ITCE272 Lab 3
Introduction to Fourier transform

Prepared by Dr. Alauddin Al-Omary

Part One: Fourier series

The Fourier series of the Square Wave
Let the function sq(t) be denoted by

$$\mathrm{sq}(t) = \begin{cases} 1 & m \le t < m + 1/2 \\ -1 & m + 1/2 \le t < m + 1 \end{cases}$$

The Fourier series associated with this function is

$$\mathrm{sq}(t) = b_0 + \sum_{k=1}^{\infty} b_k \cos(2\pi kt) + \sum_{k=1}^{\infty} a_k \sin(2\pi kt).$$

The Fourier coefficients of this 1-periodic function are given by

$$\begin{aligned} b_0 &= \int_{-1/2}^{1/2} \mathrm{sq}(t)\, dt \\ b_k &= 2\int_{-1/2}^{1/2} \cos(2\pi kt)\mathrm{sq}(t)\, dt, \qquad k > 0 \\ a_k &= 2\int_{-1/2}^{1/2} \sin(2\pi kt)\mathrm{sq}(t)\, dt, \qquad k \ge 0. \end{aligned}$$

Because sq(t) is odd, it is clear that $b_k = 0$ for all $k \ge 0$. From symmetry it is clear that:

$$a_k = 4\int_0^{1/2} \sin(2\pi kt)\, dt = 4\left.\frac{-\cos(2\pi kt)}{2\pi k}\right|_{t=0}^{t=1/2} = \begin{cases} \frac{4}{\pi k} & k \in \text{odd} \\ 0 & k \in \text{even} \end{cases}$$

Thus, we find that the Fourier series associated with sq(t) is

$$\sum_{k=0}^{\infty} \frac{4\sin(2\pi(2k+1)t)}{\pi(2k+1)}.$$

We would now like to examine the extent to which this series truly represents the square wave.

A Quick Check

Before proceeding to analyze the series we have found, it behooves us to check that the calculations were performed correctly. For a 1-periodic function like sq(t), Parseval's equation states that

$$\int_{-1/2}^{1/2} \mathrm{sq}^2(t)\,dt = b_0^2 + \frac{1}{2}\sum_{k=1}^{\infty}(a_k^2 + b_k^2).$$

In our case this means that

$$1 = \frac{1}{2}\sum_{k=0}^{\infty}\left(\frac{4}{\pi(2k+1)}\right)^2 = \frac{8}{\pi^2}\sum_{k=0}^{\infty}\frac{1}{(2k+1)^2}.$$

We can check that this sum is correct using MATLAB. One way to perform a quick check is to give MATLAB the commands

```
L = [1:2:10001];
(8/pi^2) * sum(1./L.^2)
```

MATLAB responds with

```
ans =
1.0000
```

Which is an indication that we may have performed all of the calculations correctly.

A second way to have MATLAB check this computation is to give MATLAB the commands

```
syms k
(sym('8')/sym('pi')^2) * symsum(1/(2*k+1)^2,k,0,Inf)
```

MATLAB responds to these commands with

```
ans =
1
```

Indicating that the series does, indeed, sum to 1.

Seeing" the Sum

It is not difficult to have MATLAB calculate the Fourier series and display its values. Conisder the following code:

```
t = [-500:500] * 0.001;
total = zeros(size(t));
for k = 1 : 2 : 101
total = total + (4/pi) * sin(2*pi*k*t) / k;
end
plot(t,total)
```

This code denotes a "time" vector, t, with 1001 elements. It then defines a vector to hold the sum, total, and proceeds to sum the first 51 terms in the Fourier series. The code causes MATLAB to produce the plot shown in

