



EBOOK

SIGNAL PROCESSING

October 30, 2020

Roberta Varela & Lorenzo Moretti,
Altair Technical Specialists



Table of Contents

1. General Information	7
1.1. What is Altair Compose®?	7
1.2. Why Signal Processing?	7
1.3. Data Handling	9
1.4. Time and Frequency Domains	11
1.5. Visualization	11
1.6. Applications	12
1.6.1. Control Systems	12
1.6.2. Durability	13
1.6.3. Noise, Vibration and Harshness (NVH)	13
1.6.4. Electronics	14
2. Fundamentals	15
2.1. Signal Classification	15
2.1.1. Stationary and Non-stationary Signals	15
2.1.2. Continuous-time and Discrete-time Signals	15
2.1.3. Analog and Digital Signals	15
2.1.4. Finite-length and Infinite-length Signals	16
2.1.5. Periodic and Non-Periodic Signals	16
2.2. Elementary Operations	17
2.2.1. Amplitude Scaling	17
2.2.2. Bias	17
2.2.3. Addition, Subtraction, Multiplication, Division	18
2.2.4. Time Shifting	18
2.3. Complex Exponentials	19
2.4. Sampling Time & Sampling Frequency	19
2.5. Aliasing & Nyquist-Shannon's Theorem	21
2.6. Frequency Resolution	22
2.7. Energy and Power	22
3. File Input & Output	24
3.1. Test and CAE Files	24

3.2.	Readvectorbuilder Function	25
3.3.	Readvector Function	27
3.4.	Readmultivectors Function	27
3.5.	Readcae function	28
3.6.	Other CAE/Test Functions	29
	Exercises	30
4.	Data Processing & Preparation	31
4.1.	Detrending	31
4.2.	Spike Detection & Removal	32
4.3.	Resampling	35
4.3.1.	Upsampling	35
4.3.2.	Downsampling	35
4.3.3.	Resampling to fixed rate	36
4.4.	Feature Extraction	37
	Exercises	38
5.	Time-Domain Analysis	40
5.1.	Time Response	40
5.2.	Continuous Time	40
5.3.	Discrete Time	41
5.4.	Advantages of Discrete over Continuous Time	41
5.5.	Statistical Features	43
5.5.1.	Mean	43
5.5.2.	Median	43
5.5.3.	Standard Deviation	43
5.5.4.	Root Mean Square	44
5.5.5.	Crest Factor	44
5.5.6.	Skewness	46
5.5.7.	Kurtosis	47
5.6.	Autocorrelation, Cross-Correlation and Convolution	48
5.6.1.	Autocorrelation	48
5.6.2.	Cross-Correlation	50

5.6.3.	Convolution	51
	Exercises.....	52
6.	Fourier Analysis.....	53
6.1.	Frequency Domain.....	53
6.2.	Fourier Series	55
6.2.1.	Simplification for Even Functions	59
6.2.2.	Simplification for Odd Functions	59
6.3.	Gibbs Phenomenon	59
6.4.	Discrete Fourier Transform (DFT)	60
6.4.1.	Definition	60
6.5.	Fast Fourier Transform (FFT).....	64
6.5.1.	Definition	64
6.5.2.	FFT Function	66
6.5.3.	Symmetry Considerations for FFT	68
6.5.4.	Impacts of Frequency Resolution on the FFT Result	74
6.5.5.	Impacts of Sampling Frequency on the FFT Result	75
6.6.	Inverse Fast Fourier Transform	75
6.7.	Leakage	77
6.8.	Windowing	80
6.9.	Spectrogram	84
6.10.	Short-time Fourier Transform (STFT)	87
	Exercises.....	90
7.	Power Spectral Density.....	92
7.1.	Definition	92
7.2.	Parseval Theorem	92
7.3.	Windows Correction Factor	92
7.4.	PSD from DFT	93
7.5.	Welch's Method	95
7.6.	Creation of Time-History from PSD.....	99
	Exercises.....	102
8.	Filtering	103

8.1. Definition	103
8.2. Filter Classification	103
8.2.1. Digital and Analog Filters	103
8.2.2. Task Classification	104
8.3. Mathematical Formulation	104
8.4. Filter Specifications	105
8.5. Filter Stability	107
8.6. Digital Filters	108
8.6.1. Causal and Non-Causal	109
8.6.2. Finite Impulse Response (FIR) Filters	109
8.6.3. Infinite Impulse Response (IIR) Filters	110
8.6.4. Comparison between FIR and IIR	111
8.7. Design Methods	111
8.7.1. Analog Prototyping - Design Functions	112
8.7.2. Analog Prototyping - Functions for Filter Creation	113
8.7.3. Filter Functions	114
Exercises	115
References	116
Annex	117
1. DFT as a Change of Basis	117
2. Data Handling Strategies	118
2.1. Vectorization	118
2.2. Multidimensional Matrices	120
2.3. Sparse Matrices	121
3. Other Best Practices	123
3.1. Organizational Best Practices	123
3.2. Performance Best Practices	123
Exercises - Solutions	125
Chapter 3	125
Chapter 4	128
Chapter 5	136

Chapter 6	141
Chapter 7	150
Chapter 8	155

1. General Information

1.1. What is Altair Compose®?

Altair Compose® is an all-in-one math tool that provides various functions across different domains.

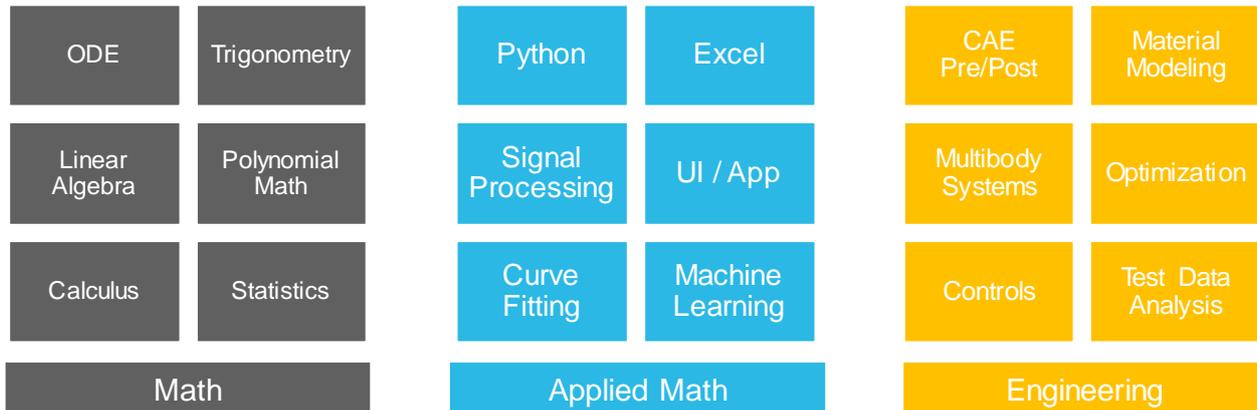


Figure 1. Multiple domains of Altair Compose®

- **Math:** Extensive coverage of functions to solve ordinary differential equations, trigonometry, linear algebra, polynomial math, calculus and statistics.
- **Applied Math:** A multilanguage environment that also supports Python, Excel reading capabilities, signal processing, creation of user interfaces and apps, curve fitting and machine learning.
- **Engineering:** CAE pre- and post-processing integration, helps with material modelling, multibody systems, optimization, controls and test data analysis.

