Sargis S Yonan

Fourier Series & Transforms with MATLAB

Electrical Engineering 103L
Signals, Systems and Transforms Lab
UC Santa Cruz
28 April 2016

Section I: Introduction

The Fourier Series Expansion is a method for approximating any periodic function by linearly combining sinusoidal terms. In this lab, I will generate approximated discrete periodic functions using the Fourier Series in MATLAB, and plot the results to visually see the accuracy of the expansion with finite terms. For the most part of my mathematical exposure, the functions I have been analyzing were in the time domain. In this lab, I will also explore the frequency domain using MATLAB to transform functions from the time domain to the frequency domain using the Fourier Transform using the FFT algorithm.

Experiments

The Fourier Series Expansion

An Even Square Function

To begin creating approximated functions using sine terms, we must first create a periodic function. I will use MATLAB to create an even square wave using the following in a MATLAB script:

```
%% Making A Square Wave
t = -10: .001: 10;
% w0 = 2pi/T --> T = pi/4
% shift to the right two units
s = .5 * square((pi/4) * (t + 2)) + .5;
plot (t, s)
axis ([-10 10 -1 2]);
grid on
hold on
```

The square wave has a period of 8 units, and a radial frequency of π radians per second. Using the `SQUARE` function in MATLAB, I had to scale and shift the graph to make the square wave an even function with an amplitude of 1 unit. After plotting this functions across a couple of periods, I received the following plot:

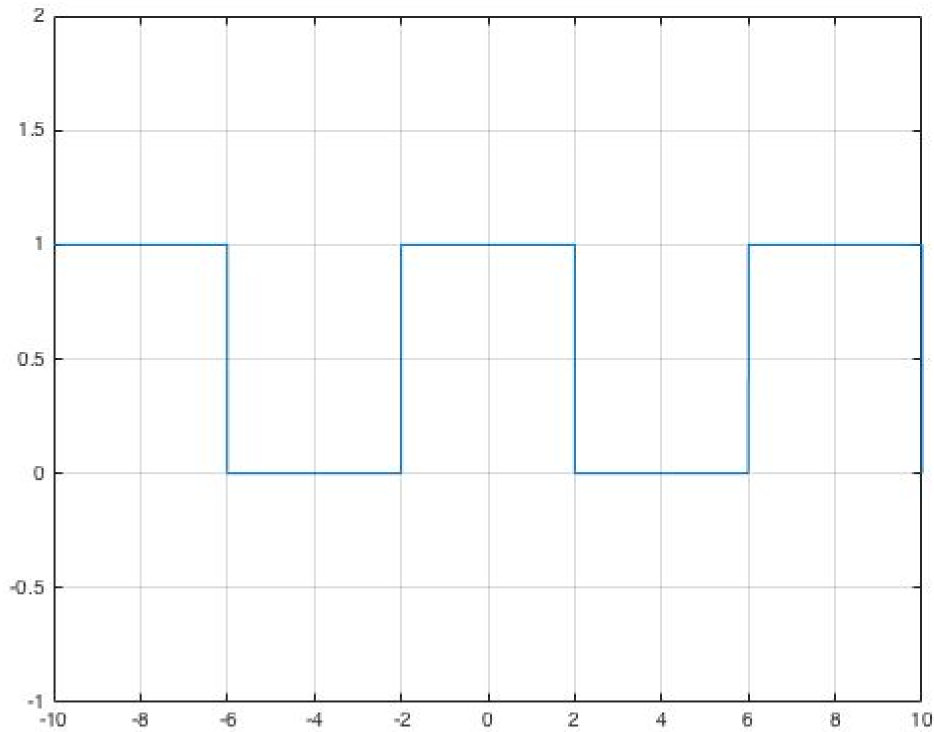

Figure 1: *An even square wave function plotted with MATLAB*

In order to use the Fourier Series on the square wave, I first had to compute the summation of the infinite series. The Fourier Series is an infinite sum that uses the scaling, summing, and frequency manipulation of sinusoidal functions to produce an approximation (expansion) of a function. The trigonometric form of the series is defined by the following:

$$x(t) = a_0 + \sum_{n=1}^{\infty} a_n cos(n\omega_0 t) + \sum_{n=1}^{\infty} b_n sin(n\omega_0 t)$$

Where a_0, a_n, and b_n are the linear terms scaling each term of the series. To calculate these coefficients as functions of n, and knowing the period, definition, and frequency of the square wave function, I used the following definitions of each:

$$Period = T_0 = 8$$
$$Frequency = \omega_0 = \frac{2\pi}{8} = \frac{\pi}{4}$$
$$Definition = x(t) = \{1\ from\ -2 \le t < 2\ \&\ 0\ from\ 2 \le t < 6\}$$

$$a_0 = \frac{1}{8}\int_0^{T_o} x(t)\,dt = \frac{1}{8}\int_{-2}^{2} 1\,dt = \frac{1}{4}+\frac{1}{4} = \frac{1}{2}$$

$$a_n = \frac{2}{T_0}\int_{T_0} x(t)cos(n\omega_0 t)\,dt = \frac{1}{4}\int_{-2}^{2} cos(n\omega_0 t)\,dt = \frac{sin(2\omega_0 n)}{4n\omega_0} - \frac{sin(-2\omega_0 n)}{4n\omega_0} = \frac{2}{n\pi}sin(\frac{\pi}{2}n),\ \forall n \in N$$

$$b_n = \frac{2}{T_0}\int_{T_0} x(t)sin(n\omega_0 t) = \frac{1}{4}\int_{-2}^{2} sin(n\omega_0 t)\,dt = \frac{-cos(2n\omega_0)}{4n\omega_0} + \frac{cos(2n\omega_0)}{4n\omega_0} = 0$$

