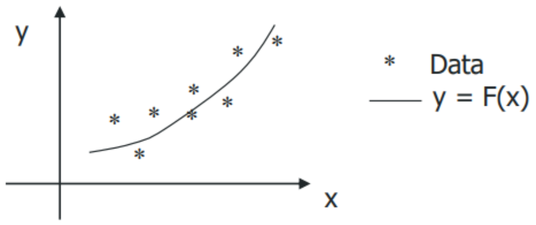
<u>FITTING AND INTERPOLATION:</u>

FITTING:

The aim of fitting is to identify the function (physical law) which optimally describes the relationships between two pair of variables x and y. Fitting is the process of finding the best mathematical function that matches the data points. The objective is to approximate the relationship between the independent and dependent variables by estimating the parameters of the model. Fitting can be linear or non-linear, depending on the nature of the model and the data.

In other words, fitting is when we have a dataset of scattered points and we find a line (or a curve) that fits the general shape of the data.



Experimental observations are inevitably uncertain to some degree. Normally, uncertainty varies for each data point. The fitting should be closer to the more certain points. To fit the curve, we always need one point more than the order (for example, if the order is 1, so a line, we need two points). They do not need to be points, but it can be any kind of constrains.

Fitting procedure consists of three phases:

1. <u>Model selection</u>: This is the process of choosing the appropriate mathematical model to fit the data, so we have to choose a family of functions. This involves selecting the type of function, the number of parameters, and the form of the equation. This phase also involves assessing the assumptions and limitations of the model.
   In the context of model selection, a "family of functions" refers to a set of mathematical functions with a common form, structure, or behaviour. For example, the polynomial functions form a family of functions, where each polynomial is defined by a different set of coefficients and degree. When choosing a family of functions for model selection, one is effectively choosing the type of relationship that is expected to exist between the independent and dependent variables.
   In case of polynomials, the family of functions is defined as:

   $$F(x) = a + bx + cx^2 + \cdots + nx^M$$

   Where $a, b, c \dots n$ are the coefficients, M is the degree of the polynomial

   In case of Lagrange polynomials, the family of functions is the defined as:

   $$F(x) = L_0(x)F_0(x_0) + L_1(x)F_1(x_1) + \cdots + L_M(x)F_M(x_M)$$

   By selecting the polynomial family of functions, one is assuming that the relationship between the independent and dependent variables can be approximated by a polynomial equation. The specific polynomial equation that best fits the data will be selected based on the optimization of the parameters (coefficients) and the evaluation of the fit using a chosen criterion.

   It's important to note that the choice of the family of functions can have a significant impact on the accuracy and reliability of the final model.

2. <u>Parameter estimation</u>: Once the model has been selected, the next step is to estimate the values of the parameters that define the model. This is done by finding the parameters that minimize the error between the observed data and the model's predictions. This phase involves selecting an appropriate optimization method and evaluating the accuracy of the parameter estimates.

3. <u>Model validation</u>: The final phase is to validate the model by evaluating its performance on new, independent data. This involves comparing the model's predictions to the actual observations and evaluating the goodness of fit. The objective is to ensure that the model generalizes well to new data and does not over-fit the training data.

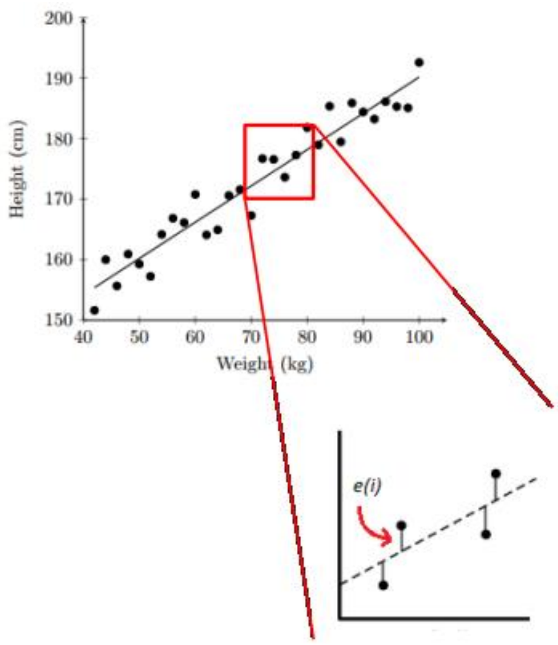
Linear fitting: Linear fitting is a type of model fitting in which the relationship between the independent and dependent variables is assumed to be linear. In other words, linear fitting assumes that the relationship can be modelled by a straight line.

The most common form of linear fitting is *linear regression*, which is a statistical method used to model the relationship between a dependent variable and one or more independent variables. In linear regression, the objective is to find the line of best fit that minimizes the difference between the observed data points and the predictions of the model.

Model selection: The equation of the line of best fit is defined as a polynomial function of the 1rst order:

$$F(x) = a_0 x + b_0$$

Where b_0 is the intercept and a_0 is the slope of the line. The parameters a_0 and b_0 are estimated using optimization methods (such as the least squares or maximum likelihood).



We want to minimize the error, which is the difference between the measured parameters and the values predicted through the function. We do not want positive and negative error to null themselves, so we minimize the sum of the square error (least square error → $g(a, b)$ in the following formula).

For a generic pair of parameters (a, b) the model error is:

$$e(i) = y_i - (ax_i + b)$$

We look for the pair which minimizes the errors (least-squares, LS):

$$J = \sum_{i=1}^{N} e_i^2 = \sum_{i=1}^{N} (y_i - ax_i - b)^2 = g(a, b)$$

Where: J is called loss function.

Parameter estimation: the goal is to find the best-fit parameters that minimize the loss function, which measures, as we said, the difference between the observed data and the predictions of the model.

Finding the minimum of the loss function J involves finding the parameter values that result in the lowest value of the loss function. One way to do this is to find the partial derivatives of the loss function with respect to each parameter, and then set them equal to zero to find the critical points. The critical points represent the points where the gradient of the loss function is zero, meaning that it is neither increasing nor decreasing at that point.

The intuition behind this is that, at a local minimum of the loss function, the gradient is zero, which means that the direction of the steepest ascent has been reversed. By finding the critical points, we can identify the parameter values that result in a minimum of the loss function.

$$\frac{dJ}{da} = 0 = -2 \sum_{i=1}^{N} x_i(y_i - ax_i - b)$$

$$\frac{dj}{db} = 0 = -2 \sum_{i=1}^{N} (y_i - ax_i - b)$$

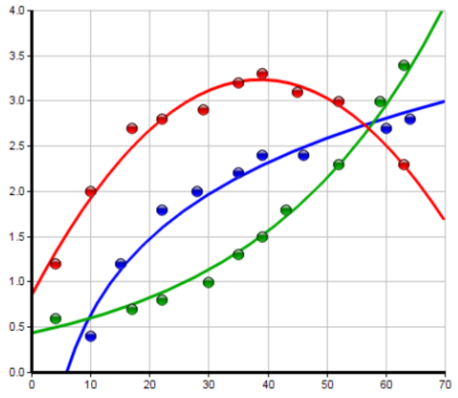
We obtain two equations with two unknown variables a and b, so we are able to find the values of a and b.

Polynomial fitting: Polynomial fitting is a technique used in signal processing and statistical analysis to fit a polynomial function to a set of data points. The goal is to find a polynomial that provides the best fit to the observed data, based on a measure of goodness-of-fit, such as the mean squared error.

The polynomial function can have any degree, and the choice of the degree of the polynomial is an important consideration in polynomial fitting. A higher degree polynomial will have more terms and can model more complex relationships between the independent and dependent variables, but it may also result in overfitting, where the model becomes too complex and fits the noise in the data instead of the underlying signal.

Model selection: The equation of the function of best fit is defined as a polynomial function of the M^st order:

$$F(x) = a_0 + b_0 x + c_0 x^2 + \cdots + n_0 x^M$$



Parameter estimation: Parameter estimation in polynomial fitting involves finding the values of the coefficients of the polynomial function that best fit the observed data. The goal is to find a polynomial that provides the best fit to the data, based on a measure of goodness-of-fit, such as the mean squared error.

In polynomial fitting, the parameters of the polynomial are estimated by minimizing a loss function, such as the mean squared error, using optimization methods. The resulting polynomial function can then be used to make predictions for new data points.

In analogy with the first-order case, the minimum is obtained by solving a system of M+1 equations, so M+1 points are needed to calculate the coefficients exactly. If we do an exact fit, there no residual terms, and there is no noise.

$$\begin{cases} \dfrac{dJ}{da} = 0 \\ \cdot \\ \cdot \\ \cdot \\ \dfrac{dJ}{dN} = 0 \end{cases} \rightarrow M+1\ equations, M+1\ unknowns$$

The general solution is:

$$\begin{aligned} f(x_1) &= a_0 + a_1 x_1 + a_2 x_1^2 + \cdots + a_N x_1^N \\ f(x_2) &= a_0 + a_1 x_2 + a_2 x_2^2 + \cdots + a_N x_2^N \\ &\vdots \\ f(x_{N+1}) &= a_0 + a_1 x_{N+1} + a_2 x_{N+1}^2 + \cdots + a_N x_{N+1}^N \end{aligned} \Rightarrow \begin{bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_{N+1}) \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & & x_1^N \\ 1 & x_2 & x_2^2 & \cdots & x_2^N \\ 1 & x_3 & x_3^2 & & x_3^N \\ & \vdots & & \ddots & \\ 1 & x_{N+1} & x_{N+1}^2 & & x_{N+1}^N \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_N \end{bmatrix}$$

$[f] = [x][a]$

$[a] = [x]^{-1}[f]$

Variables of interes

Measurement variable y_i

$$\theta = \begin{bmatrix} \hat{a} \\ \hat{b} \\ \vdots \\ \hat{n} \end{bmatrix} = \left[\sum_{i=1}^{N} \varphi_i \varphi_i^T \right]^{-1} \sum_{i=1}^{N} y_i \varphi_i^T$$

Matrix form

$$\varphi_i = [1 \quad x_i \quad x_i^2 \quad \ldots \quad x_i^M]^T$$

INTERPOLATION:

Interpolation is the process of estimating the values of a function at points where the function has not been explicitly defined. The idea is to fill in the missing values by constructing a continuous function that passes through the available data points. In other words, Interpolation can be thought of as filling in the gaps between known data points to obtain a smooth and continuous representation of the underlying function.

The goal of interpolation is to estimate the values of the underlying function at points that are intermediate between the known data points. The resulting interpolating function can be used for a variety of purposes, such as extrapolating beyond the range of the observed data, smoothing noisy data, or estimating the derivatives of the function at the known data points.

[Interpolation: sometimes an intermediate value located between measured values is needed. Extrapolation: sometimes a value located outside of the measured values is needed]

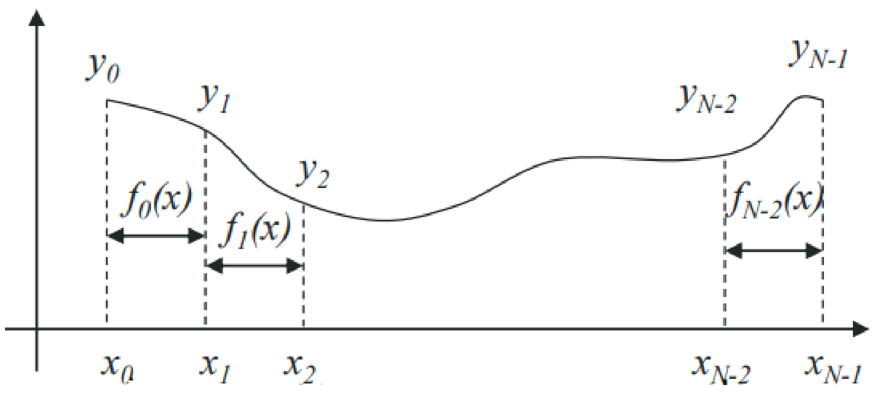
Interpolation methods can be either polynomial, spline-based, or based on other mathematical models.

Interpolation typically involves different steps:

- Collect the data: The first step is to gather the data that you want to interpolate. This data should be a set of ordered pairs, where each pair consists of an independent variable and a dependent variable.
- Choose the interpolating function: Next, you need to choose the type of interpolating function that you want to use. Common choices include polynomial interpolation, spline interpolation, and piecewise linear interpolation.
- Calculate the coefficients: Depending on the type of interpolating function you have chosen, you may need to calculate the coefficients of the function. For example, in polynomial interpolation, you need to find the coefficients of the polynomial that best fits the data. This can be done using optimization methods, such as gradient descent or conjugate gradient.
- Evaluate the function: Once you have the coefficients of the interpolating function, you can use them to evaluate the function at any intermediate point. This will give you an estimate of the underlying function at that point.

<u>*Example:*</u> Interpolation with cubic spline

Cubic spline interpolation is a type of interpolation that uses piecewise cubic polynomials to estimate the values of a function at intermediate points. In this technique, the data is divided into a set of subintervals, and a cubic polynomial is fit to the data in each subinterval.



$$f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$$

$$x_i < x < x_{i+1}$$

Task: Given N points, we have to identify (N-1) polynomial functions define by 4 parameters → 4*(N-1) unknowns.

First, we fit the data to a curve and secondly, we use the fit curve to calculate the new values (interpolation).

