Time series analysis: Signal transforms and DSP

ΕΠΛ 428: IOT PROGRAMMING

Dr. Panayiotis Kolios
Assistant Professor, Dept. Computer Science,
KIOS CoE for Intelligent Systems and Networks
Office: FST 01, 116
Telephone: +357 22893450 / 22892695
Web: https://www.kios.ucy.ac.cy/pkolios/

Πανεπιστήμιο Κύπρου

- Get a basic understanding of the Fourier Transform (FT), Discrete Fourier Transform (DFT), and learn how any function can be approximated by a series of sines and cosines

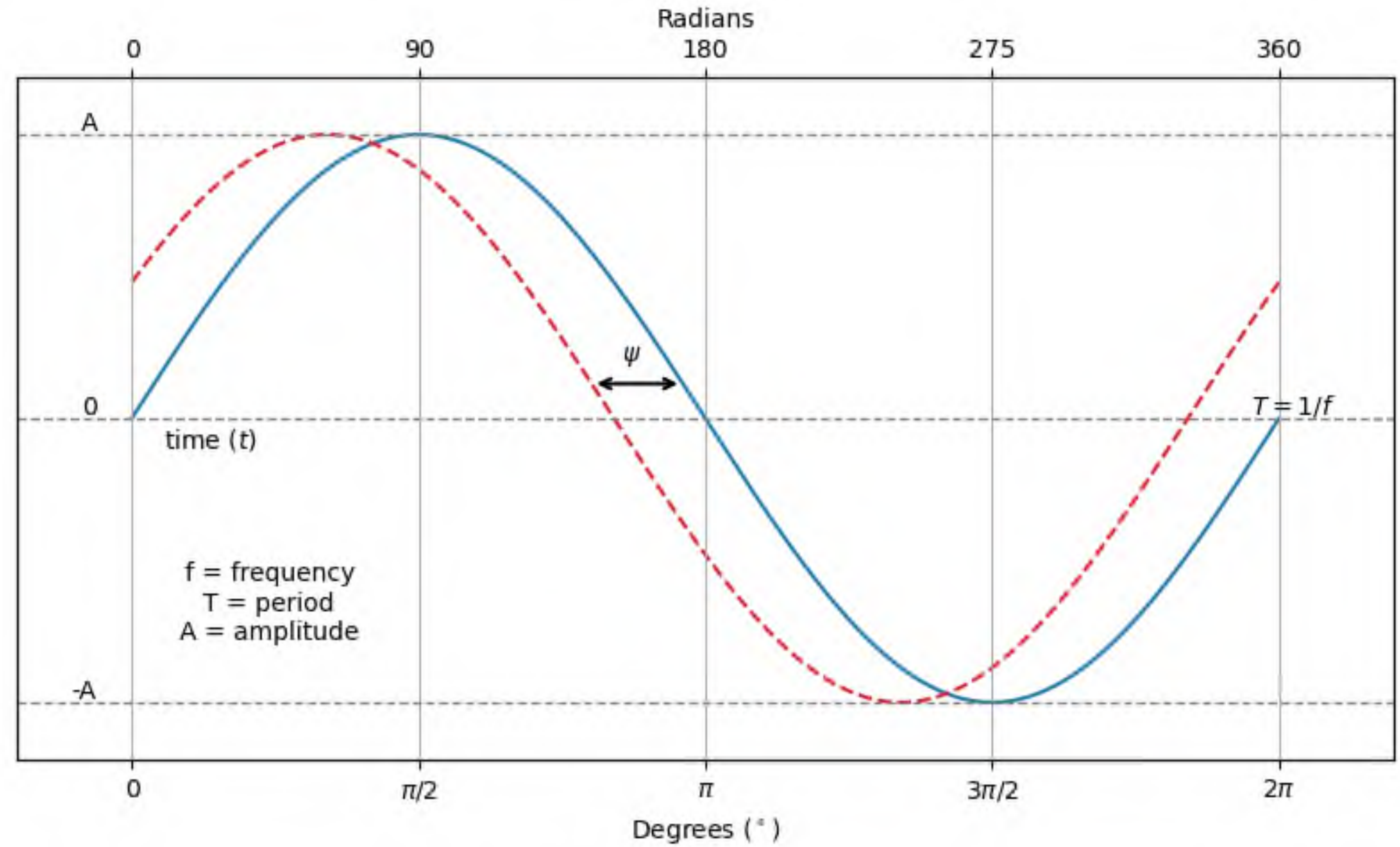
- Learn main properties of FT, FT of common signals, and practical skills needed to apply FT

- Use common filters based on FT and their usefulness to process time series data and forecasting

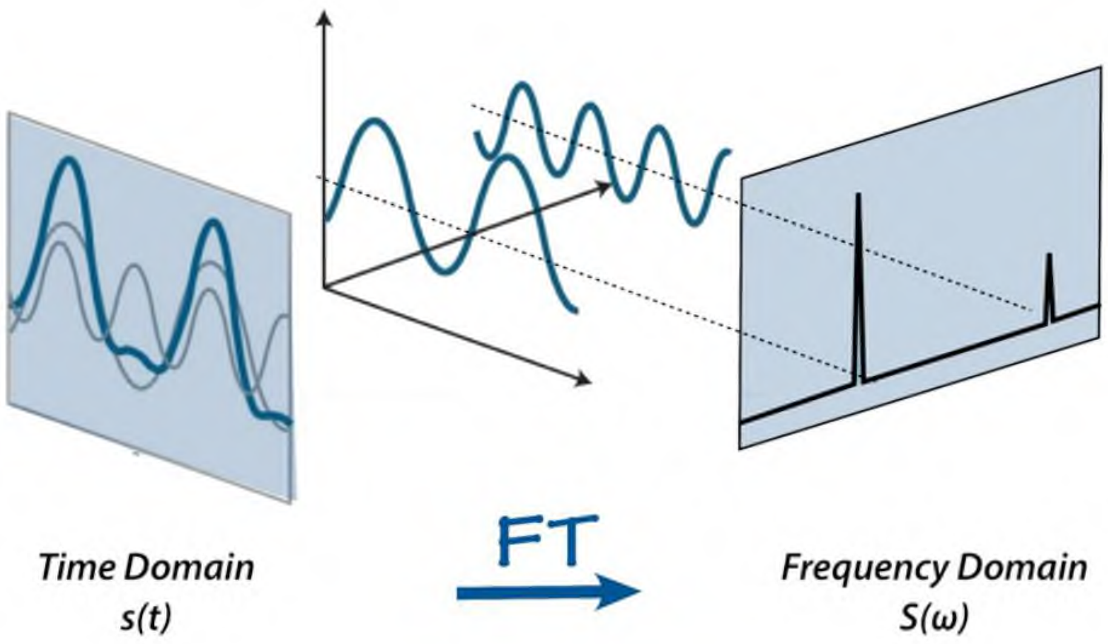
- Fourier Transform: any time signal can be decomposed into a sum of sinusoids that have different amplitude, frequencies, and phases
- Let's start from the definition of a sine wave
    - A sine wave is mathematical curve that describes smooth periodic oscillations
    - It is continuous and is described by:

$$y(t) = Asin(2\pi ft + \psi)$$

    - where A is the amplitude, f the frequency and ψ the phase in radians


Radians
0
90
180
275
360
A
0
-A
ψ
T = 1/f
time (t)
f = frequency
T = period
A = amplitude
0
π/2
π
3π/2
2π
Degrees (°)

- The FT decomposes a time signal into the sum of sines with varying amplitudes, frequencies, and phases
- The sines represent the **constituent** frequencies of the original signal
- As such, FT gives the frequency domain representation of the original signal



- The FT mathematical formulation is given by:

$$\mathcal{F}(\omega) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt$$

- where
  - t is the time
  - $f(t)$ is the continuous time signal
  - $\mathcal{F}(\omega)$ is the Fourier Transform of $f(t)$
  - $\omega = 2\pi f$ is the angular frequency
  - e is the base of the natural logarithm
  - i is the imaginary unit, satisfying $i^2 = -1$

- The FT mathematical formulation is given by:

$$\mathcal{F}(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt$$

  - $e^{-i\omega t}$ is a complex exponential that can be expressed as

$$e^{-i\omega t} = \cos(\omega t) - isin(\omega t)$$

  - $e^{-i\omega t}$ can be conceptualized as a rotating vector (phasor) in the complex plane, where ω is the speed of rotation and t is the time
  - the integral over time indicates that the transformation considers all points in time from $-\infty$ to ∞ to provide a complete representation of the signal in the frequency domain
- Since dealing with ∞ computationally is very hard, approximations have been devised
- The resulting FT shows how much each frequency is present in the signal
  - magnitude of $\mathcal{F}(\omega)$ indicates the amplitude of a particular frequency component
  - Its *phase* (angle of the complex number) indicates the phase shift of that frequency component relative to the origin

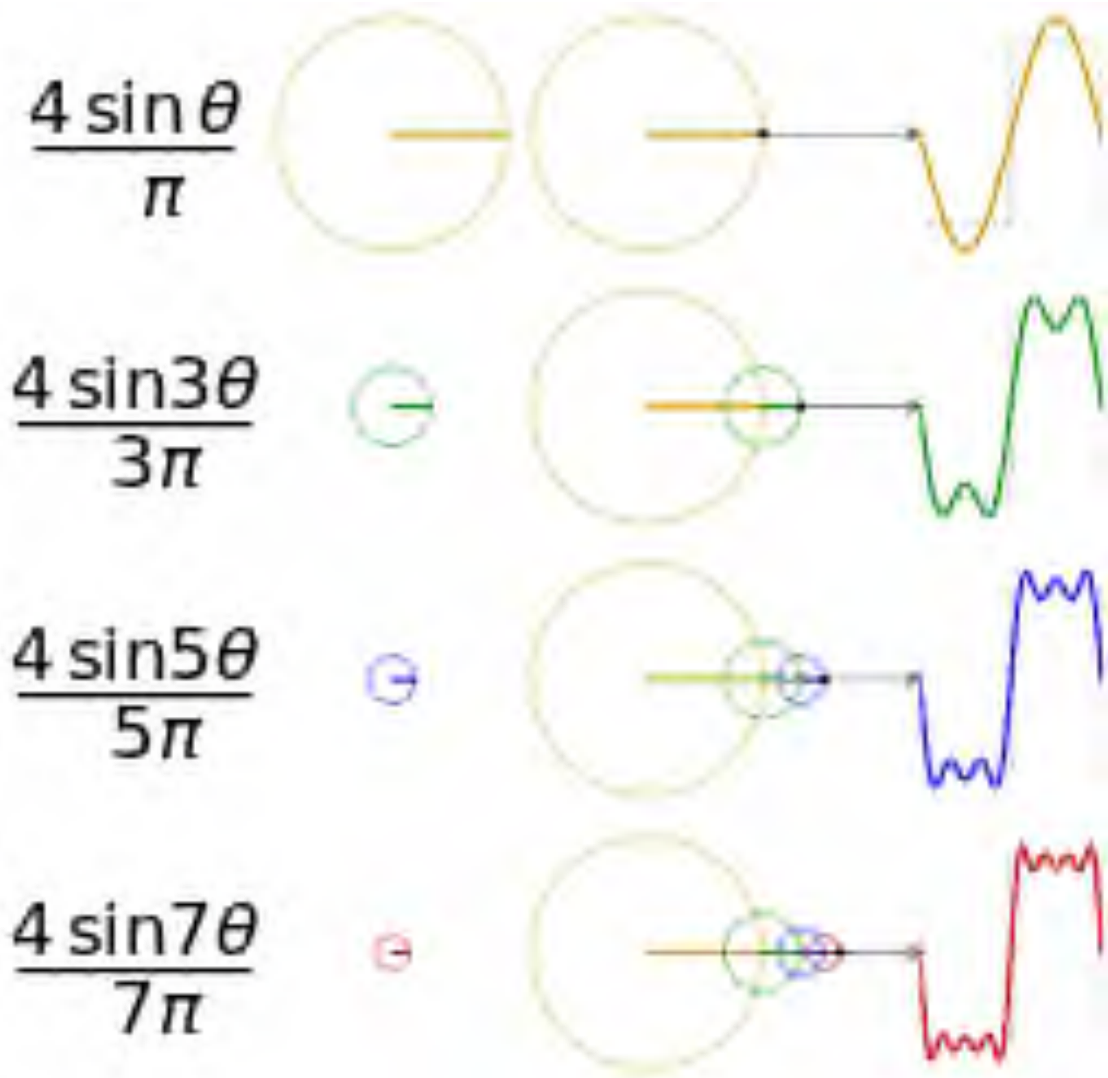
Πανεπιστήμιο Κύπρου

$$\mathcal{F}(\omega) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt$$

- The intuitive interpretation of the FT equation is that the effect of multiplying $f(t)$ and $e^{-i\omega t}$ is to subtract ω from every frequency component of $f(t)$

- Only the component that was at frequency ω can produce a non-zero value of the infinite integral

- On the other hand, all the other shifted components are oscillatory and integrate to zero


$\frac{4\sin\theta}{\pi}$
$\frac{4\sin 3\theta}{3\pi}$
$\frac{4\sin 5\theta}{5\pi}$
$\frac{4\sin 7\theta}{7\pi}$

$$f(t) = \int_{-\infty}^{\infty} \mathcal{F}(\omega) e^{i\omega t} d\omega$$

- The Inverse Fourier Transform expresses a signal $f(t)$ as a weighted summation of complex exponential functions
- It shows us that any function can be expressed as a combination of sinusoids
- The sinusoids represent are the basis

- Discrete Fourier Transform (DFT) converts a finite list of equally spaced samples of a signal into the list of coefficients of a finite combination of complex sinusoids, ordered by their frequencies

- Fast Fourier Transform (FFT) is an algorithm designed to compute the DFT and its inverse efficiently

- FFT significantly reduces the computational complexity of performing a DFT from $O(N^2)$ to $O(N \log N)$, where N is the number of samples

- This efficiency gain is particularly impactful for large datasets, making the FFT an indispensable tool in digital signal processing (DSP), image analysis, fast convolution, among others

- The DFT for a sequence $x[n]$ of length N is defined as follows:

$$X[K] = \sum_{n=0}^{N-1} x[n] e^{-\frac{i2\pi}{N}kn}$$

- for $k = 0, 1, \dots, N-1$,
- $X[K]$ are the frequency domain samples
- The FFT exploits the symmetry and periodicity properties of the complex exponential function
- Applies a divide-and-conquer strategy to decompose the DFT of a sequence into smaller DFTs, thereby reducing the overall computational effort

- The most well-known FFT algorithm is the Cooley-Tukey algorithm
- It recursively divides the DFT of a sequence into two, by separating the sequence into even and odd elements:

$$X[K] = X_{even}[K] + e^{-\frac{i2\pi}{N}k} X_{odd}[K]$$

$$X\left[K + \frac{N}{2}\right] = X_{even}[K] - e^{-\frac{i2\pi}{N}k} X_{odd}[K]$$

- this process is repeated on the subsequences until the problem is reduced to the DFT of sequences of length 1, which are trivial.
- The algorithm recombines these results to produce the final DFT

COMMON FOURIER TRANSFORMS

	Name	$f(t)$	$\mathcal{F}(\omega)$	Remarks
1	Dirac delta	$\delta(t)$	1	Constant energy at all frequencies
2	Time sample	$\delta(t-t_0)$	$e^{-i\omega t_0}$	
3	Phase shift	$e^{i\omega_0 t}$	$2\pi\delta(\omega-\omega_0)$	
4	*Signum*	$sng\ t$	$\frac{2}{i\omega}$	sign function
5	Unit step	$u(t)$	$\frac{1}{i\omega}+\pi\delta(\omega)$	
6	Cosine	$\cos(\omega_0 t)$	$\pi[\delta(\omega-\omega_0)+\delta(\omega+\omega_0)]$	
7	Sine	$\sin(\omega_0 t)$	$-i\pi[\delta(\omega-\omega_0)-\delta(\omega+\omega_0)]$	
8	Single pole	$e^{-at}u_0(t)$	$\frac{1}{i\omega+a}$	$a>0$
9	Double pole	$te^{-at}u_0(t)$	$\frac{1}{(i\omega+a)^2}$	$a>0$
10	Complex pole (cosine component)	$e^{-at}cos\omega_0 t u_0(t)$	$\frac{i\omega+a}{(i\omega+a)^2+\omega_0^2}$	$a>0$
11	Complex pole (sine component)	$e^{-at}sin\omega_0 t u_0(t)$	$\frac{\omega_0}{(i\omega+a)^2+\omega_0^2}$	$a>0$

Πανεπιστήμιο Κύπρου

- Mainly use three functions:

- fft(y): Computes the FFT $\mathcal{F}(\omega)$ of a time signal $y = f(t)$

- fftfreq(n, d): Returns the DFT sample frequencies ω; n is the windows length and d is the sample spacing (inverse of the sampling rate $f = 1/T$)

- fftshift(x): Shifts the zero-frequency component to the center of the spectrum:
  [0., 1., 2., ..., -3., -2., -1.] --> [-5., -4., -3., -2., -1., 0., 1., 2., 3., 4.]