The infinite series approximation was now known, and defined as follows:

$$x(t) = \frac{1}{2} + \sum_{n=1}^{\infty} \frac{2}{n\pi}sin(\frac{\pi}{2}n)cos(n\frac{\pi}{4}t)$$

Note, since the square wave was an even function in this example, the b_n term went to zero. Likewise, a purely odd function will produce 0 a_n terms.

Since the series has been represented, using a looping program in MATLAB, I was able to construct a program to compute n iterations of the Fourier Series representation of the square wave. I used a plotting function in the loop to produce a plot for each iteration so I could visualize the function being approximated in real time. Using the following script, which performs the computational requirements to satisfy the series representation, I received the graph in Figure 2.

```
%% FOURIER TRANSFORM LOOP
N   =   10000;      %Number of  sums
a0  =   .5;
w0  =   (pi/4)
f   =   a0;  % your approximated function
for n   =   1:1:N
% Calculating and plotting the fourier terms and coefficients
a_n =   2/(n*pi)*sin((pi/2) * n);
b_n =   0;
f   = f + a_n*cos(n*w0*t) +  b_n*sin(n*w0*t);
hold    on
plot(t,f)
pause(0.5)
end
```

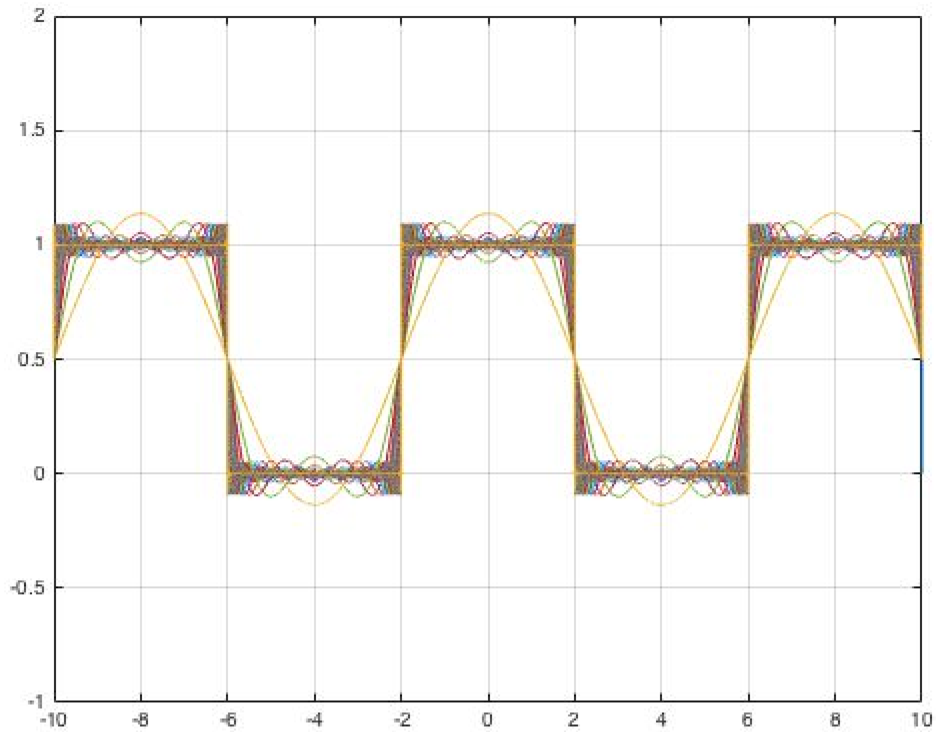

Figure 2: *The Fourier Series representation of the even square wave superimposed upon an actual even square wave*

I computed and plotted 10,000 terms of the square wave, and superimposed the original square wave from Figure 1 to see the resemblance. The function was approximated to a good degree over 10,000 iterations.

An Odd Square Function

I then went on to test the Fourier Series on an odd function. This time, I created an odd square wave using the following methods to produce and plot the square wave:

```
%% Making A Square Wave
t = -10: .001: 10;
% w0 = 2pi/T --> T = pi/4
% shift to the right two units
s = .5 * square((pi/4) * (t)) + .5;
plot (t, s)
axis ([-10 10 -1 2]);
grid on
hold on
```

The square wave from the previous section in Figure 1 was simply shifted one unit to the right to produce an odd function. The period and frequency remained the same.

