Numerical Fourier Transforms in MATLAB (R2008b)

G. Elijah Kemp

Andy Krygier

Christopher Willis

Abstract

The following discussion describes the types of steps one would take when performing a Fourier analysis on a given discrete data set. It involves some elementary details concerning importing data, performing the forward and reverse Fourier transforms and some helpful advice for those who are new to the topic and/or MATLAB. An example script (samplecode.m) is also attached going through some of these steps so that the reader can get a better understanding of taking numerical Fourier transforms in MATLAB.

NOTE: Any of the following functions described can be seen in greater detail by simply typing "help" followed by the desired function in the command window, like follows...

```
>> help load
```

Load Data

To load tab-delimited data into Matlab (such as the wave01.txt document provided for Problem Set 2), Matlab provides a function called “load”. This imports the data into arrays which can then be called later. For example, consider the following (where the text file is included in the current working directory) ...

```
>> data = load(‘wave01.txt’)
>> t = data(:,1);
>> E_t = data(:,2);
```

So, "data" is the matrix name of the imported data and using the form data(m,n) picks values in the m by n matrix (Note: using a ":" as one of the values for m or n means to use the entire column, specified by the n value). In this example the first column is time, the second column the electric field.

Forward Fourier Transform

To do a Fourier transform of data, Matlab has a fast discrete Fourier transform to perform the forward transform from time to frequency space. It can be called using "fft(Y)" where Y is the desired array of data. Note that this function will only calculate the forward transform of the y-values of the data and not the x-component, which has to be specified by the user (Refer to class notes to calculate the appropriate frequency domain from the specified time domain). Consider the following example using the above electric field from the imported data…

```
>> E_w = fft(E_t);
```

However, Matlab handles the Fourier transform slightly differently by arranging the 0-frequency component first in the output array. So to plot the data, one must use the "fftshift" function to rearrange the data into the conventional order (often easier to do all in one step as shown).

```
>> E_w = fftshift(fft(E_t));
```

Another option with the "fft" function is to use the form…

```
>> E_w = fftshift(fft(E_t,N));
```

where N is the forward transform for N points (if length(E_t)<N then the array is padded with zeroes or if length(E_t)>N then the array is truncated.

Displaying real and imaginary parts, magnitude and phase.

Let E_w be the array of complex numbers resulting from the Fourier transform. To take the real part of this array, and assign the values to the array E_w_real, use the MATLAB command …

```
>>E_w_real = real(E_w);
```

where the semicolon indicates that we do not want these data parts output to the command window. Similarly, to take the imaginary part, and assign to a workspace variable called E_w_imag, use the command …

```
>> E_w_imag = imag(E_w);
```

Many times, it is more informative to see your transformed data in terms of a magnitude and phase, rather than a real part and imaginary part. Using the abs(x) commmand in Matlab will produce the modulus of the complex data. For example …

```
>> E_mag =abs(E_w)
```

will produce an array of the same length as E_w (same number of data points) containing the modulus of each of the points in the E_w array to get the phase, use the angle(x) command. Here it would be useful to "unwrap" the data, so that angles larger than pi (radians) do not get "wrapped" around to -pi. Basically, this takes the data and adds or subtracts multiples of 2*pi appropriately so that the phase comes out without discontinuities. Using our E_w data set from above, we would generate the phase using the command

```
>> unwrap(angle(E_w))
```

where the unwrap() function unwraps the phase as described above.

Doing the reverse transform.

To do a reverse transform, apply the function ifft() to the data. Note that if you used the command …

```
>> fftshift(fft(x))
```

to get the Fourier transform with zero frequency data in the center of the array, you will also need to use the function ifftshift(x) to "undo" the shift from fftshift. This ifftshift(x) function should always be applied BEFORE taking the inverse transform. So, one should always use the form …

```
>> ifft(ifftshift(x))
```

Advice

- FFT requires 2^N data points, Matlab's routine will automagically account for this, but it could create errors. Therefore, it is ideal if the length of your data is already a power of 2.

