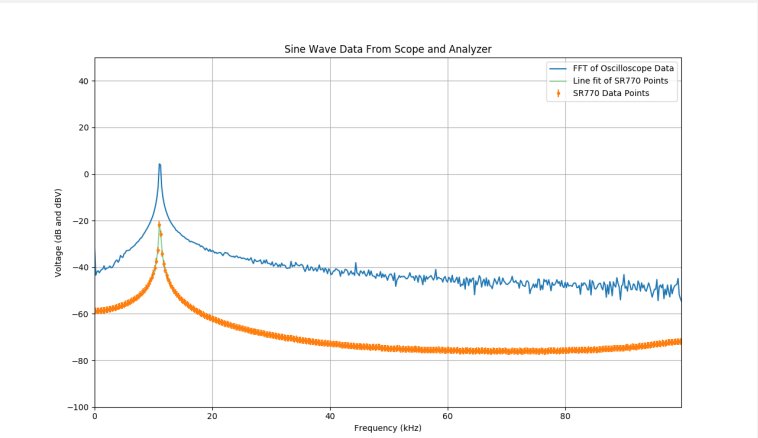

Figure 5: Analyzer data overlaid on top of Figure 4

Things are looking pretty good – however, those amplitudes don't quite match. At a glance that might seem problematic. But, if you look closer at the label on my y-axis, you'll notice that the units are not the same! The data from the SR770 is in dBV while the data from the oscilloscope is in dB. In fact, if you convert everything to a linear scale, multiply by the amplitude of the input current, and convert back to a log scale – the amplitudes just about match up:

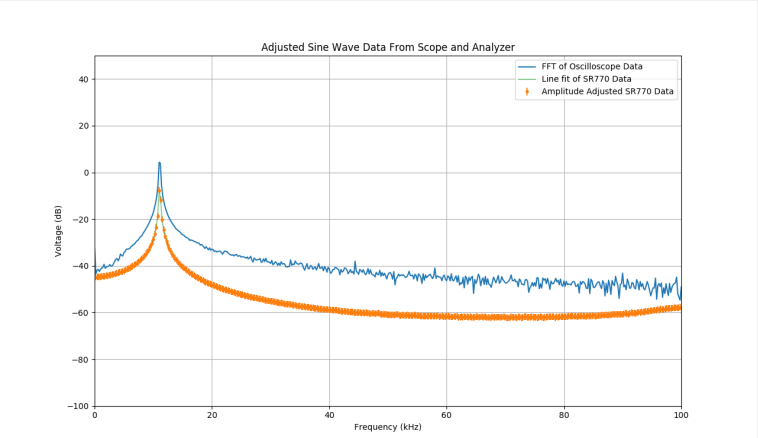

Figure 6: Amplitude matching by converting dBV to dB

The last thing to do before moving on is model the results. The plot is

starting to get a little bit busy, but modeling a sine wave is quite easy. All we have to do is plug in the frequency, transform a vector of points, and overlay the final plot in the series:

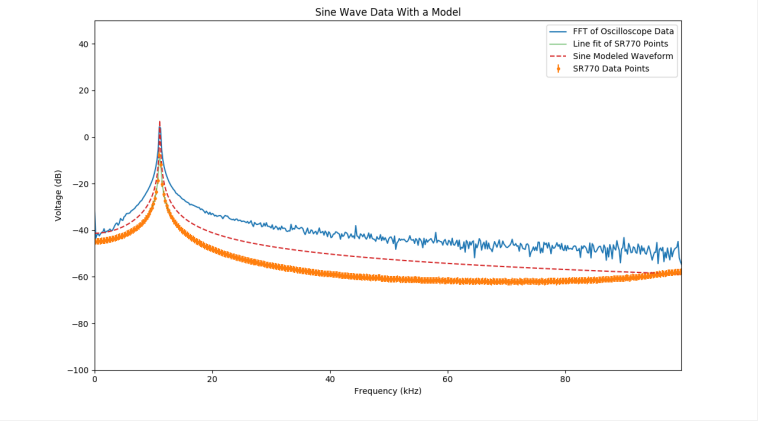

Figure 7: Adding a theoretical model to Figure 6

I'm not going to walk though the entire process for the square and triangle waves; however, here are the final plots that we produced:

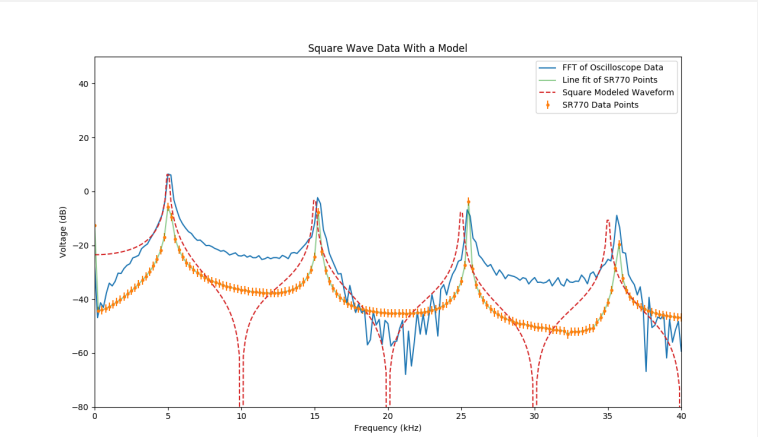

Figure 8: Square wave data (including a model)

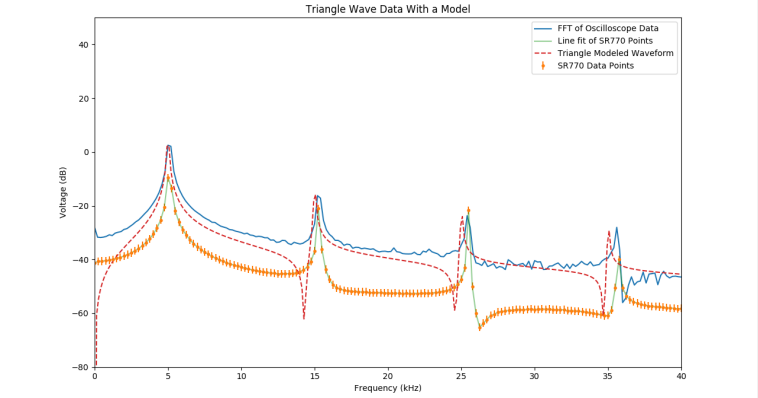

Figure 9: Triangle wave data (including a model)

2.1.2 Summer Module

The process for the summer and multiplier modules was extremely similar. The only difference is that we fed the input signal from the signal generator through the "Addition" module instead of running it directly into the scope.

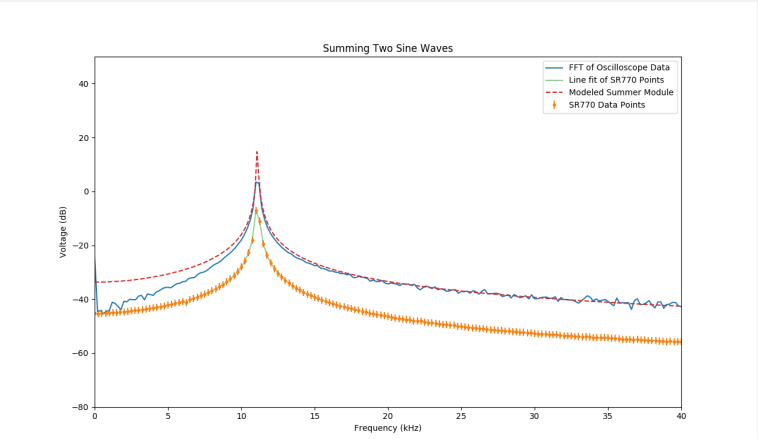

Figure 10: Summing two sine waves. Wave A has a frequency of 11.097 kHz and peak-to-peak amplitude of 5V. Wave B is 1V at 11.097 kHz.

We noticed that you cannot resolve two peaks when the frequencies are close together. For example, if we increase the frequency of the second wave to 11.597 kHz, then you can't quite see the second peak on the SR770. You can actually resolve it by transforming the scope data, but this makes sense because the scope collects roughly 6 times as many data points as the analyzer does.