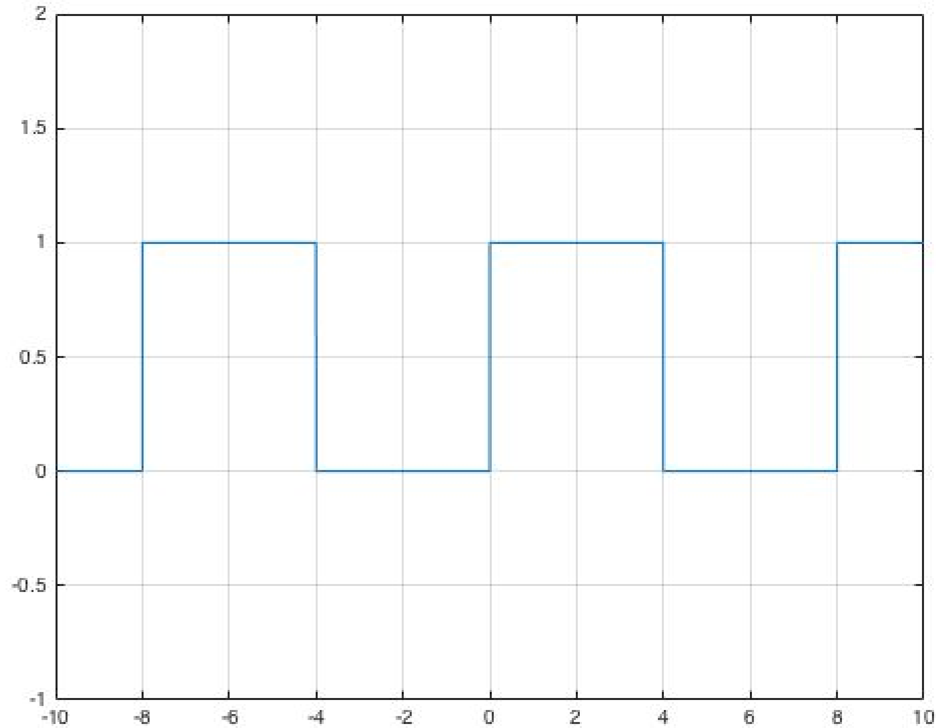

Figure 3: *An odd square wave function plotted with MATLAB*

Using a similar approach to calculating the series representation of the even square wave, I executed the needed arithmetic to generate an infinite series representation of the odd square wave.

$T_0 = 8$

$\omega_0 = \frac{2\pi}{8} = \frac{\pi}{4}$

$x(t) = \{1\ from\ 0 \le t < 4\ \&\ 0\ from\ 4 \le t < 8\}$

$a_0 = \frac{1}{8}\int_0^{T_o} x(t)\,dt = \frac{1}{8}\int_0^4 1\,dt = \frac{1}{4} + \frac{1}{4} = \frac{1}{2}$

$a_n = \frac{2}{T_0}\int_{T_0} x(t)cos(n\omega_0 t)\,dt = \frac{1}{4}\int_0^4 cos(n\omega_0 t)\,dt = 0,\ \forall n \in N$ odd function, an→ 0

$$b_n = \frac{2}{T_0}\int_{T_0} x(t)sin(n\omega_0 t) \;=\; \frac{1}{4}\int_0^4 sin(n\omega_0 t)\,dt \;= \frac{1}{4}[\frac{-1}{n\omega_0}cos(n\omega_0 t)]\,from\ 0\ to\ 4$$

$$= \frac{-1}{4n\omega_0}cos(4n\omega_0) \;+\; \frac{1}{4n\omega_0}$$

$$b_n = \frac{-1}{\pi n}cos(n\pi) \;+\; \frac{1}{\pi n} \;= \frac{1}{\pi n}(1 - cos(n\pi))$$

$$x(t) \;=\; a_0 + \sum_{n=1}^{\infty} a_n cos(n\omega_0 t) + \sum_{n=1}^{\infty} b_n sin(n\omega_0 t)$$

I then plugged in the coefficients I calculated into the Trigonometric Fourier Series Definition, and received the following.

$$x(t) \;=\; \frac{1}{2} + \sum_{n=1}^{\infty} \frac{1}{\pi n}(1 - cos(n\pi))sin(n\frac{\pi}{4}t)$$

Using a for loop in MATLAB, I calculated just 100 iterations of the odd square wave approximation using the following script, and plotted the each iteration and I superimposed the expansion upon the original wave in Figure 3, into Figure 4.

```
xlabel('time (t)');
ylabel('x(t)')
title('Trig Fourier Series of Odd Square Wave')
%% TRIG. FOURIER TRANSFORM LOOP
N   =   100;      %Number   of  sums
a0  =   .5;
w0  =   (pi/4);
f   =   a0;  % approximated function
for n   =   1:1:N
a_n =   0;
b_n =   (1/(pi*n))*(1-cos(n*pi));
f   = f + a_n*cos(n*w0*t) +  b_n*sin(n*w0*t);
hold on
plot(t,f)
pause(0.5)
end
```