OML stands for *Open Matrix Language* and is the main language used in Altair Compose, whose syntax is compatible with MATLAB® and Octave. It means its adoption is non-disruptive and can coexist with existing math tools. Finally, all the features mentioned above are within the standard installation accessible with Altair Units, via their patented licensing system business model, without need for toolboxes.

1.2. Why Signal Processing?

In a nutshell, a “signal” describing the evolution of a phenomenon and its field of study is called “signal processing”. There are many good reasons to carry out signal processing, such as:

- Extract useful information from sensors that cannot be measured, aiming to reduce the complexity of the signal into meaningful features.
- Improve transmission, storage efficiency and signal quality.

- Have a clear understanding of phenomena in countless applications, such as durability, noise, vibration and harshness (NVH), controls, electronics, finance, genomics, seismology and many others.

It is important to define the goals of the workflow, which can be quite diverse, but they all involve the idea of manipulating the information contained in signals, such as:

- **Model signals:** Synthesize the information about a physical phenomenon.
- **Analyze:** Extract features in both time and frequency domains
- **Classify:** Identify signal attributes using parameters, such as skewness and kurtosis in the time domain
- **Compress:** Reduce the need to store data by decreasing the necessary size to accurately represent the signal.
- **Filter:** Clean up the signal to remove unwanted components in the time and frequency domains.
- **Transmit:** Optimize bandwidth use to send data.

It is mandatory to understand the nature of the signal to be analyzed, because it will tell the engineer which procedure is the most appropriate to be used. For example:

- If the signal is **transient**, like a chirp whose frequency increases with time, short time Fourier transform is more suitable because it determines the sinusoidal frequency content of local sections

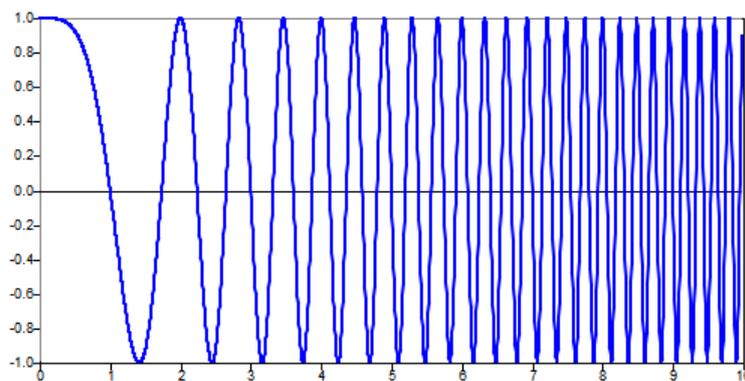


Figure 2. Example of chirp signal

- If the signal is **stationary**, such as sine waves, Fast Fourier Transform is pertinent because it assumes periodicity of the signal.

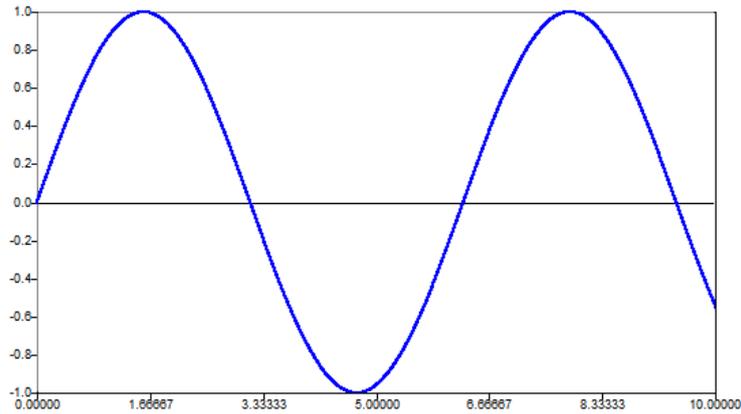


Figure 3. Example of sine wave

- If the signal is **random**, like a road profile or a turbulent air flow, then the computation of Power Spectral Density is the best approach to extract information.

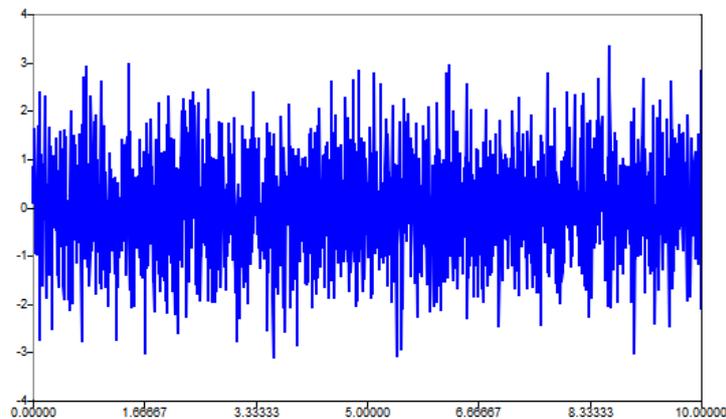


Figure 4. Example of random signal

Most signal processing procedures are based on 3 fundamental steps:

- 1) Data handling
- 2) Time and frequency domain analysis
- 3) Visualization

Altair Compose provides functions to efficiently address signal processing workflows in all these steps.

1.3. Data Handling

With respect to **data import**, as shown in Figure 5, Altair Compose can directly read numerous CAE and test data formats, both time and frequency based. From CAE, it supports files from multiple Finite Element Method (FEM) solvers, Computational Fluid Dynamics (CFD) and Multibody Systems (MBS). From test, it supports various formats from different acquisition systems.

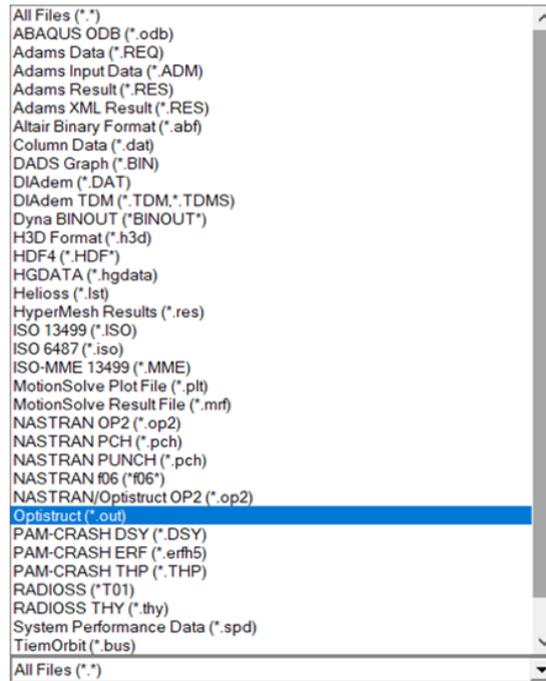


Figure 5. Supported file formats in Altair Compose

Regarding **data preparation**, there is a broad coverage of different methods to prepare data, like:

- Detrending.
- Spike removal and detection.
- Resampling.
- Peak detection.