We introduce some constrains between functions:

- The polynomials are chosen such that they are continuous (functions of contiguous intervals must coincide in the common knot). [2(N-2) conditions]:

$$f_{i-1}(x_i) = f_i(x_i) = y_i$$

- The function must pass the first and last point of the interval:

$$f_0(x_0) = y_0$$
$$f_{N-2}(x_{N-1}) = y_{N-1}$$

- The first derivative must be continuous at the boundary points (common knots). So, the first derivative must coincide in the common knots [N-2 conditions]:

$$f'_{i-1}(x_i) = f'_i(x_i)$$

- The second derivative must be continuous at the boundary points (common knots). So, the second derivative must coincide in the common knots [N-2 conditions]:

$$f''_{i-1}(x_i) = f''_i(x_i)$$

- The second derivative must be equal to zero in the first and last points [2 conditions]:

$$f''_0(x_0) = 0$$

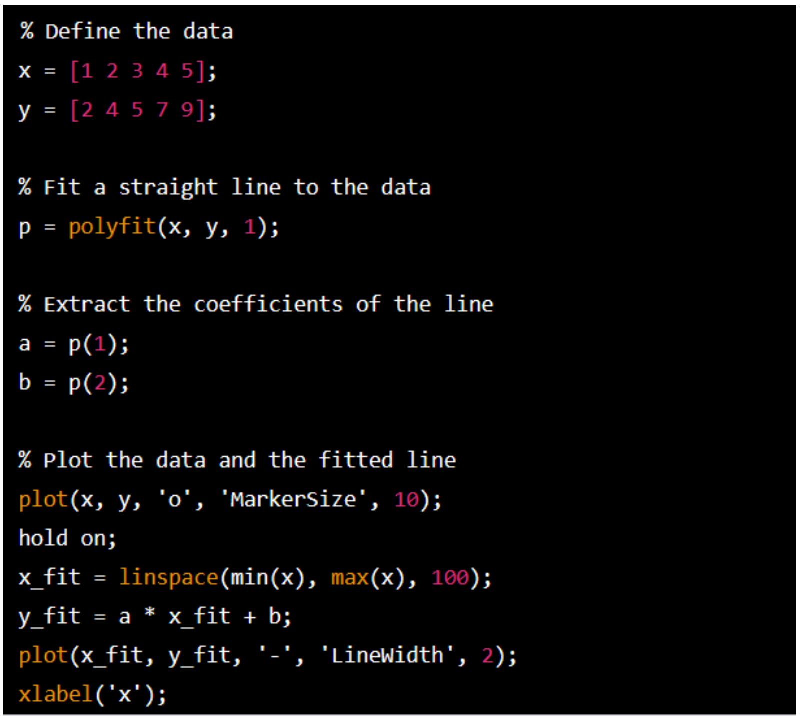
$$f''_{N-2}(x_{N-1}) = 0$$

Globally, we have 4(N-1) conditions, so we are able to identify 4(N-1) unknowns.

These constrains ensure that the resulting interpolating function is smooth and has desirable properties such as being free of overshoot and undershoot, and having minimal oscillation between the knots.

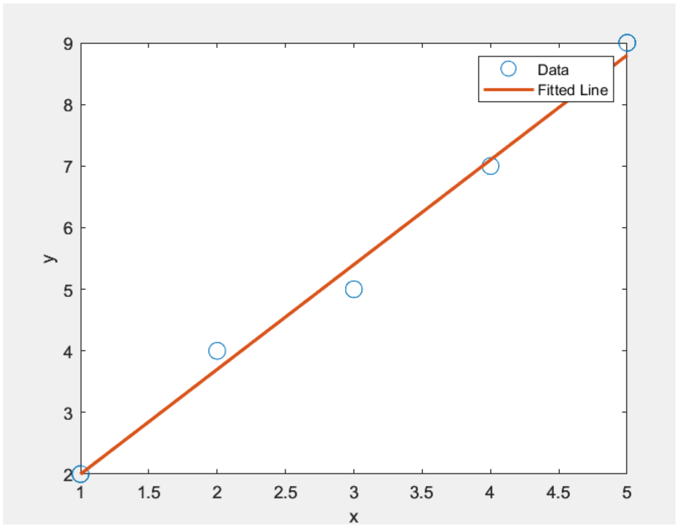
LAB:

Build a MATLAB script to estimate the parameter of a linear model y=a+bx describing the relationships between two sets of measures x e y. Verify the script using a simulated signal.



```
% Define the data
x = [1 2 3 4 5];
y = [2 4 5 7 9];

% Fit a straight line to the data
p = polyfit(x, y, 1);

% Extract the coefficients of the line
a = p(1);
b = p(2);

% Plot the data and the fitted line
plot(x, y, 'o', 'MarkerSize', 10);
hold on;
x_fit = linspace(min(x), max(x), 100);
y_fit = a * x_fit + b;
plot(x_fit, y_fit, '-', 'LineWidth', 2);
xlabel('x');
```

In this example, polyfit is used to fit a straight line to the data, which is defined by the x and y arrays. The first argument to polyfit is the independent variable (x), the second argument is the dependent variable (y), and the third argument is the degree of the polynomial to fit (1 for a straight line). The resulting polynomial coefficients are stored in the p array, with the first element (p(1)) being the slope (a) and the second element (p(2)) being the intercept (b). The fitted line is then plotted along with the data using plot.

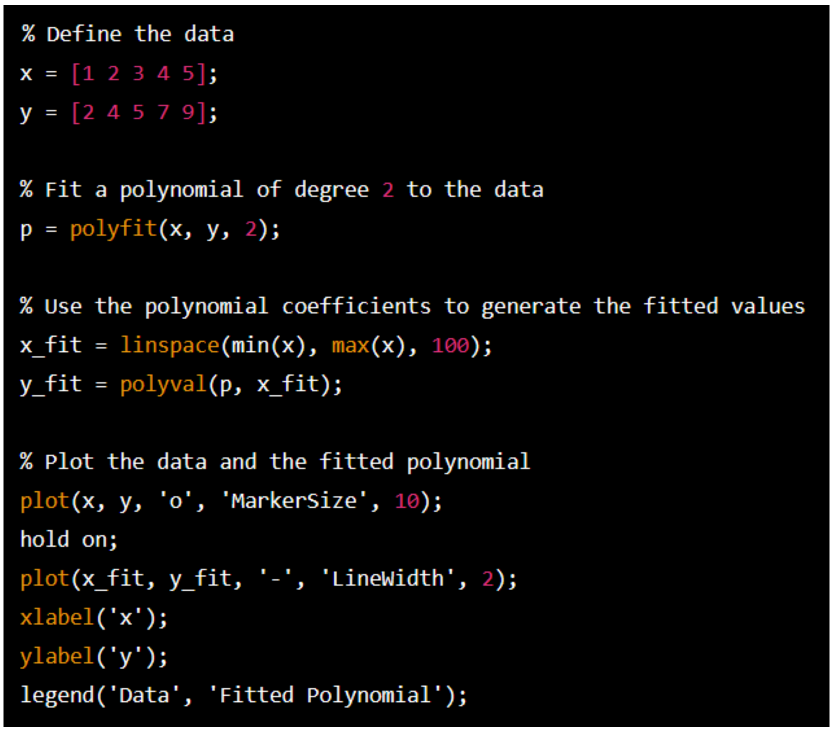


polyfit → p=polyfit(x,y,n) returns the coefficients for a polynomial $p(x)$ of degree n, that is a best fit (in least-squares sense) for the data in y. The coefficients in p are in descending powers (potenze discendenti), and the length of p is $n+1$.

$$p(x) = p_1 x^n + p_2 x^{n-2} + \cdots + p_n x + p_{n+1}$$

polyval → y=polyval(p,x) evaluates the polynomial p at each point in x. The argument p is a vector that follows the structure of the output of the polyfit function.

Example for polyval:

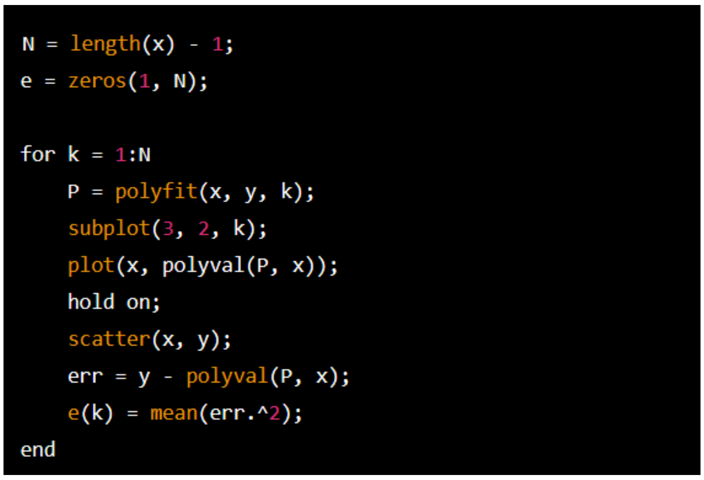


```
% Define the data
x = [1 2 3 4 5];
y = [2 4 5 7 9];

% Fit a polynomial of degree 2 to the data
p = polyfit(x, y, 2);

% Use the polynomial coefficients to generate the fitted values
x_fit = linspace(min(x), max(x), 100);
y_fit = polyval(p, x_fit);

% Plot the data and the fitted polynomial
plot(x, y, 'o', 'MarkerSize', 10);
hold on;
plot(x_fit, y_fit, '-', 'LineWidth', 2);
xlabel('x');
ylabel('y');
legend('Data', 'Fitted Polynomial');
```

In this example, polyfit is used to fit a second-degree polynomial to the data, which is defined by the x and y arrays. The first argument to polyfit is the independent variable (x), the second argument is the dependent variable (y), and the third argument is the degree of the polynomial to fit (2 in this case). The resulting polynomial coefficients are stored in the p array. The polyval function is then used to generate the fitted values using the polynomial coefficients. Finally, the fitted polynomial is plotted along with the data using plot.

Which is the effect of the model order on the model errors?

Sometimes we need to evaluate the order to use to fit data. So, we fit data x and y using polynomial of **increasing order** and we compute the mean quadratic error for each order, as follows:



```
N = length(x) - 1;
e = zeros(1, N);

for k = 1:N
    P = polyfit(x, y, k);
    subplot(3, 2, k);
    plot(x, polyval(P, x));
    hold on;
    scatter(x, y);
    err = y - polyval(P, x);
    e(k) = mean(err.^2);
end
```

This code performs a polynomial fit for increasing degrees of polynomials from 1 to N, and plots the fitted curves along with the data points in a subplot matrix with 3 rows and 2 columns. The mean squared error of the fit for each degree of polynomial is also calculated and stored in the vector e.

We can plot the error and see its behaviour (on the x axis the order and on the y axis the corresponding value of the error).

In general, as the order of the polynomial is increased, the error in the fit will decrease for the training data, as the polynomial can capture more complex patterns and better fit the data. However, as the order increases further, the error on new or out-of-sample data may start to increase again, due to overfitting.

Overfitting occurs when the polynomial becomes too flexible and starts to fit the noise or random fluctuations in the data rather than the underlying trend. This can result in a polynomial that looks good on the training data but performs poorly on new data or out-of-sample data. The relationship between the order of the polynomial and the error will thus be influenced by the presence of overfitting, and the balance between model complexity and goodness of fit.

To find an appropriate balance between model complexity and goodness of fit, it is common to use techniques such as cross-validation. Another method is the elbow criterion: we select the value of the order at the elbow. For example, the point after which the sum of squared residuals starts decreasing in a flat or linear way.

spline→ s=spline(x,y,xq) returns a vector of interpolated values s corresponding to the query points xq. The values of s are determined by cubic spline interpolation of x and y.

[A query point is a point in a dataset for which you want to predict a value based on the values at other points in the dataset]

The function takes three input arguments:

x: A vector of independent variables (or "knots") that define the original data points; y: A vector of dependent variables (or "ordinates") that define the original data points; xq: A vector of independent variables at which the interpolated curve will be evaluated.

The spline function generates a piecewise cubic polynomial representation of the interpolated curve and returns the values of the curve at the desired independent variables specified by xq.

DISCRETE FOURIER TRANSFORM:

Analysis of signals and systems often requires to move from time to frequency domain. The discrete Fourier transform (DFT) is a mathematical technique used to transform a finite sequence of discrete values (e.g., a signal sampled in time or space) into its equivalent frequency domain representation. The output of the DFT is a set of complex coefficients, each representing the magnitude and phase of a sinusoidal component present in the original signal.

The DFT is used for many things:

- Signal Decomposition and Harmonic Analysis: In signal processing, signals are often represented as the sum of simpler waveforms known as harmonics. The DFT can be used to analyze a signal and determine its harmonic components, enabling the extraction of important information about the signal's content and behaviour. By analyzing the DFT of a signal, we can determine its spectral content, which provides insights into its frequency composition and can be used to identify periodic and non-periodic components.
- Linear System Response and Characterization: In control systems and signal processing, the DFT can be used to study the response of linear systems to signals. The DFT of the impulse response of a linear system can be used to determine its frequency response, which provides information about the system's behaviour at different frequencies. This is important for characterizing and analyzing linear systems, such as filters and communication channels, and for designing systems that meet certain performance specifications.
- Linear Filter Design: The DFT is often used to design linear filters, which are systems used to process signals by removing or suppressing certain frequency components. Filter design can be formulated as an optimization problem, and the DFT provides a convenient tool for analyzing the frequency-domain behaviour of filters. By carefully choosing the frequency response of a filter, it is possible to implement systems that perform a variety of signal processing tasks, such as noise reduction, signal enhancement, and signal separation.