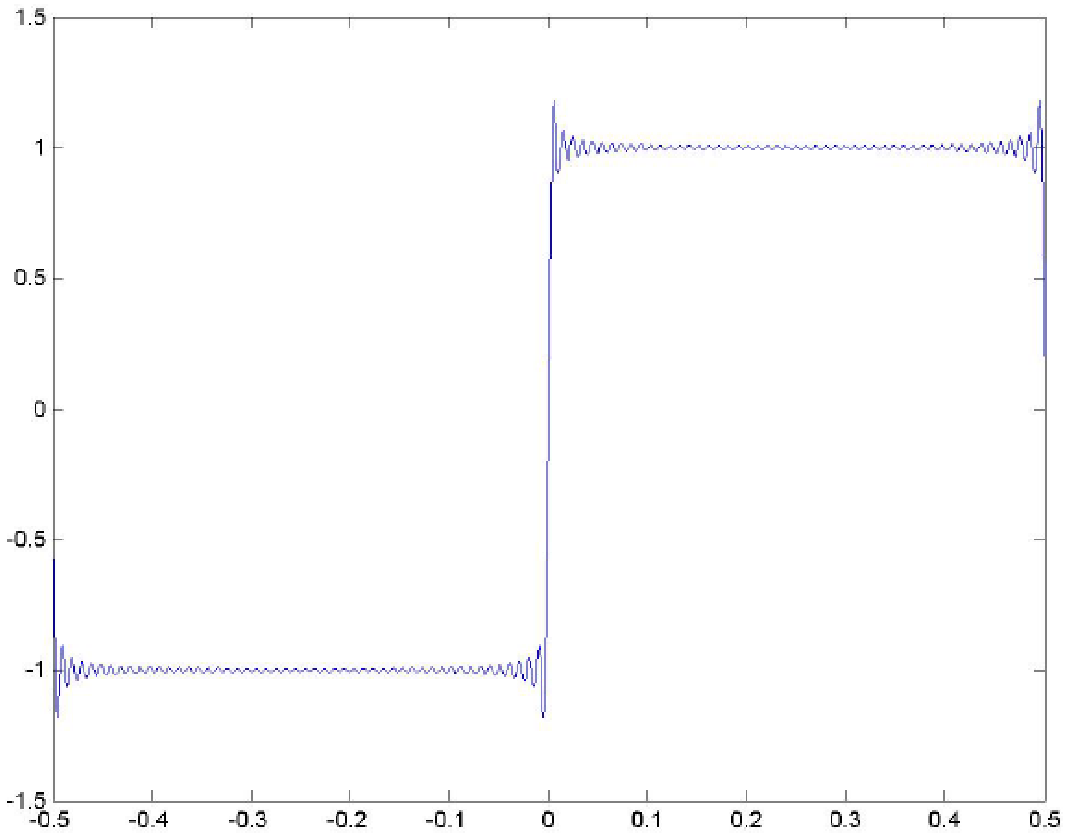


Figure 5.1: Summing the Fourier series. The "ringing" associated with Gibb's Phenomenon is clearly visible.

The Experiment

Let

saw(t) = t

for -1/2 < t ≤1/2 and continue saw(t) periodically outside of this region.

1. Calculate the Fourier coefficients associated with saw(t).
2. Check Parseval's equation for saw(t) both numerically and symbolically.
3. Sum the first 3 terms in the Fourier series and plot the Fourier series as a function of time.
4. Sum the first 10 terms in the Fourier series and plot the Fourier series as a function of time.
5. Sum the first 50 terms in the Fourier series and plot the Fourier series as a function of time.

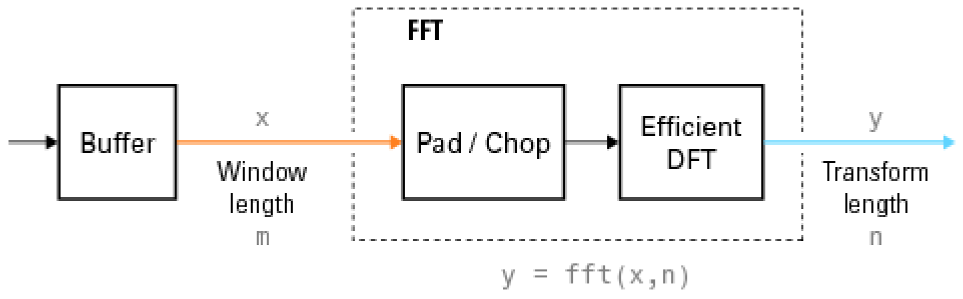
When summing the Fourier series, make sure that the time samples you take are sufficiently closely spaced that you clearly see Gibb's phenomenon.

Part 2: DFT and FFT

Introduction

DFTs with a million points are common in many applications. Modern signal and image processing applications would be impossible without an efficient method for computing the DFT.