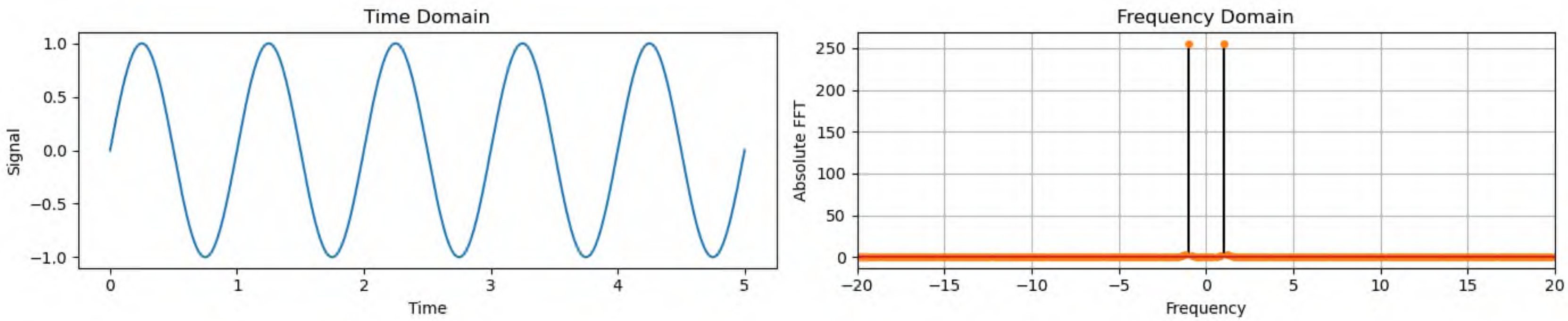
Πανεπιστήμιο Κύπρου

```
# function that creates both a time domain and frequency domain plot
def plot_time_freq(t, y):
    # Converts Data into Frequncy Domain
    freq = np.fft.fftfreq(t.size, d=t[1]-t[0])
    Y = abs(np.fft.fft(y))

    # Time domain plot
    plt.figure(figsize = [14,3])
    plt.subplot(1,2,1)
    plt.plot(t,y)
    plt.title('Time Domain')
    plt.xlabel('Time')
    plt.ylabel('Signal')

    # Frequency domain plot
    plt.subplot(1,2,2)
    markerline, stemline, baseline =
plt.stem(np.fft.fftshift(freq),np.fft.fftshift(Y),
'k', markerfmt='tab:orange')
    plt.setp(stemline, linewidth = 1.5)
    plt.setp(markerline, markersize = 4)
    plt.title('Frequency Domain')
    plt.xlabel('Frequency')
    plt.xlim(-20, 20)
    plt.ylabel('Absolute FFT')
    plt.grid()

    plt.tight_layout()
    plt.show()
```

EXAMPLE APPROXIMATIONS GIVEN BY FFT

Name	$f(t)$	$\mathcal{F}(\omega)$
Sine	$\sin(\omega_0 t)$	$-i\pi[\delta(\omega-\omega_0)-\delta(\omega+\omega_0)]$

```
# time series of a sine wave with a frequency of 1Hz
freq = 1
time = np.linspace(0, 5, 512)
y_sine = np.sin(2 * np.pi * freq * time)
plot_time_freq(time, y_sine)
```



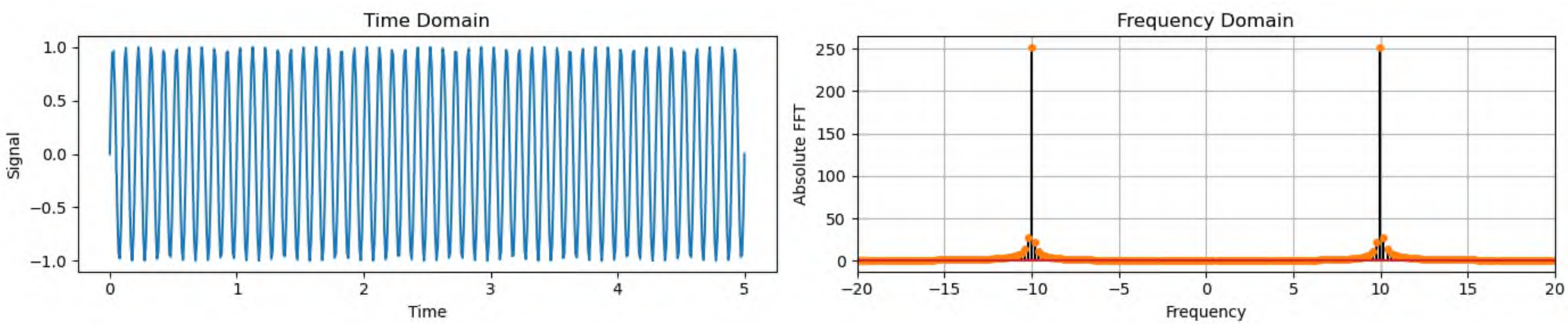
- Observe that in the frequency domain there are 2 spikes at 1Hz and -1Hz
- i.e., the frequency of the sine wave
- symmetry between the left and right side across the 0Hz

Πανεπιστήμιο Κύπρου

EXAMPLE APPROXIMATIONS GIVEN BY FFT

Name	$f(t)$	$\mathcal{F}(\omega)$
Sine	$\sin(\omega_0 t)$	$-i\pi[\delta(\omega-\omega_0)-\delta(\omega+\omega_0)]$

```
# time series of a sine wave with a frequency of 10Hz
freq = 10
time = np.linspace(0, 5, 512)
y_sine = np.sin(2 * np.pi * freq * time)
plot_time_freq(time, y_sine)
```

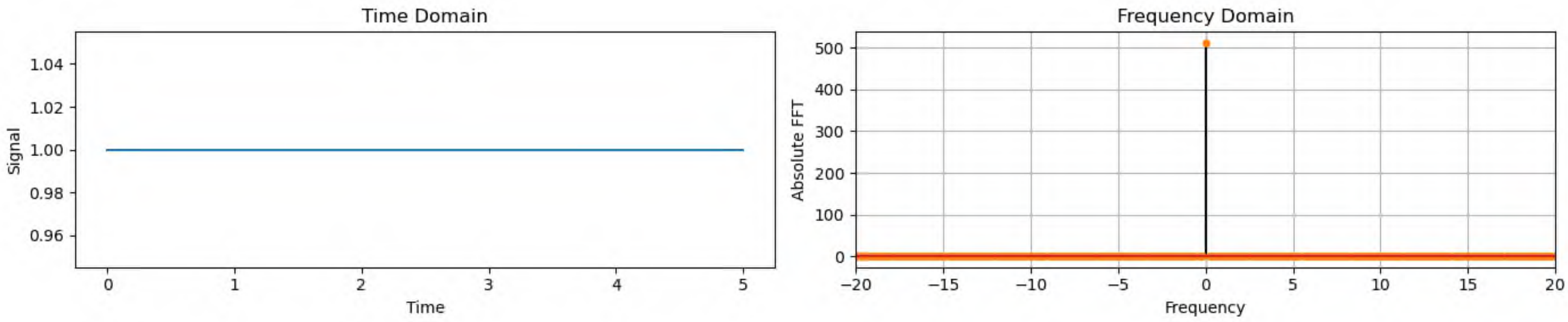


- Observe the 2 spikes at 10Hz and -10Hz
- i.e., the frequency of the sine wave
- symmetry between the left and right side across the 0Hz

EXAMPLE APPROXIMATIONS GIVEN BY FFT

Name	$f(t)$	$\mathcal{F}(\omega)$
Dirac delta	$\delta(t)$	1
Constant	1	$\delta(t)$

```
# Constant amplitude signal in Time Domain
y_constant = np.ones(time.shape)

plot_time_freq(time, y_constant)
```



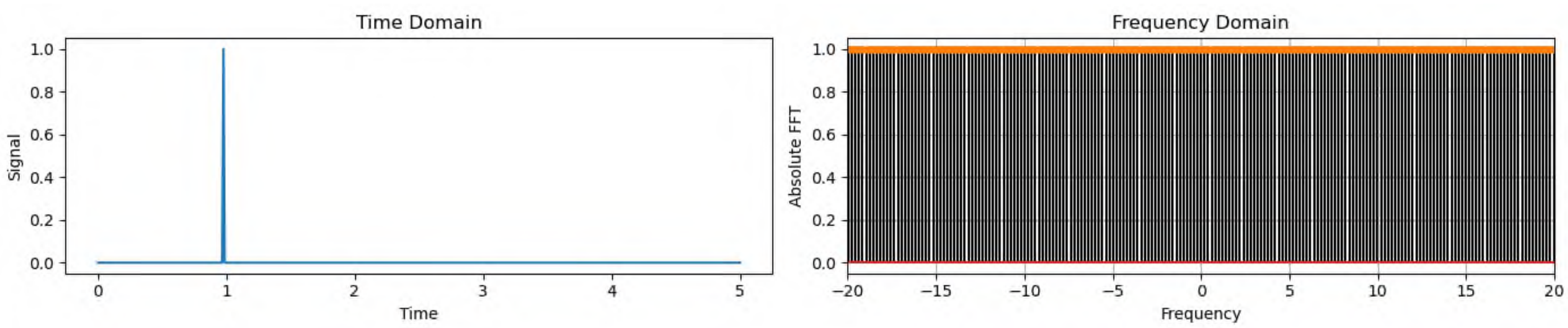
- Constant values in time domain correspond to a Dirac delta at the frequency domain at 0 Hz
- For example, a constant equal to 1 in the time domain, creates to a spike at 0Hz in the frequency domain

Πανεπιστήμιο Κύπρου

EXAMPLE APPROXIMATIONS GIVEN BY FFT

Name	$f(t)$	$\mathcal{F}(\omega)$
Dirac delta	$\delta(t)$	1
Constant	1	$\delta(t)$

```
# Dirac Delta signal in Time Domain
y_delta = np.zeros(time.shape)
y_delta[100] = 1

plot_time_freq(time, y_delta)
```



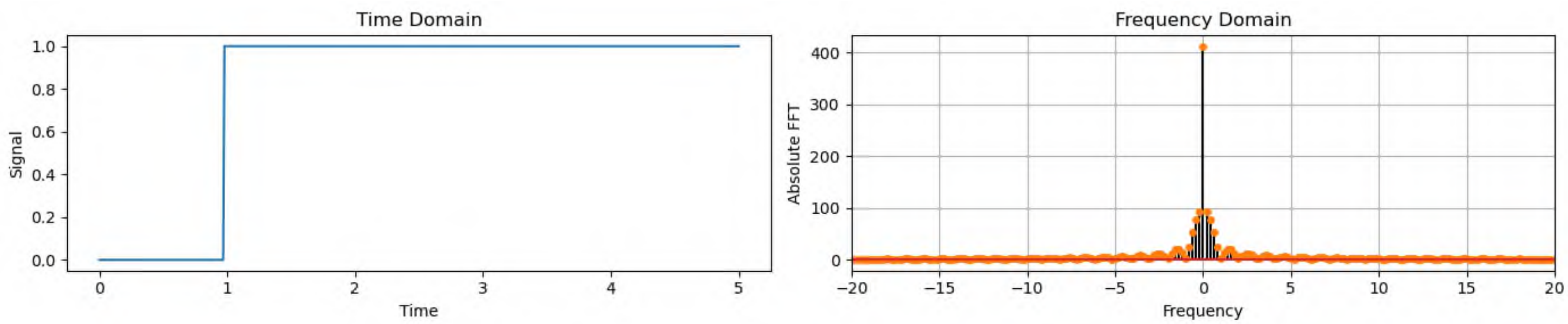
- Dirac delta in time corresponds to a constant in the frequency domain
    - i.e., contains all frequency components

Πανεπιστήμιο Κύπρου

EXAMPLE APPROXIMATIONS GIVEN BY FFT

Name	$f(t)$	$\mathcal{F}(\omega)$
Unit step	$u(t)$	$\frac{1}{i\omega} + \pi\delta(\omega)$

```
# Unit step signal in Time Domain
y_step = np.zeros(time.shape)
y_step[100:] = 1

plot_time_freq(time, y_step)
```

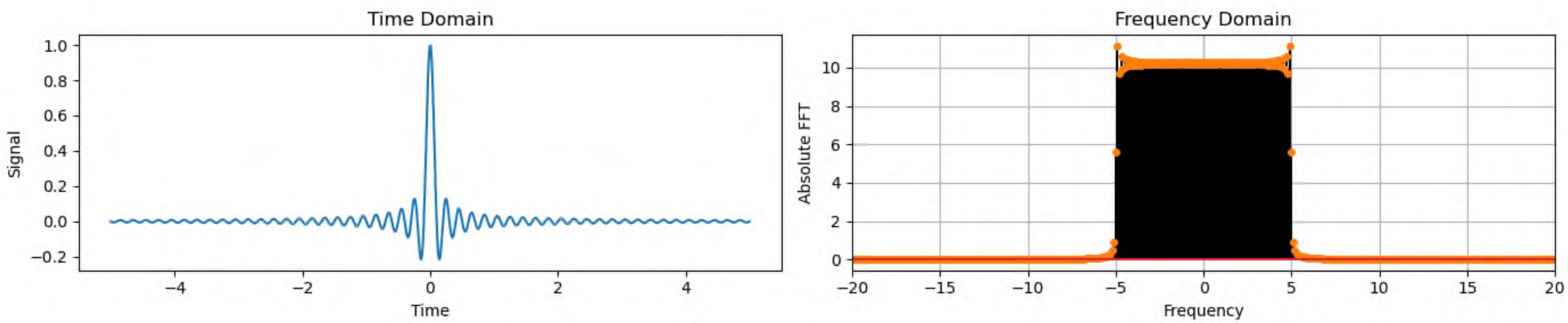


Πανεπιστήμιο Κύπρου

EXAMPLE APPROXIMATIONS GIVEN BY FFT

Name	$f(t)$	$\mathcal{F}(\omega)$
Sinc	$\dfrac{\sin(At)}{\pi t}$	$\begin{cases} 1, & \lvert\omega\rvert < A \\ 0, & otherwise \end{cases}$

```
# Sinc function
time = np.linspace(-5, 5, 1024)
freq=5
y_sinc = np.sin(2 * np.pi * freq * time) / (2 * np.pi * freq * time)
plot_time_freq(time, y_sinc)
```



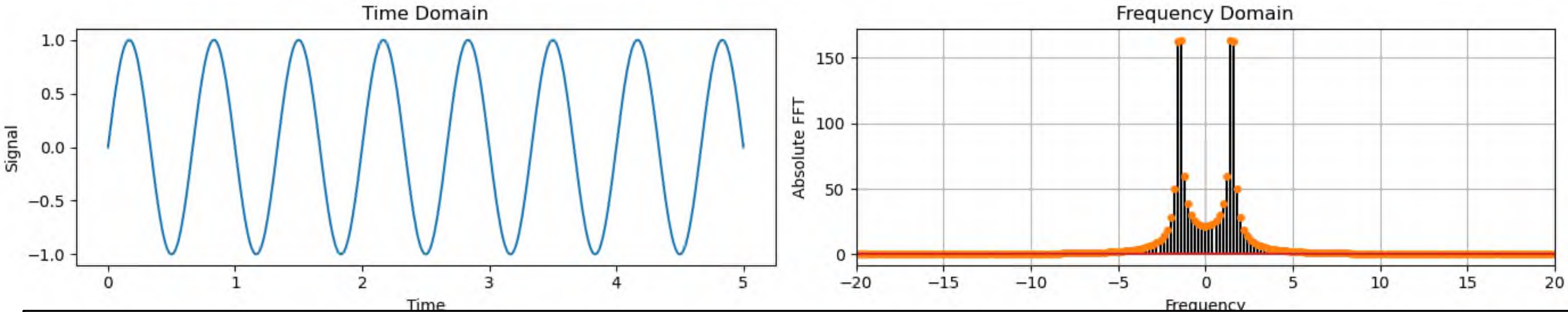
Πανεπιστήμιο Κύπρου

- Analyze a signal that contains:
    1. A sine wave representing seasonality
    2. a parabolic function representing a trend
    3. and uniformly distributed random noise

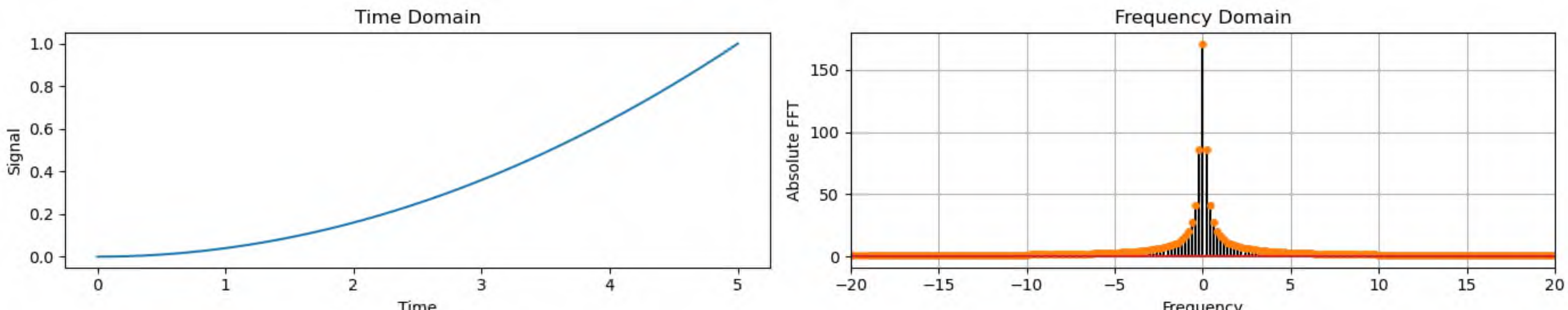
- We will observe that in the Frequency Domain:
    - **Seasonal** sine wave has components at -1.5 and +1.5
    - **Trend** has low frequency components (close to 0)
    - **Noise** has components across all frequencies.