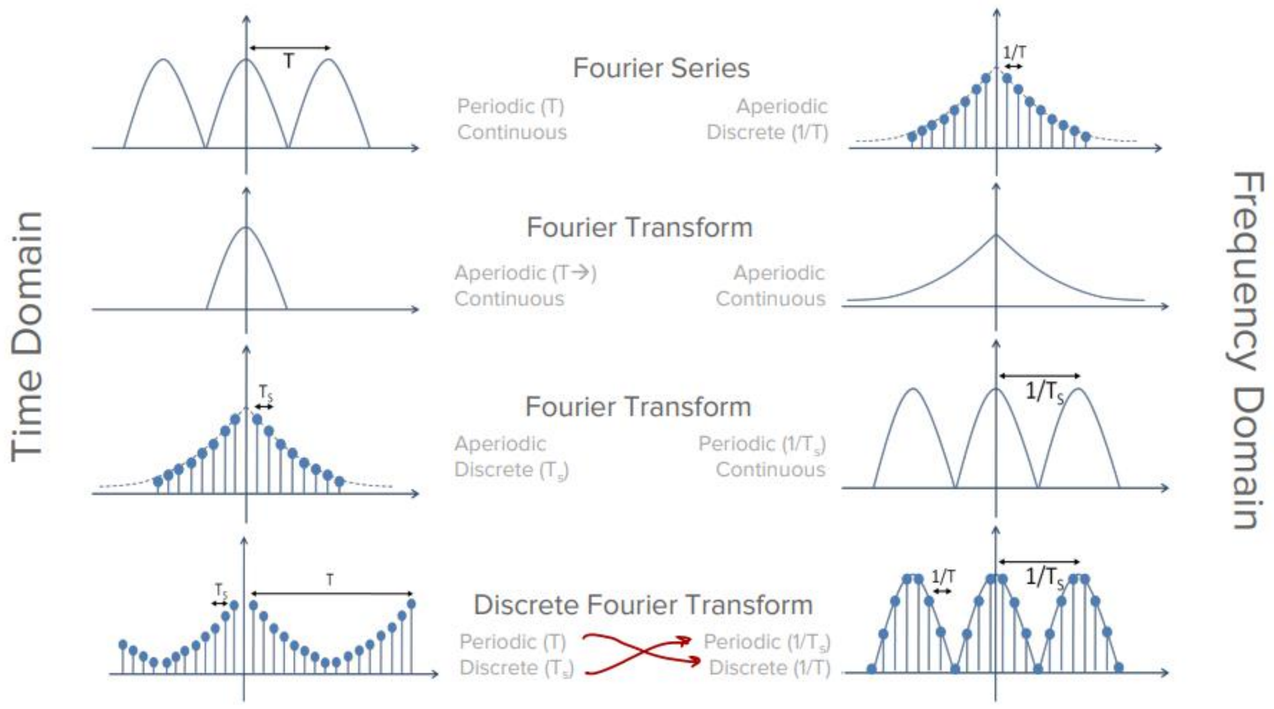
The DFT is defined as:

$$Y[k] = \sum_{n=0}^{N-1} y[n] e^{-\frac{jn2\pi k}{N}} \qquad for\ k = 0,1,2,\dots,N-1$$

Where: $Y[k]$ is the k-th complex-valued DFT coefficient, j is the imaginary unit ($j^2 = -1$), e is the base of the natural algorithm, N is the number of samples [$\Delta f = \frac{f_s}{N} = \frac{1}{NT_s} \rightarrow$ the larger is N, the smaller is the step between two successive frequencies].

The term $e^{-\frac{jn2\pi k}{N}}$ is known as the twiddle factor and it represent a complex exponential that changes phase with respect to n and k.

In essence, the DFT decomposes a discrete signal into a set of complex-valued sinusoids with different frequencies and amplitudes. Each coefficient $Y[k]$ represents the contribution of a sinusoidal component with frequency $k \cdot \frac{f_s}{N}$ to the original signal $y(n)$, where f_s is the sample rate. The magnitude of $Y[k]$ represents the strength or amplitude of the sinusoidal component, and the phase of $Y[k]$ represents its phase relationship with respect to the other components.



Fundamental note: In the digital world, both signal and its transform are discrete, and we know that:

- Discrete frequencies imply periodic signals
- Discrete time signals imply periodic transform

In real practice, we have to process signals which are observed in a given time-window and non-necessarily periodic. How can we handle this?

We assume that the signal is periodic with a period equal to the observation window, i.e. there is an infinite number of signal repetitions outside the observed window.

By assuming we are considering a limited window of time, the signal repeats itself outside that window for infinite windows. This is a fundamental hypothesis to process DFT on a signal.

Practical issue: we notice that the DFT computes values of $Y[k]$ only at multiples of $\frac{1}{NT_s}$, being N the number of samples. So, we can only see a part. The solution to this problem is to increase the number of samples (N) or increase the sampling rate.

Zero padding: it is a technique that consists of adding a series of zeros at the end of the signal. This will not alter the information content of the signal and also it does not improve frequency resolution (the capacity to distinguish two closely spaced frequencies), which will always be given by $\frac{f_s}{N}$ (where N is the number if real samples). The addition of zeros is useful obtain a denser representation of the signal in the frequency domain (high density spectrum), so to reduce the inter-frequency spacing and to interpolate points in frequencies. This allows us to see a more detailed power spectrum.

For example: we have the following signal of N samples

$$x(n) = [x(0), x(1), x(2), \dots, x(N-1)]$$

We perform zero-padding:

$$x(n) = [x(0), x(1), x(2), \dots, x(N-1), 0,0,0,0]$$

With the addition of these zeros, we are increasing the number of points between 0 and 2π, so we can obtain a better representation of magnitude and phase response. So, the way to improve the frequency resolution is to pad zeros to the given signal $x(n)$.

Effect of windowing: as we said, we assume that the signal is periodic with a period equal to the observation window, so the DFT is computed on a finite set of data points, extracted from a period hypothetical infinite signal $y_h[n]$. In practice, we have:

$$y[n] = y_h[n]w[n]$$

Where $w[n]$ is the truncation window.

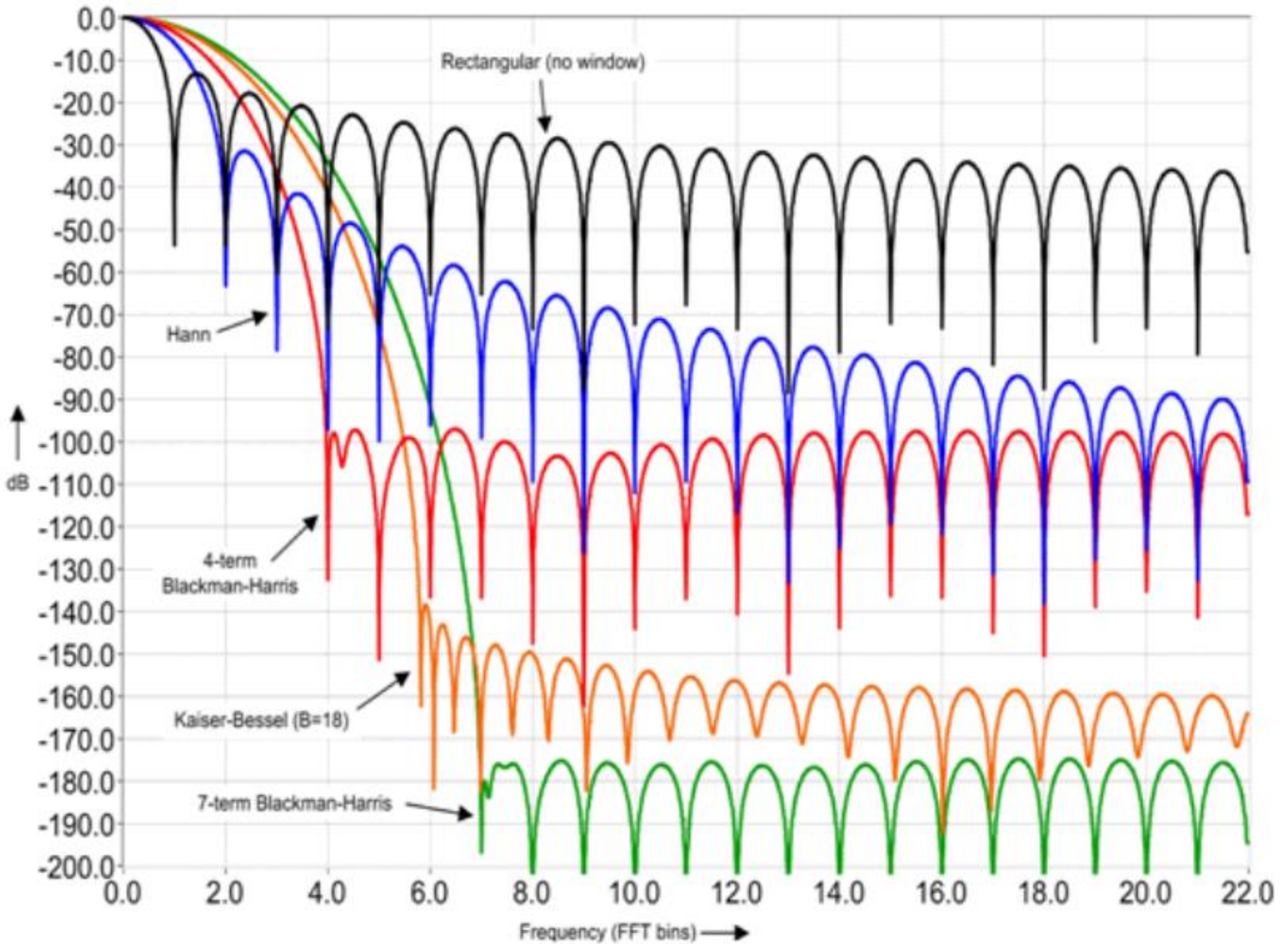
Thus, in the frequency domain we have:

$$Y[k] = Y_h[n] * W[k]$$

(Remember that multiplication in time corresponds to convolution in frequency).

When we cut the original signal with a window, we have to choose the right one in order to balance ripples and main lobe.

There are several types of windows:



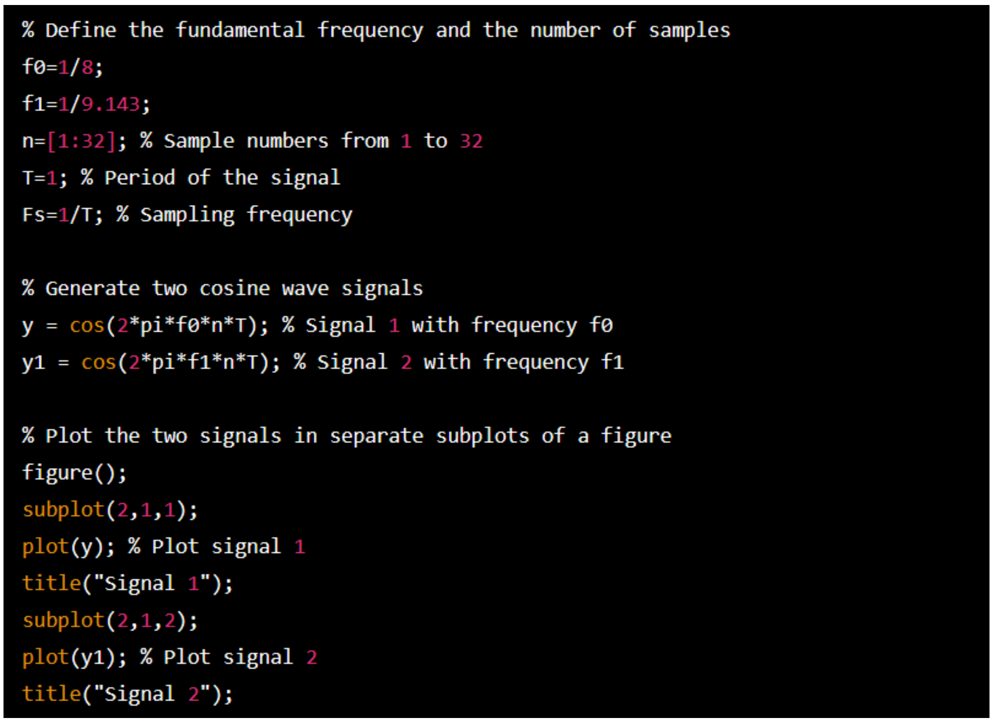
REMEMBER: window must be applied before zero padding!!!

LAB:

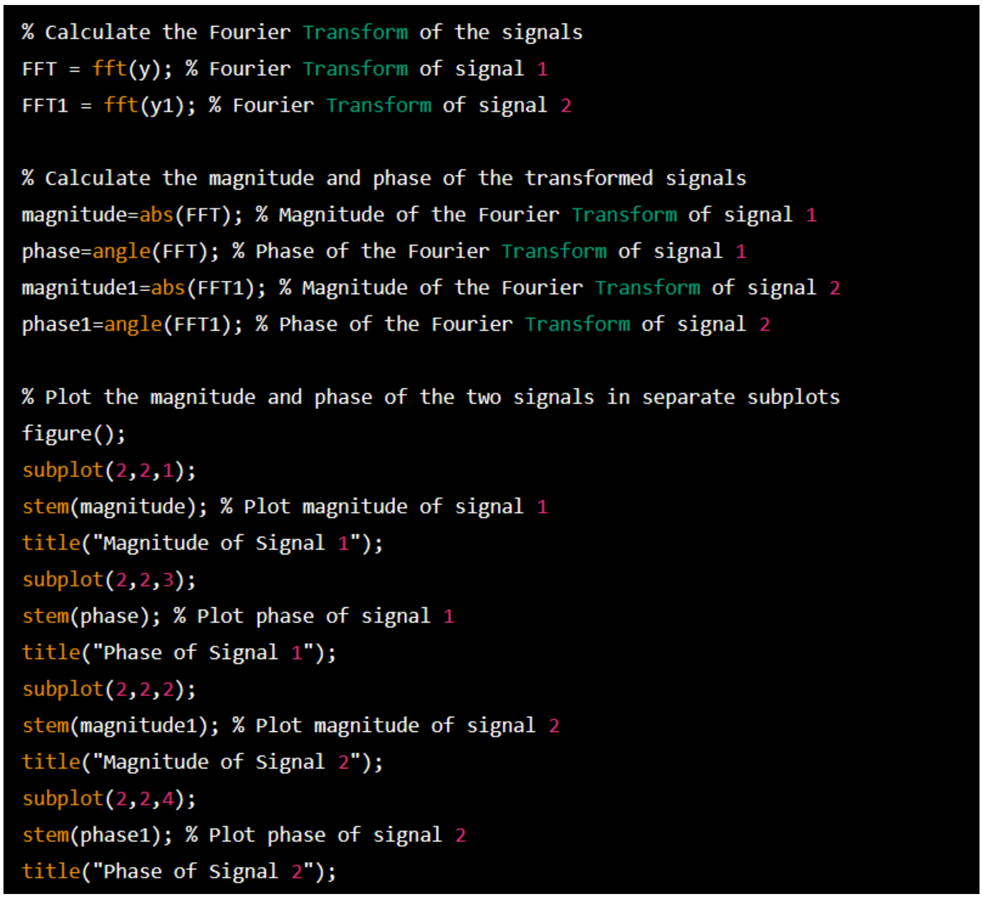
Let's consider the series $y[n] = cos\ (2\pi f_0 nT)s$ *and the following cases: 1) n=[1:32]; T=1 s; f0 = 1/8 Hz*