Direct application of the definition of the DFT (see Discrete Fourier Transform (DFT)) to a data vector of length *n*requires *n* multiplications and *n* additions—a total of $2n^2$ floating-point operations. This does not include the generation of the powers of the complex *n*th root of unity *ω*. To compute a million-point DFT, a computer capable of doing one multiplication and addition every microsecond requires a million seconds, or about 11.5 days.
Fast Fourier Transform (FFT) algorithms have computational complexity $O(n \log n)$ instead of $O(n^2)$. If *n* is a power of 2, a one-dimensional FFT of length *n* requires less than $3n \log_2 n$ floating-point operations (times a proportionality constant). For *n* = 220, that is a factor of almost 35,000 faster than $2n^2$.
The MATLAB® functions `fft`, `fft2`, and `fftn` (and their inverses `ifft`, `ifft2`, and `ifftn`, respectively) all use fast Fourier transform algorithms to compute the DFT.
When using FFT algorithms, a distinction is made between the *window length* and the *transform length*. The window length is the length of the input data vector. It is determined by, for example, the size of an external buffer. The transform length is the length of the output, the computed DFT. An FFT algorithm pads or chops the input to achieve the desired transform length. The following figure illustrates the two lengths.



The execution time of an FFT algorithm depends on the transform length. It is fastest when the transform length is a power of two, and almost as fast when the transform length has only small prime factors. It is typically slower for transform lengths that are prime or have large prime factors. Time differences, however, are reduced to insignificance by modern FFT algorithms such as those used in MATLAB. Adjusting the transform length for efficiency is usually unnecessary in practice.

The FFT in One Dimension

Introduction

The MATLAB `fft` function returns the DFT `y` of an input vector `x` using a fast Fourier transform algorithm:

```
y = fft(x);
```

In this call to `fft`, the window length `m = length(x)` and the transform length `n = length(y)` are the same.

The transform length is specified by an optional second argument:

```
y = fft(x,n);
```

In this call to `fft`, the transform length is `n`. If the length of `x` is less than `n`, `x` is padded with trailing zeros to increase its length to `n` before computing the DFT. If the length of `x` is greater than `n`, only the first `n` elements of `x` are used to compute the transform.

Basic Spectral Analysis

The FFT allows you to efficiently estimate component frequencies in data from a discrete set of values sampled at a fixed rate. Relevant quantities in a spectral analysis are listed in the following table. For space-based data, replace references to time with references to space.

Quantity	**Description**
`x`	Sampled data
`m = length(x)`	Window length (number of samples)
`fs`	Samples/unit time
`dt = 1/fs`	Time increment per sample
`t = (0:m-1)/fs`	Time range for data
`y = fft(x,n)`	Discrete Fourier transform (DFT)
`abs(y)`	Amplitude of the DFT
`(abs(y).^2)/n`	Power of the DFT
`fs/n`	Frequency increment
`f = (0:n-1)*(fs/n)`	Frequency range
`fs/2`	Nyquist frequency

Consider the following data `x` with two component frequencies of differing amplitude and phase buried in noise.

```
fs = 100;                                % Sample frequency (Hz)
t = 0:1/fs:10-1/fs;                      % 10 sec sample
x = (1.3)*sin(2*pi*15*t) ...           % 15 Hz component
  + (1.7)*sin(2*pi*40*(t-2)) ...        % 40 Hz component
  + 2.5*gallery('normaldata',size(t),4); % Gaussian noise;
```

Use `fft` to compute the DFT `y` and its power.

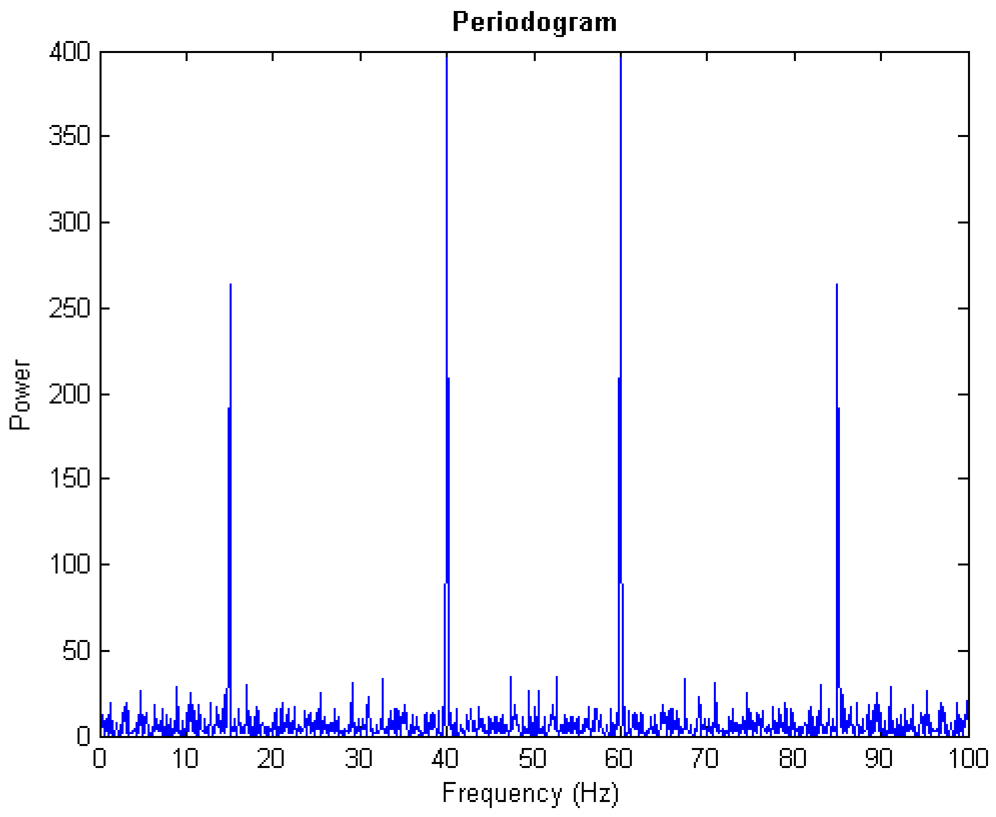
```
m = length(x);          % Window length
n = pow2(nextpow2(m));  % Transform length
y = fft(x,n);           % DFT
f = (0:n-1)*(fs/n);     % Frequency range
power = y.*conj(y)/n;   % Power of the DFT
```

