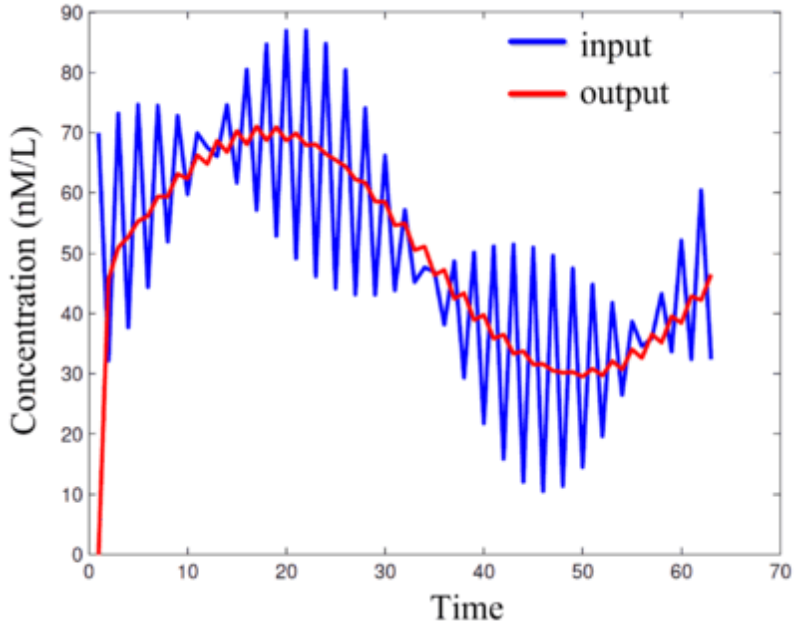
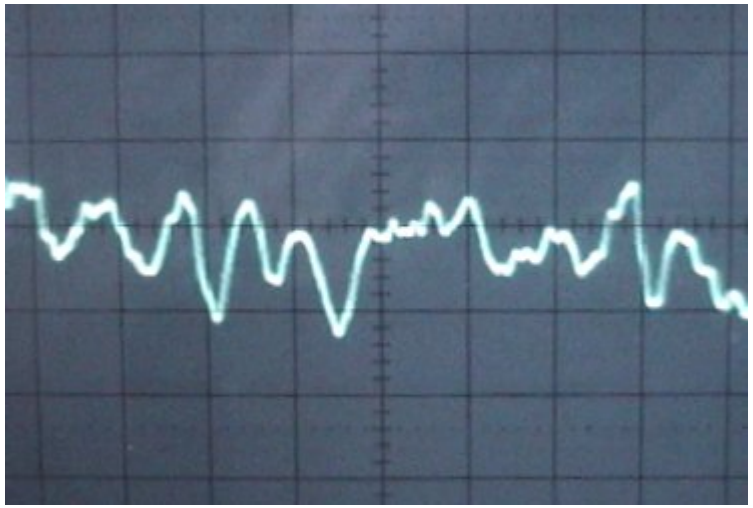
Introduction to Signal Processing and Analysis in R

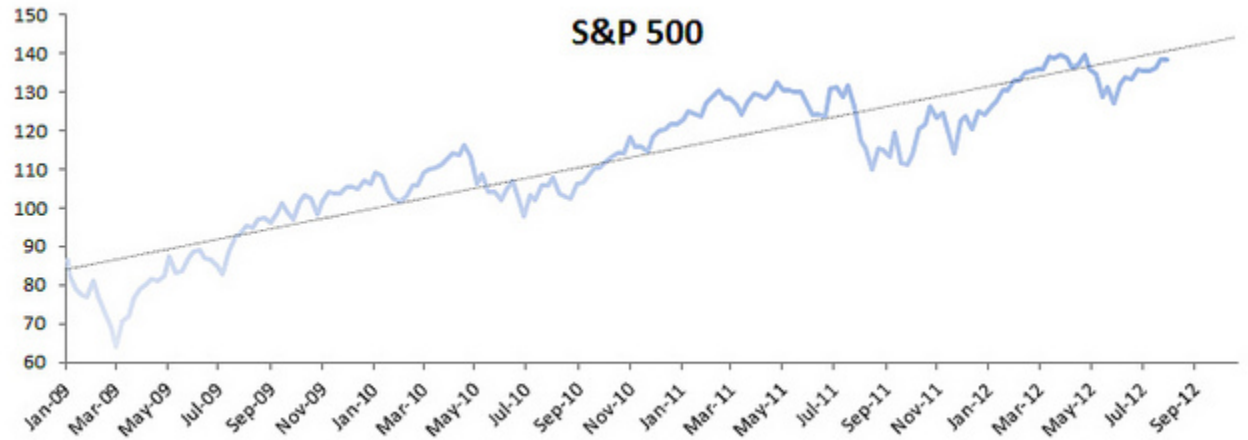
With applications to Time Series Data in Economics and Finance



What are signals?