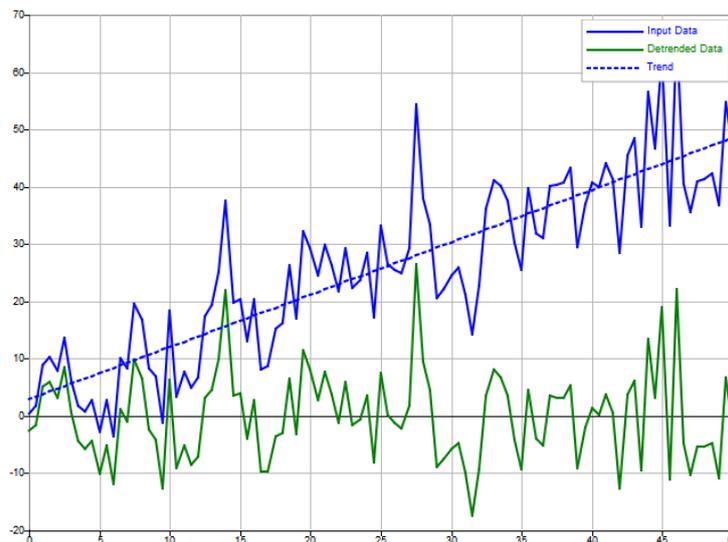


Figure 6. Example of original and detrended signals

It is also possible to perform feature extraction using the statistics library to find parameters such as maximum, minimum, mean, standard deviation and root mean square (RMS) level. Design and analysis of digital and analog filters may also be carried out.

1.4. Time and Frequency Domains

The second basis of signal processing is about time and frequency domains. Time domain analysis refers to the study of signals whose information varies with respect to time, whereas frequency domain analysis refers to the representation of signals as a function of frequency. It is possible to carry out signal analysis in both representations because Altair Compose supports transformation of signals from time to the frequency domain and vice-versa.

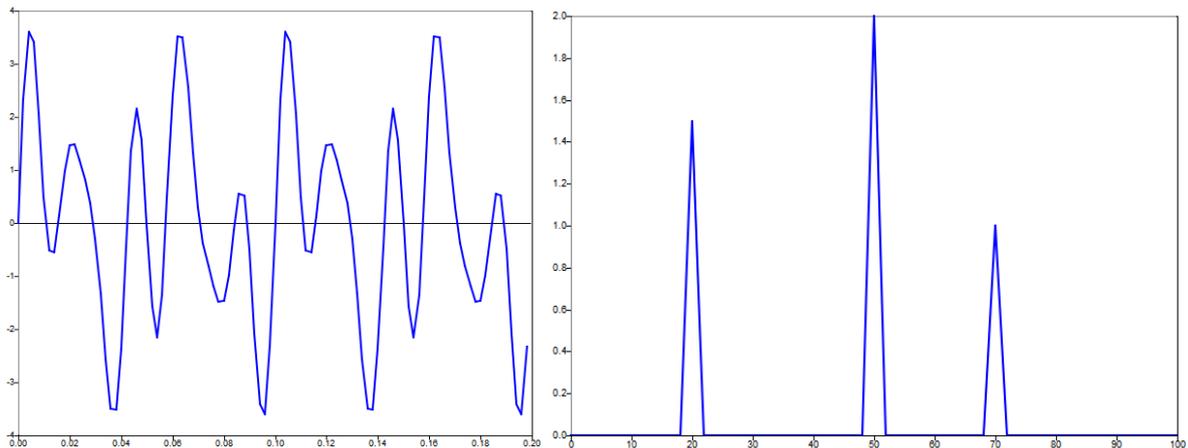


Figure 7. Time-domain signal and respective frequency-domain representation

1.5. Visualization

The third step of any signal processing workflow. Any analysis becomes more comprehensive when useful 2D and 3D plots are created to provide richer insight from calculations and data. There is a wide variety of plot types and customization parameters, and multiple graphs may be plotted together in subplots mixing different visualization types, such as 3D plot, polar plot, discrete sequence and surface.

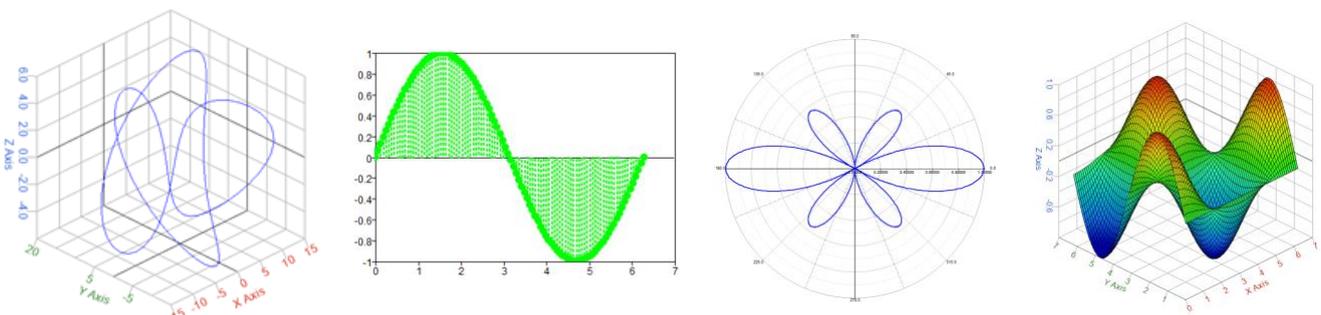


Figure 8. Some examples of the plot types created

The visual representation helps to address the mathematically complex concepts of this field, improving the understanding and the exploration of events related to the signals.

1.6. Applications

Some examples of signal processing are given from its wide range of applications from different industries, such as:

- Control systems
- Data mining
- Electronics
- Financial markets
- Genomics
- Durability
- Noise, vibration and harshness (NVH)
- Seismology

And some examples will be briefly mentioned.

1.6.1. Control Systems

There is an intimate relationship between control systems and signal processing because they rely deeply on Laplace transform in continuous-time signals and Z transform in discrete-time signals when operations with filters are needed; see Chapter 8.

Regarding the Laplace transform:

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

Its formal definition means that it transforms a function of a real variable t , which is often time, to a function of a complex variable s , which is complex frequency. Defining $z = e^{st}$, the Z transform is proportional to the Laplace transform applied for a discrete-time signal:

$$F_d(e^{sT}) = \sum_{n=0}^{\infty} f_d(nT)e^{-snT} = F_d(z) = \sum_{n=0}^{\infty} f_d(nT)z^{-n}$$

Where n represents the n -th sample and T is the period.

Although control systems are usually more interested in time-domain behavior, control engineers rely deeply on the usage of visualization tools such as Bode and Nyquist plots, which refer to frequency domain information. Digital signal processing is commonly more interested in frequency-domain, but both are part of the same mathematical formulation; see Chapter 6.

The bottom line is that signal processing has a set of tools that can be used for control systems, such as filters.

1.6.2. Durability

Durability tests are strongly data-driven to evaluate damage and life of components based on:

- 1) **Formulate the problem:** Durability specifications directly affect how the data should be acquired and therefore how the signal will be processed afterwards. It is the same situation with the test specifications.
- 2) **Set the instrumentation and data acquisition:** Instrumentation determines parameters like sampling frequency and strain gauge parameters to reduce the noise, disturbances and the acquisition process.
- 3) **Prepare the signal:** Drifts and spikes should be removed in time-domain, whereas filters in the frequency-domain may also be applied.
- 4) **Process the signal:** Signal processing will navigate from the time-domain to the frequency-domain, compute Power Spectral Density (PSD) and generate a time history based on a PSD, for instance.
- 5) **Interpret the results:** To calculate the damage and the maximum displacement of a component.