FFT OF SIGNAL WITH TREND AND NOISE

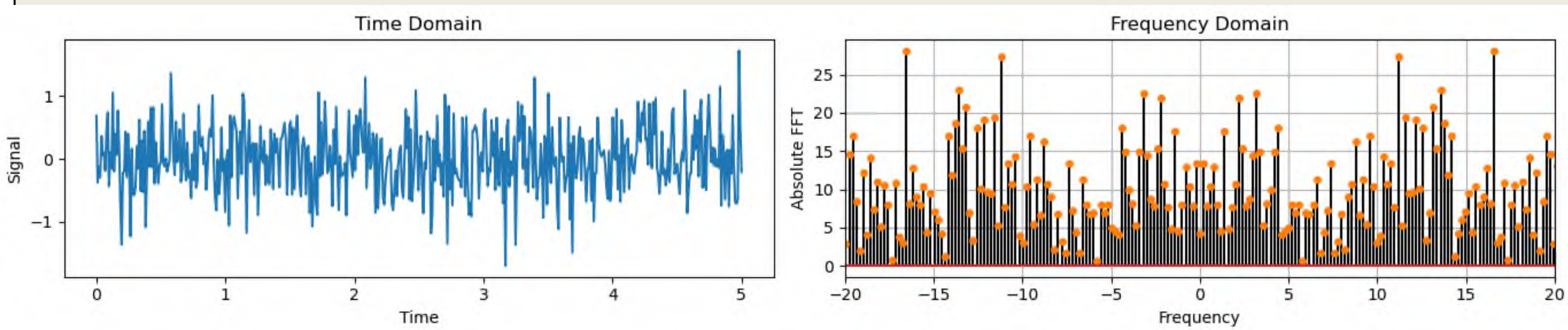
```
# Seasonality of signal
time = np.linspace(0, 5, 512)
freq = 1.5
y_sine = np.sin(2 * np.pi * freq * time)
plot_time_freq(time, y_sine)
```




```
# Extract trend
y_trend = (0.2 * time)**2
plot_time_freq(time, y_trend)
```




```
# Extract residuals
y_noise = 0.5 * np.random.randn(len(time))
plot_time_freq(time, y_noise)
```



- Consider a signal with sampling frequency and time vector that defines the length of our signal
- Fundamental frequency resolution $\frac{Sampling\ frequency}{\#\ of\ FFT\ points}$

```
Fs = 1234      # Sampling frequency in Hz
duration = 4   # seconds
t = np.arange(0, duration, 1/Fs)  # Time vector
print(f"Fundamental frequency resolution: {Fs/len(t):.2f}")
```

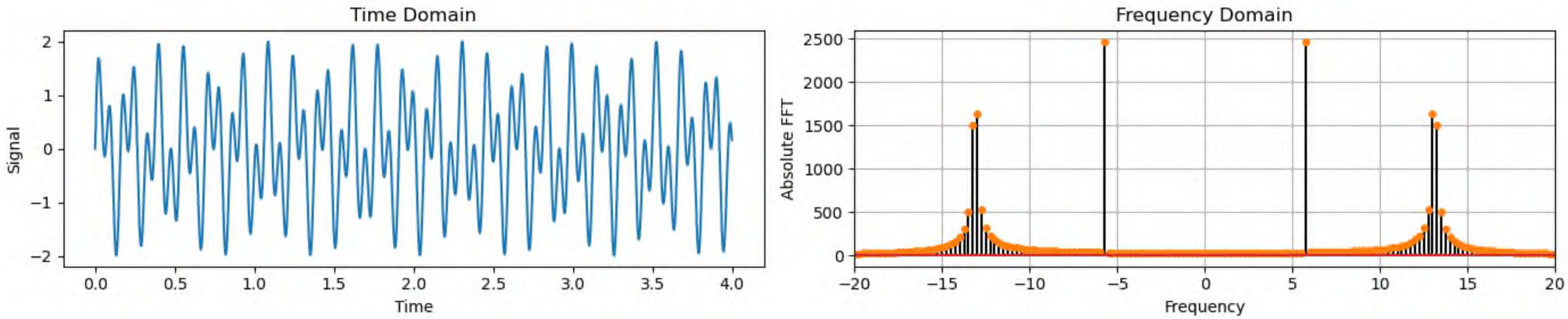
```
Fundamental frequency resolution: 0.25
```

- FFT looks at a window of the signal, of a specific length
  - If the signal's waveform doesn't complete a whole number of cycles within this window, the edges of the window essentially "cut off" part of the waveform
  - FFT assumes the signal outside this window is zero, which is rarely true for real signals
- This "cutting" makes the waveform look different from its true form, leading to inaccuracies in the frequency domain representation
- Spectral leakage occurs when the signal's frequency components do not align exactly with the FFT's frequency bins
  - the energy of the signal gets spread across multiple bins around the true frequency

Πανεπιστήμιο Κύπρου

- A demonstration example where we sum two sinusoids:
  - The first has a frequency multiple of the fundamental frequency resolution, the second doesn’t

```
freq1 = 5.75    # Frequencies of the 1st sinusoid in Hz
freq2 = 13.12   # Frequencies of the 2nd sinusoid in Hz

y_sine = np.sin(2 * np.pi * freq1 * t) + np.sin(2 * np.pi * freq2 * t)
plot_time_freq(t, y_sine)
```



- In the second component, we observe many small non-zero frequencies around the actual frequency, 13.12
  - i.e., spectal leakage.
  - Common phenomenon when analyzing real-world signals

Πανεπιστήμιο Κύπρου

- FT is **linear**, meaning that the transform of a sum of signals is the sum of their transforms
- For any two signals $f(t)$ and $g(t)$, and any constant a, b:
  - $\mathcal{F}(af(t) + bg(t)) = a\mathcal{F}(f(t)) + b\mathcal{F}(g(t))$
  - Allows for the analysis of complex signals by breaking them down into simpler components
- Shifting a signal in **time** results in a phase shift in its Fourier Transform
  - If $f(t - t_0)$ is a time-shifted version of $f(t)$, its FT is:
  - $\mathcal{F}\big(f(t - t_0)\big) = e^{-i\omega t_0}\mathcal{F}(f(t))$
  - time delays correspond to phase shifts in the frequency domain, without affecting the amplitude of the frequency components

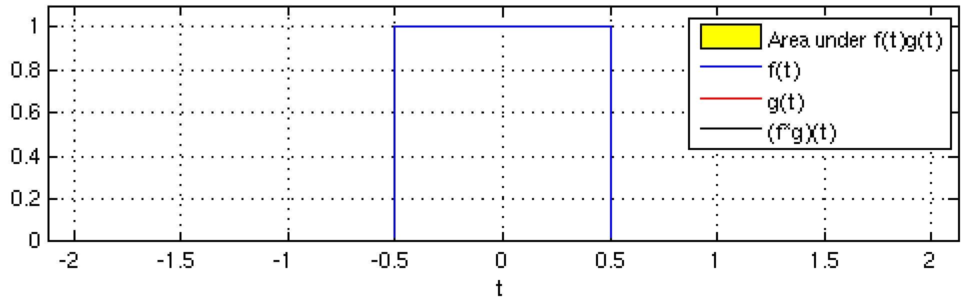
Πανεπιστήμιο Κύπρου

- Frequency shift: Modulating a signal by a sinusoid results in a shift in the frequency domain
- If g(t) $= e^{i2\pi f_0} f(t)$, then the FT of g(t) is:
  - $\mathcal{F}\big(g(t)\big) = \mathcal{F}(\omega - 2\pi f_0)$
  - Modulate communication signals to specific bands
- Scaling:
  - Let $f(at)$ be a scaled version of $f(t)$ then the FT is:
  - $\mathcal{F}\big(f(at)\big) = \frac{1}{|a|}\mathcal{F}(\frac{\omega}{a})$
  - indicates that compressing a signal in time domain expands its spectrum in the frequency domain, and vice versa

- Convolution
  - Informally, convolution represents how much one function f overlaps with another function g as it slides across the domain
  - The Fourier transform converts convolution in time domain into multiplication in the frequency domain
  - The convolution is defined as:

$$f(t) * g(t) = \int_{-\infty}^{\infty} f(t) g(t-\tau) d\tau$$

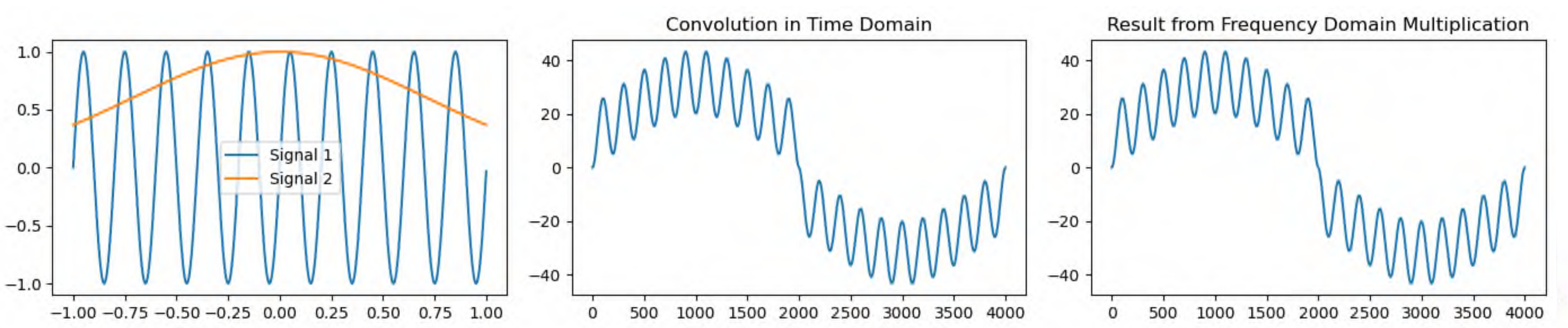
$$\mathcal{F}(f(t) * g(t)) = \mathcal{F}(f(t)) \cdot \mathcal{F}(g(t))$$



Πανεπιστήμιο Κύπρου

```
Fs = 1000  # Sampling frequency in Hz
T = 1  # Length of signal in seconds
t = np.arange(-T, T, 1/Fs)  # Time vector
y1 = np.sin(2 * np.pi * 5 * t) # first signal
y2 = np.exp(-t ** 2) # second signal

# convolution in the time domain
convolution_time = np.convolve(y1, y2, mode='full')

# Pad the signals with N-1 zeros to avoid circular convolution
y1_pad = np.pad(y1, (0, t.size-1), 'constant')
y2_pad = np.pad(y2, (0, t.size-1), 'constant')

# Compute the Fourier Transforms
Y1 = np.fft.fft(y1_pad)
Y2 = np.fft.fft(y2_pad)

# Multiplication in frequency domain and inverse transform
convolution_freq_domain = np.fft.ifft(Y1 * Y2)
```

```
plot_convolution(t, y1, y2, convolution_time, convolution_freq_domain)
```



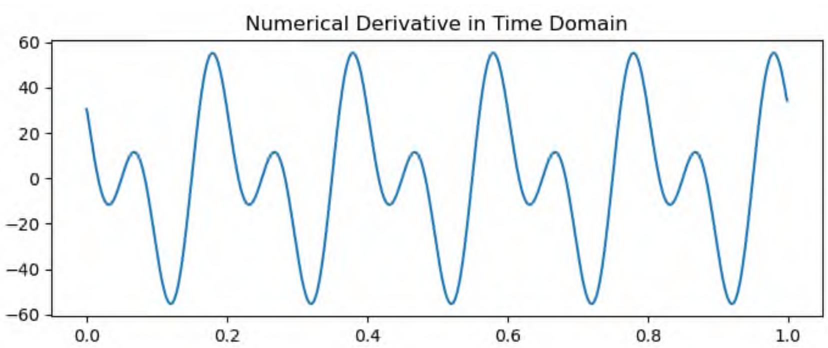
- Differentiation
  - The FT of the derivative of a function is proportional to the frequency times the FT of the function itself:
  - $\mathcal{F}\left(f'(t)\right) = i\omega\mathcal{F}\left(f(t)\right)$
  - Conversely, the inverse FT of $\frac{d^n\mathcal{F}(\omega)}{d\omega^n}$ is given by the multiplication of $f(t)$ by $(it)^n$, where n is the order of the differentiation

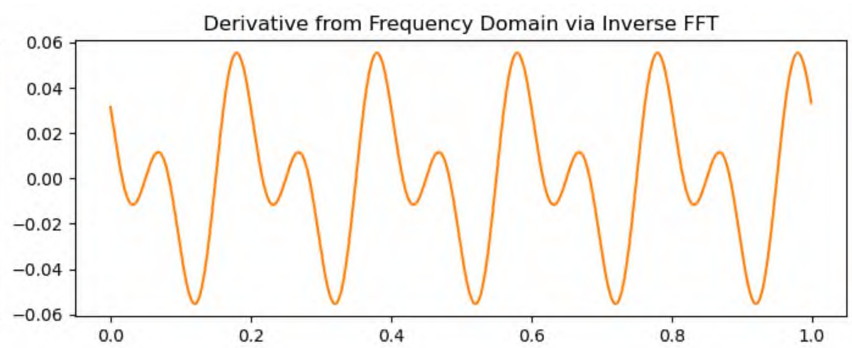
$$\mathcal{F}^{-1}\left(\frac{d^n\mathcal{F}(\omega)}{d\omega^n}\right) = (it)^n f(t)$$

DIFFERENTIATION EXAMPLE

```
Fs = 1000  # Sampling frequency in Hz
T = 1  # Length of signal in seconds
t = np.arange(0, T, 1/Fs)  # Time vector
y = np.sin(2 * np.pi * 5 * t) + 0.5 * np.cos(2 * np.pi * 10 * t) # Signal

dy_dt_numerical = np.gradient(y, t) # Numerical derivative

Y = np.fft.fft(y) # Fourier Transform of the signal
omega = 2 * np.pi * np.fft.fftfreq(t.size, T) # Angular frequency vector
dY_dt_frequency_domain = 1j * omega * Y # Differentiate in the frequency domain
dy_dt_from_frequency_domain = np.fft.ifft(dY_dt_frequency_domain) # Inverse FT
to get the derivative in the time domain
```

```
plot_derivative(t, dy_dt_numerical, dy_dt_from_frequency_domain)
```





- Integration
  - The integration of a signal in the time domain corresponds to a specific transformation in the frequency domain
  - FT of the integral of $f(t)$ is:

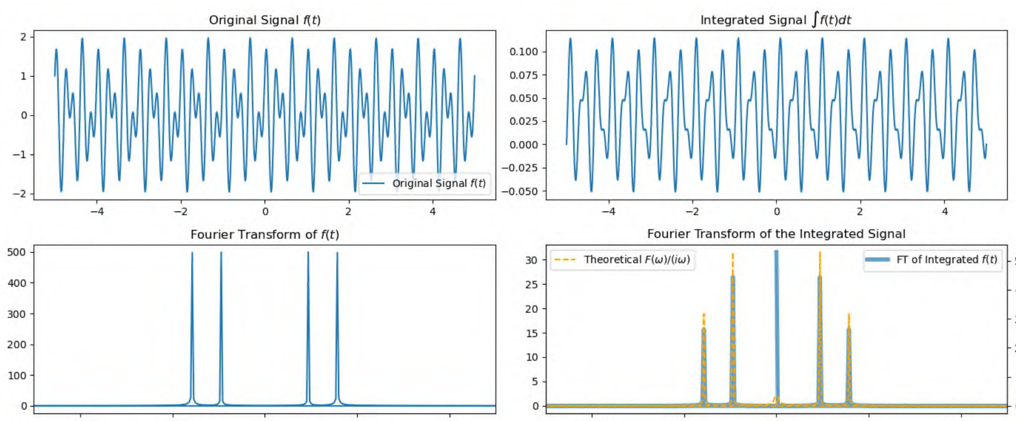
$$\mathcal{F}\left(\int_{-\infty}^{t} f(\tau)d\tau\right) = \frac{\mathcal{F}(\omega)}{i\omega} + \pi\mathcal{F}(0)\delta(\omega)$$

  - $\delta(\omega)$ is the Dirac delta function

INTEGRATION EXAMPLE

```
sampling_rate = 1000
T = 2.0 / sampling_rate
N = 1000
t = np.linspace(-5, 5, N)
f_t = np.sin(2 * np.pi * 5 * t) + np.cos(2 * np.pi * 3 * t) # Original signal
# Integrate f(t) in the time domain
integrated_f_t = cumulative_trapezoid(f_t, t, initial=0)
# Compute the Fourier Transform of the integrated signal
Integrated_F_w = np.fft.fft(integrated_f_t)
# Theoretical relationship
# Compute the Fourier Transform of the original signal
F_w = np.fft.fft(f_t)
omega = 2 * np.pi * np.fft.fftfreq(t.size, T)
# Note: To avoid division by zero at omega=0, we use np.where to handle the
omega=0 case separately.
# The delta function's contribution at omega=0 is theoretical and not directly
applicable in discrete FFT.
Theoretical_F_w = np.where(omega != 0, F_w / (1j * omega + 1e-10), 0) # Should
be equal to Integrated_F_w
```

```
plot_integration(t, f_t, integrated_f_t, F_w, Integrated_F_w, Theoretical_F_w)
```