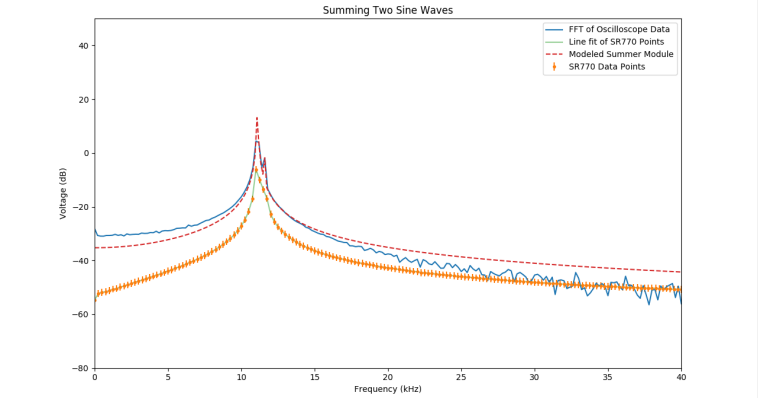

Figure 11: Summing two sine waves - SR770 shows a single poorly localized peak. Wave A is 5V at 11.097 kHz. Wave B is 1V at 11.597 kHz.

Increasing the frequency of the second wave just a touch more, we are able to resolve two peaks on the SR770:

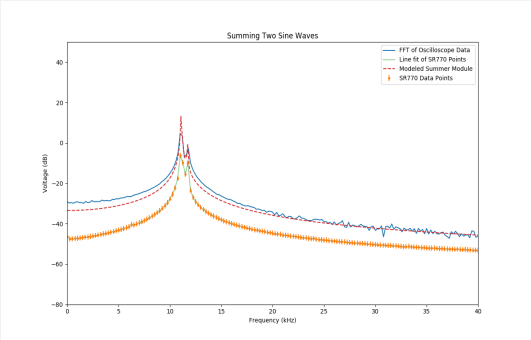

Figure 12: Summing two sine waves - SR770 shows two peaks. Wave A is 5V at 11.097 kHz. Wave B is 1V at 11.697 kHz.

If we separate the frequency substantially, the second peak becomes even more obvious:

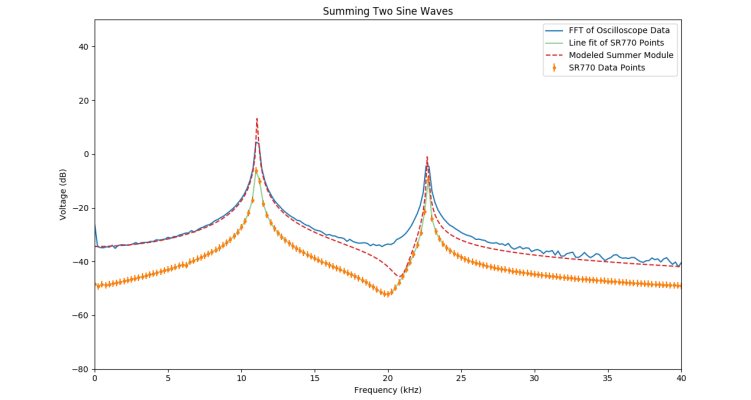

Figure 13: Summing two sine waves - two well resolved peaks. Wave A is 5V at 11.097 kHz. Wave B is 1V at 22.697 kHz.

2.1.3 Multiplier Module

Moving on to the multiplier module, we see very similar results:

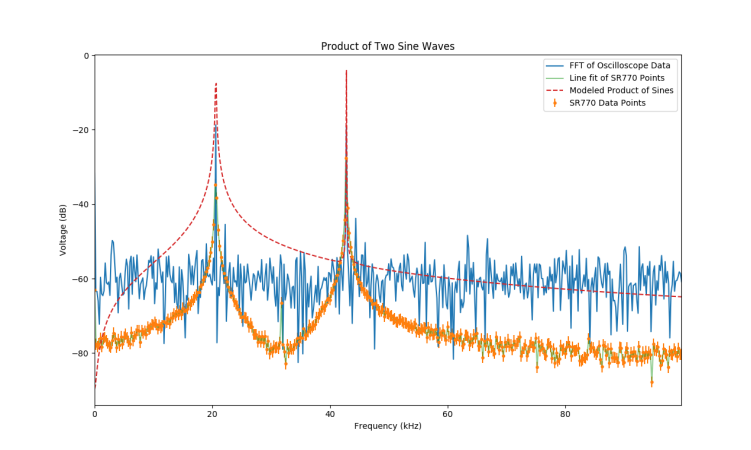

Figure 14: Product of two sine waves - two peaks present. Wave A is 5V at 11.097 kHz. Wave B is 1V at 31.697 kHz.

