Working with Signals

Table of Contents

7.1Introduction

A signal is a way of conveying information. Gestures, semaphores, images, sound, all can be signals. Technically – signal is a function of time, space, or another observation variable that conveys information. If you're working on a computer, or using a computer to manipulate your data, you're almost-certainly working with digital signals. All manipulations of the data are examples of digital signal processing.

There are many definitions of signal. For example,

Signal is a time varying physical phenomenon which is intended to convey information.

OR

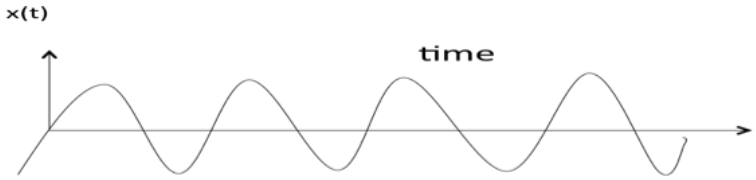
Signal is a function of time.

OR

Signal is a function of one or more independent variables, which contain some information.

Example: voice signal, video signal, signals on telephone wires etc.

Note: Noise is also a signal, but the information conveyed by noise is unwanted hence it is considered as undesirable.



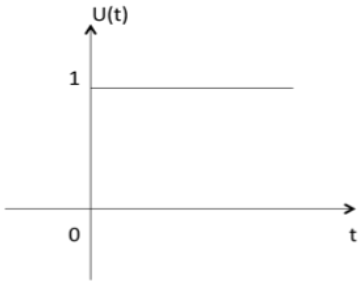
However, Signal processing is an electrical engineering subfield that focuses on analysing, modifying, and synthesizing signals such as sound, images, and scientific measurements.

7.2 Signals Basic Types

Here are a few basic types of signals:

Unit Step Function: Unit step function is denoted by u(t). It is defined as

$$u(t) = \begin{cases} 1 & t \geqslant 0 \\ 0 & t < 0 \end{cases}$$



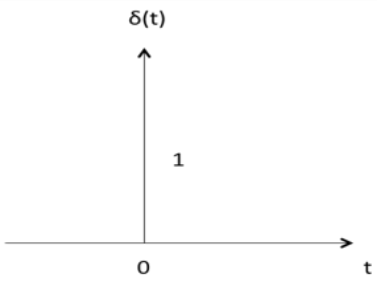
- It is used as best test signal.

- Area under unit step function is unity.

Unit Impulse Function

Impulse function is denoted by δ(t). and it is defined as

$$\delta(t) = \begin{cases} 1 & t = 0 \\ 0 & t \neq 0 \end{cases}$$



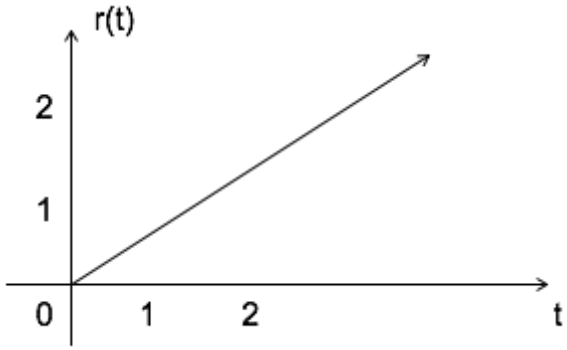
$$\int_{-\infty}^{\infty} \delta(t)dt = u(t)$$

$$\delta(t) = \frac{du(t)}{dt}$$

Ramp Signal

Ramp signal is denoted by r(t), and it is defined as

$$r(t) = \begin{cases} t & t \geqslant 0 \\ 0 & t < 0 \end{cases}$$



$$\int u(t) = \int 1 = t = r(t)$$

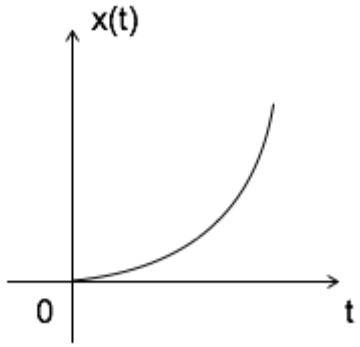
$$u(t) = \frac{dr(t)}{dt}$$

Area under unit ramp is unity.

Parabolic Signal

Parabolic signal can be defined as

$$x(t) = \begin{cases} t^2/2 & t \geqslant 0 \\ 0 & t < 0 \end{cases}$$



$$\iint u(t)dt = \int r(t)dt = \int tdt = \frac{t^2}{2} = parabolicsignal$$

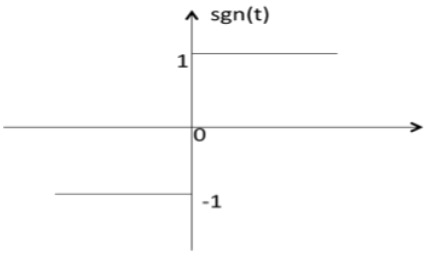
$$\Rightarrow u(t) = \frac{d^2x(t)}{dt^2}$$

$$\Rightarrow r(t) = \frac{dx(t)}{dt}$$

Signum Function

Signum function is denoted as sgn(t). It is defined as

$$\text{sgn(t)} = \begin{cases} 1 & t > 0 \\ 0 & t = 0 \\ -1 & t < 0 \end{cases}$$



sgn(t) = 2u(t) – 1

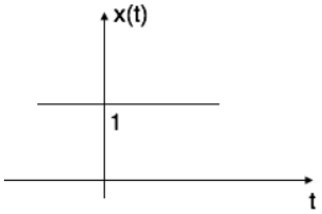
Exponential Signal

Exponential signal is in the form of

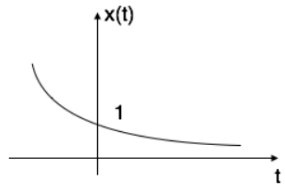
$$x(t) = e^{\alpha t}$$

The shape of exponential can be defined by αα.

Case i: if $\alpha = 0 \rightarrow x(t) = e^0 = 1$

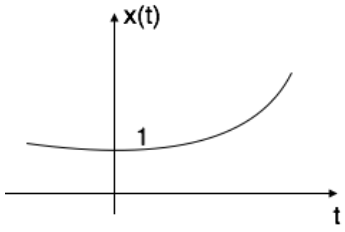


Case ii: if α < 0 i.e. -ve then x(t) = $e^{-\alpha t}$. The shape is called decaying exponential.



Case iii: if α > 0 i.e. +ve then x(t) = $e^{\alpha t}$. The shape is called raising exponential.

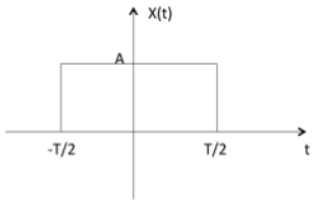
.



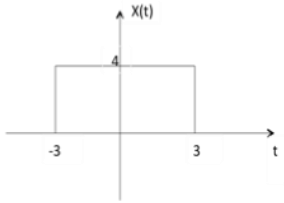
Rectangular Signal

Let it be denoted as x(t) and it is defined as

$x(t) = A\ rect\left[\frac{r}{T}\right]$



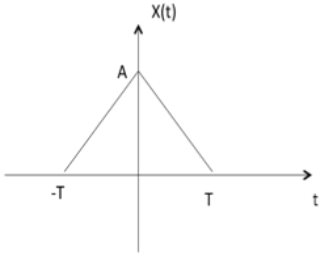
ex: $4\ rect\left[\frac{r}{6}\right]$



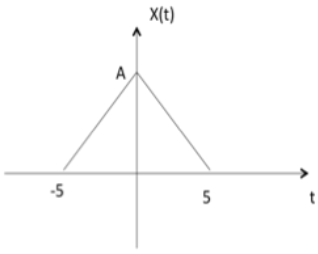
Triangular Signal

Let it be denoted as x(t)