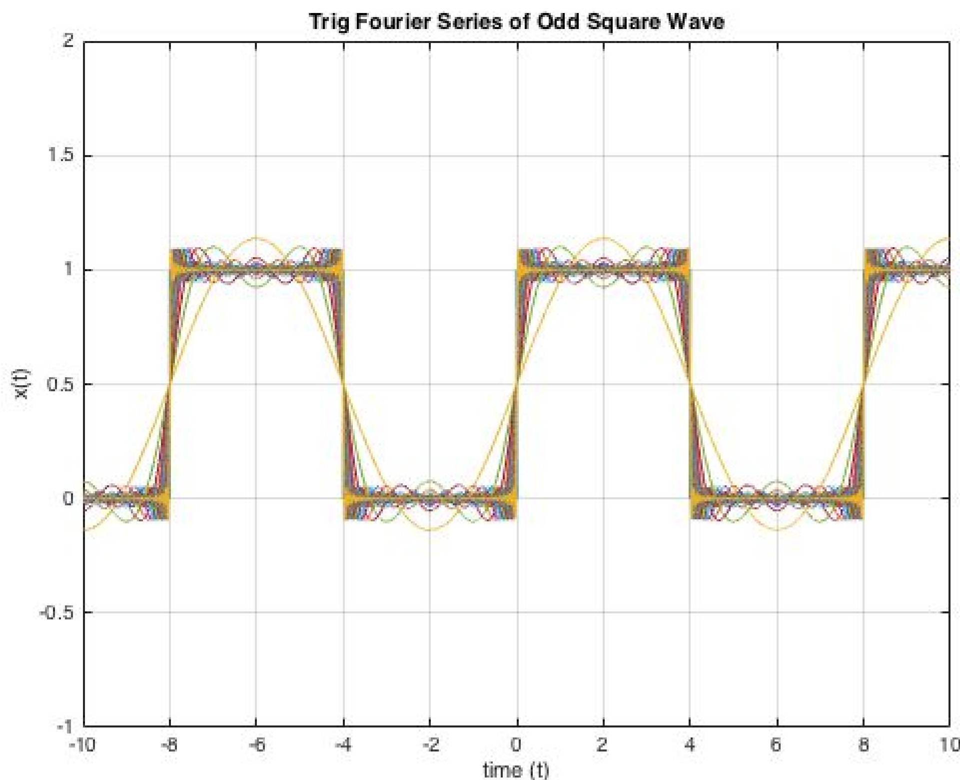

Figure 4: *The Fourier Series representation of the odd square wave superimposed upon an actual odd square wave*

The series created a very similar looking odd square wave, even with just 100 iterations of the series.

In both of the series representations that I computed and plotted, an overshoot before and after each edge of the square waves can be observed. This overshoot is referred to the Gibbs Phenomenon, and it occurs because when approximating a discontinuous functions, a sine wave has trouble perfectly reproducing discontinuities since it is a continuous function by nature. In the even square wave example, I produced the series for the square wave over 10,000 iterations, and for the odd square, I produced 100 iterations. It can be observed on the top most squares on both images have slightly different overshoots. The overshoot on the 100 iteration square wave is .2 units over and under the intended amplitudes. The 10,000 iteration wave has only a .1 unit overshoot. This implies, and backs the theory, that as the number of iterations of the Fourier series go to infinity, the Gibbs Phenomenon does not exist. The wave should be perfectly replicated over an infinite number of iterations. The error in the approximation (Gibbs Phenomenon) also decreases with more iterations.

In order to produce the series representations for the square waves to begin with, I used the Trigonometric Fourier Series representation definition, but there also exists an exponential form:

$$\sum_{n=-\infty}^{\infty} C_n e^{jnw_0 t}$$

As I will prove, the two series representations are equal. The trigonometric Fourier series turns out to be identical to the exponential fourier series form. By using the definitions of the two, and Euler's identity to substitute: $e^{j\pi t} = cos(\pi t) + jsin(\pi t)$, we have:

$$proof: \quad x(t) = a_0 + \sum_{n=1}^{\infty} a_n cos(n\omega_0 t) + \sum_{n=1}^{\infty} b_n sin(n\omega_0 t) = \sum_{n=-\infty}^{\infty} C_n e^{jnw_0 t}$$

Since cosine is even and sine is odd, we will expand the cosine term as follows:

$$By\ Component: \ x_n(t)e^{jnw_0 t} + x_{-n}e^{-jnw_0 t}$$

$$= x_n(t)cos(n\omega_0 t) + x_{-n}cos(-n\omega t) + j(x_n(t)sin(n\omega_0 t) + x_{-n}sin(-n\omega_0 t))$$

$$= (x_n(t) + x_{-n}(t))cos(n\omega_0 t) + j(x_n(t) - x_{-n}(t))sin(n\omega_0 t)$$

$$\forall n, \ we\ have: \ x_0 + \sum_{n=1}^{\infty} (x_n(t) + x_{-n}(t))cos(n\omega_0 t) + j(x_n(t) - x_{-n}(t))sin(n\omega_0 t)$$

$$Since\ cos(n\omega_0 t) = \frac{e^{jn\omega_0 t} + e^{-j\omega_0 nt}}{2} \ and\ sin(n\omega_0 t) = \frac{e^{jn\omega_0 t} - e^{-j\omega_0 nt}}{2j}$$

$$we\ have, \ a_0 = x_0, \ a_n = x_n + x_{-n} \ s.t.\ n \geq 1, \ and\ b_n = j(x_n - x_{-n})$$

This means that any period function, whether discrete or continuous, can be represented as an infinite series of exponential **or** sinusoidal terms, or be used to produce a very close approximation of the original function.