- Parseval's theorem
  - Relates the total energy of a signal in the time domain and in the frequency domain
  - Energy is preserved in the Fourier transform
  - Mathematically:

$$\int_{-\infty}^{\infty} |f(t)|^2 dt = \int_{-\infty}^{\infty} |\mathcal{F}(\omega)|^2 d\omega$$

  - For discrete signals and their DFT, this can be approximated as:

$$\sum_{n=0}^{N-1} |x[n]|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X[K]|^2$$

Πανεπιστήμιο Κύπρου

ENERGY CONSERVATION EXAMPLE

```python
Fs = 1000  # Sampling frequency in Hz
T = 0.5  # Length of signal in seconds
t = np.arange(0, T, 1/Fs)  # Time vector

f_t = np.sin(2 * np.pi * 5 * t) + 0.5 * np.sin(2 * np.pi * 10 * t)  # Signal
F_w = np.fft.fft(f_t) # Fourier Transform

# Compute the energy in the time domain
energy_time_domain = np.sum(np.abs(f_t) ** 2)

# Compute the energy in the frequency domain
energy_freq_domain = np.sum(np.abs(F_w) ** 2) / t.size

print(f"Energy in the time domain: {energy_time_domain}")
print(f"Energy in the frequency domain: {energy_freq_domain}")
```

```
Energy in the time domain: 312.5
Energy in the frequency domain: 312.5
```

Πανεπιστήμιο Κύπρου

DIGITAL FILTERS

- In the context of signal processing and time series analysis, a **filter** is a tool used to modify or enhance a signal by selectively amplifying certain frequencies and attenuating others

- A **filter** processes a signal to remove unwanted components or features, such as noise, or to extract useful information from the signal

- **Filters** are characterized by their:
  - frequency response
  - transfer function

- The frequency response of a filter characterizes the filter's output in the frequency domain
- It describes how the amplitude and phase of the output signal vary across different frequencies of the input signal
- In other words, it shows how much each frequency component of the input signal is attenuated or amplified by the filter and how the phase of these components is shifted
- The frequency response is typically represented as a plot of the filter's gain (or attenuation) and phase shift as functions of frequency
- This response is crucial for understanding how the filter will affect a signal's spectral content
- There are different categories of filters based on the frequency response
    - **Low-pass filters** – allow frequencies below a certain cutoff to pass through while attenuating higher frequencies
    - **High-pass filters** – do the opposite of LPF
    - **Band-pass filters** – allow frequencies within a certain range to pass through
    - **Band-stop filters** – attenuate frequencies within a certain range

- The transfer function describes how the filter modifies the amplitude and phase of components of the input signal at different frequencies
- To define it, we must first introduce the Laplace transform and the Z-transform
- Laplace transform
  - Laplace transform of a function $f(t)$, defined for $t \geq 0$, is given by:

$$\mathrm{F}(s) = \int_0^\infty f(t)e^{-st}dt$$

  - where $s = \sigma + i\omega$ is a complex number
  - FT is a special case of Laplace transform with $s = j\omega$
  - Not all functions that have a Laplace transform will have a FT
  - FT is useful to analyze only the frequency content while Laplace is useful to analyze the overall system behavior, including stability and transient response

Πανεπιστήμιο Κύπρου

- Z-transform
  - Relationship between z-transform and DFT
  - The z-transform of a discrete-time signal $x[n]$ is defined as:

$$X(z) = \sum_{n=-\infty}^{\infty} x[n] z^{-n}$$

  - where z is a complex variable ($z = re^{i\theta}$)
  - DFT is a special case of z-transform evaluated on the unit circle in the complex plane, i.e., when $z = e^{i\omega}$
  - The z-transform encompasses a broader range of analysis, allowing for the examination of signals and systems inside and outside the unit circle, which is useful for stability analysis and filter design in the z-domain

- Transfer function for analog filters (continuous time) it is defined as the ratio of the output signal and the input signal of the filter in the Laplace domain:

$$H(s) = \frac{Y(s)}{X(s)}$$

- For digital filters (discrete time) the ratio is defined in the Z-domain:

$$H(z) = \frac{Y(z)}{X(z)}$$

To summarize:

- The frequency response describes how the filter responds to sinusoidal inputs across a range of frequencies
- The transfer function gives a more general mathematical model of the filter that applies to all inputs, not just sinusoidal inputs
- Transfer functions are described in the Laplace or Z-domain, providing a broader analysis toolset for system properties
- In contrast, the frequency response is a subset of the transfer function evaluated on the imaginary axis in the Laplace domain or on the unit circle in the Z-domain

- A plot that displays the transfer function of a system (a filter in our case)
- It displays the amplitude (usually in decibels, i.e., logarithmic units) of a system as a function of the frequency
- It also displays the phase of a system as a function of the frequency
- Let's create a Bode plot for a simple low-pass filter as an example
    - The transfer function for a first-order low-pass filter can be expressed as:

$$H(s) = \frac{\omega_c}{s + \omega_c}$$

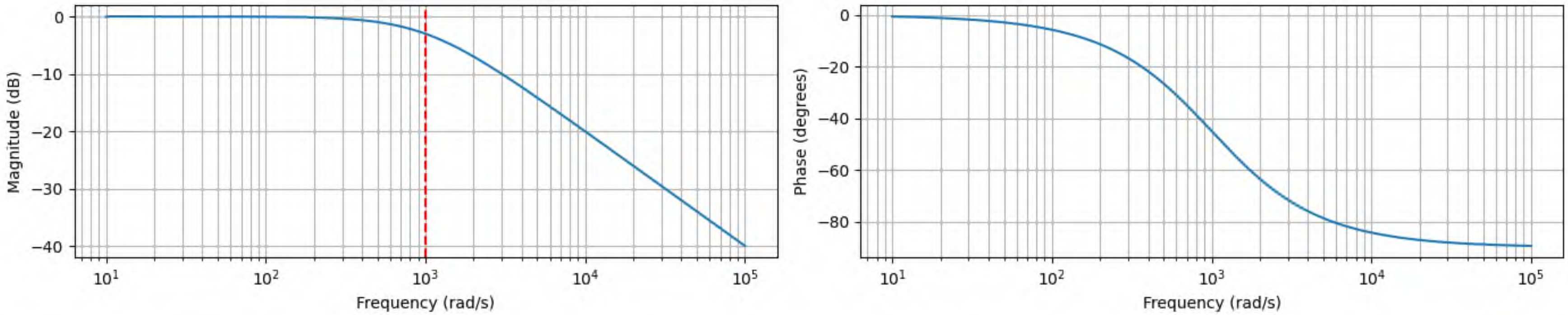
    - ω_c: cutoff frequency of the filter, frequency at which the output signal's power is reduced to half its input power

Πανεπιστήμιο Κύπρου

```
# Transfer function coefficients for H(s) = omega_c / (s + omega_c)
omega_c = 1000  # Cutoff frequency in rad/s
num = [omega_c]  # Numerator coefficients
den = [1, omega_c]  # Denominator coefficients

w, H = signal.freqs(num, den, worN=np.logspace(1, 5, 512)) # Compute frequency
response
mag = 20 * np.log10(abs(H))  # Convert magnitude to dB
phase = np.angle(H, deg=True)  # Phase in degrees
```

```
def make_bode_plot(w, mag, phase, omega_c=None, omega_signal=None):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 3))
    ax1.semilogx(w, mag)  # Bode magnitude plot
    ax1.set_ylabel('Magnitude (dB)')
    ax1.set_xlabel('Frequency (rad/s)')
    ax1.grid(which='both', axis='both')
    if omega_c != None:
        if type(omega_c) is list:
            for o in omega_c:
                ax1.axvline(o, color='red', linestyle='--') # Cutoff frequency
        else:
            ax1.axvline(omega_c, color='red', linestyle='--') # Cutoff frequency
    if omega_signal != None:
        ax1.axvline(omega_signal, color='tab:green', linestyle='--')
    ax2.semilogx(w, phase)  # Bode phase plot
    ax2.set_ylabel('Phase (degrees)')
    ax2.set_xlabel('Frequency (rad/s)')
    ax2.grid(which='both', axis='both')
    plt.tight_layout()
    plt.show()
```

BODE PLOT



Magnitude plot (left)

- Shows how the filter attenuates signals at different frequencies.
- Frequencies lower than the cutoff frequency ($\omega_c = 1000$ rad/s) are passed with little attenuation
- Frequencies higher than the cutoff frequency are increasingly attenuated

Phase plot (right)

- Illustrates the phase shift introduced by the filter across different frequencies
- For a low-pass filter, the phase shift goes from 0 degrees at low frequencies to -90 degrees at high frequencies
- The transition occurs smoothly across the frequency range, with a noticeable change around the cutoff frequency.

- As we mentioned earlier, a LPF passes signals with a frequency lower than a certain cutoff frequency
- LPFs provide a smoother form of a signal, removing short term fluctuations
- One common application is to get rid of the high-frequency components of the noise
- Moving averages are a type of low pass filters
- Representative LPFs include the Hann Window, the Tukey Filter, and the Butterworth Filter

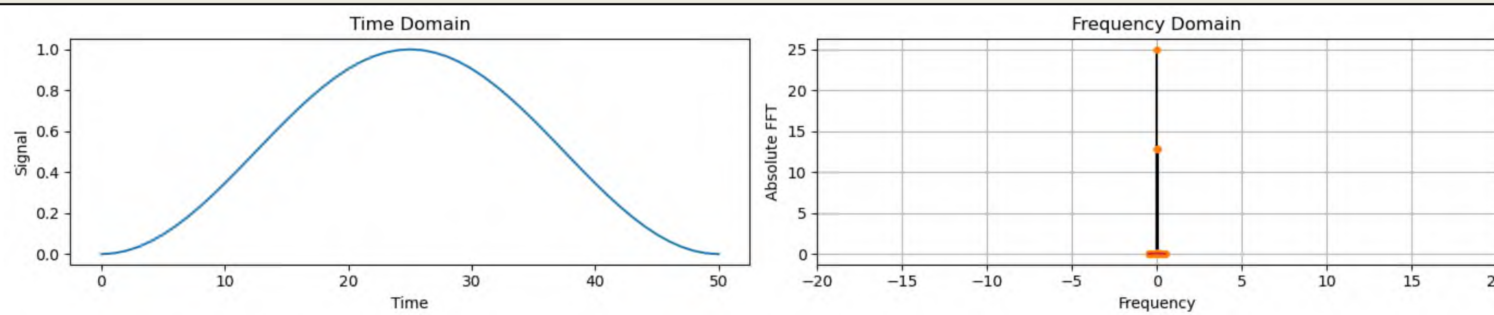
Hann Window

- The Hann window, named after Julius von Hann, is sometimes referred to as “Hanning” or as “raised cosine”
- The shape of the filter in time domain is one lobe of an elevated cosine function
- On the interval $n \in [0, N-1]$ the Hann window function is:

$$w(n) = 0.5\left[1 - \cos\left(\frac{2\pi n}{N-1}\right)\right] = sin^2\left(\frac{\pi n}{N-1}\right)$$

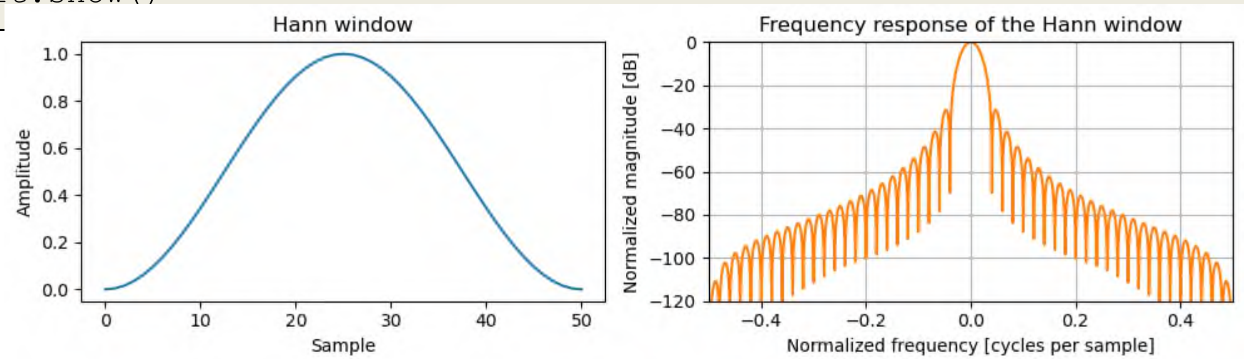
- let’s generate a Hann window of size 51 and plot the window and compute its frequency response using $w(n)$

```
# Hann window
window = signal.windows.hann(51)
plot_time_freq(np.arange(len(window)), window)
```



- Very compressed signal and thus apply transformation

```
# Hann plot
def plot_hann():
    window = signal.windows.hann(51)
    plt.figure(figsize = [10,3])
    plt.subplot(1,2,1)
    plt.plot(window)
    plt.title("Hann window")
    plt.ylabel("Amplitude")
    plt.xlabel("Sample")
    plt.subplot(1,2,2)
    A = np.fft.fft(window, 2048) / (len(window)/2.0)
    freq = np.linspace(-0.5, 0.5, len(A))
    response = np.abs(np.fft.fftshift(A / abs(A).max()))
    response = 20 * np.log10(np.maximum(response, 1e-10))
    plt.plot(freq, response, color='tab:orange')
    plt.axis([-0.5, 0.5, -120, 0])
    plt.title("Frequency response of the Hann window")
    plt.ylabel("Normalized magnitude [dB]")
    plt.xlabel("Normalized frequency [cycles per sample]")
    plt.grid()
    plt.tight_layout()
    plt.show()
```

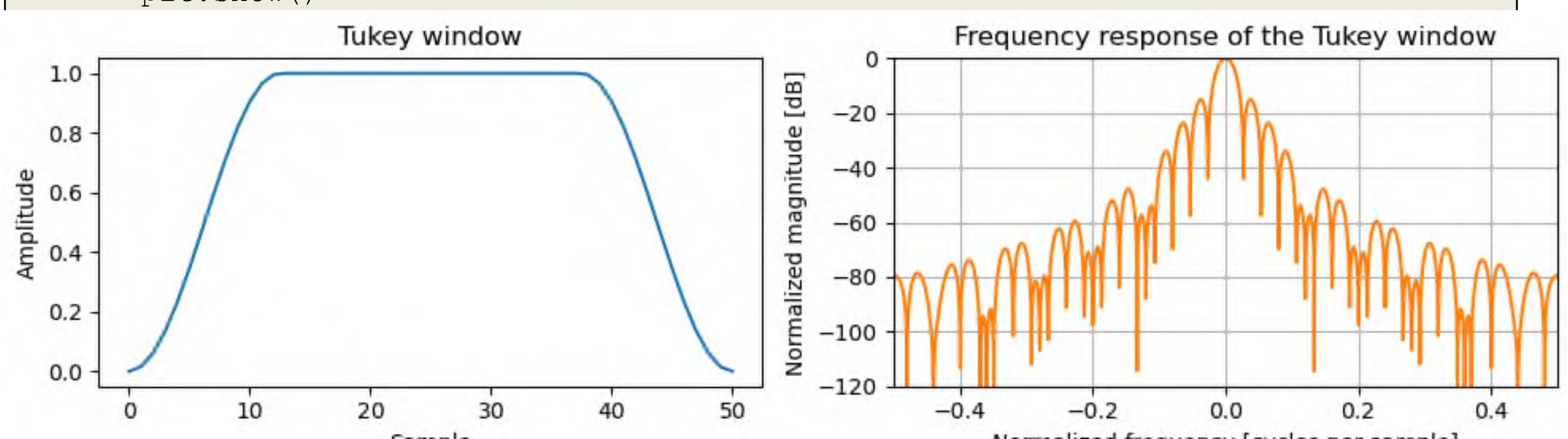


- Also known as the Tukey window or "tapered cosine window"
- Regarded as a cosine lobe of width $aN/2$ that is convolved with a rectangular window of width $\left(1-\frac{a}{2}\right)N$
- The filter in the time domain is given by:

$$w(n)=\begin{cases}\frac{1}{2}\left[1+\cos\left(\pi\left(\frac{2\pi}{a(N-1)}\right)\right)\right] & 0\le n<\frac{a(N-1)}{2}\\ 1 & \frac{a(N-1)}{2}\le n\le (N-1)(1-\frac{a}{2})\\ \frac{1}{2}\left[1+\cos\left(\pi\left(\frac{2\pi}{a(N-1)}-\frac{2}{a}+1\right)\right)\right] & (N-1)(1-\frac{a}{2})<n\le (N-1)\end{cases}$$