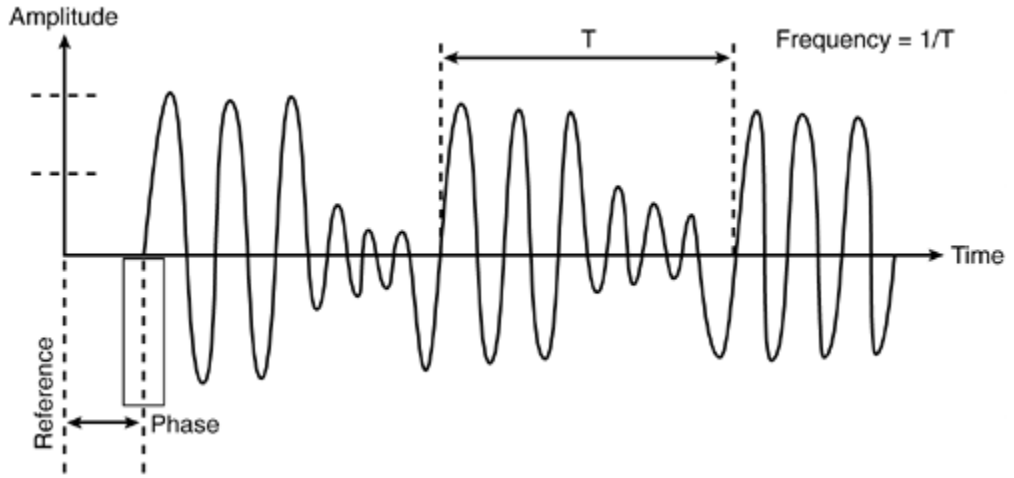
- Signals are sequences of quantities or values that vary in time: voltage in a telephone line, daily stock prices, annual production outputs of a company
- Signals can be **continuous** or **discrete**, in either time or amplitude.
- The signals we most often encounter in Economics, Finance and Business are called **time series** and are usually (but not always) discrete in time and amplitude.





What are signals?

- Some basic properties of simple signals constructed from sine and cosine waves, **amplitude**, **frequency**, **phase**:



- Signals can be continuous or discrete in time and amplitude. For us, real world signals/time series we work with on computers are **always discrete in both time and frequency**.

Why signal processing?

- So say you have certain time series such as stock price, commodity values, interest rates etc. that you are interested in.
- Signal processing techniques allow an analyst to **clean** data or **extract information** from it that may not be apparent.
- Sometimes it is necessary to clean data by **removing unwanted components** such as seasonality and random noise in order to make underlying trends more apparent and or to improve the quality of the signal
- Signal processing is done in R using the powerful **signal** package

Introduction to filters

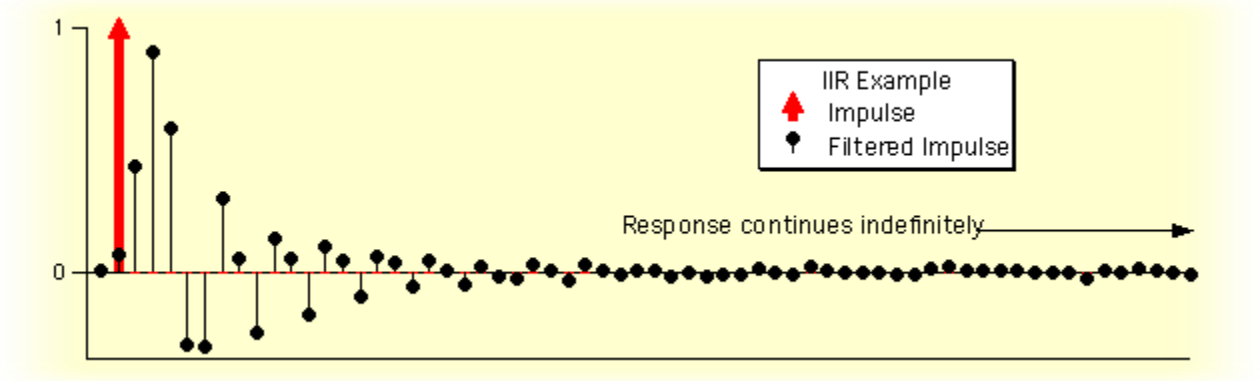
- All signals (periodic or non periodic) are comprised of finite or infinitely many simple sine and cosine waves of varying frequency and amplitude. This is the basis of the power **Fourier Theory** which allows us to look at signals from a frequency domain perspective, this is the analysis part. More on this in a bit.
- **Filters** are mathematical operations that produce an output signal from an input signal. Filters can be used to attenuate or amplify certain frequencies from the input signal.
- Think of a coffee filter: it filters out large solid particles, leaving the liquid coffee as output, similarly signal filters leave out certain components of the signal from the output