And setting both input waves to the same frequency:

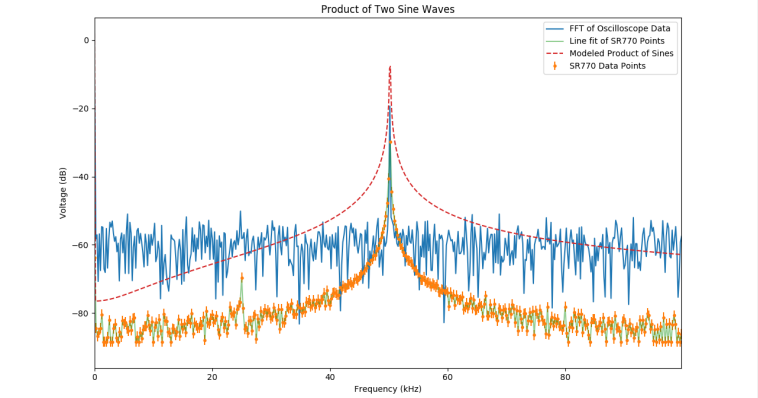

Figure 15: Product of two sine waves - single peak present. Wave A is 5V at 25.097 kHz. Wave B is 1V at 25.097 kHz

2.1.4 Buried Treasure Module

The final introductory activity required us to use the "Buried Treasure" module. This module contains an assortment of noise along with a sine wave whose amplitude is large enough to dominate and appear on the spectrum analyzer. There are three different treasures – treasure A, treasure B, and treasure C. We were successful in finding the sine waves for A and B. We opted to skip C because of time constraints.

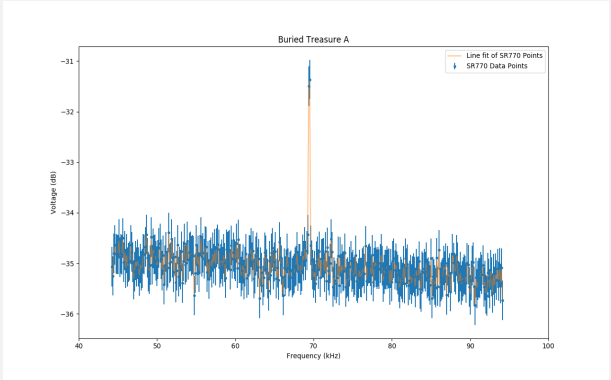

Figure 16: Buried treasure 'A' hides a sine wave of approximately 70 kHz

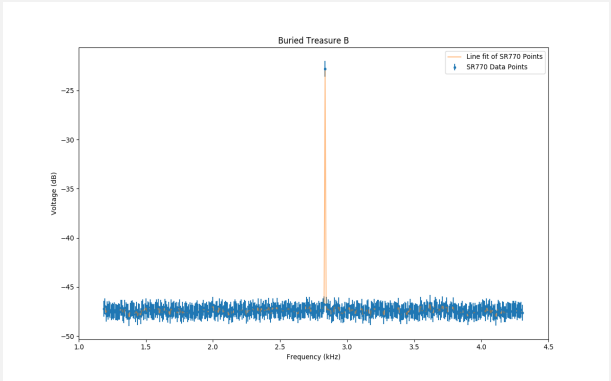

Figure 17: Buried Treasure 'B' hides a sine wave of approximately 2.75 kHz

2.2 LRC Filter

After completing the introductory exercises, we proceeded to investigate the LRC circuit. Our goal was to characterize the circuit's response to a variety of conditions. The circuit's components were all in series to each other and we measured the V_{out} as the voltage across the capacitor:

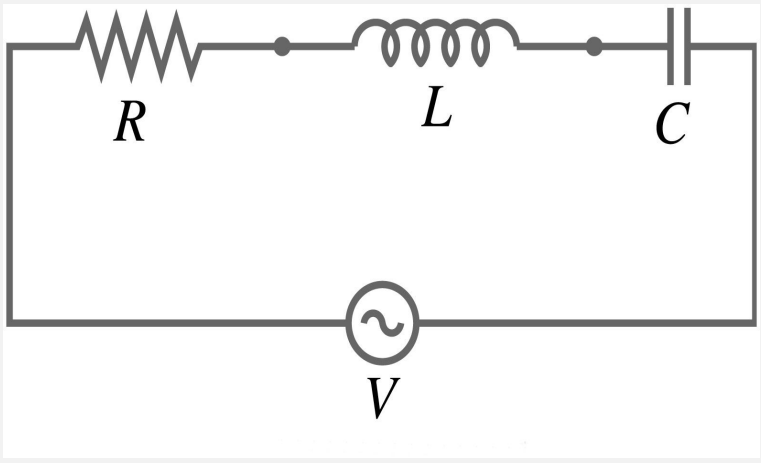

Figure 18: LRC circuit diagram with components in series
(Image courtesy of augustana.edu)