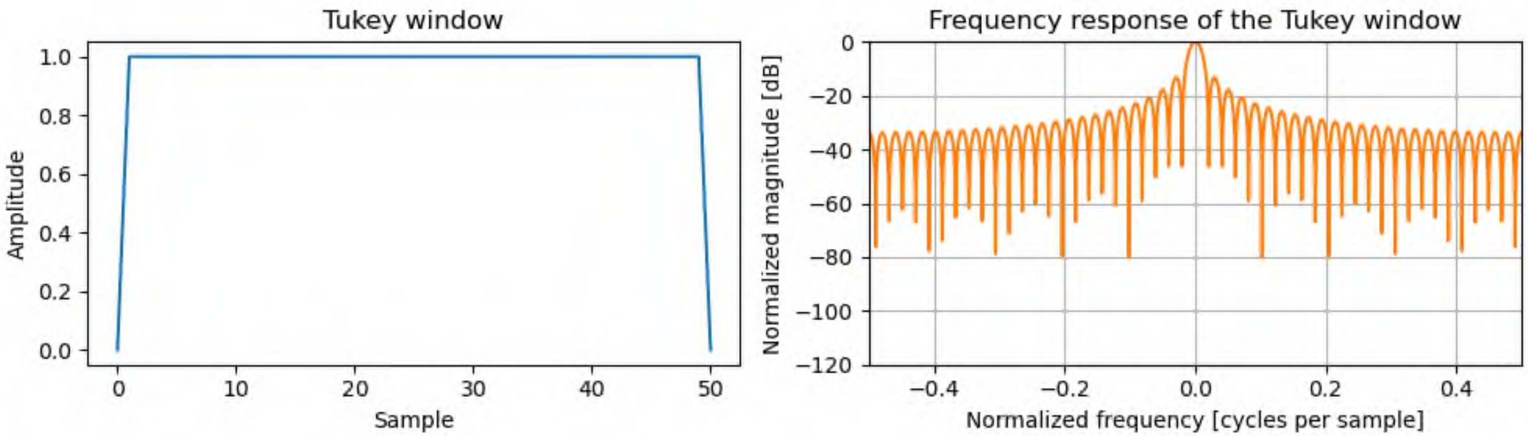
Tukey filter

```
# Tukey filter
def plot_tukey(alpha):
    window = signal.windows.tukey(51, alpha=alpha)
    plt.figure(figsize = [10,3])
    plt.subplot(1,2,1)
    plt.plot(window)
    plt.title("Tukey window")
    plt.ylabel("Amplitude")
    plt.xlabel("Sample")
    plt.subplot(1,2,2)
    A = np.fft.fft(window, 2048) / (len(window)/2.0)
    freq = np.linspace(-0.5, 0.5, len(A))
    response = 20 * np.log10(np.abs(np.fft.fftshift(A / abs(A).max())))
    plt.plot(freq, response, color='tab:orange')
    plt.axis([-0.5, 0.5, -120, 0])
    plt.title("Frequency response of the Tukey window")
    plt.ylabel("Normalized magnitude [dB]")
    plt.xlabel("Normalized frequency [cycles per sample]")
    plt.grid()
    plt.tight_layout()
    plt.show()
```



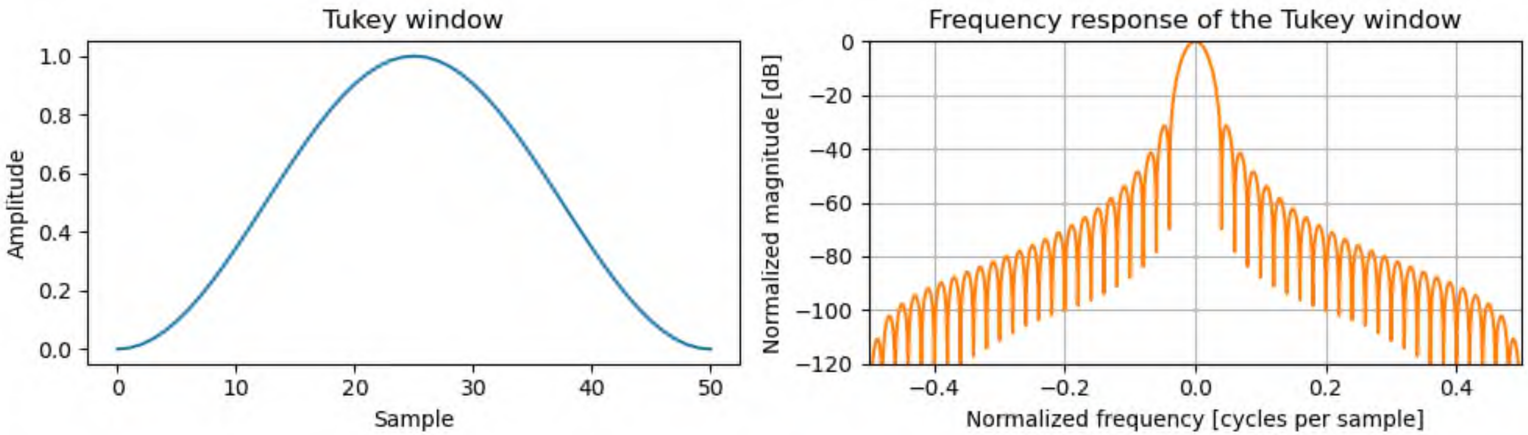
- When $a \rightarrow 0$, Tukey converges to a rectangular window

```
# Tukey filter
plot_tukey(1e-10)
```



- When $a = 1$, Tukey becomes a Hann window

```
# Tukey filter
plot_tukey(1)
```



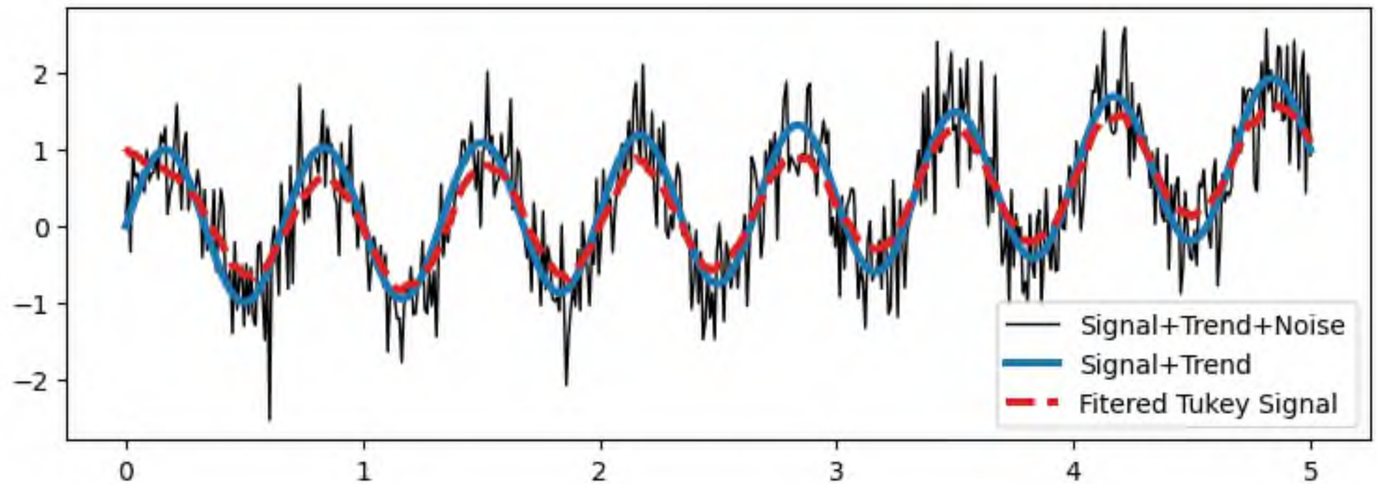
- Lets apply the filter to a noise signal by mutiplying the signal and the filter in the frequency domain

```
# Create a function to show the results
def filter_plot(time, y_noisy, y_clean, y_filtered, legend_names, alpha=1):
    plt.figure(figsize=[9,3])
    plt.plot(time, y_noisy, 'k', lw=1)
    plt.plot(time, y_clean, 'tab:blue', lw=3)
    plt.plot(time, np.real(y_filtered), 'tab:red', linestyle='--', lw=3,
alpha=alpha)
    plt.legend(legend_names);
```

```
# Create the signal
time = np.linspace(0, 5, 512)
freq = 1.5
y_sine = np.sin(2 * np.pi * freq * time)
y_trend = (0.2 * time)**2
y_noise = 0.5 * np.random.randn(len(time))
noisy_signal = y_sine + y_trend + y_noise
```

- the output should show a reduction in the high frequency components (effect of both Hann and Tukey)
- Use IFFT to map the output back to the time domain

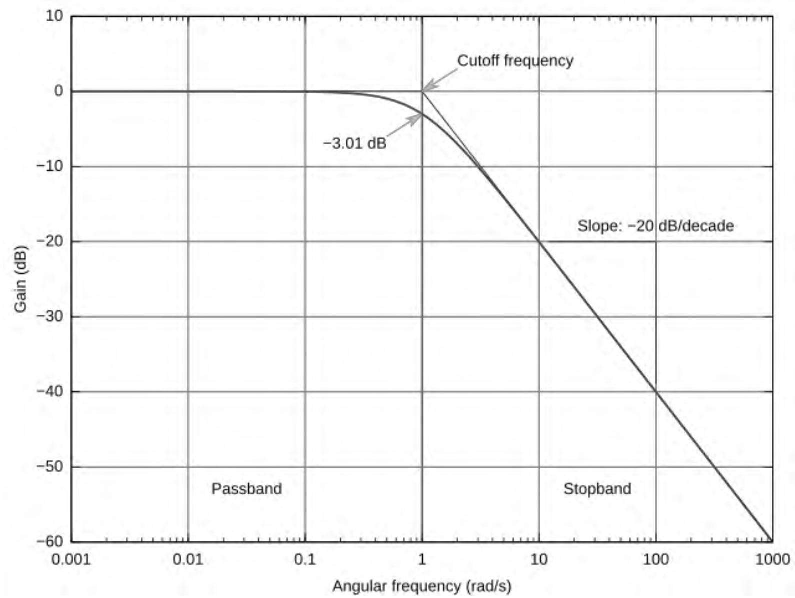
```
# Set the filter parameters
alpha=0.1
div_factor = 16 # use powers of 2
win_len = int(len(time) / div_factor)
print(f"Window length: {win_len}")
# Compute window
window = signal.windows.tukey(win_len, alpha=alpha)
# Compute frequency response
response = np.fft.fft(window, len(time))
response = np.abs(response / abs(response).max())
# Apply filter
Y = (np.fft.fft(noisy_signal))
y_tukey = np.fft.ifft(Y*response)
filter_plot(time, noisy_signal, y_sine+y_trend, y_tukey,
['Signal+Trend+Noise','Signal+Trend','Fitered Tukey Signal'])
```



- “Filtered Signal” matches very closely the “Signal + Trend”
- Most of the noise has been removed

Πανεπιστήμιο Κύπρου

- The Butterworth filter is designed to have a frequency response as flat as possible in the passband
- The passband is the part of the filter that allows certain the frequencies (left of the green line in the figure below)
- Conversely, the stopband is the part of the filter that rejects/dampens the frequency (right of the green line)
- The Butterworth filter avoids ripples both in the passband and in the stopband, providing a smooth transition between these regions



Πανεπιστήμιο Κύπρου

- Contrarily to the two window-based filters that we have just seen (Hann and Tukey), the Butterworth filter is defined directly in the frequency domain
- As such, it does not have a proper “shape” that we can visualize in the time domain
- Therefore, we will use the Bode plot, which is the tool for visualizing a frequency response
- The magnitude response of an N-th order Butterworth low-pass filter can be described by the following equation:

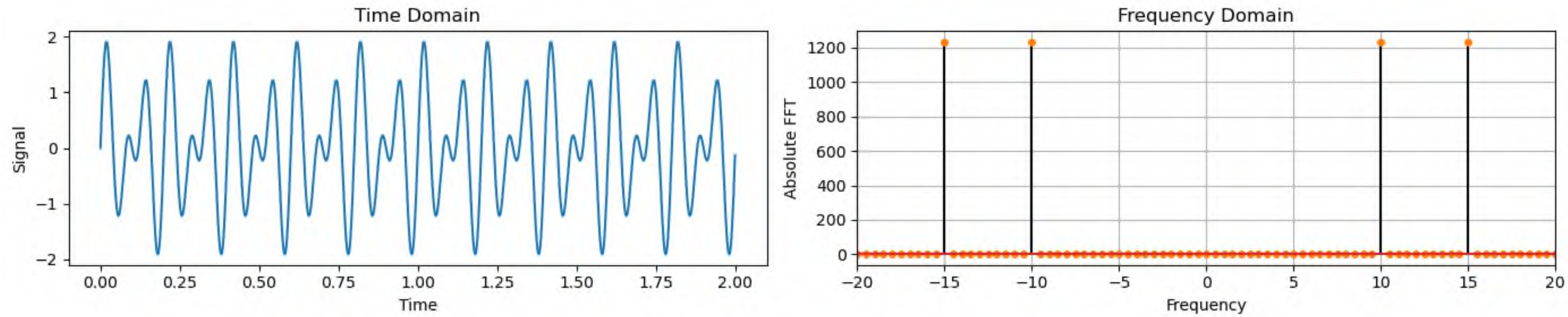
$$|H(\omega)| = \frac{1}{\sqrt{1 + \left(\frac{\omega}{\omega_c}\right)^{2N}}}$$

- where $|H(\omega)|$ is the magnitude of the frequency response of the filter, ω is the frequency of the input signal, ω_c the cutoff frequency, N determines the steepness of the filter’s roll-off beyond the cutoff frequency

Πανεπιστήμιο Κύπρου

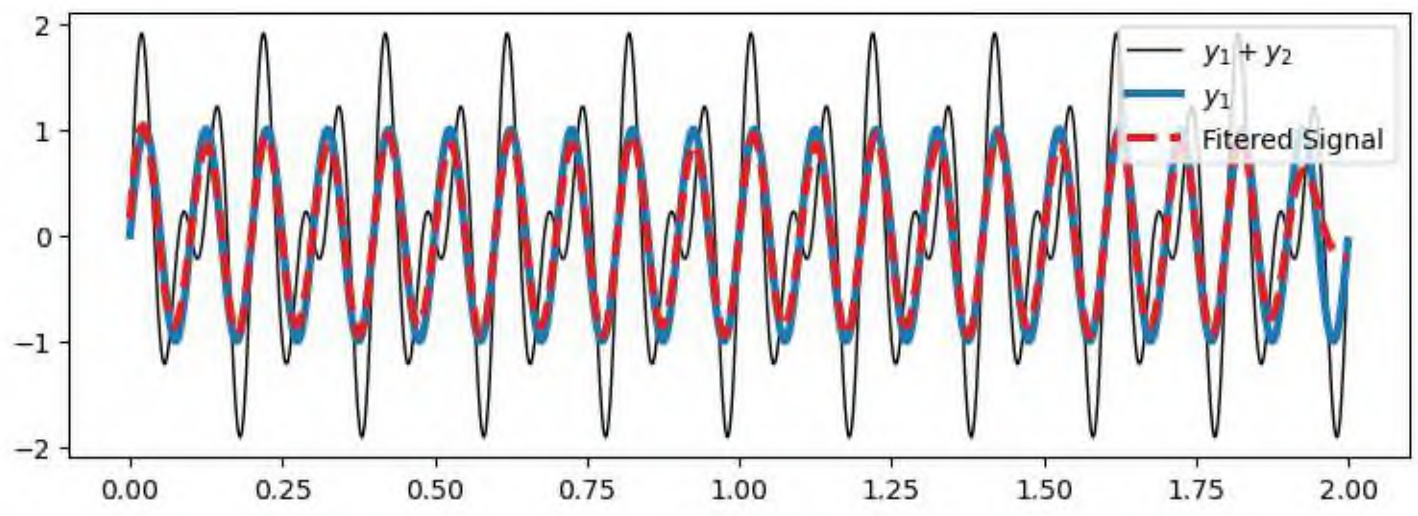
- Frequencies much lower than ω_c pass through with little attenuation
- Frequencies much higher than ω_c are significantly attenuated
- The transition between the passband and the stopband is smooth
- The steepness of this transition increasing with the filter order N
- Example
  - Let's consider a signal given by the sum of two sinusoids
  - The first sinusoid $y_1$has a frequency $f_1 = 10$Hz
  - The second sinusoid $y_2$has a frequency $f_2 = 15$Hz

```
Fs = 1234      # Sampling frequency in Hz
duration = 2   # seconds
t = np.arange(0, duration, 1/Fs)  # Time vector
freq1, freq2 = 10, 15
y1 = np.sin(2 * np.pi * freq1 * t)
y2 = np.sin(2 * np.pi * freq2 * t)
y_12 = y1 + y2
plot_time_freq(t, y_12)
```



- We want to design a low-pass filter to remove y_2, the 15Hz component, from the signal.
- We can use a LPF with cut frequency $f_c = 12\text{Hz}$
- Before, we applied the filter by multiplying the FFT of the filter and the signal and then computed the inverse
- We could use the `filtfilt` function, which directly applies a digital filter to a signal

```
freq_c = 12
# Numerator (B) and denominator (A) polynomials of the filter
B, A = signal.butter(N=6, Wn=freq_c, btype='lowpass', analog=False, fs=1234)
# Apply the filter
y_low_butter = signal.filtfilt(B, A, y_12)
# plot
filter_plot(t, y_12, y1, y_low_butter,['$y_1 + y_2$','$y_1$','Fitered Signal'])
```



- We see that the 10Hz component, y_1, is recovered in the filtered signal
- To remove the noise from the noisy signal we previously decompose (sinusoid + trend + noise)
- We also specified the cutoff frequency f_c in Hertz and we can equivalently express the cutoff as an angular frequency ω_c