Introduction to filters

- Some filter terminology: **linear time-invariant**
  - linear means that the output signal is a linear combination of the filter and input signal. In other words, linear filters satisfy the **scaling** and **superposition** principles. Let $\mathbf{S_1}$ and $\mathbf{S_2}$ be signals, $\boldsymbol{a}$ be a scalar quantity, and $\mathcal{F}$ be a linear filter. Then:
    - scaling: $\mathcal{F}(\boldsymbol{a}\,\mathbf{S_1}) = \boldsymbol{a}\;\mathcal{F}(\mathbf{S_1})$
    - superposition: $\mathcal{F}(\mathbf{S_1} + \mathbf{S_2}) = \mathcal{F}(\mathbf{S_1}) + \mathcal{F}(\mathbf{S_2})$
  - The term time invariant means that whatever time shift is applied to the input to the filter the same shift is applied to the output, and the output is unchanged by the time shift operation in any other way
  - LTI filters are the most common type of filters and have the benefit of being easy to implement and compute as well as having analytic solutions and frequency response functions

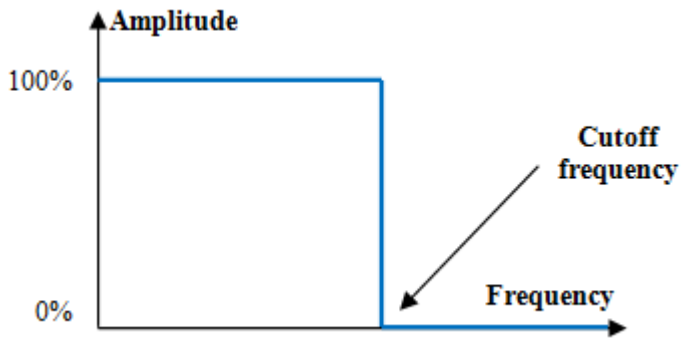
Introduction to filters

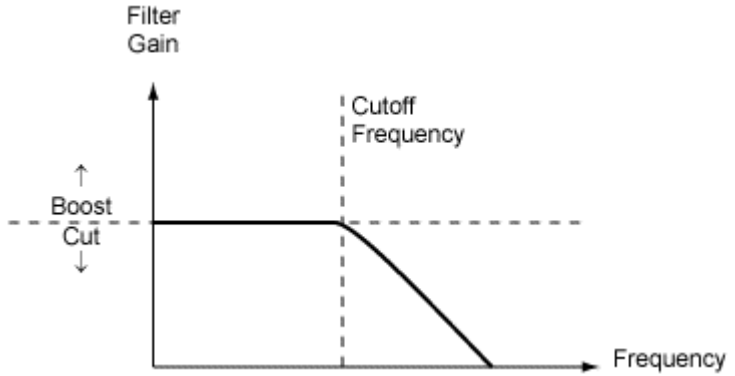
- Two types of discrete time (or digital) filter useful for time series applications: **Infinite impulse response** and **finite impulse response**
- IIR filters: low pass, high pass, bandpass and notch (we will focus on these 4 types)
- FIR filters: ARMA/ARIMA, moving average smoothing



Low pass filters

- The most basic type of IIR filter is called a **low pass filter**. This filter allows frequencies of an input signal to pass without weakening of amplitude (attenuation) up to a certain frequency called the **corner** or **cut-off** frequency
- After this point the filter attenuates frequencies at a certain rate according to design (in technical terms, the amount of attenuation or **roll-off** depends on the number of poles in the filter transfer function)
- But for our purposes we will let R take care of the details for us, and we will specify only a cut-off frequency and a desired roll-off