$x(t) = A\left[1 - \frac{|t|}{T}\right]$

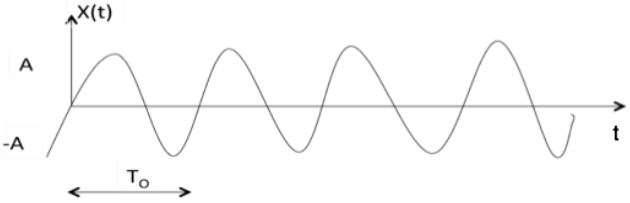


ex: $x(t) = A\left[1 - \frac{|t|}{5}\right]$



Sinusoidal Signal

Sinusoidal signal is in the form of x(t) = A cos(w0±ϕw0±ϕ) or A sin(w0±ϕw0±ϕ)



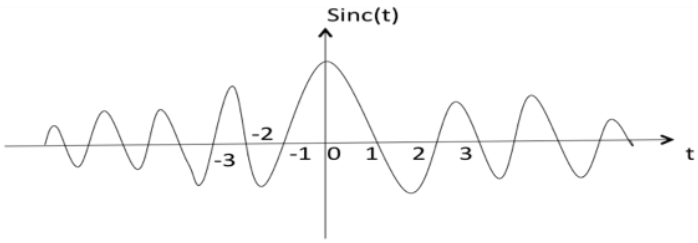
Where $T_0 = 2\pi w0$

Sinc Function

It is denoted as sinc(t) and it is defined as sinc

$$(t) = \frac{sin\pi t}{\pi t}$$

$= 0$ for $t = \pm 1, \pm 2, \pm 3 \ldots$

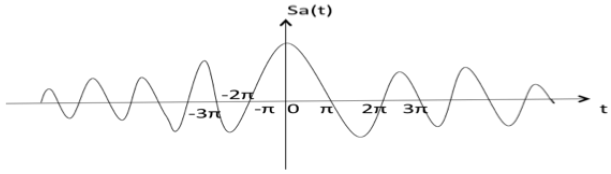


Sampling Function

It is denoted as sa(t) and it is defined as

$$sa(t) = \frac{sint}{t}$$

$= 0$ for $t = \pm\pi, \pm 2\pi, \pm 3\pi \ldots$



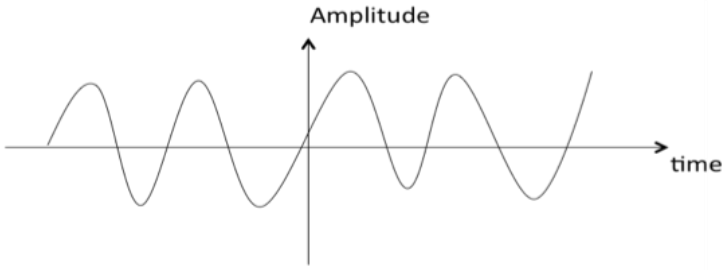
7.3 Classification of Signals

Signals are classified into the following categories:

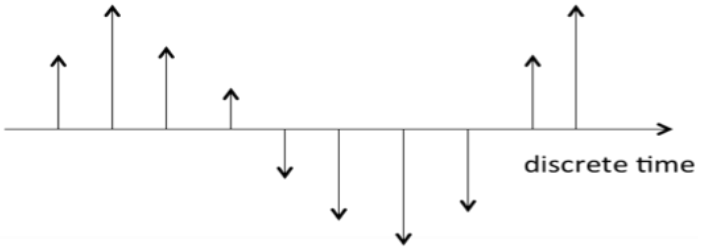
- Continuous Time and Discrete Time Signals
- Deterministic and Non-deterministic Signals
- Even and Odd Signals
- Periodic and Aperiodic Signals
- Energy and Power Signals
- Real and Imaginary Signals

7.3.1 Continuous Time and Discrete Time Signals

A signal is said to be continuous when it is defined for all instants of time.

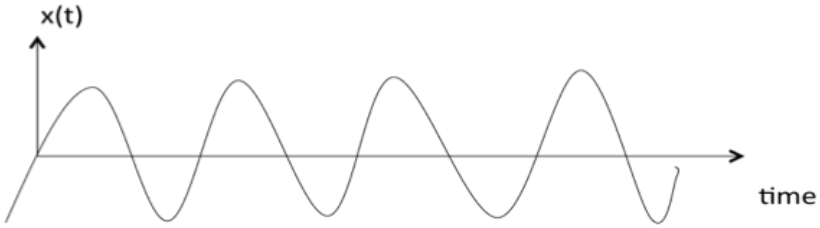


A signal is said to be discrete when it is defined at only discrete instants of time/

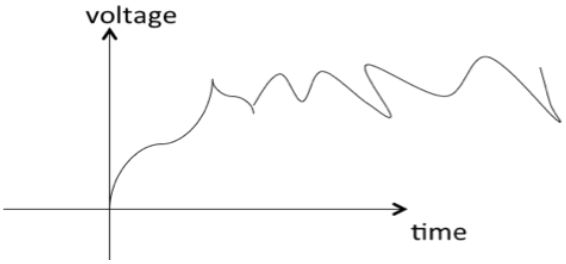


7.3.2 Deterministic and Non-deterministic Signals

A signal is said to be deterministic if there is no uncertainty with respect to its value at any instant of time. Or, signals which can be defined exactly by a mathematical formula are known as deterministic signals.



A signal is said to be non-deterministic if there is uncertainty with respect to its value at some instant of time. Non-deterministic signals are random in nature hence they are called random signals. Random signals cannot be described by a mathematical equation. They are modelled in probabilistic terms.



7.3.3 Even and Odd Signals

A signal is said to be even when it satisfies the condition x(t) = x(-t)

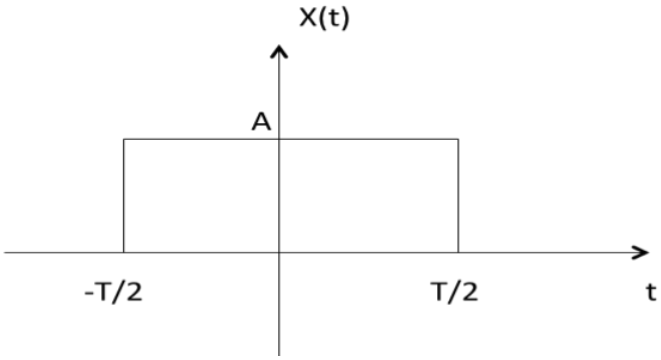
Example 1: t^2, t^4… cost etc.

Let x(t) = t^2

x(-t) = (-t)^2 = t^2 = x(t)

∴, t^2 is even function

Example 2: As shown in the following diagram, rectangle function x(t) = x(-t) so it is also even function.



A signal is said to be odd when it satisfies the condition x(t) = -x(-t)

Example: t, t3 ... And sin t

Let x(t) = sin t

x(-t) = sin(-t) = -sin t = -x(t)

∴,∴, sin t is odd function.

Any function $f(t)$ can be expressed as the sum of its even function $f_e(t)$ and odd function $f_o(t)$.

$f(t) = f_e(t) + f_0(t)$

where

$f_e(t) = ½[f(t) + f(-t)]$

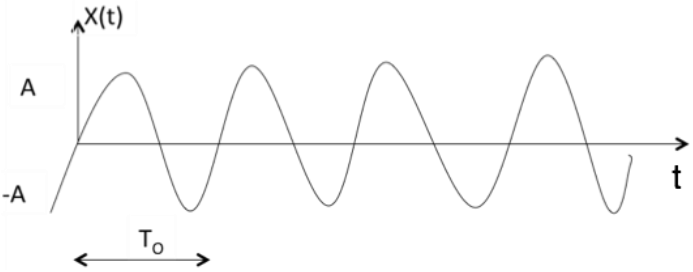
7.3.4 Periodic and Aperiodic Signals

A signal is said to be periodic if it satisfies the condition x(t) = x(t + T) or x(n) = x(n + N).

Where

T = fundamental time period,

1/T = f = fundamental frequency.



The above signal will repeat for every time interval T_0 hence it is periodic with period T_0.

However, a signal that does not repeat itself after a specific interval of time is called an aperiodic signal. By applying a limiting process, the signal can be expressed as a continuous sum (or integral) of everlasting exponentials.

7.3.5 Energy and Power Signals

A signal is said to be energy signal when it has finite energy.

$$\text{Energy } E = \int_{-\infty}^{\infty} x^2(t)dt$$

A signal is said to be power signal when it has finite power.

$$\text{Power } P = \lim_{T\to\infty} \frac{1}{2T} \int_{-T}^{T} x^2(t)dt$$

NOTE:A signal cannot be both, energy and power simultaneously. Also, a signal may be neither energy nor power signal.