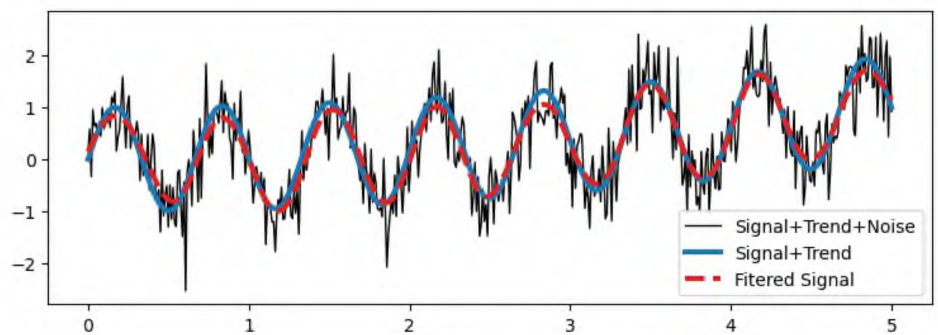
Πανεπιστήμιο Κύπρου

- Compute to what angular frequency ω the sinusoid's frequency corresponds to

```
freq = 1.5 # Frequency in Hz
time = np.linspace(0, 5, 512) # Time vector
# Compute the sampling resolution
fs = 1/(time[1]-time[0])
# Compute the Nyquist frequency
nyq = 0.5 * fs
# Compute the angular frequency
omega = freq / nyq
print(f" ω = {omega:.3f}")
```

- To allow the sinusoid frequency $\omega = 0.029$ and cut the higher ones, we can set $\omega_c = 0.05$

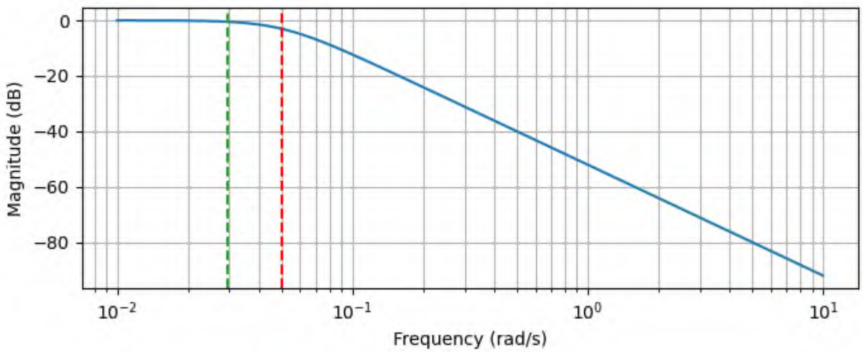
```
omega_c = 0.05
B, A = signal.butter(N=2, Wn=omega_c, btype='lowpass')
# Apply the filter
y_low_butter = signal.filtfilt(B, A, noisy_signal)
# plot
filter_plot(time, noisy_signal, y_sine+y_trend, y_low_butter,
            ['Signal+Trend+Noise','Signal+Trend','Fitered Signal'])
```

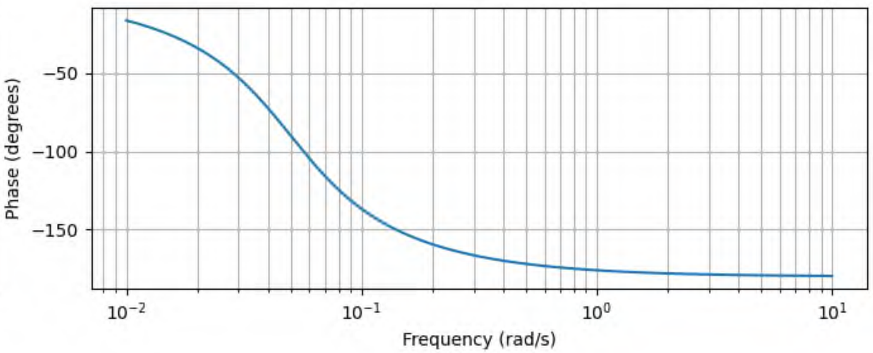


Πανεπιστήμιο Κύπρου

- Finally, we compute the Bode plot to see the shape of the LPF
- In green, we plot the frequency of the signal
- In red, we plot the cutoff frequency of the filter

```
B, A = signal.butter(N=2, Wn=omega_c, btype='lowpass', analog=True) # For the plot, we need the analog response
w, H = signal.freqs(B, A, worN=np.logspace(-2, 1, 512)) # Compute frequency response
magnitude = 20 * np.log10(abs(H))  # Convert magnitude to dB
phase = np.angle(H, deg=True)  # Phase in degrees

make_bode_plot(w, magnitude, phase, omega_c, omega)
```





Πανεπιστήμιο Κύπρου

HIGH PASS FILTERS (HPFS)

- Allows signals with frequencies higher than a certain cutoff frequency to pass through, while attenuating (reducing) signals with frequencies below the cutoff

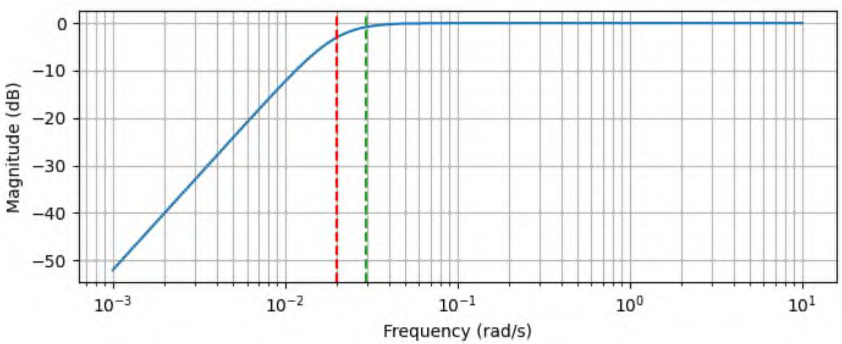
- LPFs are used to smooth out signals, remove noise, and preserve the basic shape of the signal without the details provided by high-frequency components

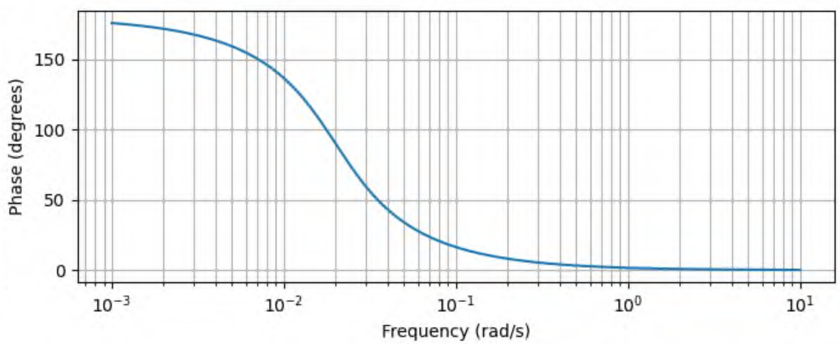
- Conversely, HPFs are often used to enhance or isolate quick changes in signals (e.g., edges in images or high-frequency sounds in audio)

- One application of HPFs is to filter out low frequency components and get rid of the trend

BUTTERWORTH FILTER AS A HIGH PASS FILTER

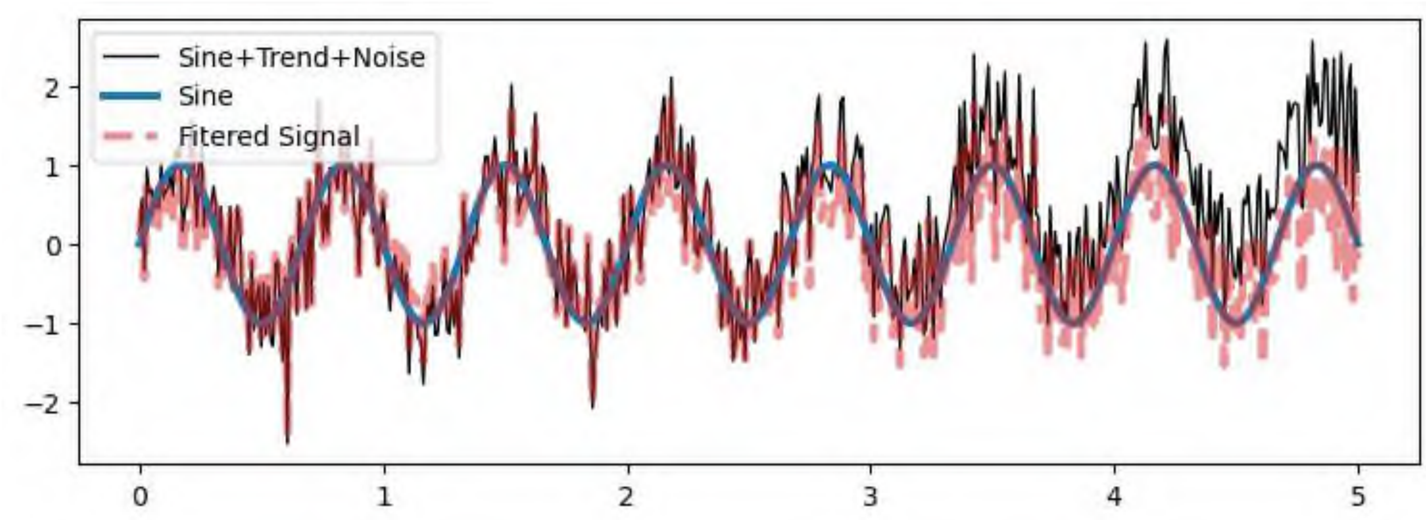
```
# High Pass Butterworth filter
N   = 2     # Filter order
omega_c = 0.02 # Cutoff frequency
B, A = signal.butter(N, omega_c, btype='highpass', output='ba', analog=True)

w, H = signal.freqs(B, A, worN=np.logspace(-3, 1, 512)) # Compute frequency
response
magnitude = 20 * np.log10(abs(H))  # Convert magnitude to dB
phase = np.angle(H, deg=True)  # Phase in degrees

make_bode_plot(w, magnitude, phase, omega_c, omega)
```





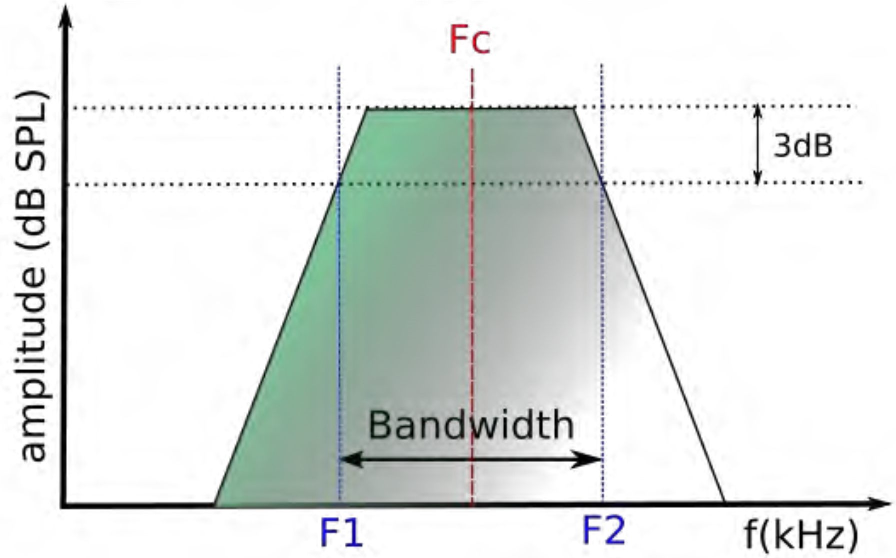
- The magnitude plot is flipped left-to-right
- The phase plot is shifted upwards, but the shape is unchanged

```
B, A = signal.butter(N, omega_c, btype = 'highpass', output='ba')

# Apply the filter
y_high_butter = signal.filtfilt(B, A, noisy_signal)

# plot
filter_plot(time, noisy_signal, y_sine, y_high_butter,
            ['Sine+Trend+Noise','Sine','Fitered Signal'], alpha=0.5)
```

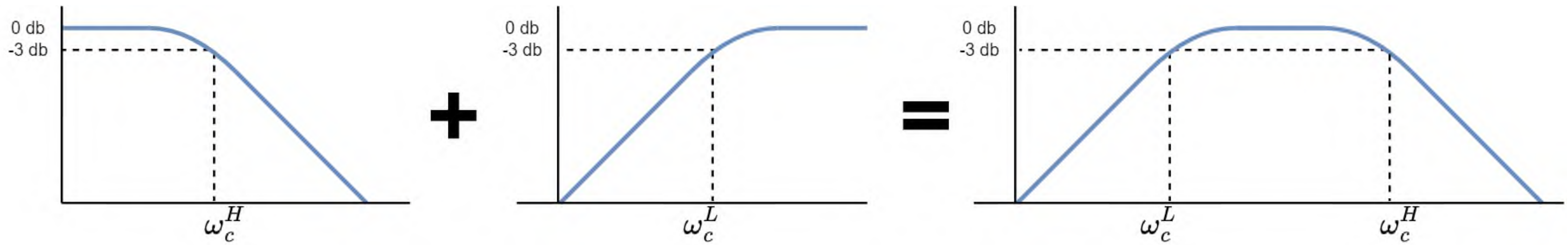


- The trend has been removed

- A band pass filter (BPF) passes frequencies within a certain range and rejects frequencies outside that range

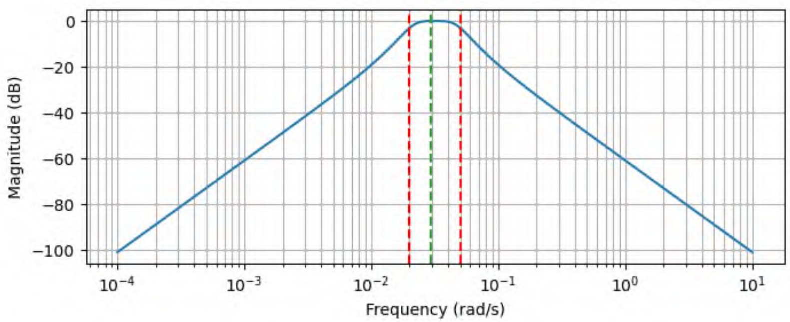


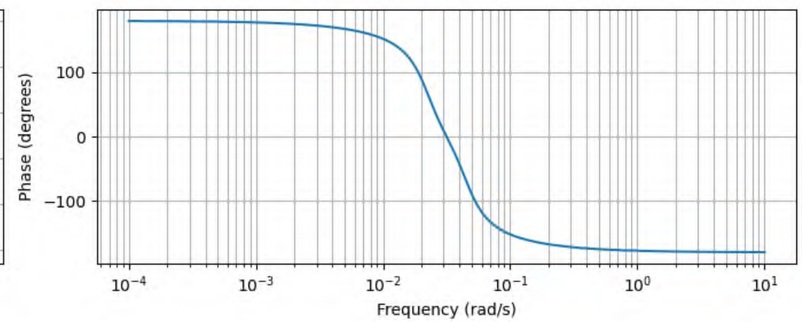
- A BPF can be seen as a combination of a high pass and low pass filters that remove both the high and low frequency components of a signal



Implement BPF using a LPF and HPF Butterworth

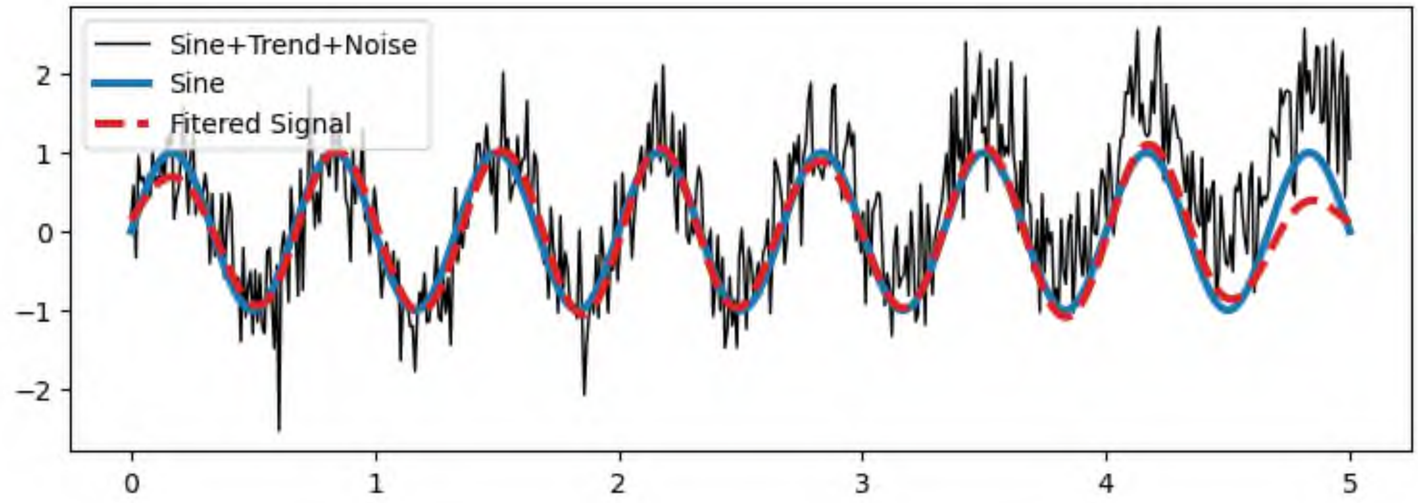
```
# Band Pass Butterworth filter
N  = 2    # Filter order
omega_c = [0.02, 0.05] # Cutoff frequencies
B, A = signal.butter(N, omega_c, btype = 'bandpass', output='ba', analog=True)

w, H = signal.freqs(B, A, worN=np.logspace(-4, 1, 512)) # Compute frequency
response
magnitude = 20 * np.log10(abs(H))  # Convert magnitude to dB
phase = np.angle(H, deg=True)  # Phase in degrees

make_bode_plot(w, magnitude, phase, omega_c, omega)
```