We are provided with (and verify) the values for R, L, and C in this circuit. This makes it easy to calculate the theoretical fundamental resonant frequency using

$$\omega_0 = \frac{1}{\sqrt{LC}} \tag{2}$$

Plugging in the values:

$$L = 68mH$$

$$R = 1000\Omega$$

$$C = 10nF$$

We expect this circuit to have a natural frequency of 6103.313 Hz. In the coming sections, I will show that this is indeed the case.

The theoretical models in this section got interesting. In general, the transfer function of a circuit is a complex function that takes the form

$$\tilde{H}(\omega) = \frac{\tilde{V}_{out}}{\tilde{V}_{in}} \tag{3}$$

In class, we derived $\tilde{H}(\omega)$ for this particular circuit

$$\tilde{H}_{LRC}(\omega) = \frac{1}{1 - \omega^2 LC + i\omega RC} \tag{4}$$

This gives a magnitude of

$$\|\tilde{H}_{LRC}(\omega)\| = \sqrt{\frac{1}{\omega^2 R^2 C^2 + (1 - \omega^2 LC)^2}} \tag{5}$$

And a phase of

$$\phi = -\arctan\frac{Im}{Re} = -\arctan\frac{\omega RC}{1 - \omega^2 LR} \tag{6}$$

Both of these equations will be invaluable in the following sections.

2.2.1 Single Frequency Drive

As before, the first step in understanding and characterizing the circuit is to pass a single frequency through it and plot the results. For the remainder of the lab, I will not be plotting the oscilloscope data. At this point, it should be obvious that I understand the process of transforming data between the time and frequency domains.

When we pass a single frequency sine wave through the LRC, we find that there is a resonance at the input's frequency regardless of its value. For example, when we pass through a sine wave of 12 kHz, the output looks like this:

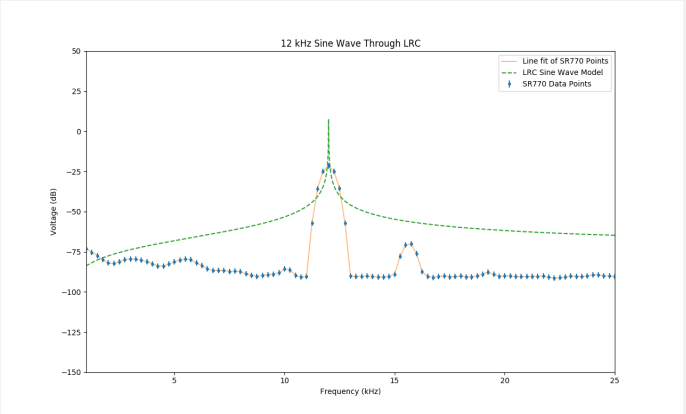

Figure 19: LRC circuit response to a single frequency sine wave with a frequency of 12 kHz

The lab manual makes a point to note that this behavior is unique to single frequency sines. If we were to pass a single frequency square wave with a frequency of 12 kHz through the circuit, the output would not necessarily have a peak at 12 kHz. Just to be thorough, here is a plot of the circuit's response to an 18 kHz sine wave:

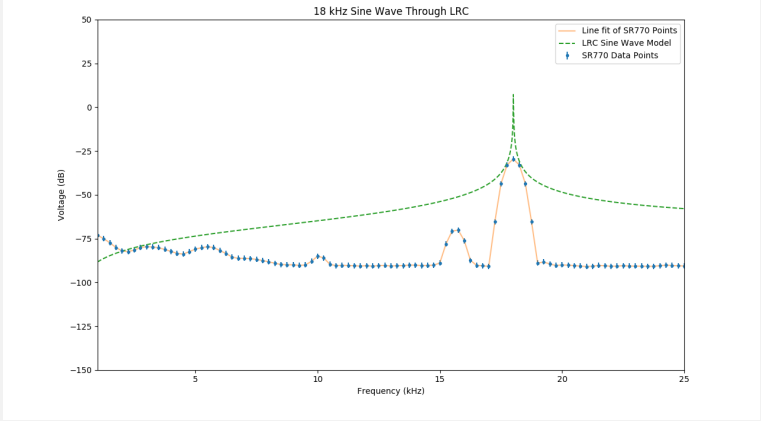

Figure 20: LRC circuit response to a single frequency 18 kHz sine wave

2.2.2 Noise Drive

After driving the LRC with several single frequency inputs – it was time to investigate the circuit's response to noise. Typical white noise consists of a Gaussian distribution of sine waves over a range of frequencies. We used the "noise" setting on the buried treasure module because its frequencies were low enough to avoid causing hardware issues.