The Fourier Transform

The Fourier Transform is a method of decomposing functions of time into functions of frequencies that are inherent to the functions themselves. This is useful to the field of engineering because of the transforms ability to help an engineer figure out the occurrences of certain frequencies in a system. From the transform, one can compute the prominent frequencies in a circuit, for example, to see the dominant frequencies and harmonics in the circuit. This could help an engineer optimize their design to reduce extraneous frequencies coming from the circuit, possibly reducing energy consumption, or interferences. The magnitude of each frequency in the frequency domain of the Fourier Transform gives the power of the system at that point, and the phase of the frequency gives the phase shift of that point, in frequency space, from a basic sine wave in the time domain.

It turns out that the Fourier Transform is closely related related to the Fourier Series. Recall, that the Fourier Series breaks any periodic function into a series of sinusoidal terms. The Fourier Transform takes those sinusoids from the series approximation, and generates a spectrum of the sinusoids that compose the function in question. The magnitude of the Fourier Transform of a periodic function at a given point, is the amplitude of the point’s component in the Fourier Series Representation. The phase of that point in the spectrum of the frequency domain of the function, is the phase shift of the Fourier Series Representation equivalent component’s wave from a sine function of the same frequency. This close relationship between the two operations comes from the fact that the Transform was derived from the Fourier Series itself.

Fast Fourier Transforms

The Fast Fourier Transform is an algorithm for quickly computing the Fourier Transform of a signal. This method is useful for computing the transform of a signal using a computer, or on the fly using a spectrum analyzer. The FFT algorithm computes the Fourier Transform with the same accuracy as directly computing the Fourier Transform by definition, at any term, with much less time. As a caveat, when using a computer, a small floating point approximation exists in the result of an FFT. By converting the elements of the function in question, to a matrix representation called a Discrete Fourier Matrix, we can represent any discrete signal as a matrix of DC and AC components, when multiplied by a unitary scalar. Using the fact that most matrices that exist are sparse matrices (meaning, most of the matrix’s elements are zero), a

transformation using matrix multiplication can be sped up from a runtime efficiency of $\Theta(n^2)$ to $\Theta(nlogn)$ for n elements. This is because the matrix is represented as a data structure of linked nodes, instead of arrays, where the zero elements are NULLed out and skipped over. Since any product of an element and zero during matrix multiplication is zero, the product is simply skipped over in the linked list (something not efficiently executed using arrays) making the transformation drastically faster than traditionally calculating the transform by definition using a nested for-loop.

To compute the FFT of the even square wave in the first part of the lab in MATLAB, we simply use the `FFT` function with the function in the time domain as its argument, as follows:

```
ts = 0.01;
L = 1e5;
t = (0:L - 1);

y = .5 * square((pi/4) * (t + 2)) + .5;

% Transforming
N = 2^nextpow2(L);
Y = abs(fft(y, N)/L);
f = 1/(ts) * linspace(-0.5, 0.5, N);

% Plotting
subplot(211)
plot(t, y), title ('Time Domain')
subplot(212)
plot(f, 20*log10(fftshift(Y)))
title('Frequency Domain')
```

This script also considers the function in the time domain, and plots the transform in the frequency domain in a subplot along with the function the time domain representation.
The script produces the following graph:

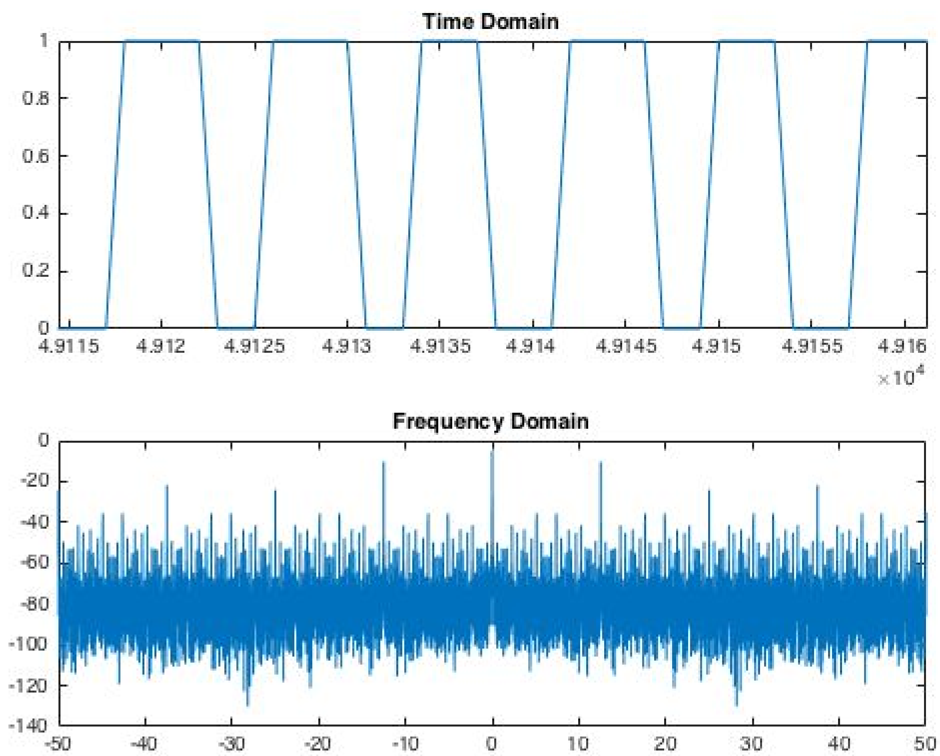

Figure 5: *Even square wave Fourier Transform*

The peaks in the frequency spectrum of the even square wave are where the function produces the most power. This case, the frequency of $\approx 1 \approx \frac{\pi}{4} = .78$, produces the most power, in the middle of the bode plot in Figure 5.

I also computed the FFT of a sawtooth wave using a similar script as before using the `SAWTOOTH` function to produce a sawtooth wave with a period of 2π, and frequency of 1.

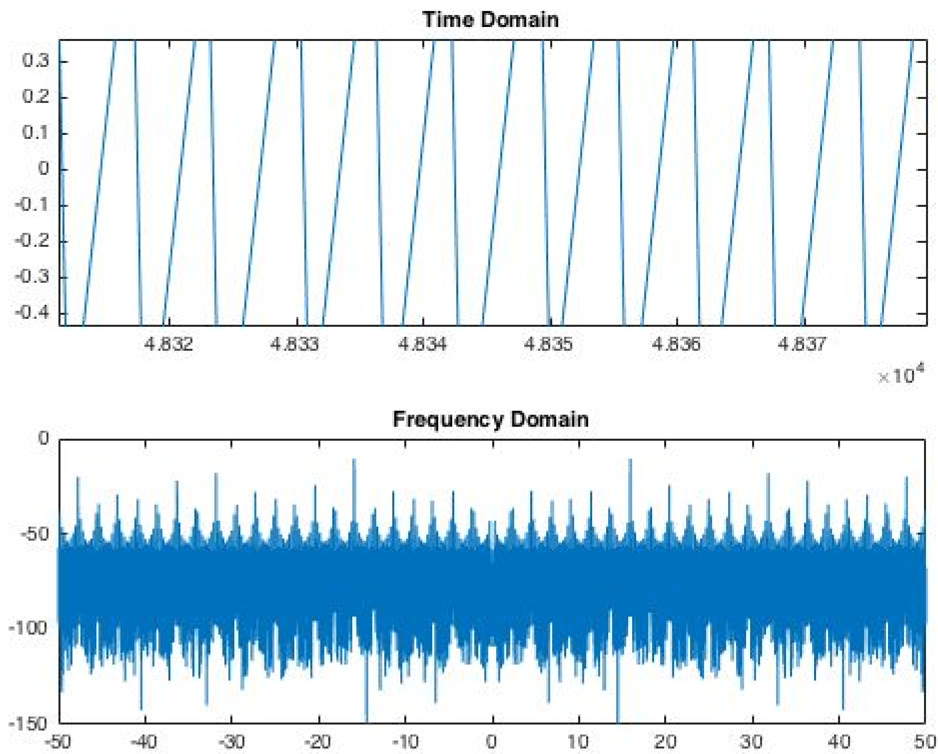

Figure 6: *A sawtooth Fourier Transform*

The sawtooth wave in question has a frequency of 1 radian per second. As a result, the Fourier Transform has peaks at nearly every point. This is because these peaks have significant power due to their harmonics with the resonant frequency of 1. Each of these peaks is a mode of the frequency of the sawtooth wave i.e. every peak produces a decent amount of power with respect to other points. The points that are significantly higher are those where the sawtooth most frequently runs at.

Conclusion

In conclusion, I now have a better grasp of the concepts covered in this lab, including The Fourier Series Expansion, The Fourier Transform, The Fast Fourier Transform Algorithm, and some useful applications of these mathematical methods using MATLAB. Knowing that any function can be represented as a linear combination of sinusoids, and that those terms can be broken down into their component frequencies and plotted in the frequency domain will surely be useful engineering skills in the future when producing higher quality designs and products.