2) n=[1:32]; T=1 s; f0 = 1/9.143 Hz

plot the DFT coefficients (modulus and phase) as a function of the frequency (using Matlab function stem) and set the frequency axis correctly.



```
% Define the fundamental frequency and the number of samples
f0=1/8;
f1=1/9.143;
n=[1:32]; % Sample numbers from 1 to 32
T=1; % Period of the signal
Fs=1/T; % Sampling frequency

% Generate two cosine wave signals
y = cos(2*pi*f0*n*T); % Signal 1 with frequency f0
y1 = cos(2*pi*f1*n*T); % Signal 2 with frequency f1

% Plot the two signals in separate subplots of a figure
figure();
subplot(2,1,1);
plot(y); % Plot signal 1
title("Signal 1");
subplot(2,1,2);
plot(y1); % Plot signal 2
title("Signal 2");
```



```
% Calculate the Fourier Transform of the signals
FFT = fft(y); % Fourier Transform of signal 1
FFT1 = fft(y1); % Fourier Transform of signal 2

% Calculate the magnitude and phase of the transformed signals
magnitude=abs(FFT); % Magnitude of the Fourier Transform of signal 1
phase=angle(FFT); % Phase of the Fourier Transform of signal 1
magnitude1=abs(FFT1); % Magnitude of the Fourier Transform of signal 2
phase1=angle(FFT1); % Phase of the Fourier Transform of signal 2

% Plot the magnitude and phase of the two signals in separate subplots
figure();
subplot(2,2,1);
stem(magnitude); % Plot magnitude of signal 1
title("Magnitude of Signal 1");
subplot(2,2,3);
stem(phase); % Plot phase of signal 1
title("Phase of Signal 1");
subplot(2,2,2);
stem(magnitude1); % Plot magnitude of signal 2
title("Magnitude of Signal 2");
subplot(2,2,4);
stem(phase1); % Plot phase of signal 2
title("Phase of Signal 2");
```

In order to <u>rescale the frequency axis</u>:

```
Nfft=length(y);
Fbins=((0:1/Nfft:1-1/Nfft)*Fs);
```

The first line, Nfft=length(y);, calculates the length of the signal y and stores it in the variable Nfft. The second line, Fbins=((0:1/Nfft:1-1/Nfft)*Fs);, calculates the bin frequencies of the Fourier Transform. The 0:1/Nfft:1-1/Nfft generates a vector of values from 0 to 1 with a step size of 1/Nfft. This vector represents the fraction of the total number of samples. The *Fs scales the fraction to the actual frequency in Hz. The result is stored in the variable Fbins. This line of code is generating the frequency axis for the plots of the magnitude and phase of the Fourier Transform:

```
figure()
subplot(2,2,1)
stem(Fbins,magnitude)
subplot(2,2,3)
stem(Fbins, phase)
subplot(2,2,2)
stem(Fbins, magnitude1)
subplot(2,2,4)
stem(Fbins, phase1)
```

Notiamo che i due grafici delle magnitude sono differenti, perchè la f1 non è una frequenza perfettamente rappresentabile dalla FFT (multiplo di frequenze rappresentabili), come lo è invece f0), quindi la trasformata cerca di rappresentarla nell'intorno. L'altra ragione è che nel secondo grafico del segnale vediamo che c'è una discontinuità e se dovessimo replicarlo/specchiarlo, la trasformata rifletterebbe questa discontinuità con una serie di piccoli picchi di frequenza diversi da zero.

To evaluate the effect of different types of windows, we have to evaluate the trade-off between width of the main lobe and amplitude of the lateral lobes.

It is worth remembering to normalize by the window characteristics. The normalization is done by the some of the elements of the window.

<u>Triangular window</u>:

```
L = length(n);
tw = triang(L);

y_tw=(y.*tw')/sum(tw); %normalization = weighting the signal for the windows values
(/sum(tw))

y_tw_zp=[y_tw, zeros(1, 32)]; %zero padding

FFTtw=fft(y_tw_zp); %Fourier Transform

magnitude_tw=abs(FFTtw);
phase_tw=angle(FFTtw);
```

<u>Blackman window</u>:

```
L = length(n);
bw = blackman(L);

y_bw=(y.*bw')/sum(bw); %normalization

y_b_zp=[y_b, zeros(1, 32)]; %zero padding
```

```
FFTb=fft(y_b_zp);%Fourier Transform

magnitude_b=abs(FFTb);
phase_b=angle(FFTb);
```

Hamming window:

```
N=length(y);
h=hamming(L);

%SIGNAL 1
y_h=(y.*h')/sum(h); %normalization


y_h_zp=[y_h, zeros(1, 32)]; %zero padding

FFTh=fft(y_h_zp); %Fourier Transform

magnitude_h=abs(FFTh);
phase_h=angle(FFTh);
```

Let's consider the following sequence, sum of two sinusoids: $y[n] = cos\,(2\pi f_0 nT) + cos\,(2\pi(f_0 + \Delta f)nT)$
Where n=[1:150], T=2s, f0=1/8 Hz.
Compute the DFT using zero-padding =128 points and three different windows (Triangular, Blackman, Hamming) when: 1) Δf*=8/(NT), 2)* Δf*=4/(NT), 3)* Δf*=2/(NT).*
After plotting the results, verify how the window selection may affect frequency resolution, i.e. the capability to detect the two sinusoids.

[remember: frequency resolution cannot be modified with zero padding]

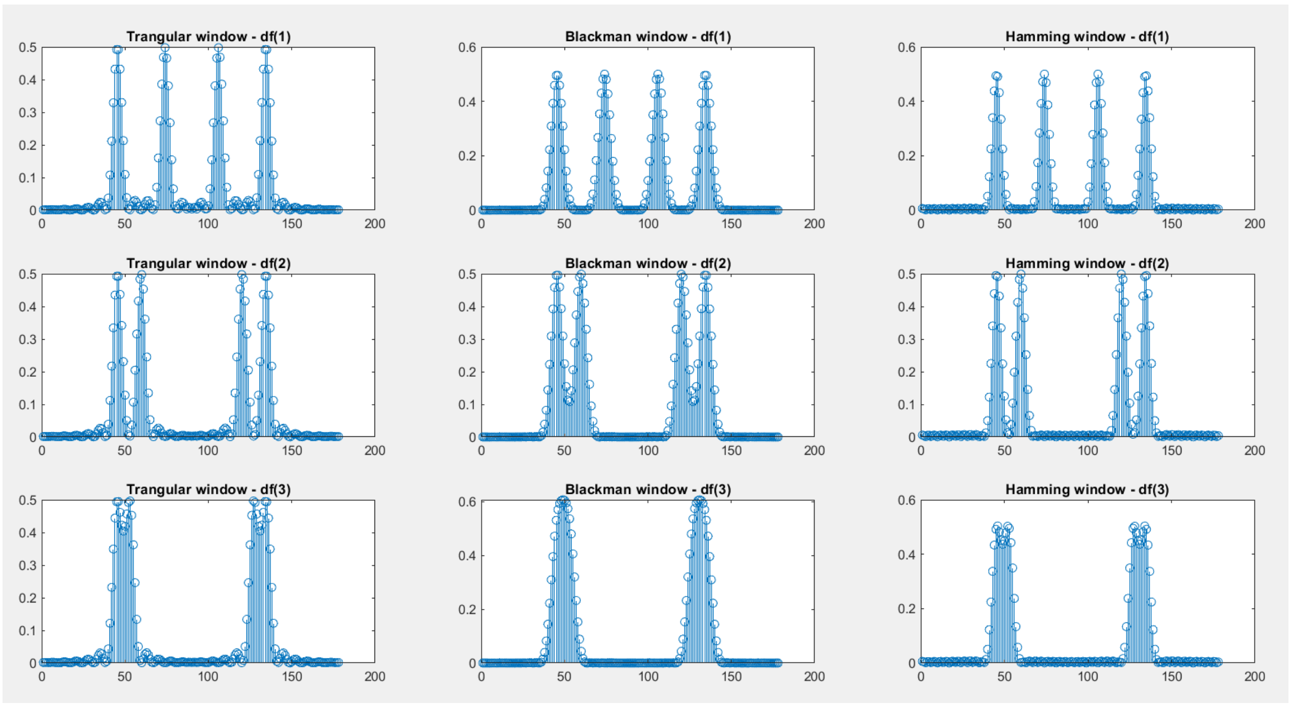
```
f0=1/8;
n=[1:50];
N=length(n);
T=2;

df=[8/(N*T), 4/(N*T), 2/(N*T)];

%we define the three signals
y1 = cos(2*pi*f0*n*T)+cos(2*pi*(f0+df(1))*n*T);
y2 = cos(2*pi*f0*n*T)+cos(2*pi*(f0+df(2))*n*T);
y3 = cos(2*pi*f0*n*T)+cos(2*pi*(f0+df(3))*n*T);
```

% We repeat the following code for all the three signals (here I just report the one for one signal to see the syntax)

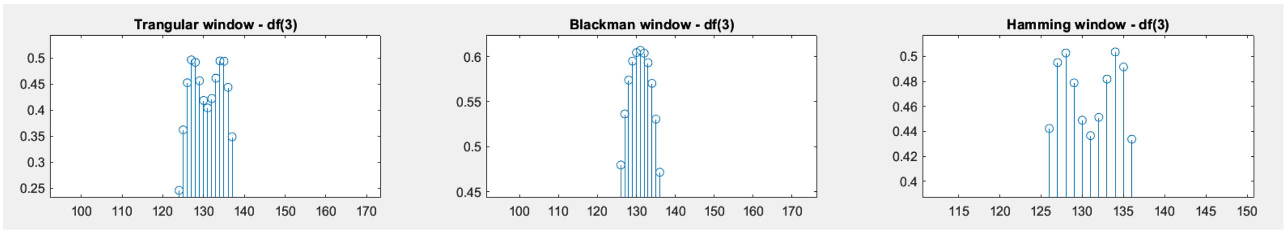
```
%% Triangular window
L = length(n);
tw = triang(L);

y1_tw=(y1.*tw')/sum(tw); %normalization

y1_tw_zp=[y1_tw, zeros(1,128)]; %zero padding

FFTtw1=fft(y1_tw_zp); %Fourier Transform

magnitude_tw1=abs(FFTtw1);
phase_tw1=angle(FFTtw1);
```

```
%% Blackman window

b=blackman(L);
y_b1=(y1.*b')/sum(b); %normalization

y_b1_zp=[y_b1, zeros(1, 128)]; %zero padding

FFTb1=fft(y_b1_zp); %Fourier Transform

magnitude_b1=abs(FFTb1);
phase_b1=angle(FFTb1);

%% Hamming window

h=hamming(L);
y1_h=(y1.*h')/sum(h); %normalization

y1_h_zp=[y1_h, zeros(1, 128)]; %zero padding

FFTh1=fft(y1_h_zp); %Fourier Transform

magnitude_h1=abs(FFTh1);
phase_h1=angle(FFTh1);
```

After doing this we subplot the magnitude and phase response of the three signals corresponding to the three different windows:



We notice that the lower the frequency resolution, the more important is the selection of the window to be used. In fact, in the case where the resolution is smaller (df(3)), triangular window and hamming window slightly show the two peaks, instead blackman window does not (third line of the graph above):



stem(x)→ This function generates a stem plot of the input data x. A stem plot is a type of plot that shows the magnitude of a signal as dots connected by a line to a baseline, where the dots represent the samples of the signal and the line represents the progression of the signal over time.

fft(x)→ This function calculates the fast Fourier transform (FFT) of the input signal x. The FFT is a mathematical algorithm that transforms a time-domain signal into its frequency-domain representation. The FFT provides information about the frequencies present in the signal, allowing you to analyze and manipulate the signal in the frequency domain.

fftshift(x)→ This function shifts the zero-frequency component of the FFT of the input signal x to the center of the output fft array. The zero-frequency component is the DC component of the signal, which represents the average value of the signal. This function is useful to visualize a Fourier Transform with the zero frequency component in the middle of the spectrum. Shifting the zero-frequency component to the center of the output makes it easier to interpret the FFT results.

abs(x)→ This function calculates the magnitude of the input x, which is the absolute value of x.

angle(x)→ This function calculates the phase angle of the input x, which is the angle of the complex representation of x in the complex plane.

blackman(n)→ This function generates a Blackman window of length n. A window function is applied to a signal to reduce the spectral leakage, which is the spreading of the spectral content of the signal into other frequency bins. The Blackman window is a type of window function that provides a balance between main-lobe width and side-lobe height.

hamming(n)→ This function generates a Hamming window of length n. The Hamming window is a type of window function that provides a balance between the width and height of the main lobe.

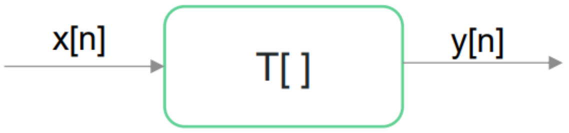
triang(n)→ This function generates a triangular window of length n. The triangular window is a type of window function that has a triangular shape and is often used for simple signal analysis.

rectwin(n)→ this function generates a rectangular window of length(n). A rectangular window is a type of window function that is simply a rectangular shape with a flat top and zero values at its sides. The rectangular window is commonly used in signal processing for its simplicity and ease of implementation, but it is often not the best choice for spectral analysis due to its poor spectral sidelobes.