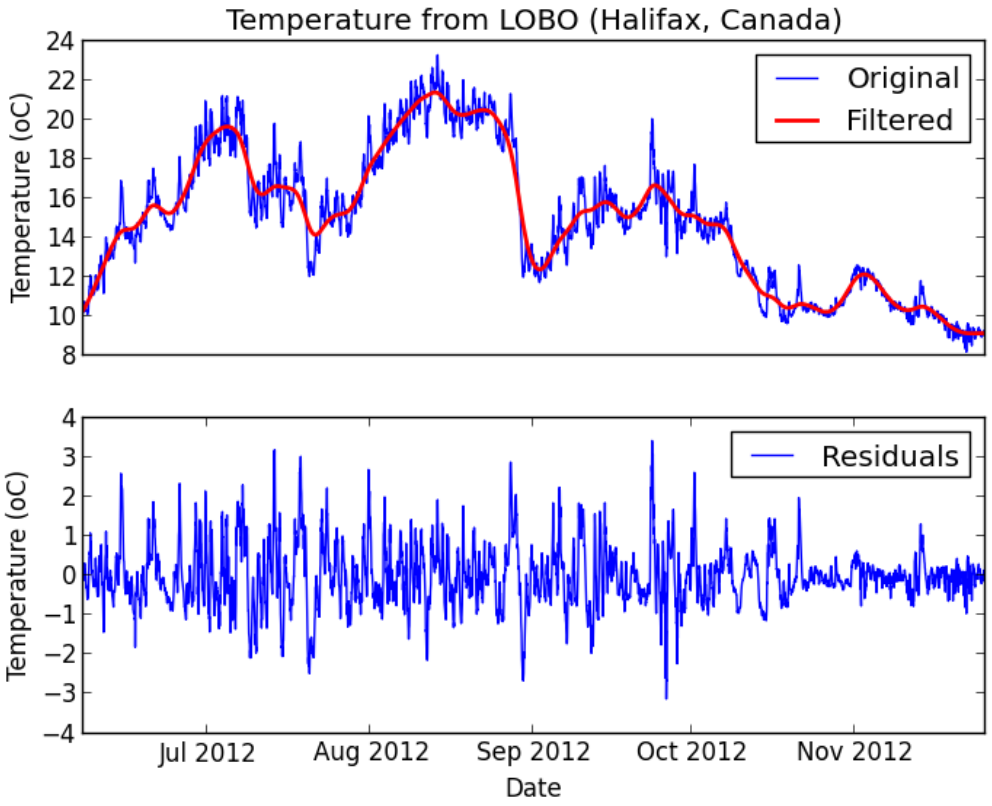




Low pass filters

- LPFs are among the most useful types of IIR filters as they can remove noise and abrupt jumps from any periodic or aperiodic signal and reveal an underlying **trend**, or used to **preprocess and clean** a signal before feeding it to more advanced techniques.
- Before moving on, please install the singal package into R:

```
> install.packages('signal')
```



Low pass filters

- For our purposes will we use a **Butterworth filter** in the low-pass configuration to clean up signals
- The Butterworth filter has a maximally flat passband (highly desirable property) i.e. there is very little variation in the gain/attenuation of frequencies before the cut-off point
- The command in the signal library is

$$\text{butter}(\text{n}, \text{W}, \text{type} = \text{c}(\text{"low"}, \text{"high"}, \text{"stop"}, \text{"pass"})$$

- n is the order of the filter, W is the cut off frequency or frequencies, type is the type of filter

Low pass filters

- Lets a simple example: we can generate a sinewave signal corrupted by some periodic noise

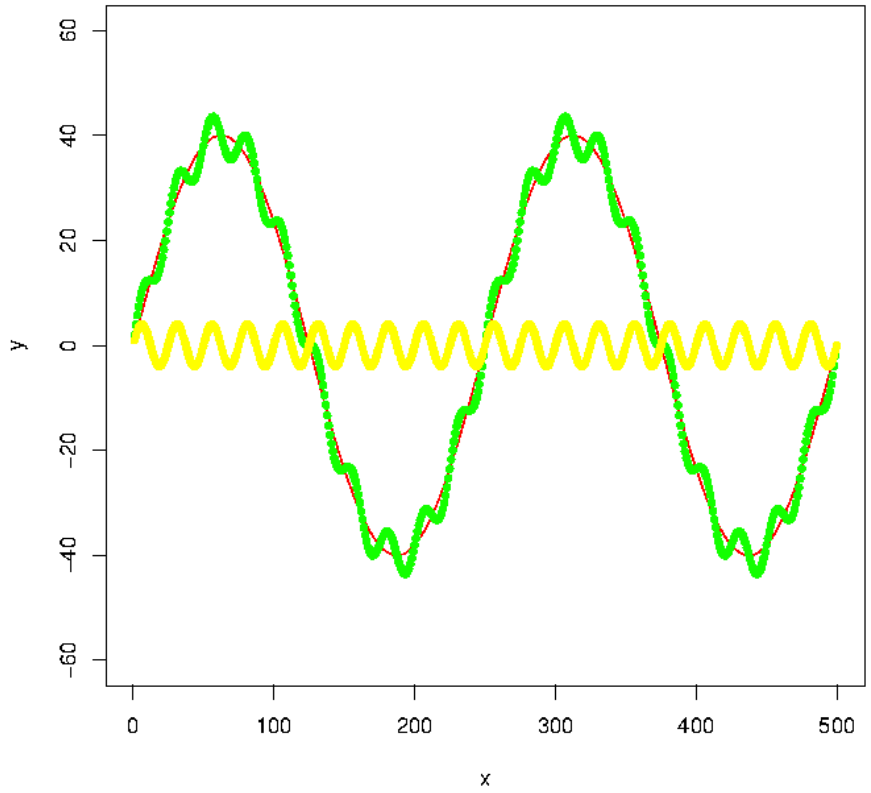
```
number_of_cycles = 2
max_y = 40

x = 1:500
a = number_of_cycles * 2*pi/length(x)

y = max_y * sin(x*a)
noise1 = max_y * 1/10 * sin(x*a*10)

plot(x, y, type="l", col="red", ylim=range(-1.5*max_y,1.5*max_y,5))
points(x, y + noise1, col="green", pch=20)
points(x, noise1, col="yellow", pch=20)
```