`nextpow2` finds the exponent of the next power of two greater than or equal to the window length (`ceil(log2(m))`) and `pow2` computes the power. Using a power of two for the transform length optimizes the FFT algorithm, though in practice there is usually little difference in execution time from using `n = m`.
To visualize the DFT, plots of `abs(y)`, `abs(y).^2`, and `log(abs(y))` are all common. A plot of power versus frequency is called a *periodogram*.

```
plot(f,power)

xlabel('Frequency (Hz)')
ylabel('Power')
```

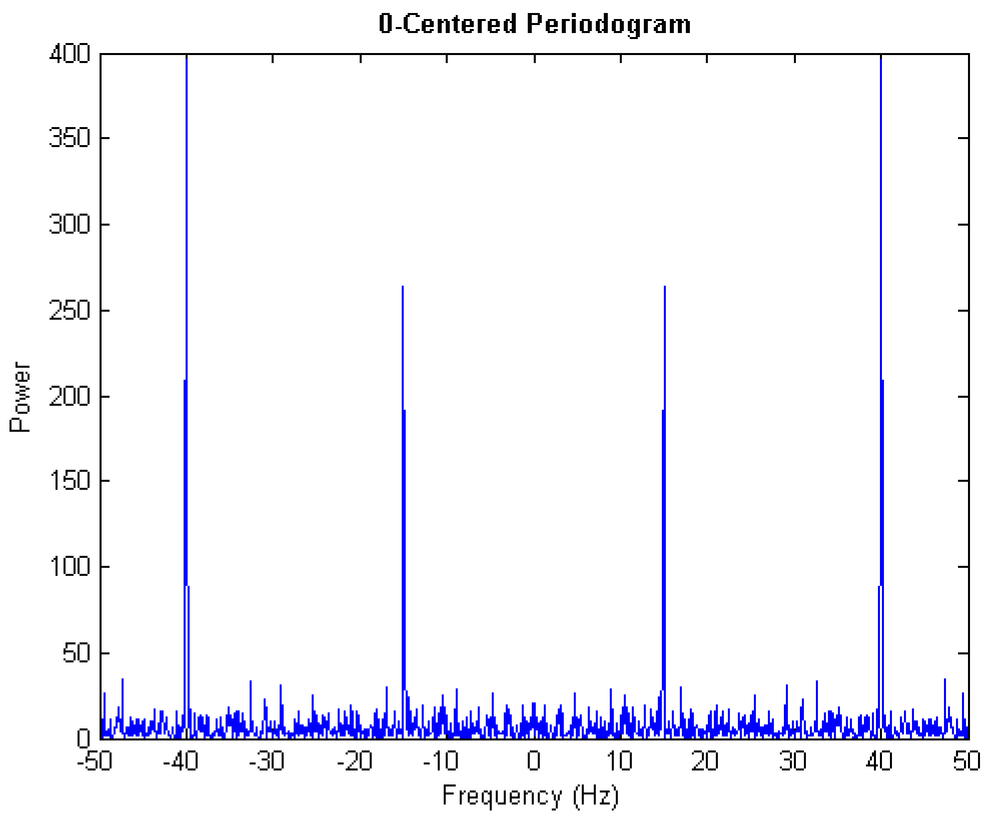
```
title('{\bf Periodogram}')
```



The first half of the frequency range (from 0 to the Nyquist frequency `fs/2`) is sufficient to identify the component frequencies in the data, since the second half is just a reflection of the first half.
In many applications it is traditional to center the periodogram at 0. The `fftshift` function rearranges the output from `fft` with a circular shift to produce a 0-centered periodogram.

```
y0 = fftshift(y);          % Rearrange y values
f0 = (-n/2:n/2-1)*(fs/n);  % 0-centered frequency range