- When plotting complex data, plot just the real part or just the imaginary part. (Matlab's plot function ignores complex values)

- Any FFT routine assumes the signal is periodic where the period is the length of the data. This is because a real Fourier transform integrates over infinity, whereas the FFT only operates on a finite slice of time. Make sure you look at a large enough time span.

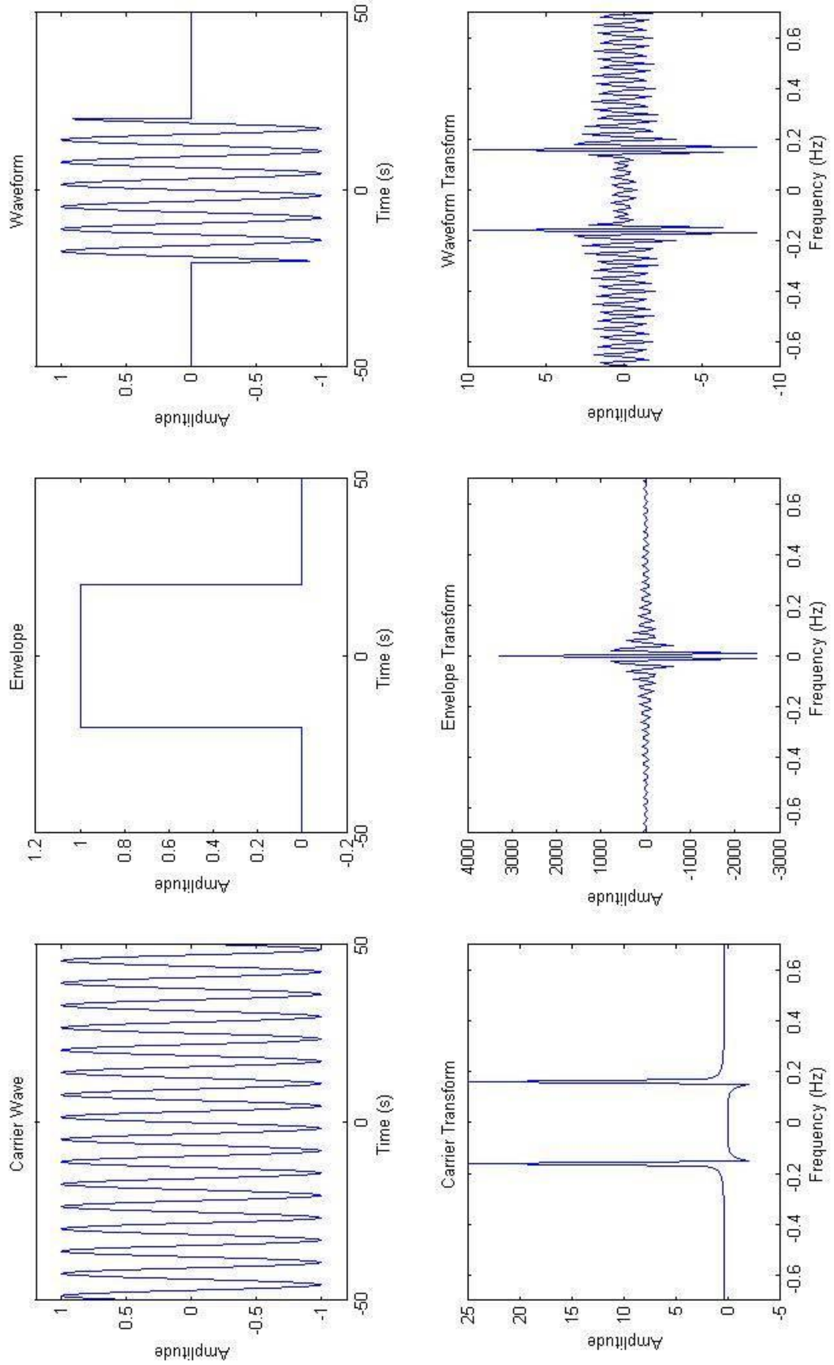
- If something doesn't look quite right, it's probably a sign error (or other simple mistake).

- Use your intuition to check the physicality of the results.