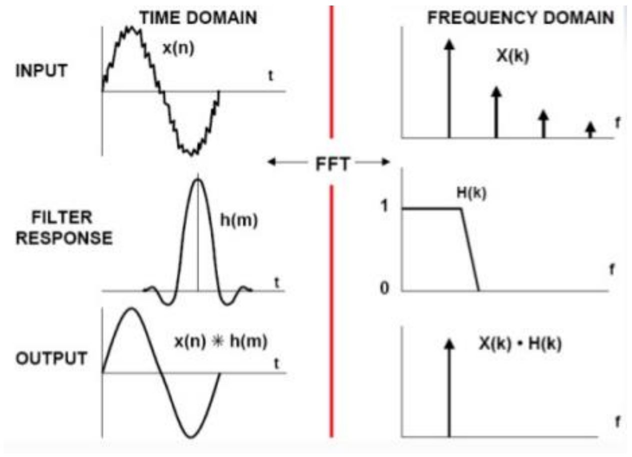
ifft(x)→ This function calculates the inverse fast Fourier transform (IFFT) of the input signal x. The IFFT transforms a frequency-domain signal back into its time-domain representation. The IFFT is the inverse operation of the FFT, and it can be used to recover the original time-domain signal from its frequency-domain representation.

DIGITAL FILTERS:

A digital filter is a mathematical algorithm that processes digital signals in order to achieve a specific outcome.



Filters transform a series of data $x[n]$ (input) in a new series of data $y[n]$ (output) in which some features/characteristics of $x[n]$ are enhanced/depressed. $T[\,]$ defines the characteristic of the filter. We consider linear, time-invariant filters.

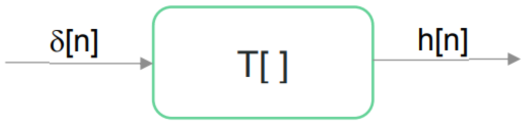


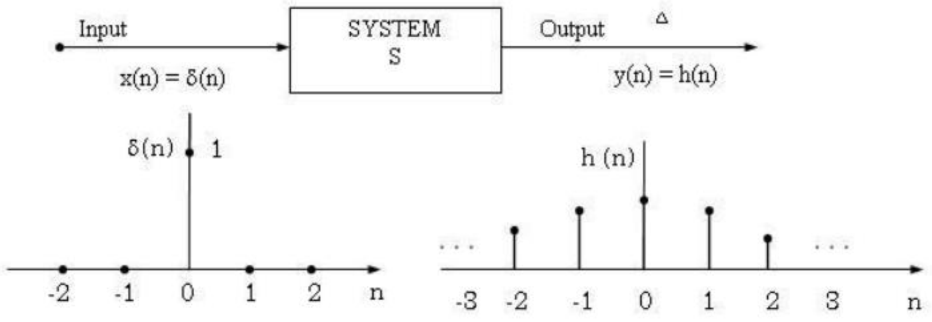
For example, the aim of digital filters is to modify the input signal in some way, such as suppressing unwanted frequencies, enhancing desired frequencies, or removing noise from the signal. In general, digital filters are designed for:

- Improve SNR
- Reduce power noise
- Artifact reduction
- Extracting features
- Waves (matched filters)
- Extraction of rhytms (ECG)

Digital filters are designed based on the mathematical model of the system that is being filtered, and the filter coefficients are chosen based on the desired filter response. The filter response determines the behaviour of the filter and can be specified in various ways, such as in terms of the magnitude response, the phase response, or the impulse response. The filter coefficients are then computed using various numerical methods, such as the least-squares method, the gradient method, or the impulse response method.

The impulse response: When an impulse is applied at the input, the output is known as impulse response of the filter. So, the impulse response represents the response of the filter to a unit impulse signal.





The impulse response contains all the information about the filter properties. The impulse response can be used to characterize the behaviour of a system, such as its stability, causality, linearity, and time-invariance.

- Stability: refers to the property of a system where its output remains bounded for all inputs, even if the input is a sequence of signals that grow indefinitely large in magnitude. A stable system is one that does not produce unbounded outputs, even if the inputs are unbounded. Stability is a crucial property for digital filters, as it ensures that the output of the filter does not grow uncontrollably and cause overflow or other numerical problems.

$$\sum_{k=-\infty}^{\infty} |h[k]| < \infty$$

- Causality: Causality refers to the property of a system where the output depends only on the present and past inputs and not on the future inputs. A causal system is one that produces outputs that are purely a function of the input up to the present time. Causality is a crucial property for digital filters, as it ensures that the filter can be implemented in real-time, where the inputs are not known ahead of time.

$$h[n] = 0 \qquad per\ t < 0$$

Stability and causality are important concepts in digital signal processing and control systems because they ensure that the system will produce well-behaved outputs for any input signals. By designing filters that are stable and causal, it is possible to ensure that the filter will produce the desired results in a predictable and reliable manner.

The impulse response of a system can be used to compute the response of the system to any input signal. This is done using the convolution operation, which is a mathematical operation that represents the relationship between the input and output of a linear, time-invariant system.

In essence, the convolution of the impulse response with an input signal computes the output signal of the system by taking into account the influence of each sample of the input signal on each sample of the output signal. By computing the convolution, it is possible to obtain the complete response of the system to any input signal, regardless of the length or shape of the input signal.

For example, if you have a filter with an impulse response $h(n)$ and an input signal $x(n)$, you can compute the output signal $y(n)$ by convolving the two signals:

$$y[n] = \sum_{k=0}^{\infty} x[k]\, h[n-k]$$

The sum from k=0 to infinity represents the cumulative effect of all samples of the input signal on the current sample of the output signal. The result of the convolution is a signal that represents the complete response of the system to the input signal. By using the convolution, it is possible to obtain the complete response of the system to any input signal, regardless of the length or shape of the input signal.

According to the impulse response, filters are divided into:

- FIR (finite impulse response), which are for sure stable.
- IIR (infinite impulse response), which can be stable or unstable.

Frequency response:

The DFT of the impulse $h[n]$ is the frequency response of the filter.

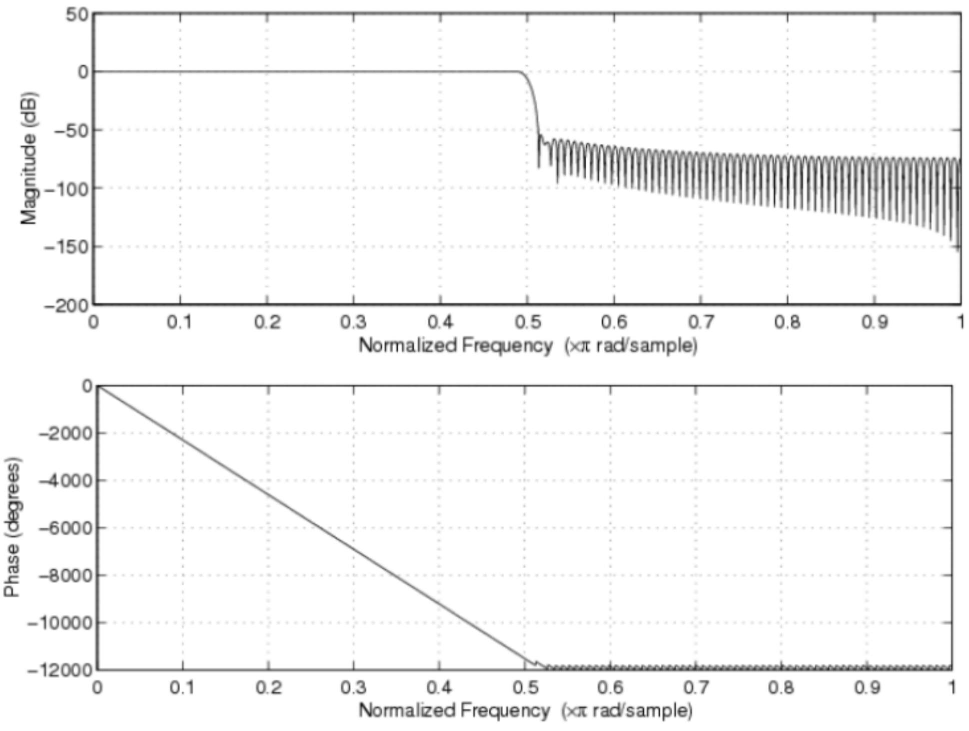
The frequency response of a system is a measure of how the system behaves at different frequencies. It is usually represented in terms of the magnitude and phase of the system's transfer function, which is the Fourier transform of its impulse response.

The magnitude response, also known as the modulus response, is a measure of the gain of the system at different frequencies. It is expressed in decibels (dB) or linear units, and it indicates how much the system amplifies or attenuates signals at different frequencies. The magnitude response is a crucial aspect of system behaviour, as it determines the overall gain of the system, the amount of distortion and noise, and the overall performance of the system.

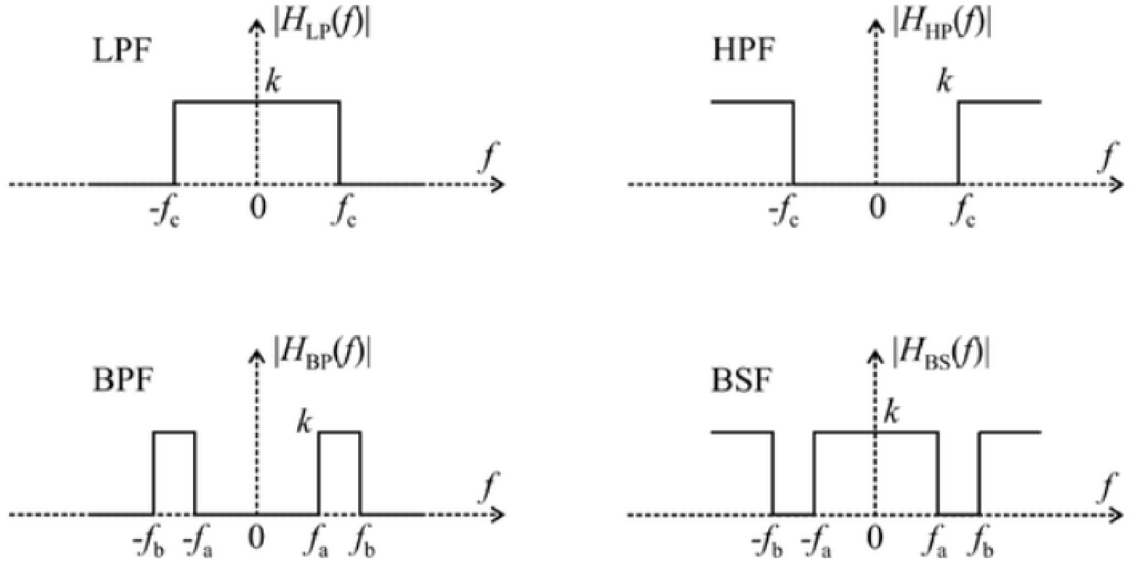
The phase response, on the other hand, is a measure of the phase shift of the system at different frequencies. It indicates how much the system delays or advances signals at different frequencies. The phase response is an important aspect of system behaviour, as it determines the timing of the output signal with respect to the input signal and the overall stability of the system. A liner phase response is required to avoid signal distortion in the filter output. All the frequencies are delayed by different delay but although they are exactly recombined in order not to introduce any distortion.

FIR filters can be designed in order to have a perfect linear phase, while IIR filters can be designed in order to have an almost linear phase, but not perfectly linear.

The frequency response of a system is usually represented using a Bode plot, which is a graph of the magnitude response against frequency on a logarithmic scale, and the phase response against frequency on a linear scale.



There are different classes of filters: Low-Pass, High-Pass, Band-pass, Band-stop



Transfer function $T[\,]$:

The transfer function is a mathematical representation of a linear time-invariant system (LTI system). It describes the relationship between the input and output signals of the system in the frequency domain. The transfer function is a complex function that represents the gain and phase shift of the system at different frequencies.

For a large class of filters $T[\,]$ can be expressed by:

$$y[n] = \sum_{k=1}^{N} a_k y[n-k] + \sum_{m=0}^{M} b_m x[n-m]$$

Where a_k and b_m are the coefficients. What identify the filters are these coefficients.

For LTI filters, the previous equation is the general form. It is the sum of two main components, the first one is a weighted sum of previous output (AR part), the second one is the weighted sum of the inputs, current and previous.

In the z-domain we have the filter transfer function is represented as a rational polynomial, and the system's behaviour can be analyzed using tools such as pole-zero plots and stability analysis:

$$H[z] = \frac{\sum_{m=0}^{M} b_m z^{-m}}{1 + \sum_{k=1}^{N} a_k z^{-k}} = \frac{b_0 z^{M-N} \prod_{m=1}^{M}(z - z_m)}{\prod_{k=1}^{N}(z - p_k)}$$

Where in the numerator $(z - z_m) = 0$ are the zeros of the system, and $(z - p_k) = 0$ are the poles of the system.

The transfer function H(z) provides information such as:

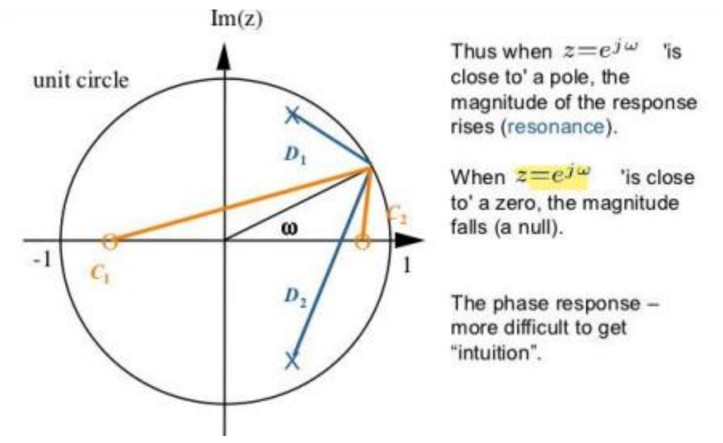
- Stability of the filter: $|p_k| < 1$ (if the modulus of the poles is less than one, the system is stable)
- Frequency response: $H[\omega] = H(z)|_{z=e^{j\omega}}$
- Rough estimation of the frequency response

[<u>Zeros and poles</u>: Zeros are points in the complex plane where the transfer function of a system equals zero. In the context of digital filters, the zeros represent the frequencies at which the filter has no gain, or zero gain. Zeros can be used to determine the behaviour of the system at certain frequencies, and they play a key role in determining the frequency response of the system.