power0 = y0.*conj(y0)/n;   % 0-centered power

plot(f0,power0)
xlabel('Frequency (Hz)')
ylabel('Power')
title('{\bf 0-Centered Periodogram}')
```



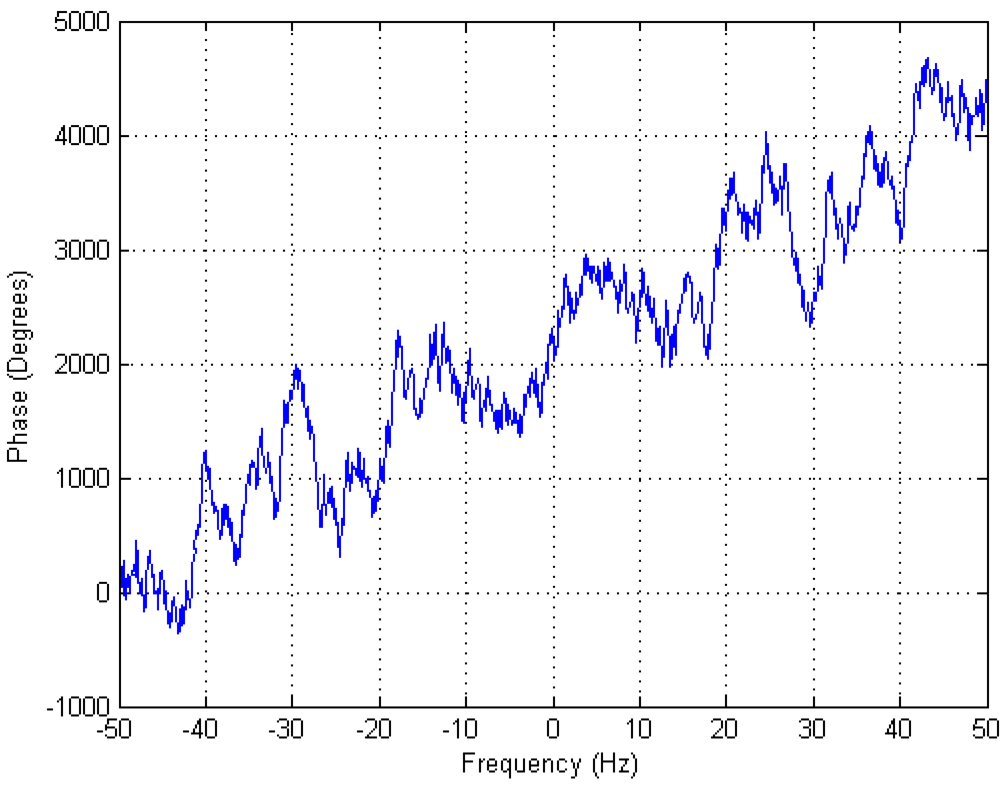
The rearrangement makes use of the periodicity in the definition of the DFT.

Use the MATLAB `angle` and `unwrap` functions to create a phase plot of the DFT.

```
phase = unwrap(angle(y0));


plot(f0,phase*180/pi)

xlabel('Frequency (Hz)')
ylabel('Phase (Degrees)')
grid on
```



Component frequencies are mostly hidden by the randomness in phase at adjacent values. The upward trend in the plot is due to the `unwrap` function, which in this case adds 2π to the phase more often than it subtracts it.

Report

1- Consider the following data `x` with two component frequencies of differing amplitude and phase buried in noise with the following parameters:
   - a- Sampling frequency = 200hz
   - b- 20 second samples
   - c- 20 and 40 hz components
   - d- Gaussian noise

2- Use FFT to compute DFT y and its power

3- Plot the frequency and power

4- Rearrange the output from fft to produce 0-centered periodogram

5- Create and plot the phase