Power of energy signal = 0

Energy of power signal = ∞

7.3.6 Real and Imaginary Signals

A signal is said to be real when it satisfies the condition x(t) = x*(t)

A signal is said to be odd when it satisfies the condition x(t) = -x*(t)

Example:

If x(t)= 3 then x*(t)=3*=3 here x(t) is a real signal.

If x(t)= 3j then x*(t)=3j* = -3j = -x(t) hence x(t) is a odd signal.

Note: For a real signal, imaginary part should be zero. Similarly for an imaginary signal, real part should be zero.

7.4 Signal parameters

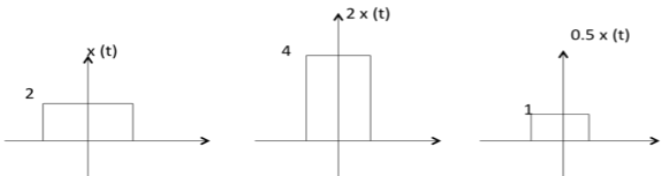
There are two variable parameters in general:

1. Amplitude
2. Time

The following operation can be performed with amplitude:

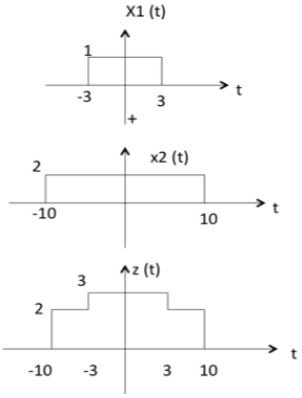
Amplitude Scaling

C x(t) is a amplitude scaled version of x(t) whose amplitude is scaled by a factor C.



Addition

Addition of two signals is nothing but addition of their corresponding amplitudes. This can be best explained by using the following example:



As seen from the diagram above,

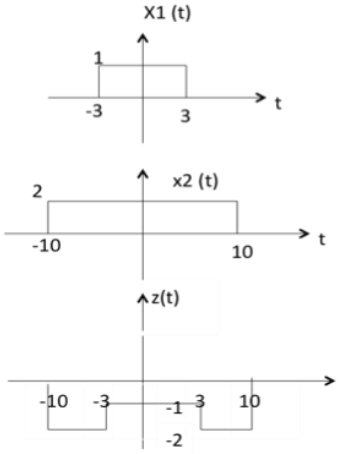
-10 < t < -3 amplitude of z(t) = x1(t) + x2(t) = 0 + 2 = 2

-3 < t < 3 amplitude of z(t) = x1(t) + x2(t) = 1 + 2 = 3

3 < t < 10 amplitude of z(t) = x1(t) + x2(t) = 0 + 2 = 2

Subtraction

subtraction of two signals is nothing but subtraction of their corresponding amplitudes. This can be best explained by the following example:



As seen from the diagram above,

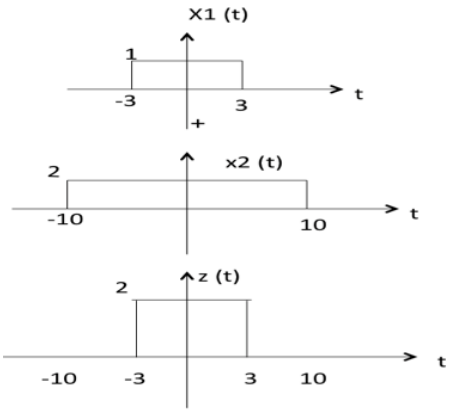
-10 < t < -3 amplitude of z (t) = x1(t) - x2(t) = 0 - 2 = -2

-3 < t < 3 amplitude of z (t) = x1(t) - x2(t) = 1 - 2 = -1

3 < t < 10 amplitude of z (t) = x1(t) + x2(t) = 0 - 2 = -2

Multiplication

Multiplication of two signals is nothing but multiplication of their corresponding amplitudes. This can be best explained by the following example:



As seen from the diagram above,

-10 < t < -3 amplitude of z (t) = x1(t) ×x2(t) = 0 ×2 = 0

-3 < t < 3 amplitude of z (t) = x1(t) ×x2(t) = 1 ×2 = 2

3 < t < 10 amplitude of z (t) = x1(t) × x2(t) = 0 × 2 = 0

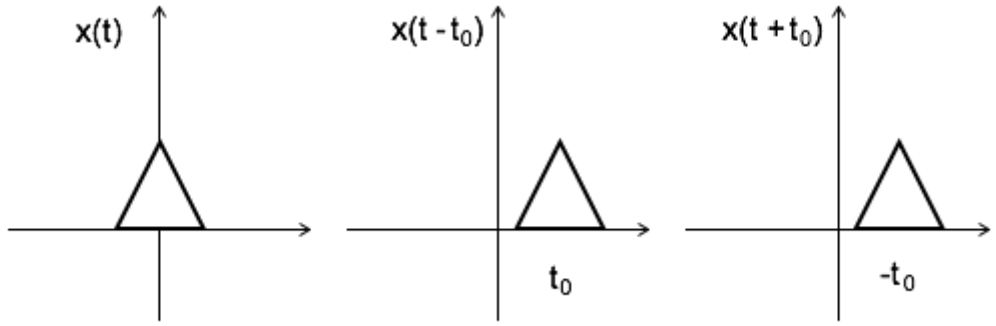
The following operations can be performed with time:

Time Shifting

$x(t \pm t_0)$ is time shifted version of the signal x(t).

$x(t + t_0)$ →→ negative shift

$x(t - t_0)$ →→ positive shift

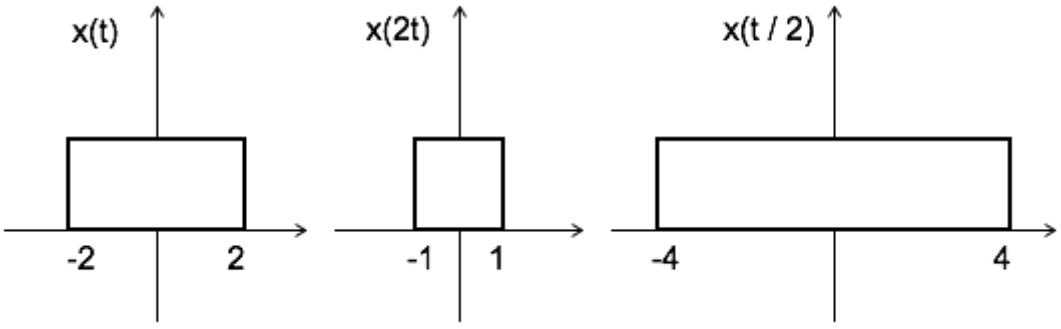


Time Scaling

x(At) is time scaled version of the signal x(t). where A is always positive.

|A| > 1 → Compression of the signal

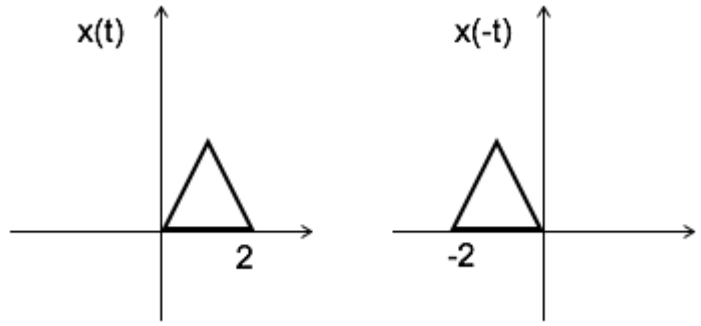
|A| < 1 → Expansion of the signal



Note: u(at) = u(t) time scaling is not applicable for unit step function.