1.6.3. Noise, Vibration and Harshness (NVH)

Owing to strong competition in the automotive industry, the increasing demand for NVH aims to improve the driver's experience by reducing vehicle noise.

Spectral analysis is also relevant to the nature and characteristics of acoustic sources because the NVH measurement procedure does not only depend on the sound pressure level but also on the spectral composition of the sound. An assessment of how critical a noise is takes into account the distributions of critical bands within human hearing.

Furthermore, it is useful to decompose experimental data to separate environmental background noise, sounds and other sources of acoustic response.

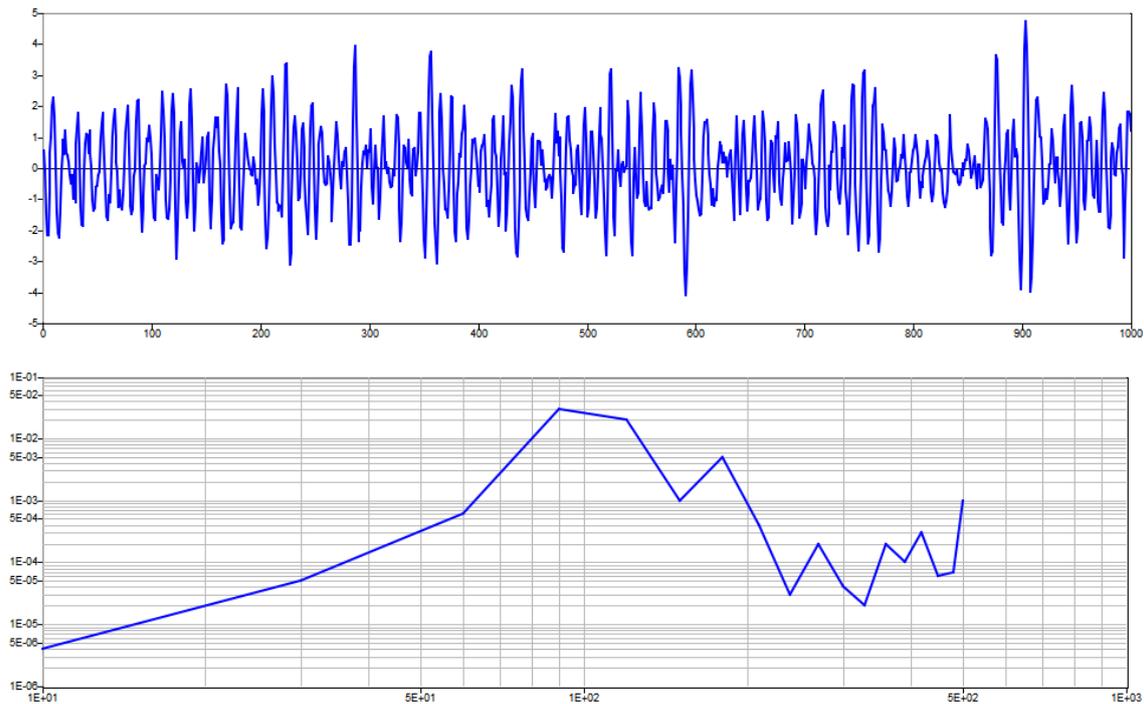


Figure 9. Conversion of time-domain signal (first plot) to PSD (second plot) to be used in an NVH analysis

1.6.4. Electronics

In electronics, there is a wide range of powerful applications using digital signal processing, such as:

- 1) **Wireless communication:** Where signal processing routines read the analog signal, convert it to digital, encode and approximate the data packet before transmitting it.
- 2) **Data compression:** Reduce the size of data packets in order to optimize the storage capacity and to the use of network bandwidth.
- 3) **Autonomous driving:** Signal processing-based systems detect vehicles as they approach and accurately locate them using GPS to compute the distance between the receiver and multiple geostationary satellites.
- 4) **Audio processing:** Used in subfields like speech recognition, noise cancellation, music identification and transmission.

2. Fundamentals

2.1. Signal Classification

The signal is a function that carries information about a phenomenon. A signal can be classified according to many criteria. Here, only those useful for this e-book are presented.

2.1.1. Stationary and Non-stationary Signals

Stationary and non-stationary are used to characterize the processes which generate the related signals. Here, for the sake of simplicity, they are referred directly to the signals.

A signal is **stationary** if a shift in time does not change its statistical properties. An example would be white noise, which is a random signal having equal intensity at different frequencies – any signal value is equally probable to occur at any two time instants because the values are standard normal; see example in 1.2.

On the other hand, when a shift in time changes the statistical properties of the signal, it is said to be **non-stationary**. A chirp signal is non-stationary because the event-to-event probabilities change over time; see Figure 2.

2.1.2. Continuous-time and Discrete-time Signals

A **continuous-time** signal is specified for any real value of the independent variable (i.e. time). For example:

$$x(t) = \sin(2\pi t)$$

Whereas a **discrete-time** signal is defined only for discrete values of the independent variable. Usually it is generated by sampling. Hence it is defined at equally spaced intervals along the time axis:

$$x(k) = \sin(2\pi k t_s) \quad , \quad k = 1, 2, 3 \dots$$

Where t_s is the sampling time.

See also: Chapter 5 for more details about discrete and continuous time approaches.

2.1.3. Analog and Digital Signals

An **analog** signal can assume any value included in a continuous range. It corresponds to a continuous y-axis.

Conversely, a **digital** signal assumes only a finite set of values. These values are usually defined as multiple integers of a certain number. It corresponds to a discrete y-axis.

A summary of the classification explained so far is shown in Table 1:

Table 1. Summary of signal classification

SIGNAL AMPLITUDE \ TIME	any real value	finite set
any real value	ANALOG CONTINUOUS-TIME SIGNAL	DIGITAL CONTINUOUS-TIME SIGNAL
finite set	ANALOG DISCRETE-TIME SIGNAL	DIGITAL DISCRETE-TIME SIGNAL

Usually when referring to analog or digital signals, it is implicitly referred respectively to analog continuous-time signal and digital discrete-time signal.

2.1.4. Finite-length and Infinite-length Signals

A **finite-length** signal has non-zero values over a finite set of values of the independent variable, such as time:

$$x(t) = \sin(2\pi t), \quad \forall t: t_1 \leq t \leq t_2$$

Whereas, an **infinite-length** signal has non-zero values over an infinite set of values of the independent variable, such as time:

$$x(t) = \sin(2\pi t), \quad \forall t$$

2.1.5. Periodic and Non-Periodic Signals

A signal $x(t)$ is said to be **periodic** if there is a positive constant T_0 such that

$$x(t) = x(t + T_0)$$

where T_0 = period of the signal.

Hence, a periodic signal remains unchanged when time-shifted by multiple integers of its period. It follows that periodic signals can be generated by time-shifting finite-length signals by their time length. Classic examples of periodic signals are sine and cosine waves:

$$x(t) = \sin(2\pi t) \quad x(t) = \cos(2\pi t)$$

Unlike periodic signals, **non-periodic signals** do not have a definite period when it repeats itself and their frequency content changes over time. A chirp signal is an example of non-periodic signal because it has a variable frequency; see Figure 2.

2.2. Elementary Operations

When talking about signal processing, the basic operations which can be performed on signals are discussed.