Note:
Period of signal = 250
Period of noise = 25



Low pass filters

- Now we can construct Buttworth filters to filter out the noise (yellow in previous figure) from the composite signal (green)

```
library(signal)
bf <- butter(2, 1/50, type="low")
b1 <- filtfilt(bf, y+noise1)
points(x, b1, col="red", pch=20)
```

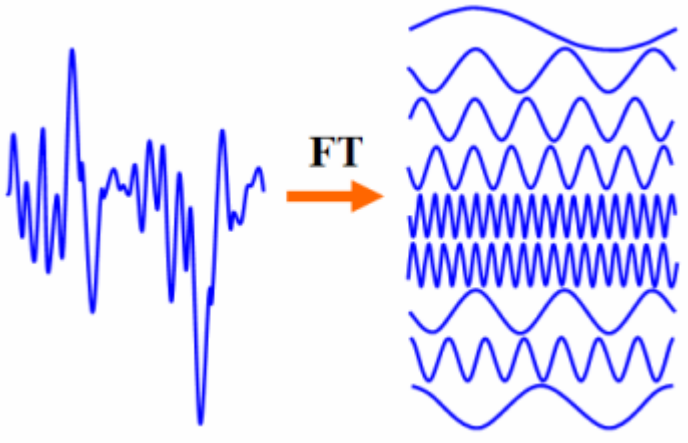
Here the filter corner frequency = 1/50 Hz < 1/25 Hz, which means that the periodic noise will be filtered out

- Experiment with different noise frequencies, filter orders and different types of noise to see what results you get! It's usually a matter of intuition to pick the best filter and cut-off frequencies

Other filters

- The butter function in the R signal package can be used to build other filters, namely highpass, notch (or bandstop) and bandpass filters.
- These filters have applications in signal processing for scientific, social and economic data
- Check out https://cran.r-project.org/web/packages/signal/signal.pdf for more information on the Butterworth filter function and other signal processing functions
- An excellent online textbook for simple to understand theory and applied signal processing if you want to learn more: http://www.ece.rutgers.edu/~orfanidi/intro2sp/

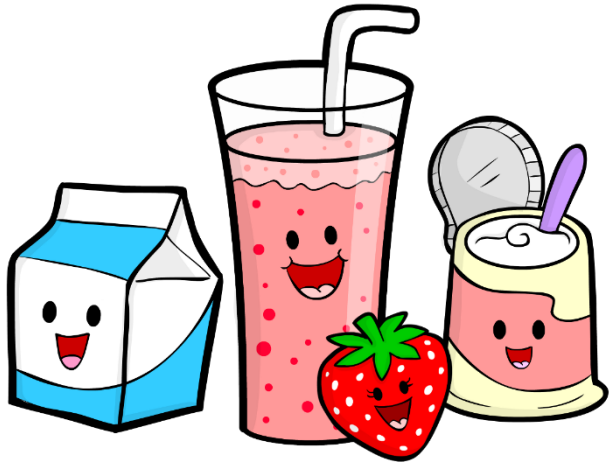
Introduction to Fourier Transform and Applications



Introduction to Fourier Transform

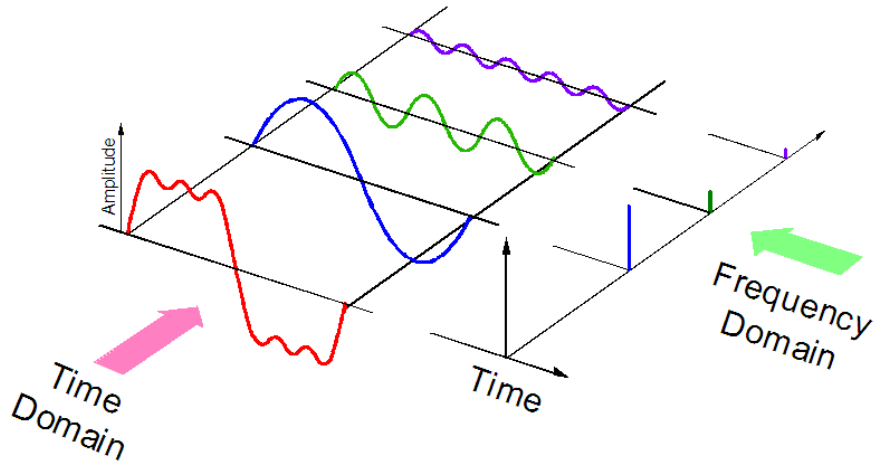
- **What does the Fourier Transform do?** Given a smoothie, it finds the recipe.
- **How?** Run the smoothie through filters to extract each ingredient.
- **Why?** Recipes are easier to analyze, compare, and modify than the smoothie itself.
- **How do we get the smoothie back?** Blend the ingredients.

http://betterexplained.com/articles/an-interactive-guide-to-the-fourier-transform/