Time Reversal

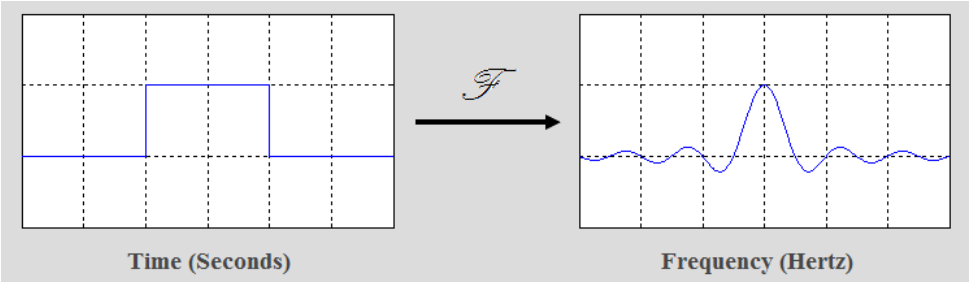
x(-t) is the time reversal of the signal x(t).



7.6 Fourier transform

7.6.1 Definition of Fourier Transform

The Fourier Transform is a tool that breaks a waveform (a function or signal) into an alternate representation, characterized by sine and cosines. The Fourier Transform shows that any waveform can be re-written as the sum of sinusoidal functions.



Mathematically, The Fourier Transform is a mathematical technique that transforms a function of time, x(t), to a function of frequency, X(ω).

The Fourier Transform of a function can be derived as a special case of the Fourier Series when the period, T→∞

$$x(t) = \sum_{n=-\infty}^{+\infty} c_n e^{jn\omega_0 t}$$

where c_n is given by the Fourier Series analysis equation,

$$c_n = \frac{1}{T}\int_T x(t)e^{-jn\omega_0 t}dt$$

which can be rewritten,

$$Tc_n = \int_T x(t)e^{-jn\omega_0 t}dt$$

As $T\to\infty$ the fundamental frequency, $\omega_0=2\pi/T$, becomes extremely small and the quantity $n\omega_0$ becomes a continuous quantity that can take on any value (since n has a range of $\pm\infty$) so we define a new variable $\omega=n\omega_0$; we also let $X(\omega)=Tc_n$. Making these substitutions in the previous equation yields the analysis equation for the Fourier Transform (also called the Forward Fourier Transform).

$$X(\omega) = \int_{-\infty}^{+\infty} x(t)e^{-j\omega t}dt$$

Likewise, we can derive the Inverse Fourier Transform (i.e., the synthesis equation) by starting with the synthesis equation for the Fourier Series (and multiply and divide by T).

$$x(t) = \sum_{n=-\infty}^{+\infty} c_n e^{jn\omega_0 t} = \sum_{n=-\infty}^{+\infty} Tc_n e^{jn\omega_0 t}\frac{1}{T}$$

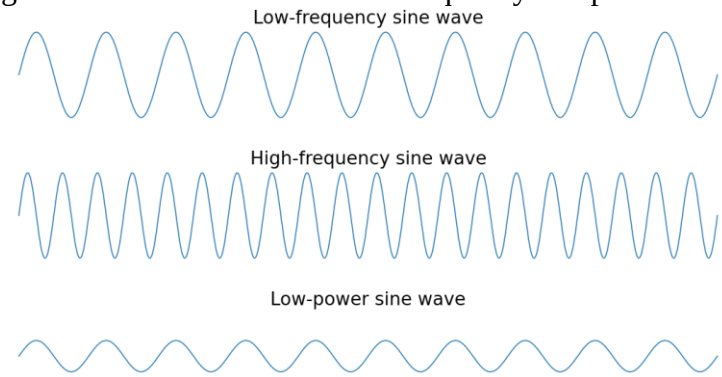
As $T \to \infty$, $1/T = \omega 0/2\pi$. Since ω_0 is very small (as T gets large, replace it by the quantity $d\omega$). As before, we write $\omega = n\omega 0$ and $X(\omega) = Tc_n$. A little work (and replacing the sum by an integral) yields the synthesis equation of the Fourier Transform.

$$x(t) = \sum_{n=-\infty}^{+\infty} X(\omega)e^{j\omega t}\frac{d\omega}{2\pi} = \frac{1}{2\pi}\int_{-\infty}^{+\infty} X(\omega)e^{j\omega t}d\omega$$

Fourier analysis is a field that studies how a mathematical function can be decomposed into a series of simpler trigonometric functions. The Fourier transform is a tool from this field for decomposing a function into its component frequencies. In short, the Fourier transform is a tool that allows you to take a signal and see the power of each frequency in it. Take a look at the important terms in that sentence:

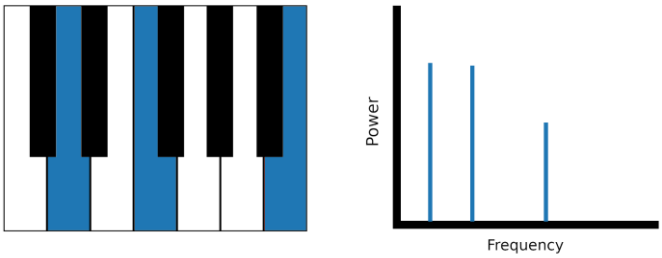
A signal is information that changes over time. For example, audio, video, and voltage traces are all examples of signals. A frequency is the speed at which something repeats. For example, clocks tick at a frequency of one hertz (Hz), or one repetition per second. Power, in this case, just means the strength of each frequency.

The following image is a visual demonstration of frequency and power on some sine waves:



The peaks of the high-frequency sine wave are closer together than those of the low frequency sine wave since they repeat more frequently. The low-power sine wave has smaller peaks than the other two sine waves. To make this more concrete, imagine you used the Fourier transform on a recording of someone playing three notes on the piano at the same time. The resulting frequency spectrum would show three peaks, one for each of the notes. If the person played one note more softly than the others, then the power of that note's frequency would be lower than the other two.

Here's what that piano example would look like visually:



7.6.2 Why Would You Need the Fourier Transform?

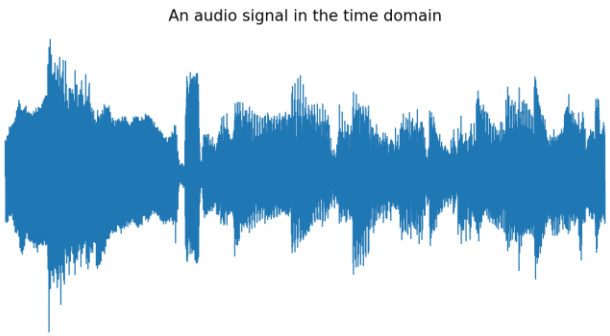
The Fourier transform is useful in many applications. For example, many music identification services use the Fourier transform to identify songs. JPEG compression uses a variant of the Fourier transform to remove the high-frequency components of images. Speech recognition uses the Fourier transform and related transforms to recover the spoken words from raw audio.

In general, you need the Fourier transform if you need to look at the frequencies in a signal. If working with a signal in the time domain is difficult, then using the Fourier transform to move it into the frequency domain is worth trying. In the next section, you'll look at the differences between the time and frequency domains.

7.6.3 Time Domain vs Frequency Domain

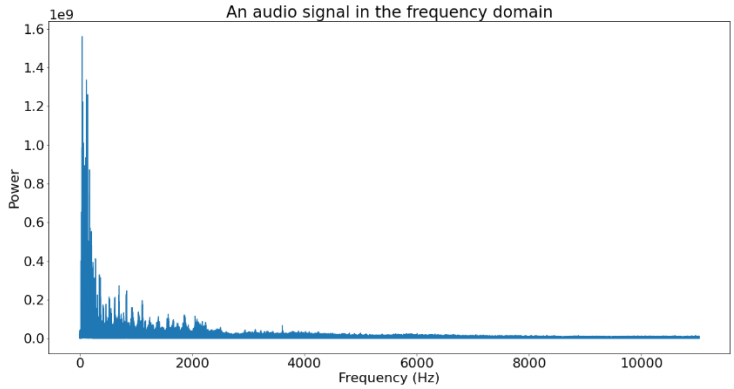
Throughout the rest of the tutorial, you'll see the terms time domain and frequency domain. These two terms refer to two different ways of looking at a signal, either as its component frequencies or as information that varies over time.

In the time domain, a signal is a wave that varies in amplitude (y-axis) over time (x-axis). You're most likely used to seeing graphs in the time domain, such as this one:



This is an image of some audio, which is a time-domain signal. The horizontal axis represents time, and the vertical axis represents amplitude.