Poles are points in the complex plane where the transfer function of a system is infinitely large. In the context of digital filters, the poles represent the frequencies at which the filter has infinite gain, or an unbounded response. Poles play a key role in determining the stability of the system, and they can be used to design filters with specific stability characteristics.

The zeros and poles of a system can be visualized in the complex plane as a pole-zero plot, which provides a graphical representation of the behaviour of the system at different frequencies. The pole-zero plot is useful for analyzing the stability, causality, and frequency response of the system, and it is used in the design of digital filters to achieve specific responses.]

We can roughly estimate the modulus of H[ω] by moving around the unitary circle and computing the distances between poles/zeros and the point on the circle:

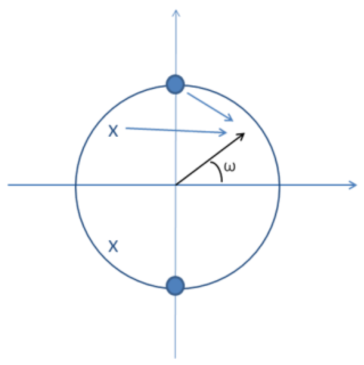


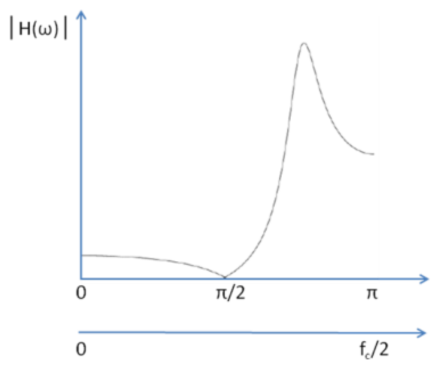
Thus when $z = e^{j\omega}$ 'is close to' a pole, the magnitude of the response rises (resonance).

When $z = e^{j\omega}$ 'is close to' a zero, the magnitude falls (a null).

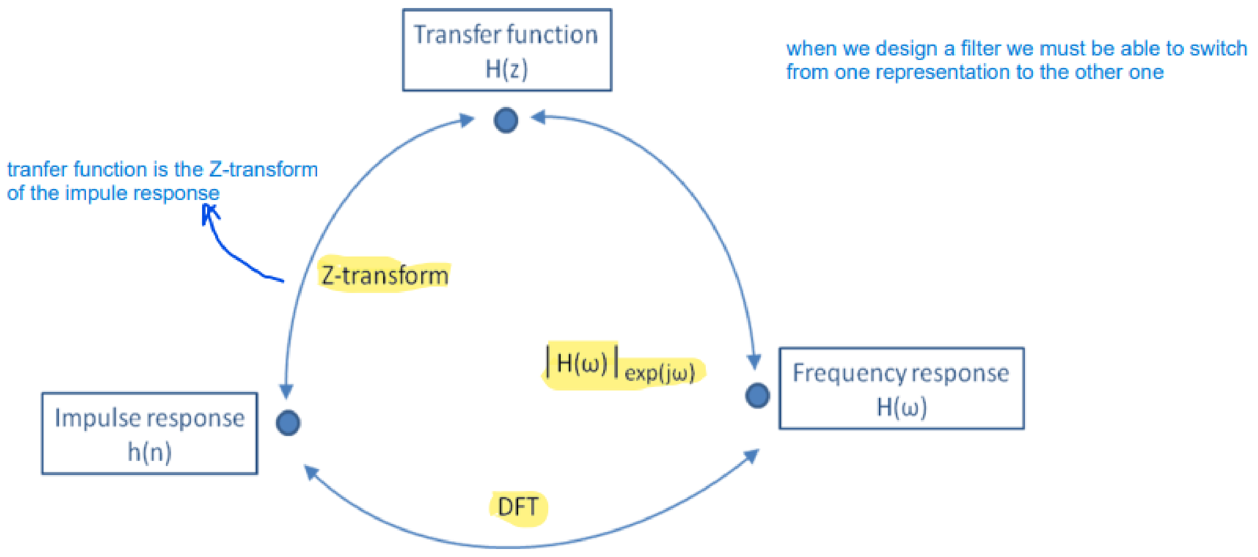
The phase response – more difficult to get "intuition".

To cancel a certain frequency place a zero on the circle in correspondence of the desired frequency. Poles and zeros in the origin do not contribute to |H(w)|.





Summary:


Transfer function H(z)
when we design a filter we must be able to switch from one representation to the other one
tranfer function is the Z-transform of the impule response
Z-transform
|H(ω)| exp(jω)
Frequency response H(ω)
Impulse response h(n)
DFT

The relationship between the time-domain signals and the z-domain signals is given by the z-transform, which is a mathematical transform that maps time-domain signals to the z-domain. The inverse of the z-transform, the inverse z-transform, maps z-domain signals back to the time domain.

FIR filter design:

Digital FIR filters are designed directly in their discrete time version. The most widespread technique is surely window design. The main idea behind window design is that of selecting an ideal filter which fulfils the basic requirements (e.g. an ideal lowpass filter, an ideal differentiator). Such ideal filter generally has non-finite and non-casual impulse response which is then truncated and shifted to obtain a finite and causal function.

When designing a lowpass filter, given the fact that an ideal lowpass filter has a single tuneable variable – its cut-off frequency – the remaining design requirements (maximum passband ripple, minimum stopband attenuation and width of the transition band) can only be met by selecting the shape and the length of the window employed for truncating the ideal impulse response. The focus is on the window employed, and thus the name “window” design.

The method is very straightforward; The things to keep in mind are:

• The minimum attenuation in stopband, As, does not depend on the length of the window L (to a very good approximation). In fact, it depends on the secondary lobes of the DTFT of the window employed. To increase it, you just need to change window.

• The maximum ripple in passband, Rp, also does not depend on the length of the window L. Therefor building a longer filter does not reduce Rp. In general, $\delta 1 \approx \delta 2$.

• The width of the transition band depends on both the window employed and the length of the window. (In fact, it depends on the width of the main lobe of the frequency response of the window.) Unfortunately, the larger As and more the width of the transition band, at given L.

FIR and IIR filters:

The impulse response or the frequency response classify digital filters. An impulse response is the response of a filter to an input impulse: $x[0] = 1$ and $x[i] = 0$ for all $i \neq 0$. The Fourier Transform of the impulse response is the frequency response of the filter which illustrates its gain when the frequencies differ.

If the impulse response of the filter falls to zero after a finite period of time, it is an FIR (Finite Impulse Response) filter. However, if the impulse response exists indefinitely, it is an IIR (Infinite Impulse Response) filter. How the output values are calculated determines whether the impulse response of a digital filter falls to zero after a finite period of time. For FIR filters the output values depend on the current and the previous input values, whereas for IIR filters the output values also depend on the previous output values.

<u>Advantages and Disadvantages of FIR and IIR Filters:</u>

The advantage of IIR filters over FIR filters is that IIR filters usually require fewer coefficients to execute similar filtering operations, that IIR filters work faster, and require less memory space.

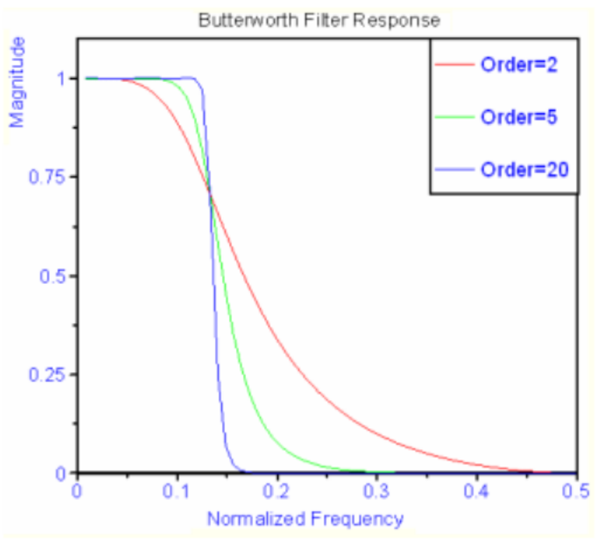
The disadvantage of IIR filters is the nonlinear phase response. IIR filters are well suited for applications that require no phase information, for example, for monitoring the signal amplitudes. FIR filters are better suited for applications that require a linear phase response.

<u>IIR Filters</u>

The output values of IIR filters are calculated by adding the weighted sum of previous and current input values to the weighted sum of previous output values. If the input values are x_i and the output values y_i, the difference equation defines the IIR filter:

$$y_i = \frac{1}{a_0}\left[-\sum_{j=1}^{N_y-1} a_i y_{i-j} + \sum_{k=0}^{N_x-1} b_k x_{i-k}\right]$$

The number of forward coefficients N_x and the number of reverse coefficients N_y is usually equal and is the filter order. The higher the filter order, the more the filter resembles an ideal filter. This is illustrated in the following figure of a frequency response of lowpass Butterworth filters with different orders. The steeper the filter gain falls, the higher the filter order is.

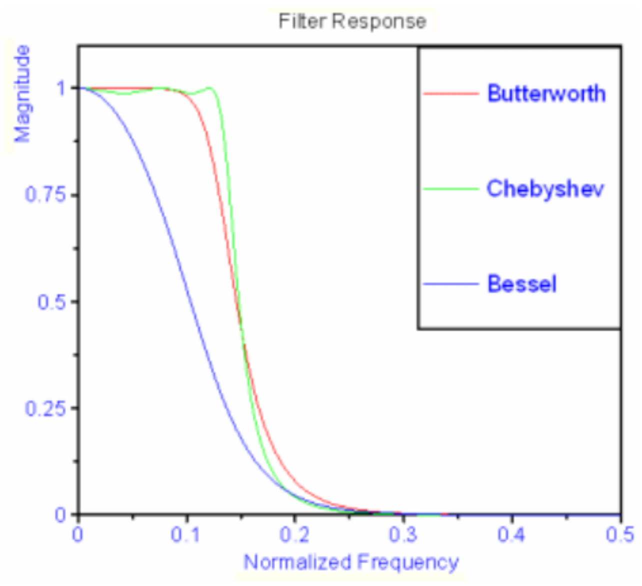


Butterworth Filters: The frequency response of the Butterworth filter has no ripples in the passband and the stopband. Therefore, it is called a maximally flat filter. The advantage of Butterworth filters is the smooth, monotonically decreasing frequency response in the transition region.

Chebyshev Filters: If the filter is the same, the frequency response of the Chebyshev filter has a narrower transition range than the frequency response of the Butterworth filter which results in a passband with more ripples. The frequency response characteristics of Chebyshev filters have an equiripple magnitude response in the passband, monotonically decreasing magnitude response in the stopband, and a sharper roll off in the transition region as compared to Butterworth filters of the same order.

Bessel Filters: The frequency response of Bessel filters is similar to the Butterworth filter smooth in the passband and in the stopband. If the filter order is the same, the stopband attenuation of the Bessel filter is much lower than that of the Butterworth filter. Of all filter types the Bessel filter has the widest transition range if the filter order is fixed.

The following figure compares the frequency response with a fixed filter order of the IIR filter types Butterworth, Chebyshev, and Bessel:



<u>FIR Filters</u>

FIR filters are also known as non-recursive filters, convolution filters, or moving-average filters because the output values of an FIR filter are described as a finite convolution:

$$y_i = \sum_{k=0}^{N-1} b_k x_{i-k}$$

The output values of a FIR filter depend only on the current and past input values. Because the output values do not depend on past output values, the impulse response decays to zero in a finite period of time. FIR filters have the following properties:

- FIR filters can achieve linear phase response and pass a signal without phase distortion.
- They are easier to implement than IIR filters.

The selection of the window function for a FIR filter is similar to the choice between Chebyshev and Butterworth IIR filters where you have to choose between side lobes near the cut-off frequencies and the width of the transition region.