2.2.1. Amplitude Scaling

It means that the signal is multiplied by a constant:

```
t = 0:0.01:2;
s = sin(2*pi*(t));
ScaleFact = 2;
plot(t,s,t,s*ScaleFact);
xlabel('time'); grid on;
legend({'original','scaled'});
```

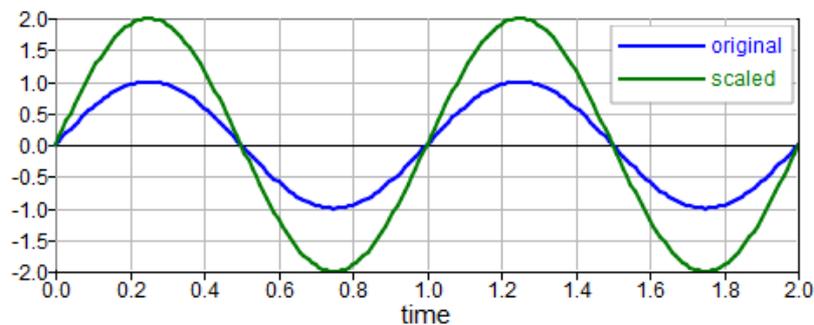


Figure 10. Example of scaling

2.2.2. Bias

It means that a constant value is added/subtracted to the signal:

```
t = 0:0.01:2;
s = sin(2*pi*(t));
bias = 1;
plot(t,s,t,s+bias);
xlabel('time'); grid on;
legend({'original','with_bias'});
```

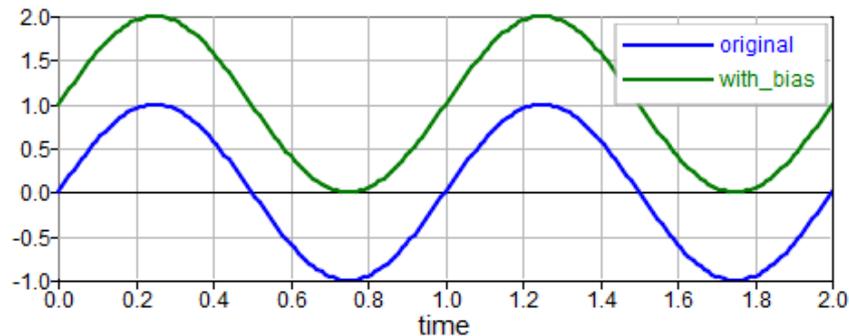


Figure 11. Example of bias

2.2.3. Addition, Subtraction, Multiplication, Division

Two or more signals can be summed together element-wise. Where discrete signals are considered, the signals must have the same number of samples.

```
t = 0:0.01:2;
s1 = sin(2*pi*(t));
s2 = cos(2*pi*(t));
s = s1+s2
plot(t,s1,t,s2,t,s);
xlabel('time'); grid on;
legend({'sig1','sig2','sum'});
```

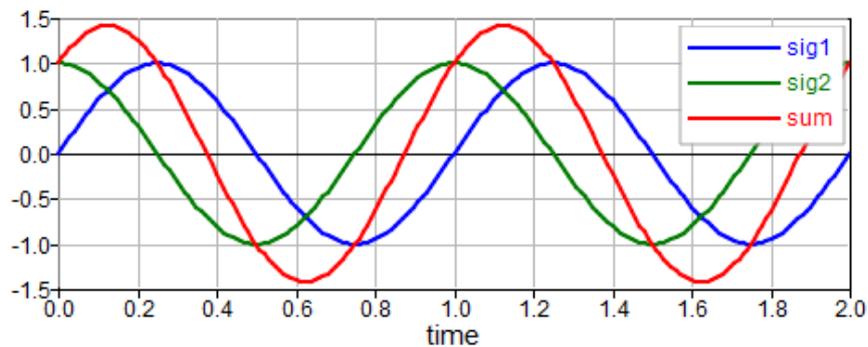


Figure 12. Example of signals that have been summed

Subtraction, multiplication, and division can be performed analogously, switching from addition operator (+) to the proper ones (−, .*, ./).

2.2.4. Time Shifting

The signal is translated forwards or backwards in time:

$$x(t) \xrightarrow{\text{shifting}} x(t + \alpha)$$

If α is positive, then the signal is anticipated but if α is negative, the signal is delayed.

```
t = 0:0.01:2;
s = sin(2*pi*(t));
Alpha = 0.4;
s1 = sin(2*pi*(t-Alpha));
s2 = sin(2*pi*(t+Alpha));
plot(t,s,t,s1,t,s2);
xlabel('time'); grid on;
legend({'original','delayed','anticipated'});
```

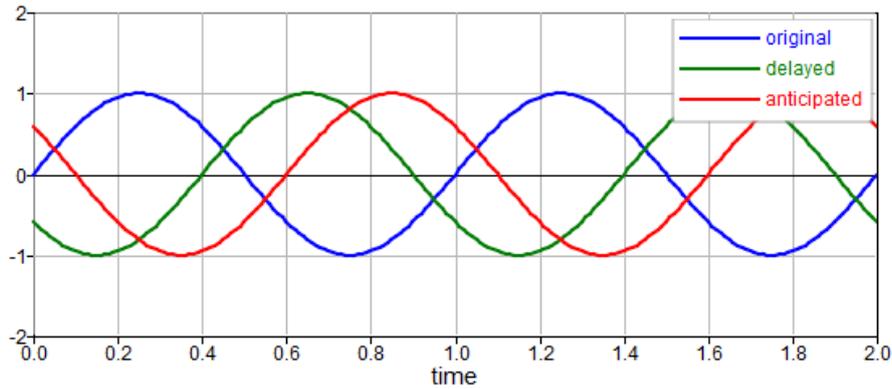


Figure 13. Example of time shifting

2.3. Complex Exponentials

In signal processing theory, the underlying math makes an extensive usage of trigonometric functions, such as sine and cosine. In many other books, Euler's formula can be used to move from these trigonometric functions to the complex exponential:

$$e^{i\varphi} = \cos \varphi + i \sin \varphi$$

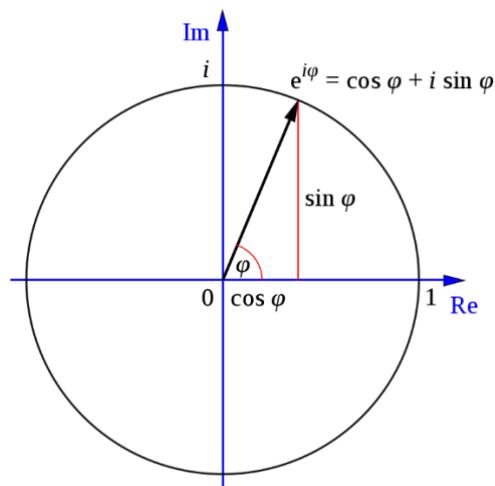


Figure 14. Trigonometric circle

The choice to move to the complex exponential form is only driven by the fact that it is easier to manipulate compared with its sinusoidal components. Hence many of the formulas presented in this e-book make use of this representation.

2.4. Sampling Time & Sampling Frequency

Given a discrete signal in the time domain and assuming that samples are equally spaced in time, the **sampling time** (t_s) is defined as the distance between two successive samples. It is also known as sampling period (T).