In the frequency domain, a signal is represented as a series of frequencies (x-axis) that each have an associated power (y-axis). The following image is the above audio signal after being Fourier transformed:



Here, the audio signal from before is represented by its constituent frequencies. Each frequency along the bottom has an associated power, producing the spectrum that you see.

7.6.4 Types of Fourier Transforms

The Fourier transform can be subdivided into different types of transform. The most basic subdivision is based on the kind of data the transform operates on: continuous functions or discrete functions. This tutorial will deal with only the discrete Fourier transform (DFT).

You'll often see the terms DFT and FFT used interchangeably. However, they aren't quite the same thing. The fast Fourier transform (FFT) is an algorithm for computing the discrete Fourier transform (DFT), whereas the DFT is the transform itself.

Another distinction that you'll see made in the scipy.fft library is between different types of input. fft() accepts complex-valued input, and rfft() accepts real-valued input.

Two other transforms are closely related to the DFT: the discrete cosine transform (DCT) and the discrete sine transform (DST).

7.6.5 Practical Example

Remove Unwanted Noise From Audio

To help build your understanding of the Fourier transform and what you can do with it, you're going to filter some audio. First, you'll create an audio signal with a high pitched buzz in it, and then you'll remove the buzz using the Fourier transform.

Creating a Signal

Sine waves are sometimes called **pure tones** because they represent a single frequency. You'll use sine waves to generate the audio since they will form distinct peaks in the resulting frequency spectrum.

Another great thing about sine waves is that they're straightforward to generate using NumPy.

Here's some python code that generates a sine wave:

```
import numpy as np
from matplotlib import pyplot as plt

SAMPLE_RATE = 44100  # Hertz
```

```
DURATION = 5  # Seconds

def generate_sine_wave(freq, sample_rate, duration):
    x = np.linspace(0, duration, sample_rate * duration, endpoint=False)
    frequencies = x * freq
    # 2pi because np.sin takes radians
    y = np.sin((2 * np.pi) * frequencies)
    return x, y

# Generate a 2 hertz sine wave that lasts for 5 seconds
x, y = generate_sine_wave(2, SAMPLE_RATE, DURATION)
plt.plot(x, y)
plt.show()
```

After you import NumPy and Matplotlib, you define two constants:

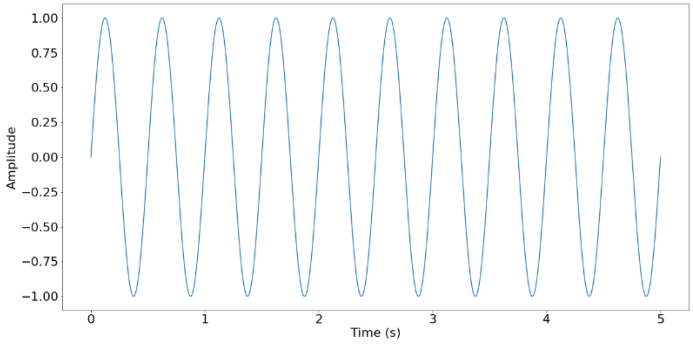
(1) SAMPLE_RATE determines how many data points the signal uses to represent the sine wave per second. So if the signal had a sample rate of 10 Hz and was a five-second sine wave, then it would have 10 * 5 = 50 data points.
(2) DURATION is the length of the generated sample.

Next, you define a function to generate a sine wave since you'll use it multiple times later on. The function takes a frequency, freq, and then returns the x and y values that you'll use to plot the wave.

The x-coordinates of the sine wave are evenly spaced between 0 and DURATION, so the code uses NumPy's linspace() to generate them. It takes a start value, an end value, and the number of samples to generate. Setting endpoint=False is important for the Fourier transform to work properly because it assumes a signal is periodic.

np.sin() calculates the values of the sine function at each of the x-coordinates. The result is multiplied by the frequency to make the sine wave oscillate at that frequency, and the product is multiplied by 2π to convert the input values to radians.

After you define the function, you use it to generate a two-hertz sine wave that lasts five seconds and plot it using Matplotlib. Your sine wave plot should look something like this:



The x-axis represents time in seconds, and since there are two peaks for each second of time, you can see that the sine wave oscillates twice per second. This sine wave is too low a

frequency to be audible, so in the next section, you'll generate some higher-frequency sine waves, and you'll see how to mix them.

Mixing Audio Signals

The good news is that mixing audio signals consists of just two steps:

1. Adding the signals together
2. **Normalizing** the result

Before you can mix the signals together, you need to generate them:

```
_, nice_tone = generate_sine_wave(400, SAMPLE_RATE, DURATION)
_, noise_tone = generate_sine_wave(4000, SAMPLE_RATE, DURATION)
noise_tone = noise_tone * 0.3

mixed_tone = nice_tone + noise_tone
```

There's nothing new in this code example. It generates a medium-pitch tone and a high-pitch tone assigned to the variables nice_tone and noise_tone, respectively. You'll use the high-pitch tone as your unwanted noise, so it gets multiplied by 0.3 to reduce its power. The code then adds these tones together. Note that you use the underscore (_) to discard the x values returned by generate_sine_wave().

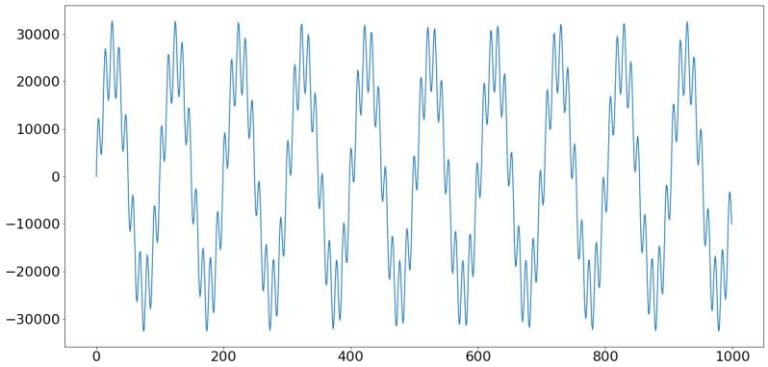
The next step is **normalization**, or scaling the signal to fit into the target format. Due to how you'll store the audio later, your target format is a 16-bit integer, which has a range from -32768 to 32767:

```
normalized_tone = np.int16((mixed_tone / mixed_tone.max()) * 32767)

plt.plot(normalized_tone[:1000])
plt.show()
```

Here, the code scales mixed_tone to make it fit snugly into a 16-bit integer and then cast it to that data type using NumPy's np.int16. Dividing mixed_tone by its maximum value scales it to between -1 and 1. When this signal is multiplied by 32767, it is scaled between -32767 and 32767, which is roughly the range of np.int16. The code plots only the first 1000 samples so you can see the structure of the signal more clearly.

Your plot should look something like this:



The signal looks like a distorted sine wave. The sine wave you see is the 400 Hz tone you generated, and the distortion is the 4000 Hz tone. If you look closely, then you can see the distortion has the shape of a sine wave.

To listen to the audio, you need to store it in a format that an audio player can read. The easiest way to do that is to use SciPy's wavfile.write method to store it in a WAV file. 16-bit integers are a standard data type for WAV files, so you'll normalize your signal to 16-bit integers:

```
from scipy.io.wavfile import write


# Remember SAMPLE_RATE = 44100 Hz is our playback rate
write("mysinewave.wav", SAMPLE_RATE, normalized_tone)
```

This code will write to a file mysinewave.wav in the directory where you run your Python script. You can then listen to this file using any audio player or even with Python. You'll hear a lower tone and a higher-pitch tone. These are the 400 Hz and 4000 Hz sine waves that you mixed.

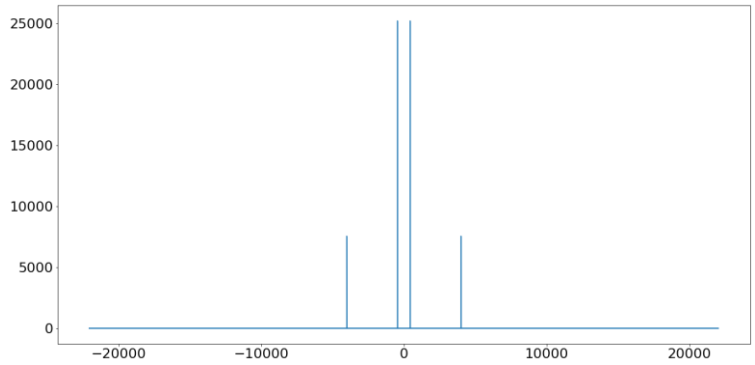
Once you've completed this step, you have your audio sample ready. The next step is removing the high-pitch tone using the Fourier transform!