Introduction to Fourier Transform

- Remember we said earlier that all signals, periodic or non-periodic can be represented by a (finite or infinite) combination of simple sines and cosines at different frequencies and amplitudes.
- This powerful fact allows us to look at signals in the **frequency domain** where we can find out what frequencies (of simple sine and cosine waves) that make the signal have the highest amplitude



Introduction to Fourier Transform

- Since we are interested in applications, we will only take look briefly at the **Discrete Fourier Transform (DFT)**, which is the **only useful Fourier Transform for time series data** invariably represented by discrete values on a computer, and discuss some aspects of the Fast Fourier Transform (FFT) algorithm implementation.
- Then we will take a look at how easy R makes Fourier Analysis of time series data and some interesting applications with respect to Economic/Financial time series.

$$DFT(FFT):$$

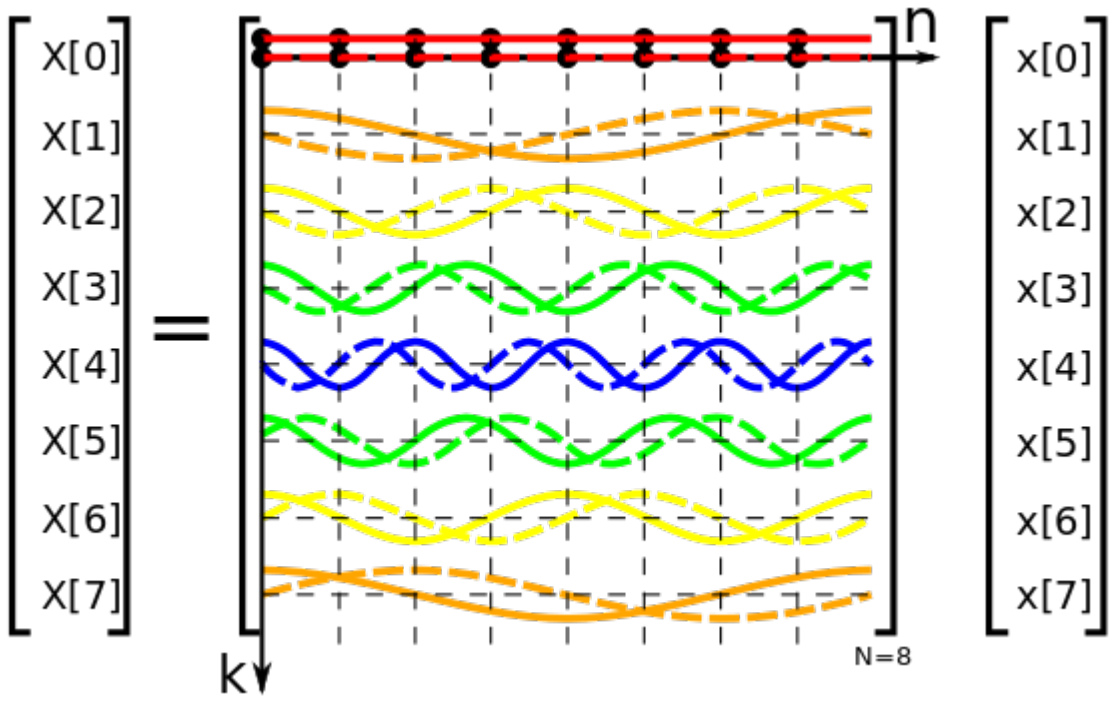
$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-j\left(\frac{2\pi}{N}\right)nk} \quad (k = 0,1,\ldots,N-1)$$

$$IDFT(IFFT):$$

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) \cdot e^{j\left(\frac{2\pi}{N}\right)nk} \quad (n = 0,1,\ldots,N-1)$$

The Discrete Fourier Transform

- The most intuitive way to represent the Discrete Transform is as a matrix multiplication. The DFT output vector (called **coefficients**) of an input vector containing a discrete time signal is produced by multiplying that input vector by a **DFT Matrix**



The Discrete Fourier Transform

- The DFT multiplies the input signal vector (a column vector) to the rows of the DFT matrix, which are the colorful sine and cosine waves in each row in the previous illustration to generate a **coefficient** on the output side
- This step is called **correlation** in which we are trying to find out how strongly a **template** signal with a certain frequency and phase exists in the input signal
- The repeats of the sine and cosine waves down the rows represent template signals of different phases
- In general the output coefficients are complex numbers with both amplitude and phase. With time series, phase is not important compared to amplitude.

The Discrete Fourier Transform

- Some important facts about the DFT:
- The DFT can only resolve a static, unchanging component in the signal (with **zero frequency**) and frequencies up to **N/2**, where N is the number of values or **samples** in the input signal vector.
- For analysis purposes we can discard coefficients with frequencies past N/2 as they are repeats of previous coefficients due to the periodicity of the complex exponentials used in calculating the DFT
- There is an **inverse DFT** that allows us to convert a complete coefficient vector back a time domain signal. This is useful for forecasting as we will see later.
- From the coefficients we can calculate **amplitudes** and **phases** of the frequencies that exist in the input signal. A plot of the amplitude and phase is called the **spectrum** of the input signal as computed by the DFT.