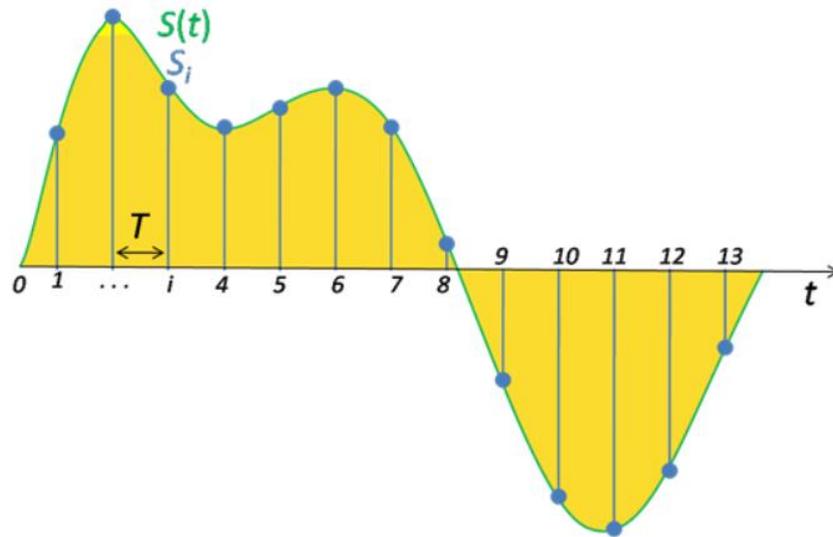


Figure 15. Sampling time representation

Conversely, the **sampling frequency** (f_s), also known as “sample rate”, is the number of samples per second. It is the reciprocal of the sampling time, hence:

$$f_s = \frac{1}{t_s}$$

The higher the sample rate, the more accurate the representation of the signal in the time domain becomes.

Consider a 6 Hz cosine wave sampled with a sampling frequency of 32 Hz.

```
f = 6;
fs = 32;
ts = 1/fs;
t = 0:ts:1;
x = 2*cos(2*pi*f*t);
```

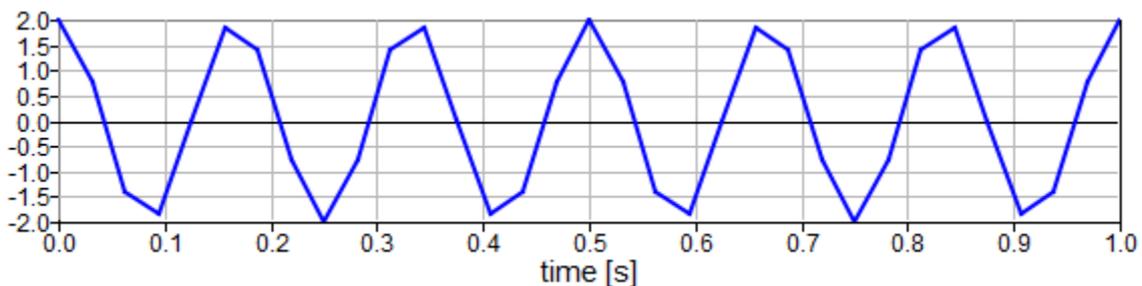


Figure 16. 6 Hz cosine wave sampled at 32 Hz

And the same cosine wave with a sampling frequency of 128 Hz:

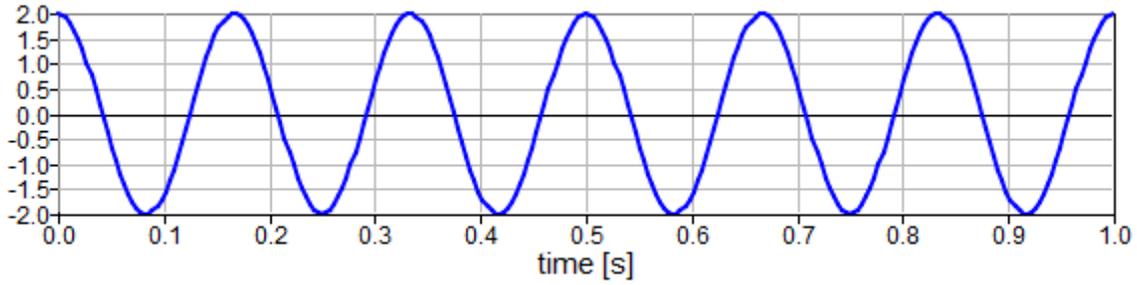


Figure 17. 6-Hz cosine wave sampled at 128 Hz

2.5. Aliasing & Nyquist-Shannon’s Theorem

Aliasing is a phenomenon that occurs when the signal is under-sampled. It causes different signals to become indistinguishable, or aliases of one another, when sampled.

For example, an 8 Hz cosine wave sampled with a 10 Hz sample rate will look like a 2 Hz cosine wave:

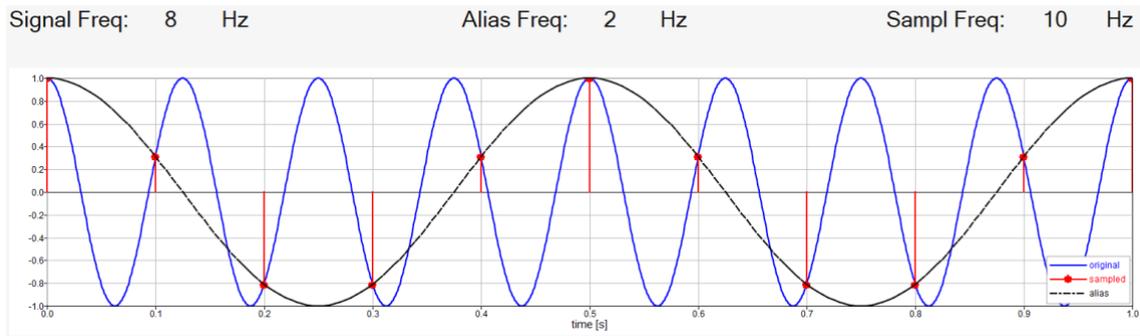


Figure 18. 8-Hz cosine wave sampled with 10 Hz

The same is true also for a 12 Hz cosine wave sampled at 10 Hz:

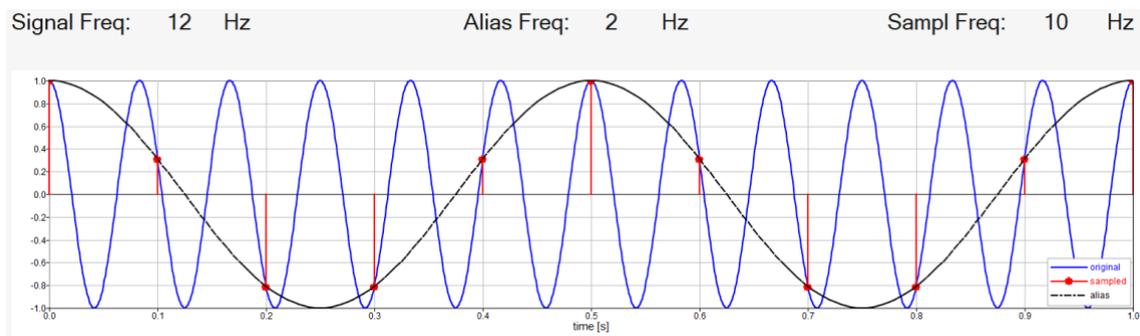


Figure 19. 12 Hz cosine wave sampled with 10 Hz

To avoid aliasing, the sample frequency must be selected properly, or an anti-aliasing filter may be used.

The sampling theorem states that properly representing a waveform requires a sample rate of at least twice the signal frequency.