Using the Fast Fourier Transform (FFT)

It's time to use the FFT on your generated audio. The FFT is an algorithm that implements the Fourier transform and can calculate a frequency spectrum for a signal in the time domain, like your audio:

```
from scipy.fft import fft, fftfreq
# Number of samples in normalized_tone
N = SAMPLE_RATE * DURATION
yf = fft(normalized_tone)
xf = fftfreq(N, 1 / SAMPLE_RATE)
plt.plot(xf, np.abs(yf))
plt.show()
```

This code will calculate the Fourier transform of your generated audio and plot it. Before breaking it down, take a look at the plot that it produces:



You can see two peaks in the positive frequencies and mirrors of those peaks in the negative frequencies. The positive-frequency peaks are at 400 Hz and 4000 Hz, which corresponds to the frequencies that you put into the audio.

The Fourier transform has taken your complicated, wibbly signal and turned it into just the frequencies it contains. Since you put in only two frequencies, only two frequencies have come out. The negative-positive symmetry is a side effect of putting **real-valued input** into the Fourier transform, but you'll hear more about that later.

In the first couple of lines, you import the functions from scipy.fft that you'll use later and define a variable, N, that stores the total number of samples in the signal.

After this comes the most important section, calculating the Fourier transform:

```
yf = fft(normalized_tone)
xf = fftfreq(N, 1 / SAMPLE_RATE)
```

The code calls two very important functions:

1. **fft()** calculates the transform itself.
2. **fftfreq()** calculates the frequencies in the center of each **bin** in the output of fft(). Without this, there would be no way to plot the x-axis on your frequency spectrum.

A **bin** is a range of values that have been grouped, like in a histogram. For more information on **bins**, see this Signal Processing Stack Exchange question. For the purposes of this tutorial, you can think of them as just single values.

Once you have the resulting values from the Fourier transform and their corresponding frequencies, you can plot them:

```
plt.plot(xf, np.abs(yf))
plt.show()
```

The interesting part of this code is the processing you do to yf before plotting it. You call np.abs() on yf because its values are **complex**. A **complex number** is a number that has two parts, a **real** part and an **imaginary** part. There are many reasons why it's useful to define

numbers like this, but all you need to know right now is that they exist. Mathematicians generally write complex numbers in the form $a + bi$, where a is the real part and b is the imaginary part. The i after b means that b is an imaginary number.

Since complex numbers have two parts, graphing them against frequency on a two-dimensional axis requires you to calculate a single value from them. This is where np.abs() comes in. It calculates $\sqrt{(a^2 + b^2)}$ for complex numbers, which is an overall magnitude for the two numbers together and importantly a single value.

Making It Faster With **rfft()**

The frequency spectrum that fft() outputted was reflected about the y-axis so that the negative half was a mirror of the positive half. This symmetry was caused by inputting **real numbers** (not complex numbers) to the transform.

You can use this symmetry to make your Fourier transform faster by computing only half of it. scipy.fft implements this speed hack in the form of rfft().

The great thing about rfft() is that it's a drop-in replacement for fft(). Remember the FFT code from before:

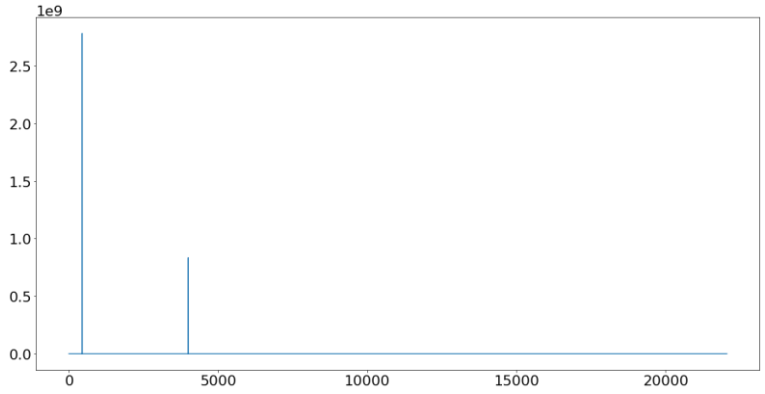
```
yf = fft(normalized_tone)
xf = fftfreq(N, 1 / SAMPLE_RATE)
```

Swapping in rfft(), the code remains mostly the same, just with a couple of key changes:

```
from scipy.fft import rfft, rfftfreq
# Note the extra 'r' at the front
yf = rfft(normalized_tone)
xf = rfftfreq(N, 1 / SAMPLE_RATE)
plt.plot(xf, np.abs(yf))
plt.show()
```

Since rfft() returns only half the output that fft() does, it uses a different function to get the frequency mapping, rfftfreq() instead of fftfreq().

rfft() still produces complex output, so the code to plot its result remains the same. The plot, however, should look like the following since the negative frequencies will have disappeared:



You can see that the image above is just the positive side of the frequency spectrum that fft() produces. rfft() never calculates the negative half of the frequency spectrum, which makes it faster than using fft().

Using rfft() can be up to twice as fast as using fft(), but some input lengths are faster than others. If you know you'll be working only with real numbers, then it's a speed hack worth knowing.

Now that you have the frequency spectrum of the signal, you can move on to filtering it.

Filtering the Signal

One great thing about the Fourier transform is that it's reversible, so any changes you make to the signal in the frequency domain will apply when you transform it back to the time domain. You'll take advantage of this to filter your audio and get rid of the high-pitched frequency.

The values returned by rfft() represent the power of each frequency bin. If you set the power of a given bin to zero, then the frequencies in that bin will no longer be present in the resulting time-domain signal.

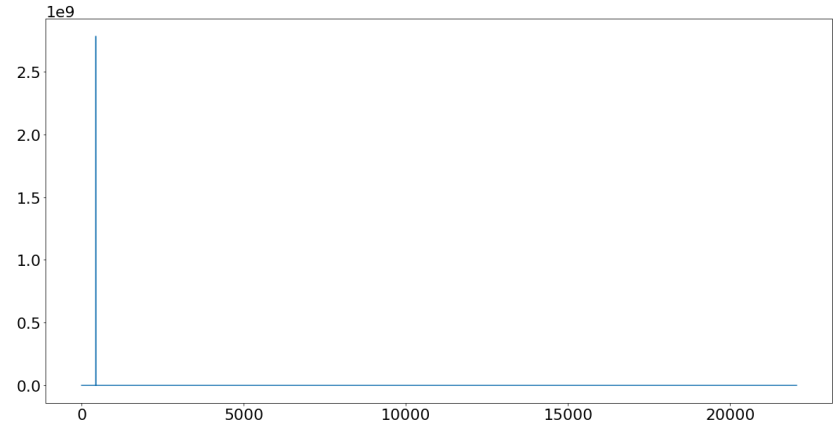
Using the length of xf, the maximum frequency, and the fact that the frequency bins are evenly spaced, you can work out the target frequency's index:

```
# The maximum frequency is half the sample rate
points_per_freq = len(xf) / (SAMPLE_RATE / 2)
# Our target frequency is 4000 Hz
target_idx = int(points_per_freq * 4000)
```

You can then set yf to 0 at indices around the target frequency to get rid of it:

```
yf[target_idx - 1 : target_idx + 2] = 0
plt.plot(xf, np.abs(yf))
plt.show()
```

Your code should produce the following plot:

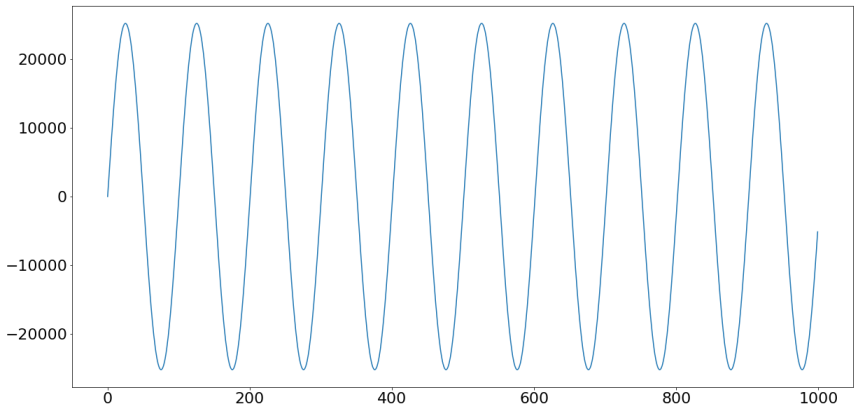


Since there's only one peak, it looks like it worked! Next, you'll apply the inverse Fourier transform to get back to the time domain.