LAB:

Let's consider the following linear FIR filter: $y[n] = \frac{1}{6}\sum_{k=0}^{5} x[n-k]$. *A) Plot zeros and poles in the complex z-plane; B) Plot the frequency response of the filter. Compare the frequency response and position of poles and zeros. C) Compute the cut-frequency when signal x[n] is sampled at: 1) 300 Hz; 2) 100 Hz. D) Plot the impulse response. Which is the filter delay? E) Describe what happens when the filter is applied twice. Compare the frequency response. F) Filter the ECG signal contained in the file ecg2.dat (signal is sampled at 250 Hz). Superimpose the ECG signals before and after filtering*

I develop the transfer function:

$$y[n] = \frac{\frac{1}{6}(x(5) + x(4) + x(3) + x(2) + x(1) + x(0))}{1}$$

```
m=[0:5];
b=1/6*[1 1 1 1 1 1];
a=[1 0 0 0 0 0];

% poles and zeros (A)
zplane(a,b)

%Frequency response (B)
[h, w]=freqz(b,a,1000); % 1000 is the number of frequency samples to compute

figure()
subplot(2,1,1)
plot(w/pi,20*log10(abs(h))) %magnitude
subplot(2,1,2)
plot(angle(h)) %phase
```

%% The location of zeros of H(z) can be used to design filters to null out specific frequencies. Zeros are placed on the unit circle, at locations corresponding to the frequencies where the gain needs to be 0

```
%% cut frequency (frequency in which the magnitude is -3 dB) (C)
Fs=300;
[h300, w300]=freqz(b,a,Fs);

figure()
subplot(2,1,1)
title('Fs = 300 Hz')
plot(w300,20*log10(abs(h300))) %magnitude
subplot(2,1,2)
plot(angle(h300)) %phase

Fs1=100;
[h100, w100]=freqz(b,a,Fs1);

figure()
subplot(2,1,1)
title('Fs=100 Hz')
plot(w100/pi,20*log10(abs(h100))) %magnitude
subplot(2,1,2)
plot(angle(h100)) %phase
```

The magnitude of the frequency response is plotted on the y-axis in dB, which is obtained by taking the logarithm of the magnitude of H(ω) and multiplying by 20. The frequency response is plotted on the x-axis in units of radians per second, which is obtained by dividing the frequency, W by pi.

```
%% impulse response (D)
[Himp, T] = impz(b, a, 30);
stem(T, Himp);
title ('Impulse Response of Digital Filter');
N=length(Himp); %N is the number of coefficients (filter order)
```

FIR filter delay → filtering a signal introduces a delay, so the output signal is shifted in time with respect to the input. IIR filters delay some frequency components more than other. They distort the input signal. Instead, FIR filters delay all frequency components by the same amount

The initial state for all samples in an FIR filter is 0. The filter output until the first input sample reaches the middle tap (the first causal sample) is called the transient response, or filter delay. Given an FIR filter which has N taps, the delay is (N - 1)/2 samples (in time: (N-1)/2fs)

The output is delayed with respect to the input. If we filter a signal, for example an ECG, what we see on the monitor what we see is delayed, it is not what is happening real time in the patient. So, knowing the delay is very important, because if the delay is too long (for example 2 minutes), the patient could die in the meantime.

```
%filter delay - 300 hz
delay=(N-1)/(2*Fs); %in time
delay_s=(N-1)/2;

% filter delay - 100 hz
delay1=(N-1)/(2*Fs1); %in time

%I apply the filter twice (E)
twice=filter(b,a,Himp);
figure()
stem(twice); %with stem I obtain the x-axis that starts from 1, I should rescale it and
make it starting from 0

%% Frequency response of the double filter
[Htwice, Wtwice]=freqz(twice, a, 1000)

figure()
subplot(2,2,1)
plot(Wtwice/pi,20*log10(abs(Htwice))) %magnitude
subplot(2,2,3)
plot(unwrap(angle(Htwice))) %phase

%plot the magnitude without log scale
subplot(2,2,2)
plot(Wtwice,abs(Htwice)) %magnitude

%% Comparison between frequency responses

figure()
plot(w,abs(h)) %magnitude
hold on
plot(Wtwice,abs(Htwice)) %magnitude
```

We notice that applying the filter twice we have a small reduction in the pass band, which is still acceptable, but we have a strong reduction of the ripples in the stop band, which is very good because we attenuate the noise better.

```
%% ecg2.dat (F)
data=importdata('ecg2.dat');
Fs_ecg=250;
ecg_filt=filter(b,a,data);

ecg_filt2=filter(b,a,ecg_filt); %signal filtered twice

filtECG=filtfilt(b,a,data)
figure()
plot(data)
hold on
plot(ecg_filt)
hold on
plot(ecg_filt2)
hold on
plot(filtECG)

legend({'ECG','ECG     filtered',     'ECG     filtered     twice',     'ECG     filt
filt'},'Location','northeast','Orientation','vertical')
```

I notice that with filtfilt i have the filtered version of the signal, but without delay. When we apply the filter twice, the direction is the same the first time and the second time, so the first introduces delta and the second introduces 2delta.
filtfilt applies the filter once, and then it takes the result and swipe it on the time axis (mirror it), so the delay introduced by the first filter is cancelled by the second operation it performs (the swapping of the time axis). This swapping deletes the delta (the delay).

roots(c)→ This function returns the roots of a polynomial with coefficients given in the vector c. For example, if c represents the polynomial c(z) = 1 + 2z + 3z^2, then roots(c) will return the roots of this polynomial, which are the solutions to the equation c(z) = 0.

polar(theta, r)→ This function is used to convert polar coordinates (represented by theta and r) to Cartesian coordinates. The argument theta represents the angle of the polar coordinate, and r represents the magnitude of the coordinate. The function returns a complex number, where the real part represents the Cartesian x-coordinate and the imaginary part represents the Cartesian y-coordinate.

zplane(z,p) → plots the zeros specified in a column vector z and the poles specified in a column vector p in the current figure window . The plot includes the unit circle for reference. If z and p are matrices , then zplane plots the poles and zeros in the columns of Z and P in different colours.

freqz→[h,w]=freqz(b, a, n) This function is used to compute the frequency response of a digital filter with numerator coefficients b and denominator coefficients a. The argument n specifies the number of frequency samples to compute. The function returns the magnitude and phase response of the filter at each frequency sample. h is a vector of length n, w is a vector whose values range from 0 to π.

impz→ [H, t]=impz(b,a). this function returns the impulse response of the digital filter with numerator coefficients b and denominator coefficients a. The function chooses the number of samples and returns the response coefficients in h and the sample times in t. Impzl(_) with no output arguments plots the impulse response of the filler .

filter→ filtered_x=filter(b, a, x)→ This function is used to filter an input signal x with a filter represented by the numerator coefficients b and denominator coefficients a. The function returns the filtered signal.

conv(a, b)→ This function is used to compute the convolution of two signals, represented by a and b. The convolution operation is a mathematical operation that combines two signals to produce a third signal, which represents the interaction between the two signals.

fir1(n, wn)→ This function is used to design a finite impulse response (FIR) filter. The argument n specifies the order of the filter, and wn specifies the normalized cut-off frequency. The function returns the numerator coefficients of the filter. Applying this function introduces a delay, so the output will be delayed with respect to the input.

cheby1 (n, r, wn)→ This function is used to design a Chebyshev type I filter. The argument n specifies the order of the filter, r specifies the maximum ripple in the passband, and wn specifies the normalized cut-off frequency. The function returns the numerator coefficients of the filter.

filtfilt(b, a, x)→ This function is used to apply a zero-phase filter to an input signal x. The filter is represented by the numerator coefficients b and denominator coefficients a. The function returns the filtered signal. This function applies the filter once, and then it takes the result and swipe it on the time axis (mirror it), so the delay introduced by the first filter is cancelled by the second operation it performs (the swapping of the time axis). This swapping deletes the delta (the delay). So, it filters the signal without delay.

loglog(x, y)→ This function is used to create a logarithmic plot of the data in y as a function of the data in x. The logarithmic scaling is applied to both the x-axis and the y-axis. This type of plot is useful for analyzing data that spans several orders of magnitude.

kaiser→ w=kaiser(L, beta) returns an L-point kaiser window with shape factor beta. Beta is specified as a positive real scalar. The parameter beta affects the sidelobe attenuation of the Fourier transform of the window.

butter→[b,a]=butter(n, Wn, ‘ftype’). This function desingns a Butterworth filter. n is the filter order (an integer greater than or equal to 1; Wn is the cut-off frequencies of the filter. ‘ftype’, is the type of filter and it can be: ‘low’ for a lowpass filter, ‘high’ for a highpass filter, ‘bandpass’, for a bandpass filter, or ‘stop’ for a stopband filter. The output are the coefficients of the transfer function, b the numerator coefficients and a the denominator coefficients. To apply this filter to a signal y, we use the filtfilt function as follows: y_filtered=filtfilt(b,a,y).

iir→[b,a] = iir(n, Wn, ‘ftype’). This function is used to design an irr digital filter. The iir function takes as input the desired filter order (n), the cut-off frequency or frequencies (Wn), and the type of filter (ftype) and returns the filter's transfer function in terms of its zeros (z), poles (p), and gain (k). To apply the IIR filter to a signal y, we can use the filtfilt function as follows: y_filtered=filtfilt(b,a,y)

SPECTRAL ANALYSIS:

Spectral analysis is a mathematical technique used in signal processing and engineering to study the frequency content of signals. It involves transforming a time-domain signal into its frequency-domain representation, allowing the analysis of the frequencies present in the signal, their amplitude, and phase.

In other words, spectral analysis studies how the signal is distributed across frequencies. The power spectrum (or Power Spectral Density, PSD) is used to describe the power of a signal in the frequency domain.

The power spectral density (PSD) of a signal is a measure of the distribution of its energy across different frequencies. The PSD estimator is used to estimate the PSD of a signal from a finite length sample of the signal.

There are different estimators (an estimator is something that is approaching the solution). There are two different families of estimators:

- Non parametric, based on DFT
- Parametric, based on some kind of signal modelling. We have the signal, we build a model which is supposed to have generated the signal, and then we compute the PSD from the model. This way, knowing how the model il inside, the PSD is based on the parameters of the model.

In the context of PSD estimation, non-parametric estimators are those that do not assume any specific functional form for the underlying PSD. Instead, they rely on the empirical distribution of the data and make fewer assumptions about the distribution. Examples of non-parametric PSD estimators include the periodogram and the Welch method.

Parametric estimators, on the other hand, make assumptions about the underlying PSD and model the PSD using a specific functional form, such as a sum of sinusoids. The parameters of the model are then estimated from the sample data. Examples of parametric PSD estimators include the autoregressive (AR) model and the moving average (MA) model.

In general, non-parametric PSD estimators are more robust to deviations from the assumed distribution and can provide a good estimate of the PSD for signals with complex spectra, but may have a higher variance compared to parametric estimators. Parametric PSD estimators, on the other hand, can be more computationally efficient, but may not provide an accurate estimate of the PSD for signals that deviate significantly from the assumed functional form.

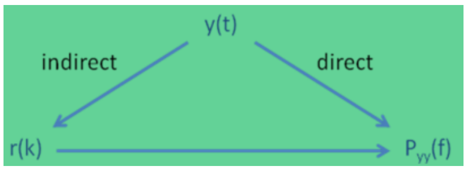
Non parametric approaches:

Non parametric approaches are based on the computation of the DFT of the signal. They are divided into two classes:

- Direct methods: direct methods compute the PSD directly from the sample data, without making any assumptions about the underlying distribution of the data. Examples of direct methods include the periodogramand Welch method. These methods are usually based on Fourier transforms and use window functions to reduce spectral leakage.
- Indirect methods: indirect methods, on the other hand, do not estimate the PSD directly. Instead, they estimate some other statistical measure, such as the covariance or auto-correlation function, and then transform it to obtain the PSD. Examples of indirect methods include the maximum entropy method, Burg method, and Yule-Walker method.

In general, direct methods are more straightforward and computationally efficient, but may have a higher variance compared to indirect methods. Indirect methods, on the other hand, can provide a more accurate estimate of the PSD, especially for signals with complex spectra, but may be more computationally intensive.

At a certain part of the process, we will use the DFT for the computation of the spectrum. So, if we learn to use DFT very well, the solution will be very easy. There are two types of non-parametric approach: direct and indirect. In the indirect we use the signal to estimate of the autocorrelation function and then using the DFT we obtain the power spectrum, while the direct method just apply the DFT to the signal.



<u>Direct methods:</u>

- Periodogram: we get the DFT of the sample data, we square it and we normalize it for the number of samples:

$$P_{xx}(f) = \frac{1}{NT}|DFT\{x\}|^2$$

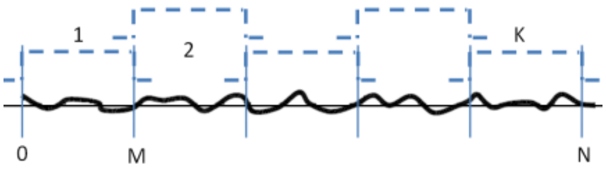
- Windowed periodogram: it is always better not to directly perform the DFT, but to use windows. The windowed periodogram performs the DFT on the windowed signal:

$$P_{xx}(f) = \frac{1}{NT}|DFT\{x(t)w(t)\}|^2$$

  Limits of the periodogram:
  - Frequency resolution is limited by N (number of samples), due to DFT properties. This limitation is inherited from the fact that we use DFT. If we have N samples, we will only be able to describe N frequencies. The delta is $\frac{2\pi}{N}$. Due to this, we have to plan the experimental session in order to collect at least N samples.
  - The estimator of $P_{yy}(f)$ is biased and non-consistent. It does not converge to the true spectrum even when the number of samples N goes to infinite. The PSD that we obtain is an estimator, which are evaluated in terms of bias. The idea is that the variance of the error computed by the estimator should be lower and lower as N increases. This does not work with the periodogram. Actually, the variance of the error for the periodogram is exactly the power spectrum, so the estimator does an error that is equal to the quantity it estimates. So, the bias is not a real problem because it can be corrected, while non -onsistency is a big problem. This is why consistent methods have been introduced: Bartlett, Welch and Daniell.

Consistent estimators:

- Bartlett method: Instead of computing a single measurement, performs several measurements on non-overlapping section of the signal, and then it averages the results.
  1. the signal is divided into K segments, each containing M points (K*M=N, number of samples)



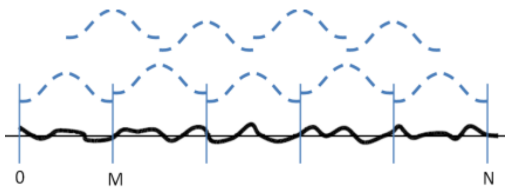
  2. the PSD is estimated in each segment, so we obtain K sub-spectra (so we have K PSDs).
  3. The estimate is obtained by average of each sub-spectrum. The larger number of segments, the better this estimator is, but if K is larger, M is lower and lower, and so we have a kind of balance.

$$P_{xx}(f) = \frac{1}{K}\sum_{k=1}^{K} P_{xx}^{k}(f)$$

Limits of Bartlett method: The larger the number of segments, the better the convergence of the estimator. However, increasing the number of segments will reduce its length and therefore the frequency resolution.

So, this method performs a balance between the convergence and the resolution

- Welch method: it is similar to the Bartlett method, but windows may overlap.