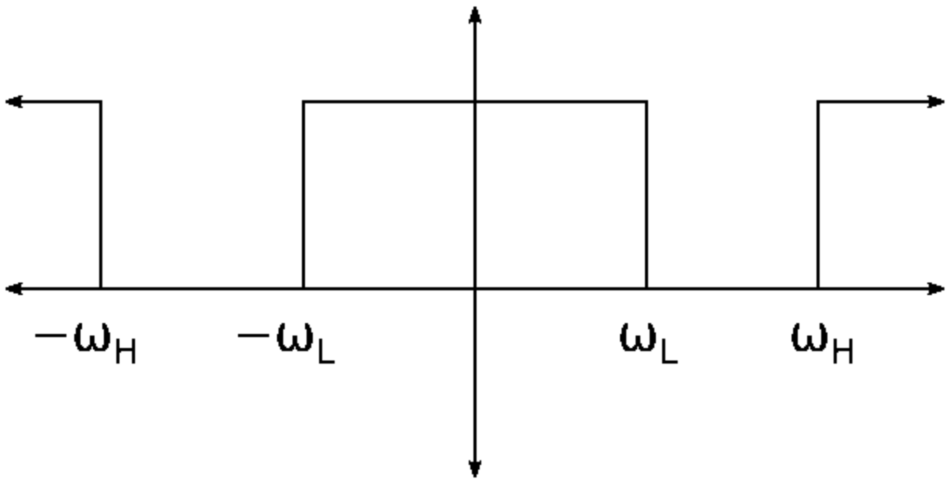
Πανεπιστήμιο Κύπρου

```
B, A = signal.butter(N, omega_c, btype = 'bandpass', output='ba')

# Apply the filter
y_band_butter = signal.filtfilt(B, A, noisy_signal)

# plot
filter_plot(time, noisy_signal, y_sine, y_band_butter,
            ['Sine+Trend+Noise','Sine','Fitered Signal'])
```



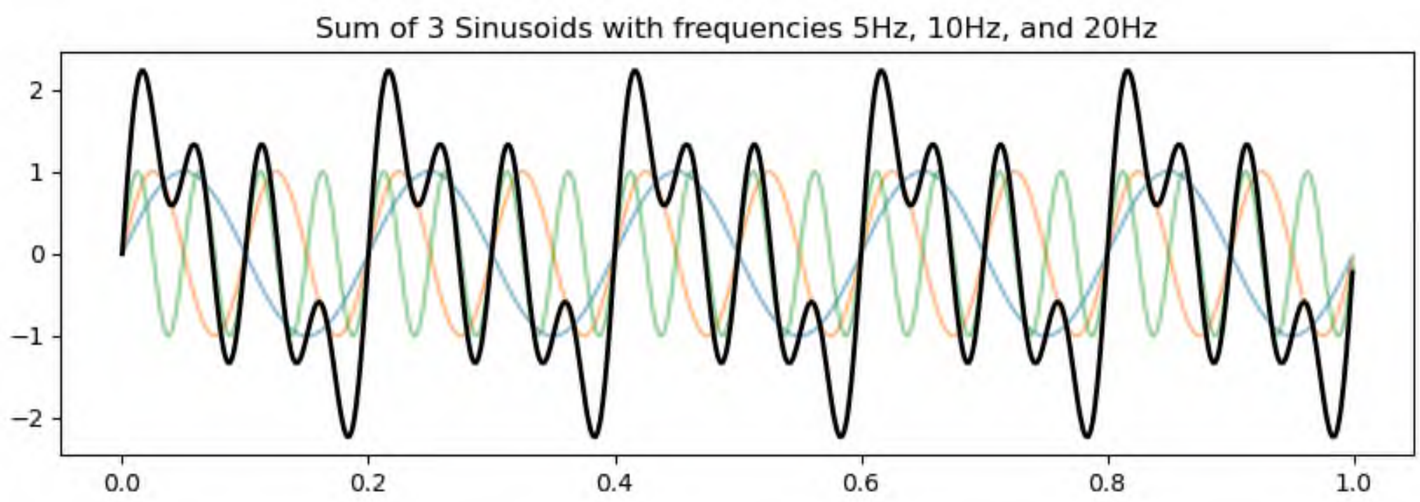
- Observe that the original signal has been de-trended, and the noise has been removed

- A band-stop filter (BSF) passes most filters unaltered, but attenuates those in a specific range to very low levels



- We use an example with 3 sinusoids

```
fs = 1000  # Sampling frequency in Hz
f1, f2, f3 = 5, 10, 20  # Frequencies of the three sinusoids in Hz
duration = 1  # seconds
t = np.arange(0, duration, 1/fs)  # Time vector

sinusoid1 = np.sin(2 * np.pi * f1 * t)
sinusoid2 = np.sin(2 * np.pi * f2 * t)
sinusoid3 = np.sin(2 * np.pi * f3 * t)

Y = sinusoid1 + sinusoid2 + sinusoid3 # Combined signal

plt.figure(figsize=(10, 3))
plt.plot(t, sinusoid1, alpha=0.5)
plt.plot(t, sinusoid2, alpha=0.5)
plt.plot(t, sinusoid3, alpha=0.5)
plt.plot(t, Y, 'k', linewidth=2)
plt.title(f'Sum of 3 Sinusoids with frequencies {f1}Hz, {f2}Hz, and {f3}Hz');
```

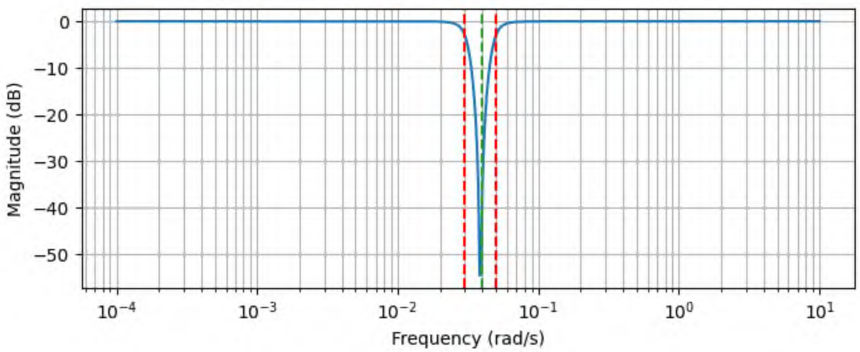


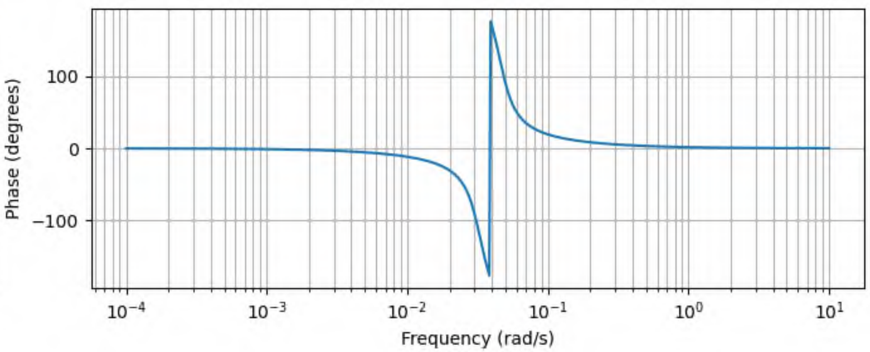
BAND-STOP FILTERS

- We use an example with 3 sinusoids

```
# Filter parameters to reject the third sinusoid
lowcut = f3 - 5   # Just below the third frequency
highcut = f3 + 5  # Just above the third frequency
# Scale the values by the sampling frequency
nyq = 0.5 * fs
omega_c_low = lowcut / nyq
omega_c_high = highcut / nyq
omega = f3 / nyq
```

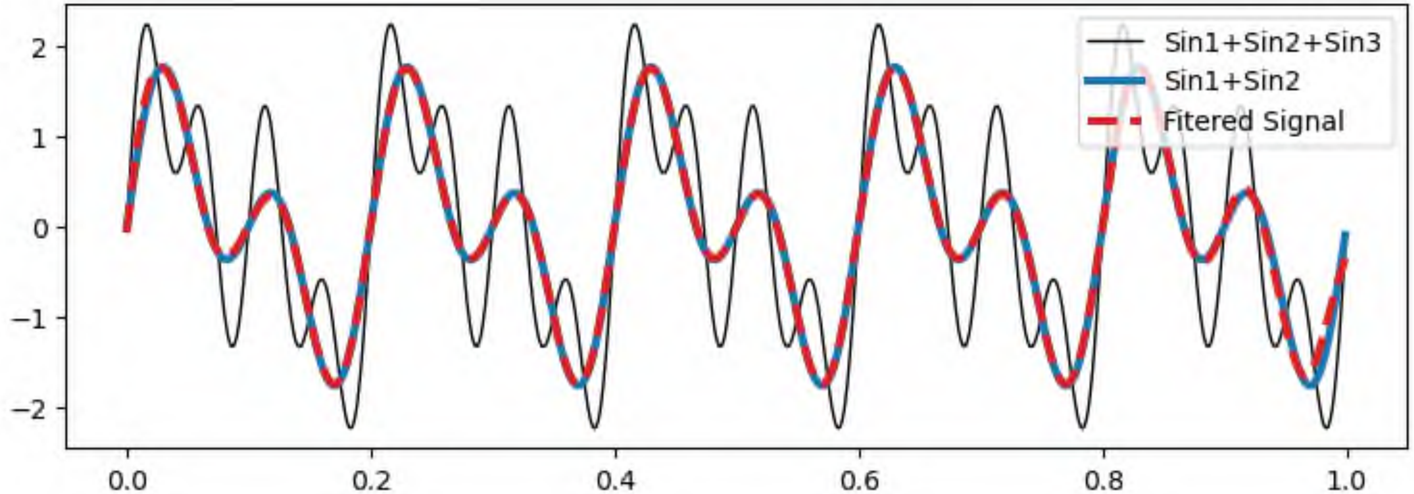
```
# Band Stop Butterworth filter
N  = 2    # Filter order
omega_c = [omega_c_low, omega_c_high] # Cutoff frequencies
B, A = signal.butter(N, omega_c, btype = 'bandstop', output='ba', analog=True)
w, H = signal.freqs(B, A, worN=np.logspace(-4, 1, 512)) # Compute frequency
response
magnitude = 20 * np.log10(abs(H))  # Convert magnitude to dB
phase = np.angle(H, deg=True)  # Phase in degrees
make_bode_plot(w, magnitude, phase, omega_c, omega)
```





Κυπρου

```python
B, A = signal.butter(N, omega_c, btype = 'bandstop', output='ba')

# Apply the filter
y_stop_butter = signal.filtfilt(B, A, Y)

# plot
filter_plot(t, Y, sinusoid1+sinusoid2, y_stop_butter,
            ['Sin1+Sin2+Sin3','Sin1+Sin2','Fitered Signal'])
```



- The effect of the third sinusoid has been completely removed from the signal
- An application of this filter is to remove a specific seasonality from the data

- Future values can be predicted using Fourier analysis of previous values
- Need to decomposing data into its frequency components, eliminating the trend, and then reconstructing the signal to predict future values
- We will define a function `fourierPrediction(y, n_predict, n_harm)` for this purpose
- The function returns `n_predict` future points of the original time series (with the linear trend) reconstructed using the specified number of harmonics `n_harm`

Πανεπιστήμιο Κύπρου

1. **Define the number of harmonics**
   - Harmonics are sine and cosine functions with frequencies that are integer multiples of a fundamental frequency
   - The input parameter n_harm specifies the number of harmonics used in the Fourier series expansion
   - By using multiple harmonics, the model can approximate the original time series more accurately
   - Using too many harmonics might overfit the data as they will start to model the noise
2. **Trend removal**
   - The function first computes a linear trend of the time series y using np.polyfit(t, y, 1), which fits a first-degree polynomial to the data, i.e., a straight line with form $\beta_1 t + \beta_0$, where β_1 is the slope (stored in p[0]) and β_0 is intercept (stored in p[1])
   - This trend is then subtracted from the original time series to obtain a detrended series y_notrend
   - Detrending is crucial for focusing the Fourier analysis on the cyclical components of the time series without the influence of the underlying trend
3. **Fourier transform**
   - The detrended time series is transformed into the frequency domain using the FFT with np.fft.fft(y_notrend)
   - The FFT algorithm computes the DFT, which represents the original time series as a sum of cosine and sine waves with different frequencies and amplitudes

Πανεπιστήμιο Κύπρου

4. **Frequency identification**
   - Use np.fft.fftfreq(n) to generate an array of frequencies associated with the components of the FFT
   - These frequencies are needed for reconstructing the signal later
5. **Sorting indexes by largest frequency components**
   - The indexes of the frequency components are sorted according to their magnitude
   - In this way, the most important frequency components of the signal (i.e., those that contribute the most to its shape) will come first
6. **Signal reconstruction and prediction**
   - The function reconstructs the time series (and extends it to predict future values) by summing the first 1 + n_harm * 2 sorted harmonic components
   - The * 2 is there because each harmonic has a positive and a negative frequency component in the FFT output
   - Each harmonic is defined by the amplitude (amp), frequency (f[i]), and phase (phase) given by the FFT
   - The reconstructed signal at each time point t is the sum of these harmonics, each represented by amp * np.cos(2 * np.pi * f[i] * t + phase)
7. **Adding back the trend**
   - the linear trend previously removed is added back to the reconstructed signal

Πανεπιστήμιο Κύπρου

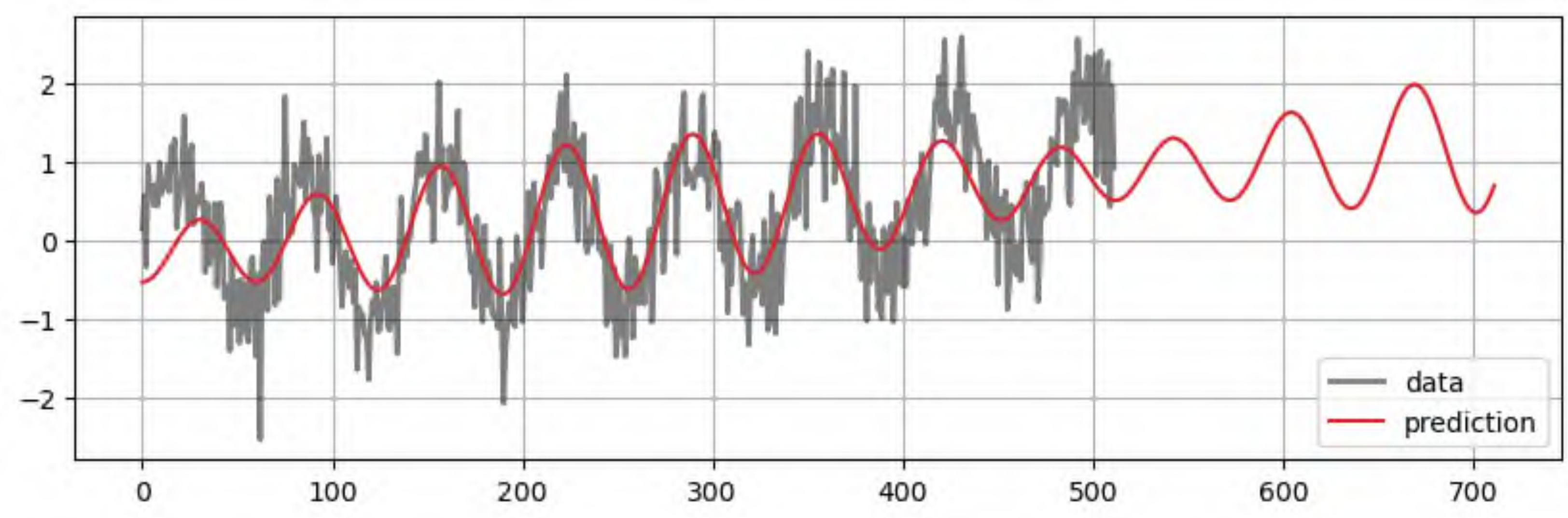
```
def fourierPrediction(y, n_predict, n_harm = 5):
    n = y.size                              # length of the time series
    t = np.arange(0, n)                     # time vector
    p = np.polyfit(t, y, 1)                 # find linear trend in x
    y_notrend = y - p[0] * t - p[1]         # detrended x
    y_freqdom = np.fft.fft(y_notrend)       # detrended x in frequency domain
    f = np.fft.fftfreq(n)                   # frequencies

    # Sort indexes by largest frequency components
    indexes = np.argsort(np.absolute(y_freqdom))[::-1]

    t = np.arange(0, n + n_predict)
    restored_sig = np.zeros(t.size)
    for i in indexes[:1 + n_harm * 2]:
        amp = np.absolute(y_freqdom[i]) / n    # amplitude
        phase = np.angle(y_freqdom[i])         # phase
        restored_sig += amp * np.cos(2 * np.pi * f[i] * t + phase)
    return restored_sig + p[0] * t + p[1] # add back the trend
```

```
def fourierPredictionPlot(y, prediction):
    plt.figure(figsize=(10, 3))
    plt.plot(np.arange(0, y.size), y, 'k', label = 'data', linewidth = 2, 
alpha=0.5)
    plt.plot(np.arange(0, prediction.size), prediction, 'tab:red', label = 
'prediction')
    plt.grid()
    plt.legend()
    plt.show()
```