Applying the Inverse FFT
Applying the inverse FFT is similar to applying the FFT:

```
from scipy.fft import irfft
new_sig = irfft(yf)
plt.plot(new_sig[:1000])
plt.show()
```

Since you are using rfft(), you need to use irfft() to apply the inverse. However, if you had used fft(), then the inverse function would have been ifft(). Your plot should now look like this:



As you can see, you now have a single sine wave oscillating at 400 Hz, and you've successfully removed the 4000 Hz noise. Once again, you need to normalize the signal before writing it to a file. You can do it the same way as last time:

```
norm_new_sig = np.int16(new_sig * (32767 / new_sig.max()))
write("clean.wav", SAMPLE_RATE, norm_new_sig)
```

When you listen to this file, you'll hear that the annoying noise has gone away!

Avoiding Filtering Pitfalls
The above example is more for educational purposes than real-world use. Replicating the process on a real-world signal, such as a piece of music, could introduce more buzz than it removes.

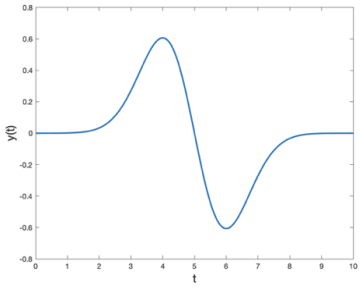
7.7 Wavelet transform

A major disadvantage of the Fourier Transform is it captures global frequency information, meaning frequencies that persist over an entire signal. This kind of signal decomposition may not serve all applications well, for example Electrocardiography (ECG) where signals have short intervals of characteristic oscillation. An alternative approach is the Wavelet Transform, which decomposes a function into a set of wavelets.

Wavelet transforms are one of the key tools for signal analysis. They are extensively used in science and engineering. Some of the specific applications include data compression, gait analysis, signal/image de-noising, digital communications, etc.

7.7.1 What's a Wavelet?

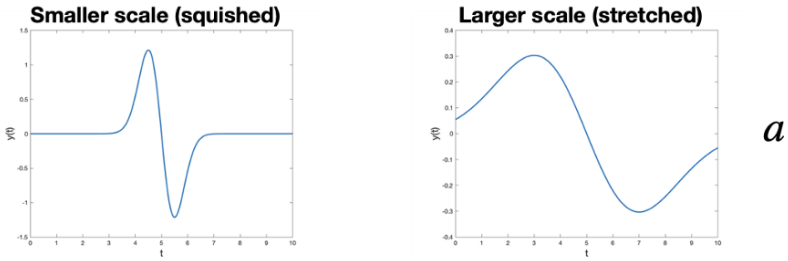
A Wavelet is a wave-like oscillation that is localized in time, an example is given below. Wavelets have two basic properties: scale and location. Scale (or dilation) defines how "stretched" or "squished" a wavelet is. This property is related to frequency as defined for waves. Location defines where the wavelet is positioned in time (or space).



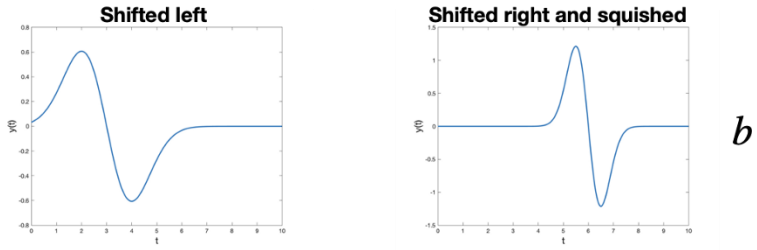
$$-(x-b)e^{\frac{-(x-b)^2/(2a^2)}{\sqrt{2\pi a^3}}}$$

First derivative of Gaussian Function

The parameter "a" in the expression above sets the scale of the wavelet. If we decrease its value the wavelet will look more squished. This in turn can capture high frequency information. Conversely, increasing the value of "a" will stretch the wavelet and captures low frequency information.

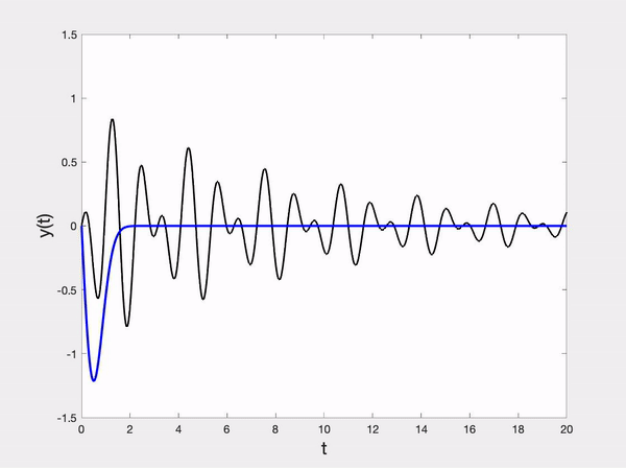


The parameter “b” defines the location of the wavelet. Decreasing “b” will shift the wavelet to the left. Increasing “b” will shift it to the right. Location is important because unlike waves, wavelets are only non-zero in a short interval. Furthermore, when analyzing a signal we are not only interested in its oscillations, but where those oscillations take place.



7.7.2 How does it work?

Let’s take a look at the animation below.



Animation of Discrete Wavelet Transform (again).

The basic idea is to compute how much of a wavelet is in a signal for a particular scale and location. For those familiar with convolutions, that is exactly what this is. A signal is convolved with a set wavelets at a variety of scales.
In other words, we pick a wavelet of a particular scale (like the blue wavelet in the gif above). Then, we slide this wavelet across the entire signal i.e. vary its location, where at each time step we multiply the wavelet and signal. The product of this multiplication gives us a coefficient for that wavelet scale at that time step. We then increase the wavelet scale (e.g. the red and green wavelets) and repeat the process.

7.7.3 Types of Wavelet Transforms

There are two types of Wavelet Transforms: Continuous and Discrete. Definitions of each type are given below. The key difference between these two types is the Continuous Wavelet Transform (CWT) uses every possible wavelet over a range of scales and locations i.e. an infinite number of scales and locations. While the Discrete Wavelet Transform (DWT) uses a finite set of wavelets i.e. defined at a particular set of scales and locations.

Continuous Wavelet Transform (CWT)

$$T(a,b) = \frac{1}{\sqrt{a}} \int_{-\infty}^{\infty} x(t)\, \psi^* \frac{(t-b)}{a} dt$$

Discrete Wavelet Transform (DWT)

$$T_{m,n} = \int_{-\infty}^{\infty} x(t)\, \psi_{m,n}(t)\, dt$$

Definitions of Continuous and Discrete Wavelet Transforms.

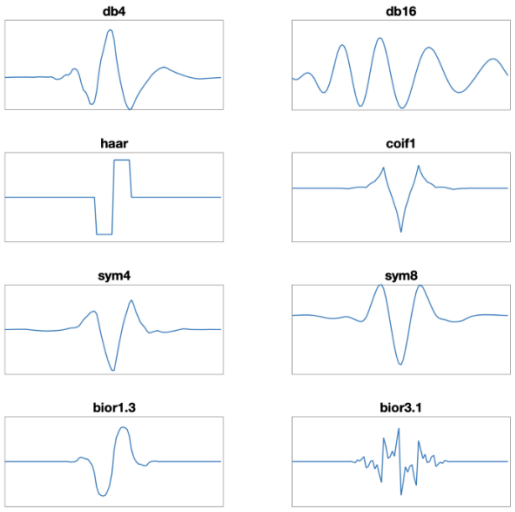
7.7.4 Why wavelets?