In this method, the number of segments may be increased without reducing the segment length M.

Limitations: The average of the PSDs works if we are assuming that the segments are independent, but in this case, there is the possibility of overlapping the segments, so they are not independent anymore, so the average of the PSDs does not work the same. It is suggested not to go beyond 50% of overlapping, in order to keep somehow the segments independent, otherwise we will count in the average a segment twice

- Daniell method: we perform the periodogram and THEN we filter the spectrum that we obtained. when we filter it there will be a delay, so the spectrum will be shifted on frequencies, so we have to have a zero-phase filter or we will have to correct by the filter delay.

Parametric methods:

The power spectral density (PSD) is obtained through the identification of a parametric model, usually an AutoRegressive (AR) Model (in which the output is given by regression of the previous outputs).

The signal is described as the output of an AR model fed by a white-noise $w(n) = WN(0, \sigma^2)$:

$$Y[n] = -\sum_{k=1}^{p} a_k y[n-k] + w[n]$$

Where a_k are the coefficients and p is the model order.

There is an AR part plus a White Noise part, whose contribution is unpredictable. So, if we want to the describe the signal Y[n], we consider w[n] as negligible.

The transfer function becomes:

$$H[z] = \frac{Y[z]}{W[z]} = \left(1 + \sum_{k=1}^{p} a_k z^{-k}\right)^{-1}$$

If the poles are closer to the unit circle, then there is a peak in the filter response and so we have a White Noise as input, and then the pole picks up some frequencies that will be the peaks in the spectrum. This is called coloured filter, because we start from a white noise and then we see some peaks in the spectrum.

If the properties of the model are known, the spectrum can be estimated as follows:

$$S_{yy}[\omega] = S_{ww}[\omega]|H[\omega]|^2$$

The AR component is modelled as a sum of p exponential terms, where each term is a function of the AR coefficients $a_1, a_2, ..., a_p$ and the frequency ω.

The white noise component is added to account for any unstructured or random component of the signal that is not accounted for by the AR component (the unpredictable part).

We compute the spectrum with the previous relationship. So, the spectrum of the output is given by the spectrum of the input (white noise) multiplied by the squared modulus of the transfer function.

$$S_{yy}[\omega] = \frac{\sigma^2 T}{|1 + \sum_{k=1}^{p} a_k z^{-k}|^2_{z=\exp(j\omega T)}} \quad (*)$$

As we see, the spectrum of the white noise is equal to:

$$S_{ww}[\omega] = \sigma^2 T$$

Where σ^2 is its variance and T represents the total power of the signal, which is proportional to the length of the signal in the time domain. By multiplying the variance of the white noise process by T, we obtain the total power of the signal in the frequency domain. $\sigma^2 T$ is a number.

Thus, the problem of estimating the spectrum is reduced to the problem of estimating the model parameters and the properties of the input noise.

Parameter estimation: Different algorithms are available and based on the minimization of a figure of merit:

$$J = \sum_{k=1}^{N} e[n]^2 = \sum_{k=1}^{N} (y[n] - \hat{y}[n])^2$$

Where $y[n]$ are the real values and $\hat{y}[n]$ are the estimated values.

There are different approaches:

- Yule-Walker: if we know the model a priori, solution can be found solving a linear system.
- Levinson-Durbin: recursive estimation of the model coefficients for any order.

Noise properties estimation: Once the model parameters have been estimated, we would like to check if the model is able to capture all the information contained in the signal. We can guarantee that the model is able to capture all the info in the signal, if there is no more info in the error. This happens when the error is a white noise process (so something unpredictable). So, if the error is a white noise process, we can compute the variance of the input noise as the variance of the error, since we are able to compute e[w], while we are not able to compute the white noise that affects the original signal (input white noise).

So, we compute the error, which is the difference between the measure and the prediction. The best prediction we can have is the auto regressive part, since the noise is not predictable. We can compute the variance of the input noise by computing the variance of the error.

$$e[n] = y[n] - \hat{y}[n] = y[n] - \sum_{k=1}^{p} a_k y[n-k]$$

In other words, in order to be a good model, the error must be a white-noise process (a process in which any sample does not carry information on the future and past samples), otherwise I was not able to capture all the info contained in the signal. Thus:

$$e[n] \rightarrow WN(0, \sigma^2)$$

Model order: To completely define the model, the value of p must be defined. There are different criteria:

- Final Prediction Error (FPE):

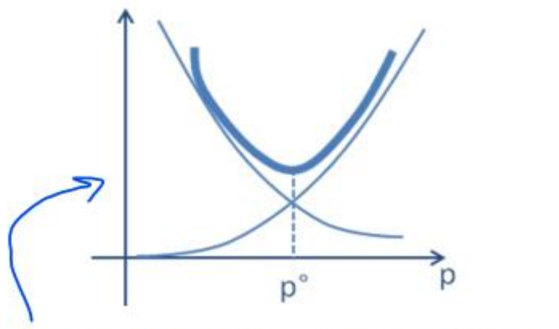
$$FPE = \sigma^2 \frac{N+p+1}{N-p-1}$$

- Akaike Information Criterion (AIC):

$$AIC = Nln(\sigma_e^2) + 2p$$

  This equation consists of two terms, the first one $Nln(\sigma_e^2)$ represents the variance of the error, and the second one $2p$ represents the penalizing factor.

  On one side, the variance of the error is decreased with the increase of the model order, while the penalization factor increases with the increase of the model order. So, we have two factors, one favouring higher model order and the other one penalizing the higher model order. They together give the minimum in the figure:



the optimal model order is in correspondance to the minimum of the blue curve

  I'm able to reduce the prediction error if I increase the model order, but this is not a good strategy, we have to find a balance. We can get the error as low as we like by increasing the model order, but on the other side we don't want to have overfitting, if we increase the order too much.

Practical issue:

- To estimate the spectrum we have
  a) estimate the model coefficients (a_k) and the model residuals (σ^2) for any order of interest.
  b) Different algorithms.
- Verify the correctness of the model (so, the whiteness of the residuals), by analyzing the properties of the error e(n), such as its autocorrelation function.
- Select the optimal model order (AIC, FPE criteria)
- Compute the power spectrum P(f), with equation $(*)$. What we have at the denominator, without the square, is exactly the modulus of the TF of a filter with an autoregressive part. The numerator is a number, that contains the variance of the error.

LAB:

*Estimates the power-spectral density of the data using **non-parametric approaches**:*

- *The periodogram method + windowing (do not use the periodogram function)*
- *Bartlett method*
- *Welch method*
- *Daniell's method*

```
data=importdata('rr1.dat'); % rr1.dat contains the series of RR intervals measured from
the ECG

% Finding the sampling frequency
RRtot=sum(data);
RRint=RRtot/length(data);
Fs=1/RRint;

% media mobile
MovMean = movmean(data,50*15);
data = data-MovMean;

% MEDIA MOBILE (GOLDEN RULE!!!!!!!!!!!!!!!!!)
Rimuovere la media ci permette di analizzare la frequenza delle oscillazioni, dal momento
che le oscillazioni sono più piccole rispetto all'andamento del segnale, e quindi lo
spettro si concentrerebbe solo nelle frequenze DC. rimuovere la media ci permette di
analizzare tutte le componenti in frequenza, anche delle oscillazioni più piccole

%% PERIODOGRAM (without using the periodogram function)

% Fast Fourier Transform (FFT):

  FFT=fft(data); % Fast Fourier Transform
  R_mag=abs(FFT); % Magnitude
  R_phase=angle(FFT); % Phase

  %Frequency bins
  n_sample=length(data);
  Fbins=((0:1/n_sample:1-1/n_sample)*Fs); %rescaling the frequency axis

  %Plot magnitude response
  figure()
  plot(Fbins, R_mag)
  title('Magnitude response')

  % Signal power information from FFT (PSD)
    psdfft=(1/(Fs*n_sample))*abs(FFT).^2;
    Fbins=((0:1/n_sample:1-1/n_sample)*Fs); %rescaling the frequency axis
```

The magnitude squared of the FFT is calculated using the "abs" and "^2" operators, which represents the power of the signal at each frequency. The factor of 1/(Fs * n_sample) is a normalization factor that ensures that the units of the PSD estimate are in terms of power per unit frequency. By dividing the magnitude squared of the FFT by (Fs * n_sample), the PSD estimate is normalized such that its integral over all frequencies represents the total power of the signal. This normalization ensures that the PSD estimate is independent of the number of samples in the signal and the sampling frequency.

```
    figure()
    plot(Fbins, psdfft)
    grid on
    title('Periodogram using FFT')
    xlabel('Frequency [Hz]')
    ylabel('Power/Frequency [dB/Hz]')


%% WINDOWED PERIODOGRAM

H = hanning(n_sample);
psdfft_windowed=(1/(Fs*n_sample))*abs(fft(data.*H)).^2;

figure()
plot(Fbins, psdfft_windowed)
grid on
title('Windowed Periodogram using FFT')
xlabel('Frequency [Hz]')
ylabel('Power/Frequency [dB/Hz]')

%% BARTLET METHOD
 % to perform Bartlett spectral estimation we can also use the pwelch
% function with overlap = 0
window_length=rectwin(50); %rectwin generates a rectangular window of length=50
noverlap=0;

[Pw,Fw] = pwelch(data,window_length,noverlap,n_sample,Fs)
figure()
plot(Fw, Pw)
grid on
title('Bartlett using Welch')
```

Away to do bartlett method is to implement the following function:

```
% function for Bartlet Spectral Estimation

function Px = bartlettPSD(x, K) %K is the number of windows and x is the signal

    Px=0;
    n1=1;
    M=floor(length(x)/K);

    for i = 1 : K
        Px=Px + periodogram(x(n1:n1+M-1))/K;
        n1=n1+M;
    end
end

%% WELCH METHOD

window_length=bartlett(20); %we use a bartlett window of length=20
noverlap=10; % 10% of overlapping

[pw, fw] = pwelch(data, window_length, noverlap, n_sample, Fs);

figure()
plot(fw, pw)
title('Periodogramma di Welch')
```

```
%% DANIELL'S METHOD

% I estimate the psd with the periodogram
psd=periodogram(data);
% and then I filter it
b=[1 1 1 1 1];
a=5;
D=filtfilt(b, a, psd)

figure()
plot(psd)
hold on
plot(D)
```

Estimate the spectral density of series in rr1.dat using a ***parametric method*** *based on autoregressive modeling. a) Use different models order (from 1 to 15). b) Find the minimum order whitening the residual. c) Estimate the optimal order using the Akaike Criterion.*

```
rr=importdata('rr1.dat');

%% Finding the sampling frequency

RRtot=sum(rr);
RRint=RRtot/length(rr);

Fs=1/RRint;

%% MEDIA MOBILE
MovMean = movmean(rr,50*15);
rr = rr-MovMean;

%% MODELLO
% N = ordine modello (proviamo da 0 a 15)
for N = 1:15
    modello=ar(rr,N);

    % Akaike criterion
    AIC(N) = aic(modello)

  % con la funzione pe calcoliamo il prediction error
    err=pe(modello,rr);
    figure(1)
    subplot(3,5,N)
    plot(err)

% calcoliamo l'autocorrelazione del prediction error (perchè il modello sia ottimale,
l'autocorrelazione dell'errore deve essere zero, perchè l'errore deve essere un rumore
bianco, quindi ogni suo punto deve essere scorrelato con ogni altro suo punto)
    [c,lags] = xcorr(err)
    figure(2)
    subplot(3,5,N)
    plot(lags,c)

    A=modello.A;
    [H,W] = freqz(1,A,256); % H is the frequency response
    H=(abs(H)).^2;
    VarNoise=modello.NoiseVariance;

    SS=(VarNoise*(1/Fs))*H;
    % loop sui subplot
    figure(3)
    subplot(3,5,N)
    plot(SS)
    % we see in the picture that for the first five orders the model is not
    % able to identify one of the two peaks, so it is not an optimal model

end

figure(4)
plot(AIC) % l'ordine migliore è in corrispondenza del minimo in figura
```

periodogram→Px=periodogram(x, window, nfft, fs). This function computes the periodogram power spectral density (PSD) estimate of a signal "x". The "window" parameter specifies the window function to use (e.g. Bartlett, Hamming, Hanning, etc.). The "nfft" parameter specifies the number of points to use in the FFT calculation. The "fs" parameter specifies the sampling frequency of the signal "x".

ar→th=ar(y,n). This function performs an autoregression (AR) model estimation on a signal "y". The "n" parameter specifies the order of the AR model to use. The output is a transfer function object "th" that represents the AR model estimate.

pe→err=pe(z, th). This function computes the prediction error of a signal "z" with respect to an AR model "th". The output is a prediction error "e".

xcorr→ [c,lags] = xcorr(a). This function computes the cross-correlation of a signal "a". The output is a cross-correlation vector c. To plot it: plot(lags, c)

pwelch→ [pw, fw] = pwelch(data, window_length, noverlap, n_sample, Fs). This function computed the Welch power spectral density estimate of a time-domain signal data using a sliding window approach. The input arguments are:

- data: a one-dimensional time-domain signal.
- window_length: the length of the window in samples
- noverlap: the number of samples that the sliding window should overlap
- n_sample: the number of points to use for the FFT (Fast Fourier Transform) calculation. If n_sample is not specified, it defaults to the value of window_length.
- Fs: the sample rate of data in Hz

The function returns two outputs:

- pw: the power spectral density estimate
- fw: the corresponding frequencies at which pw was estimated.

If we only write pxx=pwelch(x), the function returns the power spectral density (PSD) estimate pxx, of the input signal x, found usinf Welch's overlapped segment averaging estimator. By default, x is divided into the longest possible segments to obtain as close to 8 segments but without exceeding 8 segments, with 50% overlap. Each segment is windowed with a Hamming window. The modified periodograms are averaged to obtain the PSD estimate. pwelch (_) plots the Welch PSD estimate in the current figure window.