Hence:

- The Nyquist frequency is half of the sampling rate f_s
- The Nyquist rate is the minimum sampling rate that satisfies the Nyquist sampling theorem for a given signal.

When aliasing occurs, signal frequencies above the Nyquist frequency ($\frac{f_s}{2}$) are folded back and appear as lower frequency signals.

2.6. Frequency Resolution

Frequency resolution, the sampling time in the frequency domain, is the distance between successive frequency bins in the frequency domain; see: Chapter 6. It can also be thought of as the smallest frequency that can be visualized. It can be computed as:

$$f_r = \frac{1}{\text{Acquisition Time}}$$

As shown in the formula, the longer the acquisition time, the better the frequency resolution becomes. Hence, in signal processing a natural trade-off arises between quality of the signal in the time domain and quality in the frequency domain.

\uparrow *Quality in time domain* \rightarrow \downarrow *Quality in frequency domain*

Given a fixed memory size to store data, increasing the sample rate will improve the time domain representation but worsen the frequency domain, since the acquisition time gets shorter and vice versa. Extending the acquisition time improves the frequency representation of the signal but worsens in the time domain because the sample rate gets smaller.

2.7. Energy and Power

The terms signal energy and signal power are used to characterize a signal, although they are not actual measures of energy and power.

They can be computed in both time and frequency domains. Here, it is shown how the energy and power are computed in time domain. For the computation of the same quantities in the frequency domain, see: Chapter 7).

The energy of a continuous-time signal is defined as the area under the squared magnitude of the signal:

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

The energy of a discrete-time signal is defined replacing the integral with a summation:

$$E = \sum_{n=-\infty}^{\infty} |x(n)|^2$$

Where the discrete signal has a finite length, its energy becomes:

$$E = \sum_{n=0}^{N-1} |x(n)|^2$$

where N is the number of the signal samples.

At this point, knowing the energy, the computation of the average power is straightforward:

$$\text{Avg. Power} = W = \frac{\text{Energy}}{n^{\circ} \text{ of samples}} = \frac{E}{N} = \frac{\sum_{n=0}^{N-1} |x(n)|^2}{N}$$

3. File Input & Output

There are some main categories of files that are commonly used in signal processing workflows and each has a more appropriate function for importing data. Altair Compose can read inputs in multiple formats:

- **Excel™-compatible files**, such as .csv or .xlsx
- **MAT files**, which have a container-like format and store variables of various data types.
- **Test and CAE files**, generated by numerical solvers or acquisition systems
- **Other formats**, like binary, delimited files, text files and images

As the primary focus of this book is on signal processing, test and CAE files will be explored in more detail.

3.1. Test and CAE Files

Direct reading of numerous CAE and test data formats, time- and frequency-based, are readable by Altair Compose:

- **CAE**: Multiple solvers of FEM, CFD and MBS
- **Test**: Multiple acquisition systems in formats such as UNV, ISO, ASCII and CSV

The main takeaway is that no translators or converters are required to import such files.

There are 4 main functions for this purpose:

- `readvectorbuilder`
- `readvector`
- `readmultvectors`
- `readcae`

To write a new CAE file:

- `createabffile` creates an Altair Binary Format file, and
- `createh3dfile` is the main standard used by Altair within its FEM solvers.

As a good practice, it is beneficial to close the file using `releasedata` to release from memory only what has been read and `releasefileinfo` to release from memory the file as a whole.

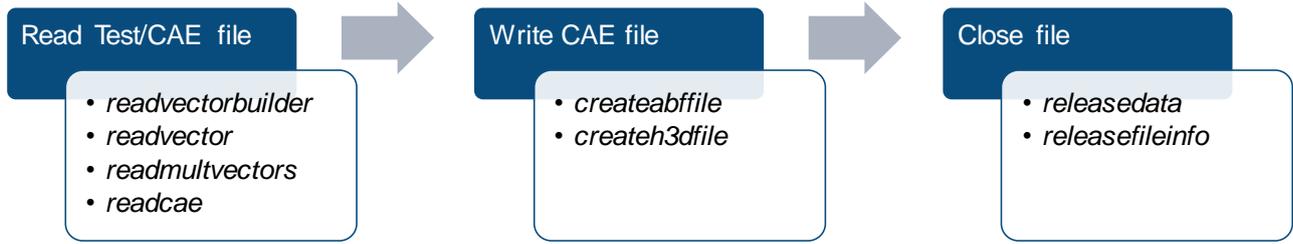


Figure 20. Main functions used to read, write and close CAE and test files

3.2. *Readvectorbuilder* Function

To import CAE and test files, *readvectorbuilder* is a utility that reads test and CAE result files and helps to create the right commands to import data. Once again, it is important to remember that a huge variety of CAE and test formats is supported; see Figure 5.

Since CAE files are usually binary and not easily accessible with other programming languages or platforms, *readvectorbuilder* (command window) launches the utility and waits for the user to load the input file and check what is in there.

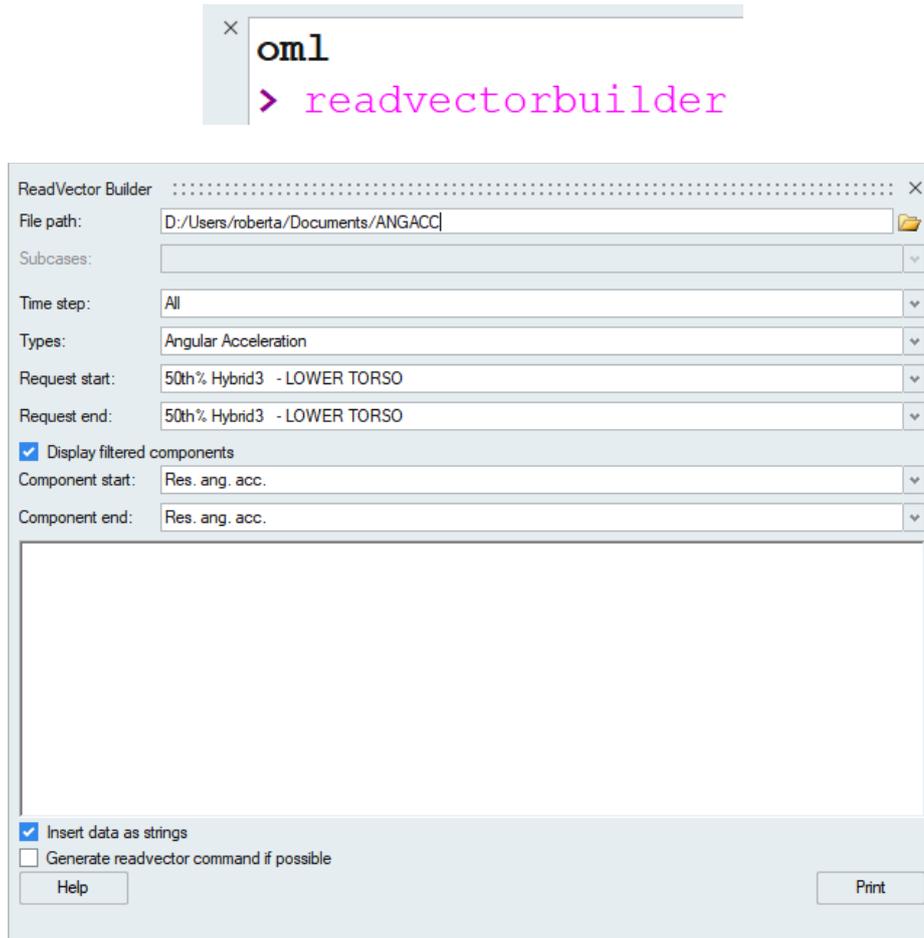


Figure 21. Steps to launch *readvectorbuilder* and load a CAE/Test file

All entities within the file are recognized: subcases, data types, requests, components and time steps.