Fourier Transforms With MATLAB

Section II: Introduction

The Fourier Transform is a method of decomposing functions of time into functions of frequencies that are inherent to the functions themselves. This is useful to the field of engineering because of the transforms ability to help an engineer figure out the occurrences of certain frequencies in a system. From the transform, one can compute the prominent frequencies in a circuit, for example, to see the dominant frequencies and harmonics in the circuit. This could help an engineer optimize their design to reduce extraneous frequencies coming from the circuit, possibly reducing energy consumption, or interferences. The magnitude of each frequency in the frequency domain of the Fourier Transform gives the power of the system at that point, and the phase of the frequency gives the phase shift of that point, in frequency space, from a basic sine wave in the time domain.

It turns out that the Fourier Transform is closely related related to the Fourier Series. Recall, that the Fourier Series breaks any periodic function into a series of sinusoidal terms. The Fourier Transform takes those sinusoids from the series approximation, and generates a spectrum of the sinusoids that compose the function in question. The magnitude of the Fourier Transform of a periodic function at a given point, is the amplitude of the point's component in the Fourier Series Representation. The phase of that point in the spectrum of the frequency domain of the function, is the phase shift of the Fourier Series Representation equivalent component's wave from a sine function of the same frequency. This close relationship between the two operations comes from the fact that the Transform was derived from the Fourier Series itself.

In this lab, we will use MATLAB to compute the Fourier Transforms in order to get functions of frequency out of functions of time. Instead of using the definition of the Fourier Transform:

$$F(\omega) = \int_{-\infty}^{\infty} f(t) exp(j\omega t)\, dt$$

We will use MATLAB's `FOURIER` function to transform symbolic representations of our systems of time into systems of inverse time.

Experiments

First I will test out MATLAB's built-in Fourier Transform function `FOURIER`. I created some symbolic variables in order to create a symbolic expression in MATLAB as we did in Lab 2. I set a function of time (f(t)), and used the `FOURIER` function to have the fourier transform of: $f(t) = 1$ computed.

```
syms w t;
f(t) = 1;
fourier(f,w)
```

MATLAB returned the following for the Fourier Transform of 1.

```
ans =
2*pi*dirac(w)
```

Using a Fourier Transform Table, it is evident that:

$$For\, f(t) = 1$$

$$Fourier(f(t)) = F(\omega) = 2\pi\delta(\omega)$$

So the result is true.

Next, I computed the Fourier Transform of a cosine function with a radial frequency of one.

Since $for\ cos(t),\ T_0 = 2\pi,\ \omega_0 = \frac{2\pi}{T_0} = \frac{2\pi}{2\pi} = 1,\ cos(t)$ has a radial frequency of 1 radian per second.

I used MATLAB's built in Fourier Transform function to give me the Fourier Transform of the cosine function in terms of omega.

```
syms w t;
fourier(cos(t), w)
```

MATLAB returned the following:

```
pi*(dirac(w - 1) + dirac(w + 1))
```

This is correct, since:

$$F(\cos\omega_0 t) = \pi[\delta(\omega-\omega_0) + \delta(\omega+\omega_0)],\ where\ \omega_0 = 1$$

$F(\omega) = \pi[\delta(\omega-1) + \delta(\omega+1)]$, MATLAB's result, is true.

Impulse Response

I then defined a 2 unit wide, 1 unit height step function using heaviside functions (seen in Figure 1), and computed the Fourier Transforms of it using MATLAB.

```
syms t w;
f(t)=heaviside(t + 1) - heaviside(t - 1);
figure
subplot(2,1,1)
ezplot(f(t),[-2 2])
fourier(f(t),w)
subplot(2,1,2)
ezplot(fourier(f(t),w),[-30 30 -.5 2])
```

MATLAB gave the result as follows:

```
ans =

- (cos(w)*1i - sin(w))/w + (sin(w) + cos(w)*1i)/w
```

MATLAB's result is true, since:

$$f(t) = u(t+1) - u(t-1)$$

$$Fourier(f(t)) = F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt = \int_{-1}^{1} 1 \times e^{-j\omega t}dt = \tfrac{-1}{j\omega}(e^{-j\omega} - e^{j\omega})$$

$$let\ a = -\omega \rightarrow \tfrac{-1}{j\omega}(e^{ja} - e^{-ja}) = \tfrac{-1}{j\omega}(2j)sin(-\omega) = \tfrac{2}{\omega}sin(\omega)$$

$$= 2sinc\omega$$

$$MATLAB's\ Answer\ = -\tfrac{(jcos\omega - sin\omega)}{\omega} + \tfrac{(jcos\omega + sin\omega)}{\omega} = \tfrac{-jcos\omega + sin\omega + jcos\omega + sin\omega}{\omega} = \tfrac{2sin\omega}{\omega} = 2sinc\omega$$

The graph of the step function along with the plot of the Fourier Function are as follows:

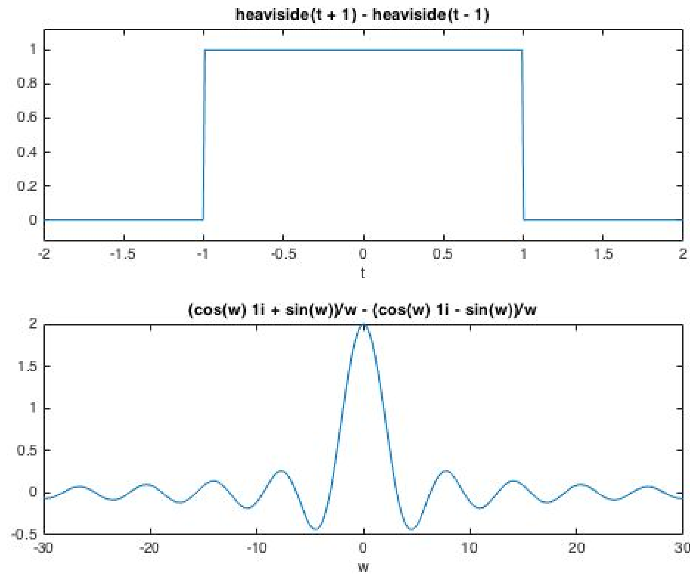

***Figure 1**: A step function, and its Fourier Transform plot*

As a final task, I computed the magnitude and phase of a Low Pass Filter in MATLAB. Knowing the impulse response of a passive low pass filter circuit, I used MATLAB's Fourier Transforming function to transform the function of time, into a function of frequency. Then, using MATLAB's `abs` and `angle` functions, I found the magnitude and phase of the function of frequency to plot the information. The plots are known as Bode plots for the circuit, and they tell us the cutoff frequencies and power outputs for the filtering circuit.

```
% declaring functions
syms w t
frequency = 66300; % 66.3 KHz
h(t) = (1/frequency)*exp(-t/frequency)*heaviside(t);
H(w) = fourier(h(t), w);

% making domain vectors
x = logspace(3, 7, 100);
substitution = subs(H(w), x);
magnitude = abs(substitution);
phase = angle(substitution);

% plotting
figure
subplot(2,1,1)
loglog(x, magnitude)
title('Magnitude of The Low Pass Filter')
xlabel('frequency (w)')
ylabel('|H(w)|')
subplot(2, 1, 2)
```

```
loglog(x, phase)
title('Phase of The Low Pass Filter')
xlabel('Radians')
ylabel('Angle of H(w)')
```

With the plots in Figure 2, it is evident at what frequencies going into the circuit allow for the most power coming out of the circuit. At around 100 KHz, the power coming out is only about half of the power coming in.

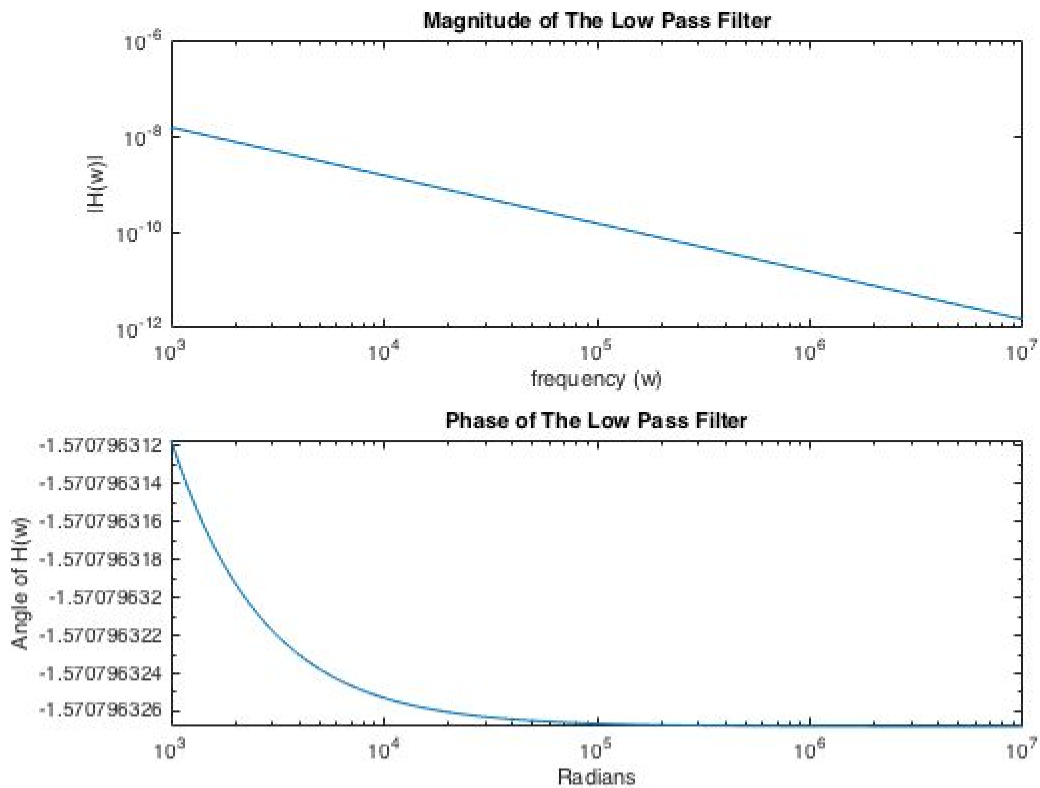

***Figure 2**: The magnitude and phase plots of a passive low pass filter*