Unfortunately, we did not record the array of input voltages. This makes it impossible to recover the units for the data collected. We can't get the amplitude parameter from experimental data. So, when we plot the model, we find that the frequency of the peak lines up perfectly while the amplitudes do not match at all.

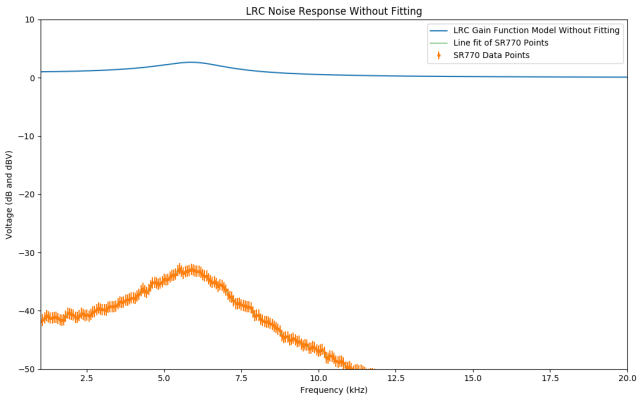

Figure 21: LRC circuit's response to a noise drive without matching amplitudes

Fortunately, we can use the curve_fit function in order to artificially recover the parameters. In other words, we can find the values for amplitude, capacitance, resistance, and inductance that best fit the theoretical model to the data. Once we do this, the plot gets much prettier.

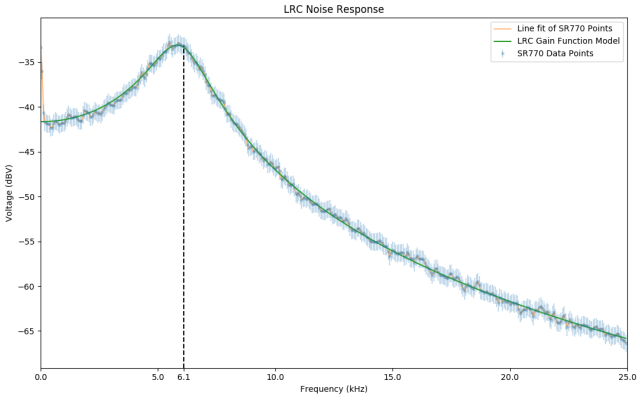

Figure 22: LRC circuit response to a noise drive

During lecture, we predicted the peak of the response to be at a frequency slightly beneath the natural frequency. Experimentally, we can see that this is

in fact the case!

2.2.3 Chirp Drive

The results of driving the LRC with a chirp are extremely similar to those of driving it with noise – except that it is possible to recover the complex phase information.

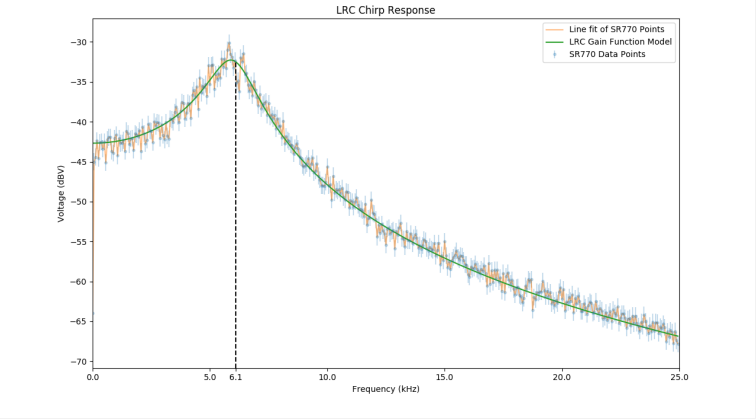

Figure 23: LRC circuit response to chirp drive

Unfortunately, all of the groups had some pretty serious issues collecting phase data. For us, it displayed correctly on the screen of the SR770, but then the data set came out awful! We collected the data twice and were still unable to get anything that looked correct. For the sake of completeness, here is the phase data that we collected:

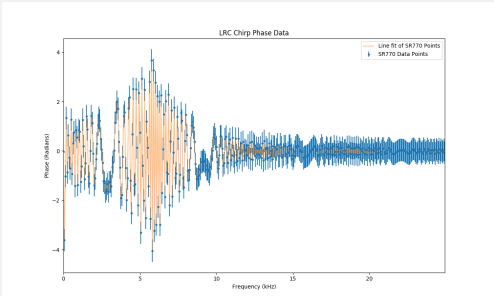

Figure 24: LRC phase data (corrupted)

It is easy enough to model the correct phase results using equation (6) and NumPy's arctan2 method:

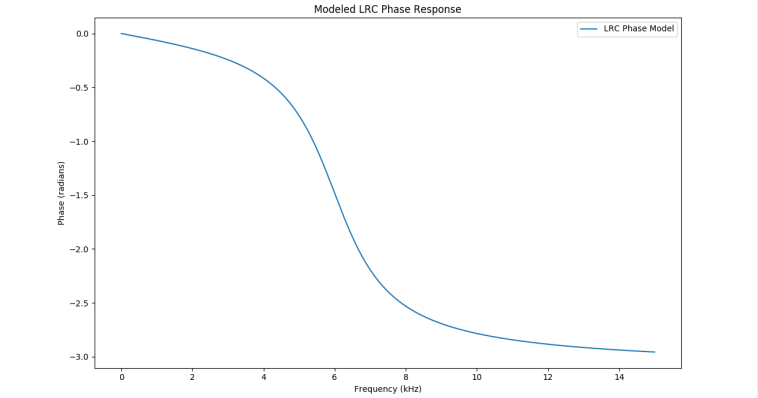

Figure 25: LRC phase model (without data)

2.2.4 Transient Response

The final activity with the LRC circuit involves passing a step function through the circuit. Plotting the results and model we see that the fundamental frequency aligns perfectly

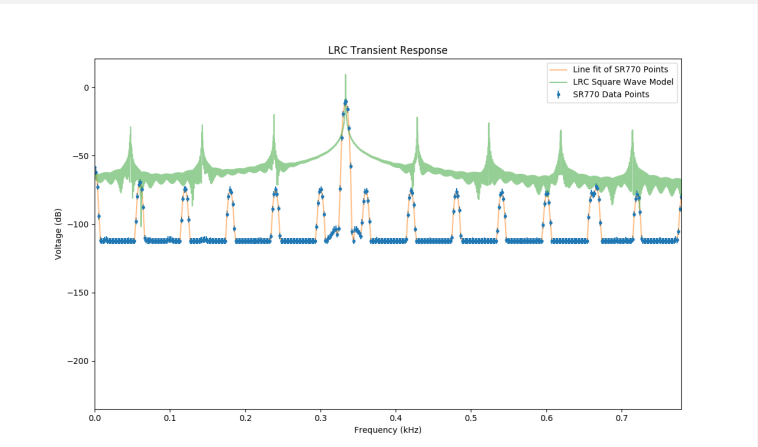

Figure 26: LRC transient response data and model

Unfortunately, the harmonics do not match. To me, it looks like there is either a unit issue, or our model is slightly too naive to catch them.

2.3 Acoustical Cavity

We did not really discuss the mathematical theory behind the acoustical cavity in lecture or lab. So, I will exclude it from here as well. The dedicated reader can easily google resources regarding the resonance of open and closed cylindrical cavities. Additionally, I will not include theoretical models for the data. The procedure for this section was almost identical to that of the preceding section, so I will present the results with a minimal amount of explanation. Refer to the lab manual or my notebook to rectify any confusion that occurs.

2.3.1 Single Frequency Drive

2.3.2 Noise Drive

2.3.3 Chirp Drive

2.3.4 Transient Response

2.4 Coupled Torsional Reed Oscillators

3 Questions

3.1 SR770 Normalization

3.2 Time Domain Usefulness

4 Code Appendix

The coding aspect of this lab was extensive enough that it seemed to warrant utilizing classes. So, I defined a class for measurements, and one for models. Within the model class, there are numerous sub classes. I took time to fully document the code, so it should be fairly straightforward. The goal of this section is to provide a 30,000 foot overview and point out a couple sneaky things. It does not aim to provide a full walk through of the code.