FT PREDICTION EXAMPLE

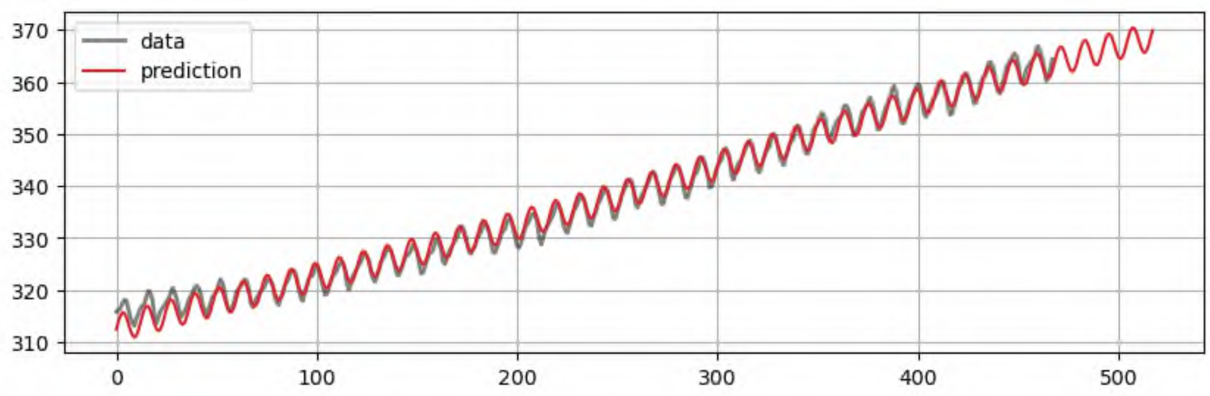
- noisy sinusoid with trend

```
prediction = fourierPrediction(noisy_signal, n_predict=200, n_harm=1)

fourierPredictionPlot(noisy_signal, prediction)
```



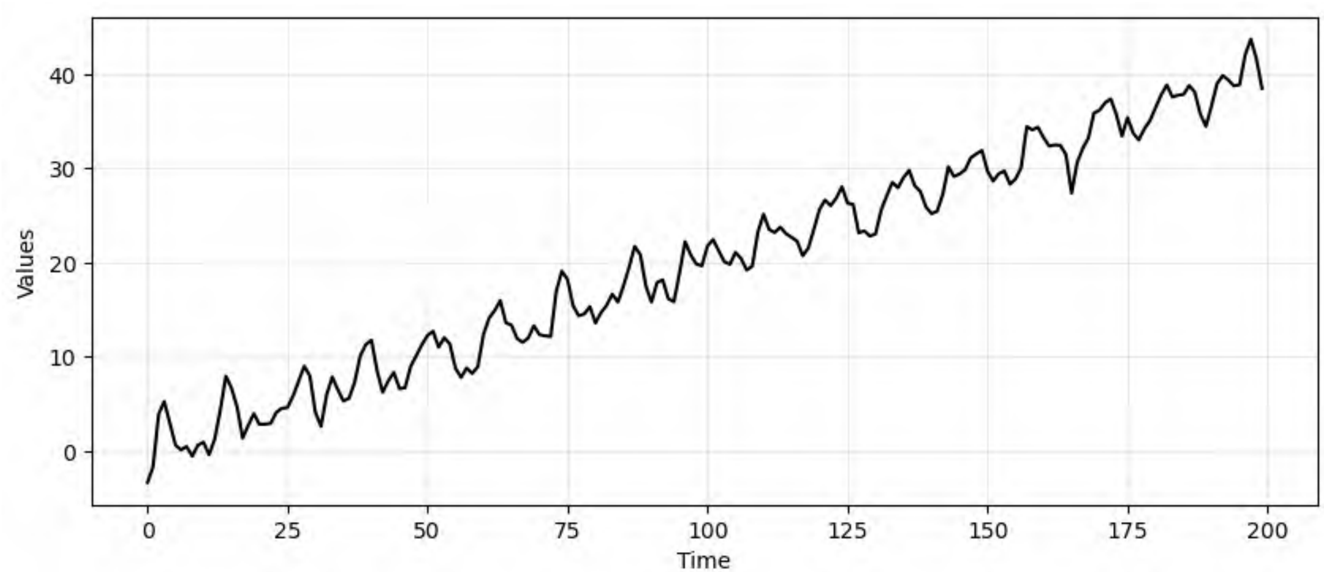
- Predict CO2 Data

```
co2 = sm.datasets.get_rdataset("co2", "datasets").data
print(co2.head())
# Convert decimal year to pandas datetime
def convert_decimal_year_to_datetime(decimal_years):
    dates = [(pd.to_datetime(f'{int(year)}-01-01') + pd.to_timedelta((year -
int(year)) * 365.25, unit='D')).date()
             for year in decimal_years]
    return dates
co2['time'] = convert_decimal_year_to_datetime(co2['time'])
# Convert the column ds to datetime
co2['time'] = pd.to_datetime(co2['time'])
print("\nConverted:\n------------------\n", co2.head())
# Resample to monthly frequency based on the ds column
co2 = co2.resample('MS', on='time').mean().reset_index()
# Replace NaN with the mean of the previous and next value
co2['value'] = co2['value'].interpolate()
print("\nResampled:\n------------------\n", co2.head())

prediction = fourierPrediction(co2['value'], n_predict=50, n_harm=1)
fourierPredictionPlot(co2['value'], prediction)
```



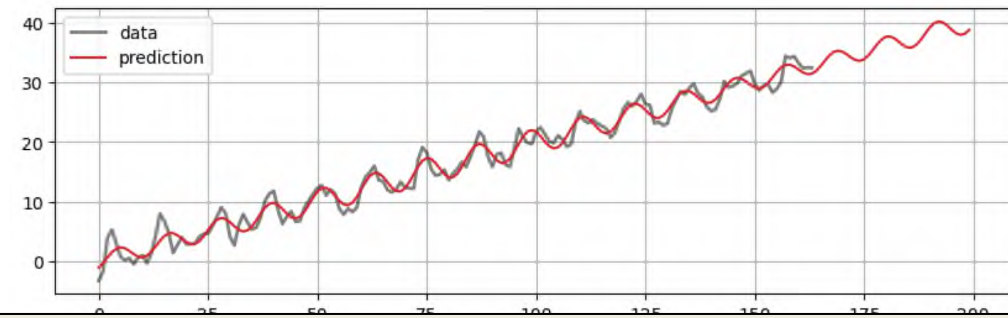
- The forecasting approach based on FT can also be used to remove trend and seasonality.

```
# Generate data from an AR(2) process
ar_data = arma_generate_sample(ar=np.array([1.0, -0.5, 0.7]), ma=np.array([1]),
nsample=200, scale=1, burnin=1000)

# Add trend and seasonality
time = np.arange(200)
trend = time * 0.2
seasonality = 2*np.sin(2*np.pi*time/12)
time_series_ar = trend + seasonality + ar_data

_, ax = plt.subplots(1, 1, figsize=(10, 4))
run_sequence_plot(time, time_series_ar, "", ax=ax);
```

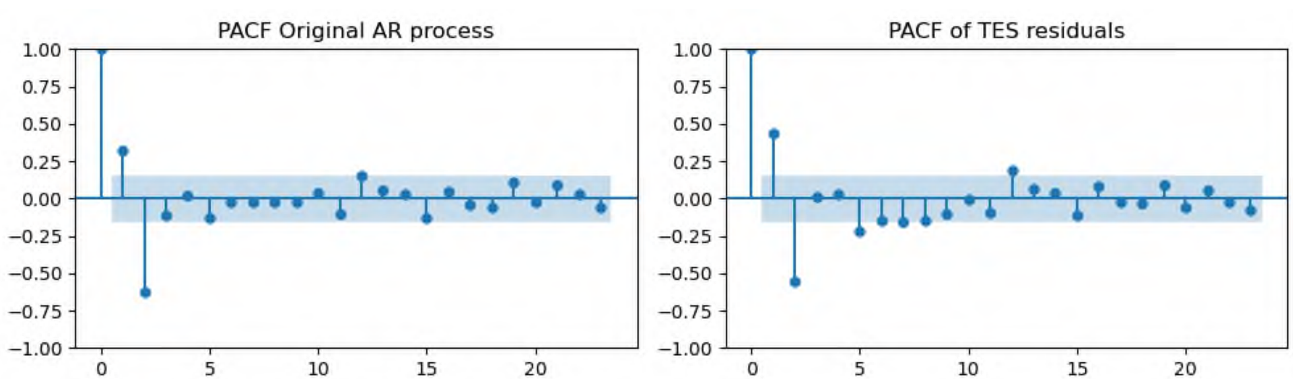


FT DETRENDING EXAMPLE

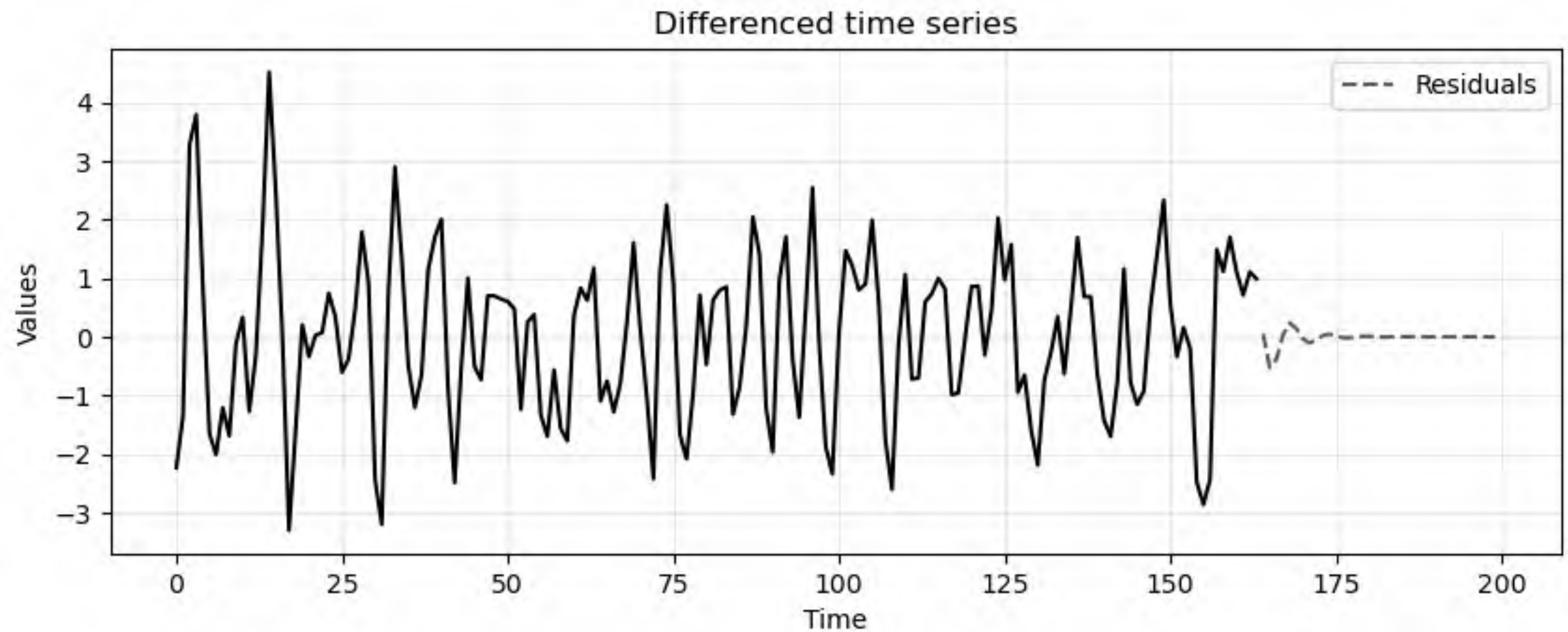
```
# Train/test split
train_data_ar = time_series_ar[:164]
test_data_ar = time_series_ar[164:]
prediction = fourierPrediction(train_data_ar, n_predict=len(test_data_ar),
n_harm=1)
fourierPredictionPlot(train_data_ar, prediction)
```



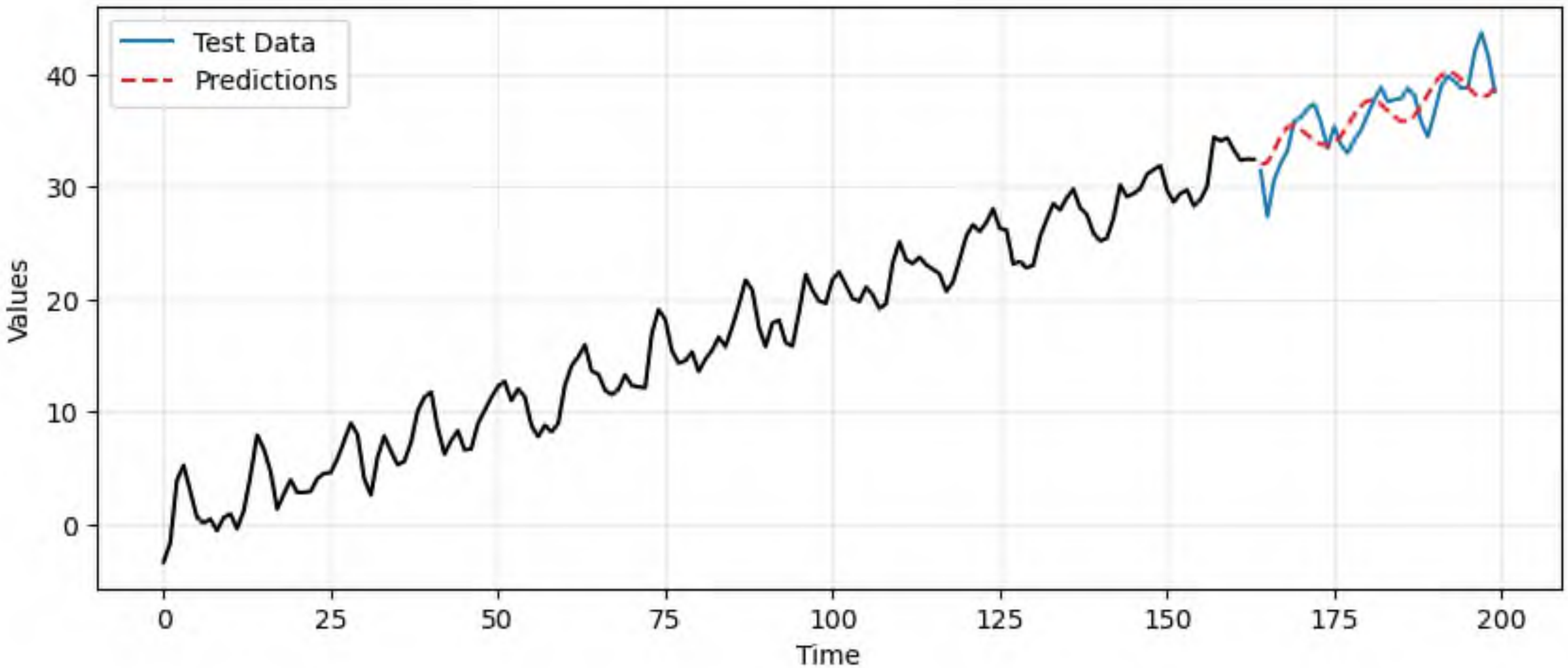

```
# Estimate trend and seasonality and remove them
trend_and_seasonality = prediction[:len(train_data_ar)]
resid = train_data_ar - trend_and_seasonality
_, axes = plt.subplots(1, 2, figsize=(10, 3))
plot_pacf(ar_data[:len(train_data_ar)], ax=axes[0], title="PACF Original AR
process")
plot_pacf(resid, ax=axes[1], title="PACF of TES residuals")
plt.tight_layout();
```



```
# Fit the model
model = ARIMA(resid, order=(2,0,0))
model_fit = model.fit()

# Compute predictions
resid_preds = model_fit.forecast(steps=len(test_data_ar))

ax = run_sequence_plot(time[:len(train_data_ar)], resid, "")
ax.plot(time[len(train_data_ar):], resid_preds, label='Residuals', linestyle='-
-', color='tab:red')
plt.title('Differenced time series')
plt.legend();
```



```python
# Add back trend and seasonality to the predictions
ft_preds = prediction[len(train_data_ar):]
final_preds = ft_preds + resid_preds

_, ax = plt.subplots(1, 1, figsize=(10, 4))
run_sequence_plot(time[:len(train_data_ar)], train_data_ar, "", ax=ax)
ax.plot(time[len(train_data_ar):], test_data_ar, label='Test Data',
color='tab:blue')
ax.plot(time[len(train_data_ar):], final_preds, label='Predictions',
linestyle='--', color='tab:red')
plt.legend();
```



- Learned some basic concepts from signal processing:
  1. A basic intuition of the FT and DFT
  2. A practical knowledge of how to apply the FFT
  3. The FT of common signals and the main properties of the FT
  4. Frequency response and transfer function concepts
  5. Different types of filters, their properties, and the Bode diagram
  6. How to use FT in forecasting tasks