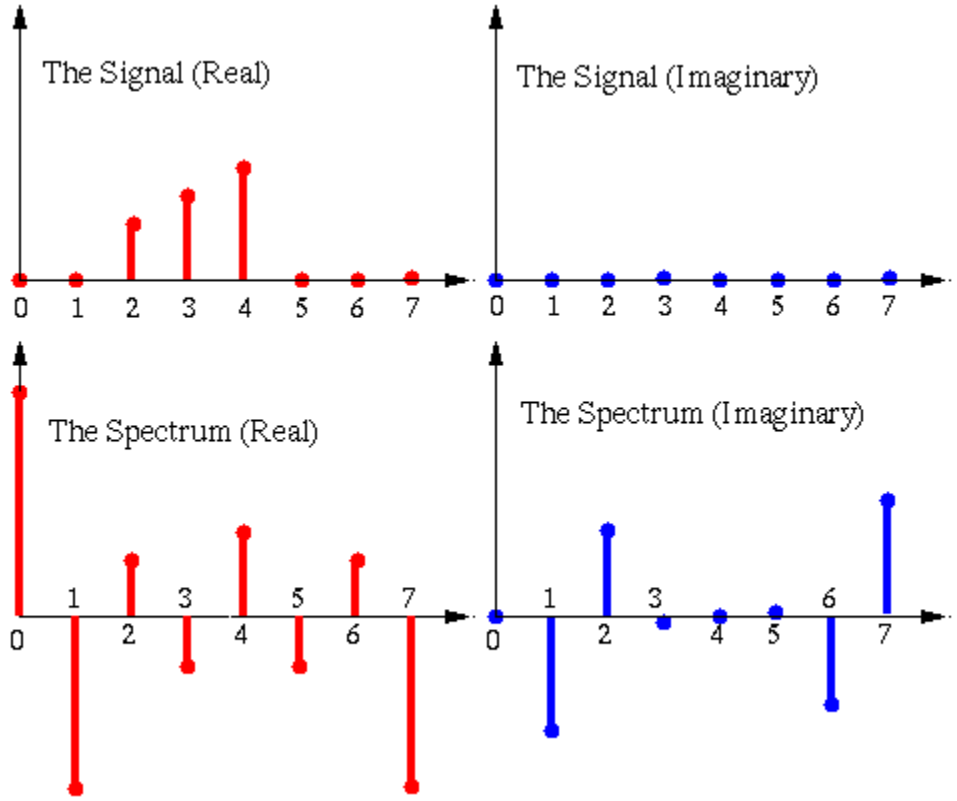
The Discrete Fourier Transform

- An example:
- The normalized amplitude and phase of the of the input signal is given by

$$|X_k|/N = \sqrt{\mathrm{Re}(X_k)^2 + \mathrm{Im}(X_k)^2}/N$$

$$\arg(X_k) = \mathrm{atan2}\left(\mathrm{Im}(X_k), \mathrm{Re}(X_k)\right) = -i \ln\left(\frac{X_k}{|X_k|}\right),$$

- From the raw coefficient output we use the equations above to calculate the spectrum



The Fast Fourier Transform algorithm

- The DFT requires a number of operations that scale quadratically with the size of the input vector. In other words, if our input vector was 8 elements, it would take 8*8 = 64 operations to calculate the DFT output. For 10 million elements (with digital audio for example) it would take 10 million squared operations.
- In the late 60's J.W. Cooley and John Tukey invented the a version of the Fast Fourier Transform algorithm than is ubiquitous today.
- This is known as the **Cooley - Tukey FFT** or simply *the Fast Fourier Transform*
- The beauty of the C-T FFT algorithm is that its operations scale on the order of N*log(N), where N is the number of elements in the input vector

The Fast Fourier Transform algorithm

- The C-T FFT algorithm exploits two things:

    1. the linear superposition property of the DFT, allowing ‘sub DFTs’ to be pieced together to create a composite DFT

    2. the ‘divide and conquer’ technique in computer science, which breaks problems down into bits, solves the bits and patches their solutions back together

- Combining these two operations allows the C-T FFT algorithm to break down the input through divide and conquer and compute sub DFTs, piecing them back together at the end.
- It turns out this method is **exponentially** faster than a regular DFT

Fourier Analysis in R

- **Finally the fun part**: after having some intuition about how DFTs work, let's see how we can use them in R to do time series analysis:
- Taking an FFT is painfully simple, we can reuse the example from earlier.

```
f <- sqrt( Re(fft(y+noise1))^2 + ( Im(fft(y+noise1))^2 )
plot(f[1:250] / f[which.max(f)], type = "h")
```