A couple of key advantages of the Wavelet Transform are:

- **Wavelet transform** can extract local spectral **and** temporal information simultaneously
- **Variety of wavelets** to choose from

The first key advantage is probably the biggest reason to use the Wavelet Transform. This may be preferable to using something like a Short-Time Fourier Transform which requires chopping up a signal into segments and performing an Fourier Transform over each segment.

The second key advantage sounds more like a technical detail. Ultimately, the takeaway here is if you know what characteristic shape you are trying to extract from your signal, there are a wide variety of wavelets to choose from to best *match* that shape. A handful of options are given in the figure below.



Some wavelet families. From top to bottom, left to right: Daubechies 4, Daubechies 16, Haar, Coiflet 1, Symlet 4, Symlet 8, Biorthogonal 1.3, & Biorthogonal 3.1.

7.7.5 Wavelets in Python

There are several packages in Python which have support for wavelet transforms:

- PyWavelets is one of the most comprehensive implementations for wavelet support in python for both discrete and continuous wavelets.
- pytorch-wavelets provide support for 2D discrete wavelet and 2d dual-tree complex wavelet transforms.
- scipy provides some basic support for the continuous wavelet transform.

PyWavelet is probably the most mature library available. Its coverage and performance are great. However, major parts of the library are written in C. Hence, retargeting the implementation for GPU hardware is not possible. That is one of the reasons for people coming up with newer implementations e.g. on top of PyTorch which provides the necessary GPU support.

7.8 Entropy

The *spectral entropy* (SE) of a signal is a measure of its spectral power distribution. The concept is based on the Shannon entropy, or information entropy, in information theory. The SE treats the signal's normalized power distribution in the frequency domain as a probability distribution, and calculates the Shannon entropy of it. The Shannon entropy in this context is the spectral entropy of the signal. This property can be useful for feature extraction in fault detection and diagnosis. SE is also widely used as a feature in speech recognition and biomedical signal processing.

The equations for spectral entropy arise from the equations for the power spectrum and probability distribution for a signal. For a signal $x(n)$, the power spectrum is $S(m) = |X(m)|^2$, where $X(m)$ is the discrete Fourier transform of $x(n)$.

Then the spectral entropy at time t is:

$$H(t) = -\sum_{m=1}^{N} P(t,m)\log_2 P(t,m).$$

Where,

H(t) = Spectral Entropy

P(t,m) = the probability distribution at time t

7.9 Singular Value Decomposition (SVD)

Singular Value Decomposition (SVD) is a powerful technique widely used in solving dimensionality reduction problems. This algorithm works with a data matrix of the form, m×n, i.e., a rectangular matrix.

The idea behind the SVD is that a rectangular matrix can be broken down into a product of three other matrices that are easy to work with. This decomposition is of the form as the one shown in the formula below:

A= UΣ$A = U \sum V^T$

Where:

- A is our m×n data matrix we are interested in decomposing.
- U is an m×n orthogonal matrix whose bases are orthonormal.
- Σ is an m×n diagonal matrix.
- V^T is the transpose of an orthogonal matrix.

Suppose we have an m×n rectangular matrix A, then we can break it down into
- an m×m orthonormal matrix U,
- m×n diagonal matrix Σ, and
- an n×n orthonormal matrix V^T.

Using the data matrix A, we can compute its SVD, i.e., matrices U, Σ and V^T as follows:

U=AA^T Columns of U are orthonormal eigenvectors of AA^T and are called the left singular vectors of A.
V= $A^T A$ Columns of V are orthonormal eigenvectors of $A^T A$ and are called the right singular vectors of A.

The matrix Σ is a diagonal matrix whose values are square roots of eigenvalues of the matrix U or V in decreasing order. These diagonal entries of matrix Σ are called the singular values.

7.10 Principal Component Analysis (PCA)

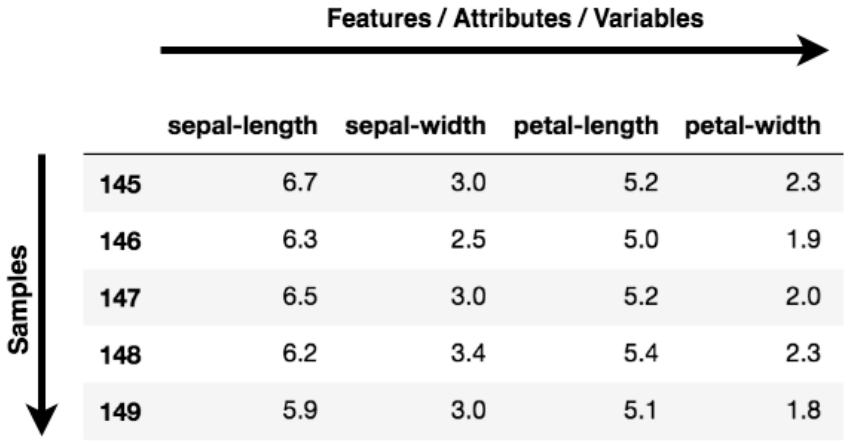
Principal Component Analysis (PCA) is a **linear dimensionality reduction** technique that can be utilized for extracting information from a high-dimensional space by projecting it into a lower-dimensional sub-space. It tries to preserve the essential parts that have more variation of the data and remove the non-essential parts with fewer variation.

According to Wikipedia, PCA is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables (entities each of which takes on various numerical values) into a set of values of linearly uncorrelated variables called principal components.

Dimensions are nothing but features that represent the data. For example, A 28 X 28 image has 784 picture elements (pixels) that are the dimensions or features which together represent that image.

One important thing to note about PCA is that it is an **Unsupervised** dimensionality reduction technique, you can cluster the similar data points based on the feature correlation between them without any supervision (or labels), and you will learn how to achieve this practically using Python!

Note: Features, Dimensions, and Variables are all referring to the same thing. You will find them being used interchangeably.



	sepal-length	sepal-width	petal-length	petal-width
145	6.7	3.0	5.2	2.3
146	6.3	2.5	5.0	1.9
147	6.5	3.0	5.2	2.0
148	6.2	3.4	5.4	2.3
149	5.9	3.0	5.1	1.8

Application of PCA?

Data Visualization: When working on any data related problem, the challenge in today's world is the sheer volume of data, and the variables/features that define that data. To solve a problem where data is the key, you need extensive data exploration like finding out how the variables are correlated or understanding the distribution of a few variables. Considering that there are a large number of variables or dimensions along which the data is distributed, visualization can be a challenge and almost impossible. Hence, PCA can do that for you since it projects the data into a lower dimension, thereby allowing you to visualize the data in a 2D or 3D space with a naked eye.

Speeding Machine Learning (ML) Algorithm: Since PCA's main idea is dimensionality reduction, you can leverage that to speed up your machine learning algorithm's training and testing time considering your data has a lot of features, and the ML algorithm's learning is too slow. At an abstract level, you take a dataset having many features, and you simplify that dataset by selecting a few Principal Components from original features.

What is a Principal Component?

Principal components are the key to PCA; they represent what's underneath the hood of your data. In a layman term, when the data is projected into a lower dimension (assume three dimensions) from a higher space, the three dimensions are nothing but the three Principal Components that captures (or holds) most of the variance (information) of your data.

Principal components have both direction and magnitude. The direction represents across which *principal axes* the data is mostly spread out or has most variance and the magnitude signifies the amount of variance that Principal Component captures of the data when projected onto that axis. The principal components are a straight line, and the first principal component holds the most variance in the data. Each subsequent principal component is orthogonal to the last and has a lesser variance. In this way, given a set of *x* correlated variables over *y* samples you achieve a set of *u* uncorrelated principal components over the same *y* samples.

The reason you achieve uncorrelated principal components from the original features is that the correlated features contribute to the same principal component, thereby reducing the original data features into uncorrelated principal components; each representing a different set of correlated features with different amounts of variation.

Each principal component represents a percentage of total variation captured from the data.