Sample Code and Plots

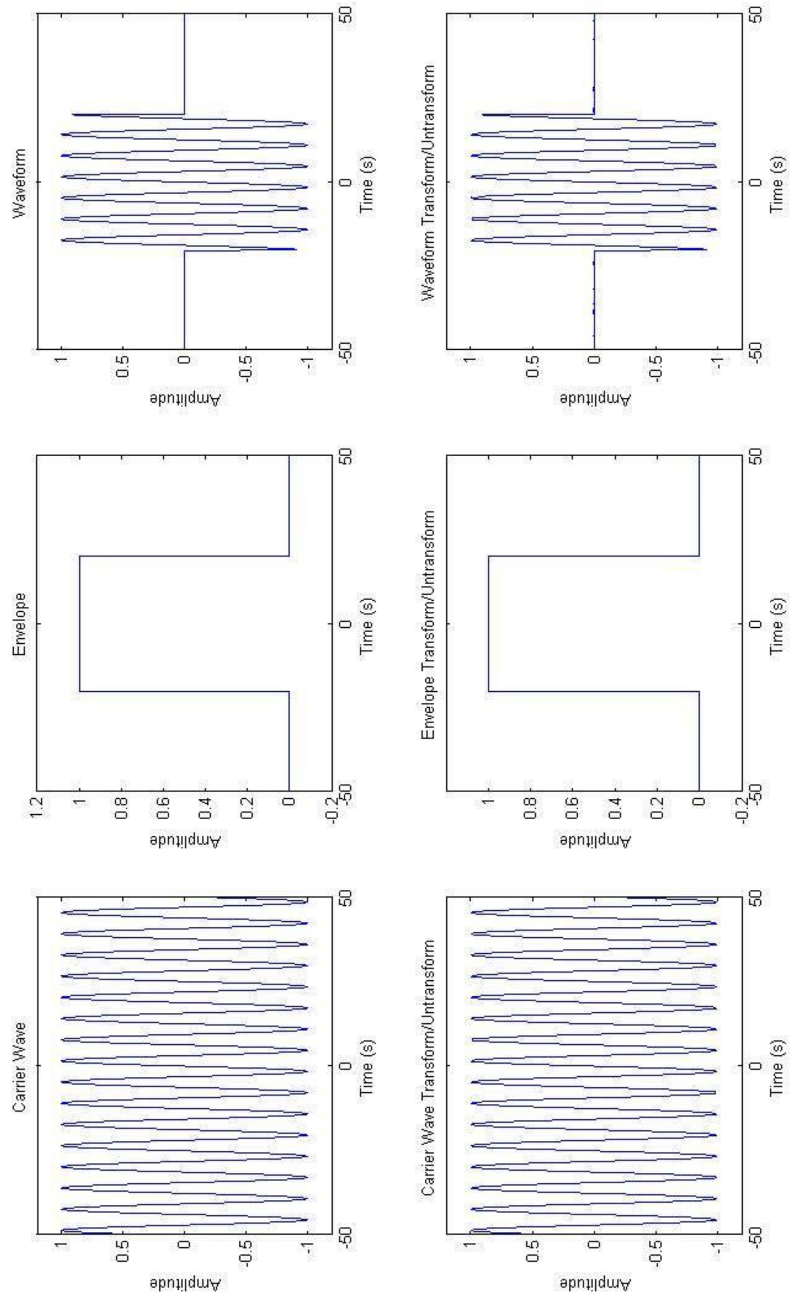
A sample Matlab m-file (samplecode.m) is included with this document. This file includes an example of the code discussed above, including the Fourier transform of a square pulse with some carrier frequency. In addition, this code verifies that taking the inverse transform of the transformed data will reproduce the original waveform.

Sample Plots: Square wave with Carrier



Sample Plots: Result of transform then inverse transform

This is just to verify that we get the same thing back when we take the inverse transform.


Waveform
Envelope
Carrier Wave
Waveform Transform/Untransform
Envelope Transform/Untransform
Carrier Wave Transform/Untransform
Amplitude
Time (s)