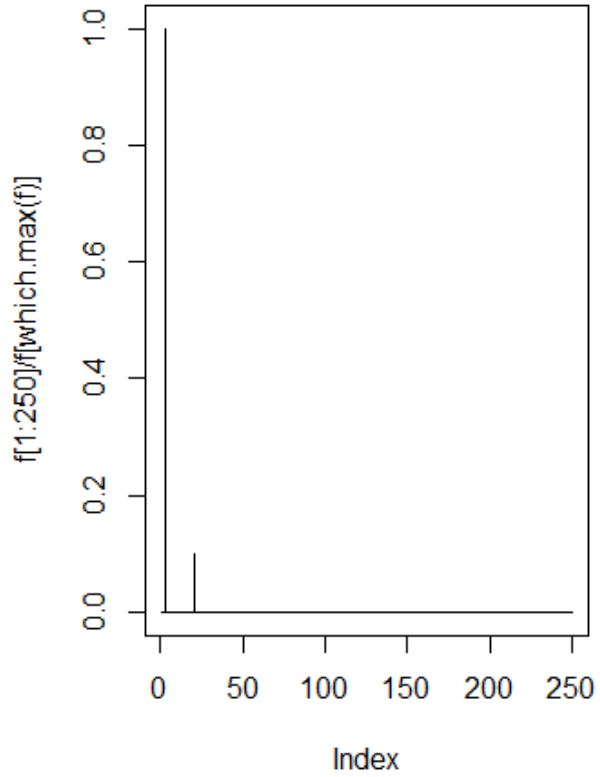
- the first line assigns the output of the FFT to f. We normalize the output of the FFT(which has both real and imaginary parts) to get our amplitude plot in the second line. The second line produces a amplitude normalized plot of the frequency spectrum of the original signal

Fourier Analysis in R

- Now we have a plot that tells us what frequency components are in our original signal.
- Notice the ‘index’ is not well defined. We still need to convert the indices into actual frequencies. The formula for this is:

freq_n = index_n * (sample rate)/(FFT size)

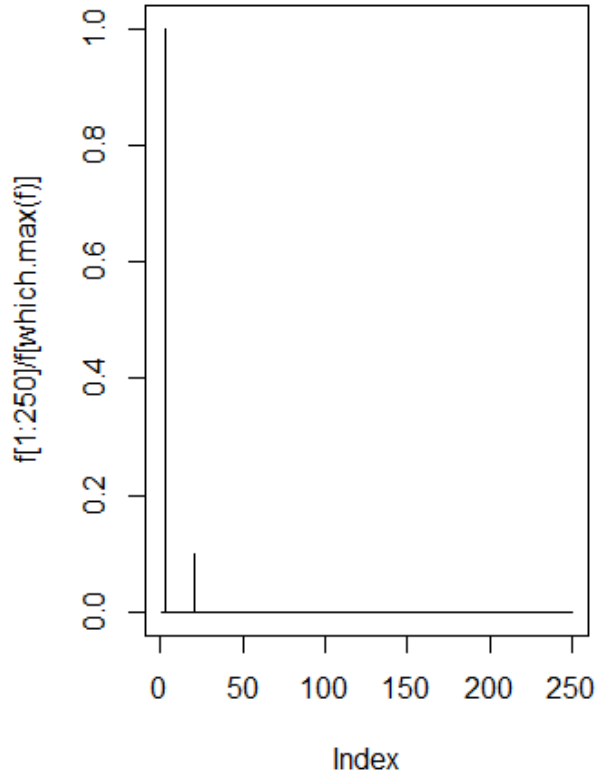
where index_n is the nth index in the spectrum plot starting from zero, the sample rate is determined beforehand (usually 1 sample/unit time for time series) and the FFT (input) size is 500 in our example



Fourier Analysis in R

- So in our example, the first large spike is at index = 2, since we start from 0 (0 frequency being static signal components) and this gives us a frequency of 2 * 1/ 500 = 2/500 = 1/250 as we specified.
- The noise frequency sees a spike at index = 20; so 20 * 1/500 = 1/25 as specified originally.

Now we can show you how to do some Time Series examples!



Fourier Analysis Forecasting

- Fourier analysis can be used in Economical/Finance forecasting. A good example comes from the paper *Fourier Analysis for Demand Forecasting in a Fashion Company* by Fumi et al.
- The method is as follows:
  1. grab time series data from a long period, separate it into a training and validation set
  2. optionally filter/clean the training set to remove high frequency noise
  3. find a linear trend in the training data and then de-trend it, we keep the trend parameters for later
  4. calculate the FFT on the detrended data and produce a frequency spectrum
  5. except for the 0 frequency component, order the frequency spectrum components in decreasing amplitude
  6. starting with one component and the zero frequency, perform an inverse FFT on the spectrum, add the trend and calculate error against validation set
  7. iterate 6, increasing number of frequency components added until combination with least error is found