4.1 Measurement Class

The measurement class has several important attributes. I carry uncertainty through all the analyzer data. Propagating error through a Fourier Transform is outside the scope of this lab, so I do not carry it through the oscilloscope data. After reviewing the data sheet, the analyzer is able to resolve data to $\pm$(0.3 dB + 0.02% of full scale) where the full scale excludes window effects and only takes into consideration the actual data points. In python, this looks like

```
# define the full scale of points
an_scale = \
    abs(max(self.an['voltage']) - min(self.an['voltage']))

# apply the uncertainty as an array
self.voltage = \
    pd.Series(uarray(self.an['voltage'], .3 + .02 * an_scale))
```

An instance of the measurement class has access to a method for taking the Fourier Transform. I'm not especially proud of the method, but it does get the job done. There are a couple sneaky things. NumPy's fft method does not automatically shift and normalize the data, so we have to do it explicitly Additionally, using the 'np.log()' method will default to the natural log. We should use 'np.log10()' instead. Lastly, we can use the 'np.fftfreq()' method to recover the frequency bins:

```
# perform the transformation on voltage
fftvolt = np.fft.fft(self.voltage)

# clean up and normalize
fftvolt = np.fft.fftshift(fftvolt)
fftvolt = 2*fftvolt/float(len(self.voltage))

# convert voltage to dB
self.fftdbv = 20.*np.log10(np.abs(fftvolt))

# determine the time step and window length
self.time_step = self.time[2]- self.time[1]
self.win_length = len(self.time)

# recover frequency bins and shift them
self.fftfreq = np.fft.fftfreq(
    self.win_length, self.time_step)

self.fftfreq = np.fft.fftshift(self.fftfreq)
```

Implementing and plotting the measurement class is very simple. For example, let's say you want to examine the 5th measurement we made. You could create a new .py file and fill it with:

```
from measurement import *
data = measurement(5)
```

Upon running this script, you will be greeted with a readout that provides all the metadata regarding the 5th measurement

```
id    filepath form_name  channel_num  frequency  amplitude
 5   511121518      sine            1    11097.0        5.0
 ...
 5  scope-511121518 sine            2    11597.0        1.0
```

From here, you should have enough information to plot the data and give it an accurate title, label, etc. In this case the full script would be:

```
from measurement import *
data = measurement(5)

data.model(
    title='Summing Two Sine Waves',
    ylabel='Voltage (dB)',
    xlabel='Frequency (kHz)',
    plotSc=False)

plt.legend()
plt.show()
```

I have caught all the errors and handled them in a way that prevent's the module from crashing. But, let's say you can't remember the ID of the measurement that you want to examine. You can use the 'measurement.query()' method to perform any SQL query on the database. For example, let's say you know that the wave in which you are interested is driven by a triangle. You could do something like

```
from measurement import query
query('SELECT * FROM info WHERE form_name = "triangle"')
```

This would print a list of all the meta data for measurements with a triangle wave input function. From there, you could pick out the ID for which you are looking and model the data.

4.2 Model Class

The model class has quite a few sub classes. All of it's sub classes have a model method which overrides the model method. The structure is fairly complicated, so here is a tree illustrating the relationships between sub classes:

```
model
    |-----fourierModel
                |-----sine
                |-----triangle
                |-----square
                |-----sineSum
                |-----sineMult
                |-----lrc
                        |-----lrcSine
                        |-----lrcSquare
                        |-----lrcMultFreq
                                    |-----lrcMultFreqGain
                                    |-----lrcMultFreqPhase
```

Implementing the model class is also very simple. Let's return to the previous example we had. We were investigating the 5th measurement. By comparing the file-path to my lab notebook, we can deduce that measurement 5 corresponds to the addition module with one wave at 11.097 kHz and the other at 11.597 kHz. Once we know that, we can examine the model classes and deduce that sineSum is probably the correct one to use. Implementation is exactly what you would expect:

```
import model

mymodel = model.sineSum(
    freqA=11.097,
    freqB=11.597,

    # using RMS amplitudes
    ampA=(5 / (2 * np.sqrt(2)),
    ampB=(1 / (2 * np.sqrt(2))))
)

mymodel.model()
plt.legend()
plt.show()
```

The form of implementation will be identical for any of the measurements. Auto completion suggestions are very helpful in knowing the order of the required parameters.