In the example given in Figure 22, the input file has information about a crash simulation. In this case, there are no subcases associated with the file, so its field is grayed out. As it is a dynamic analysis, there are different time steps and the *All* option queries all of them. Then, *Angular Acceleration* is the only result type. As shown in Figure 22, the legs of the simulation dummy are of interest from this crash analysis. Therefore, the request start is the *Upper Leg Left* and the request end is the *Upper Leg Right*. Finally, the component that corresponds to the resultant angular acceleration (*Res. ang. acc*) must be queried:

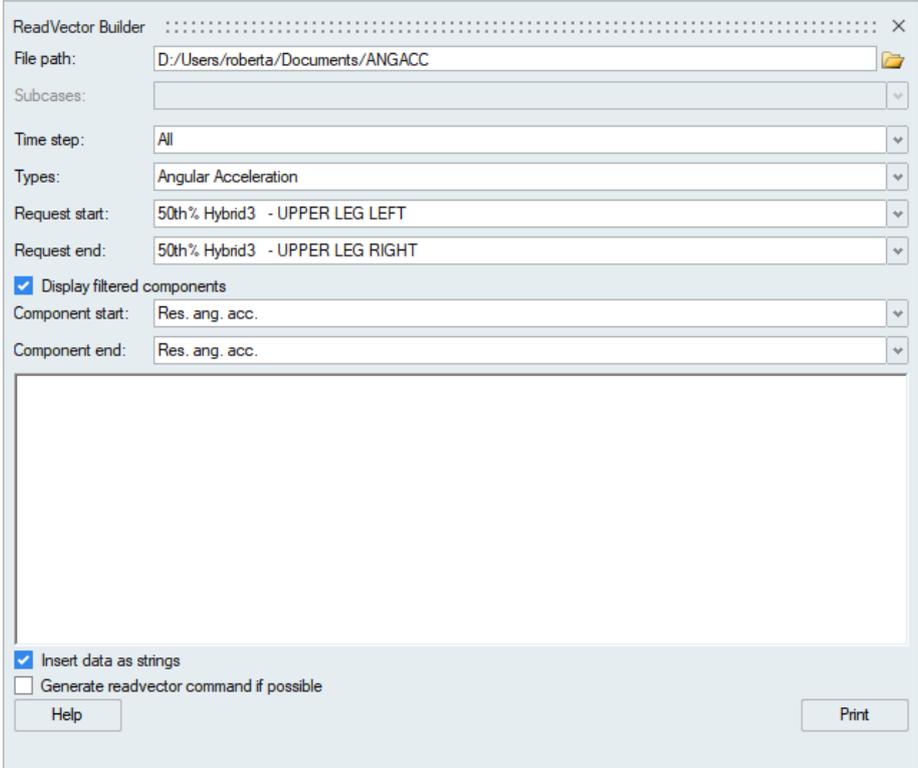


Figure 23. Example of how to query specific information from a crash simulation

Once all information to be extracted has been selected, *Print* button generates the syntax that will do the operation using the CAE functions that will be explained next. It may now be copied and pasted in your script and adapted if necessary.

```
readmultivectors('D:/Users/roberta/Documents/ANGACC','Angular
Acceleration','50th% Hybrid3 - UPPER LEG LEFT','50th% Hybrid3 -
UPPER LEG RIGHT','Res. ang. acc.','Res. ang. acc.','All')
```

3.3. Readvector Function

`Readvector` is a function that reads test and CAE files and returns a data vector, that is, the output has a single type of data.

Using the same file from the `readvectorbuilder` example, `readvector` could be used to read the angular acceleration, *Lower Torso* request and the component, which is the resultant of angular acceleration.

```
y = readvector('D:/Users/roberta/Documents/ANGACC', 'Angular
Acceleration', '50th% Hybrid3 - LOWER TORSO', 'Res. ang. acc.');
```

In this case, the variable `y` in fact is a vector, as this function allows only a single type of data to be queried at a time. It corresponds to the information of all the 151 time steps of angular acceleration of the lower torso region coming from a crash simulation.

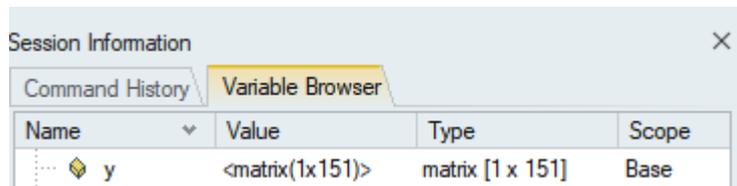


Figure 24. Output vector generated by `readvector` function

3.4. Readmultivectors Function

The `Readmultivectors` function extracts multiple types of information in a single command, such as several data types or several components at once, consequently the output is a matrix. Instead of declaring multiple `readvector` commands, `readmultivectors` performs the same task with less operations.

Using the file from the previous function, it allows not only to query crash results of angular acceleration in the *Lower Torso*, but also in the *Head* with a single command, for instance.

```
y_multivectors =
readmultivectors('D:/Users/roberta/Documents/ANGACC', 'Angular
Acceleration', '50th% Hybrid3 - LOWER TORSO', '50th% Hybrid3 -
HEAD', 'Res. ang. acc.', 'Res. ang. acc.', 'All');
```

If `readvector` is used for the same purpose, it is not possible to get both results simultaneously and that is why one command for each request would be needed.

```
y = readvector('D:/Users/roberta/Documents/ANGACC', 'Angular
Acceleration', '50th% Hybrid3 - LOWER TORSO', 'Res. ang. acc.');
```

```
y2 = readvector('D:/Users/roberta/Documents/ANGACC', 'Angular
Acceleration', '50th% Hybrid3 - HEAD', 'Res. ang. acc.');
```

3.5. *Readcae* function

The functions `readvector` and `readmultivectors` support arguments that could represent a single entity to be queried or a range of entities, giving its start and end. If one wants to query a list of entities and not necessarily a range, such as querying an entity list comprised by 1, 10, 15, 16, 17 and 30 – it could mean a list of channels in a Test file or a list of elements in a CAE File.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

Figure 25. Entity list and specific indices must be queried

Using `readvector` or `readmultivectors`, it would only be possible to give the start ID, 1 in this case, and the end ID, which is 30, which means that all the information in-between would be read for no reason.

```
readmultivectors(filename, subcase, type, 1, 30, startComponent, endCom
ponent, timeStep)
```

By using a third CAE reader function called *readcae*

```
readcae(filename, subcase, type, [1, 10, 15:17, 30], componentList, time
Step, outputType)
```

With the same crash simulation file, it is now possible to query results exclusively from *Lower Torso* and *Upper Torso*. If *readvectorbuilder* is launched to help build the syntax to query these results, the request list shows that the angular accelerations in the *Head* are between these 2 requests of interest, meaning that `readmultivectors` command would query all requests between the start, which is *Lower Torso*, and the end, the *Upper Torso*; see Figure 26.

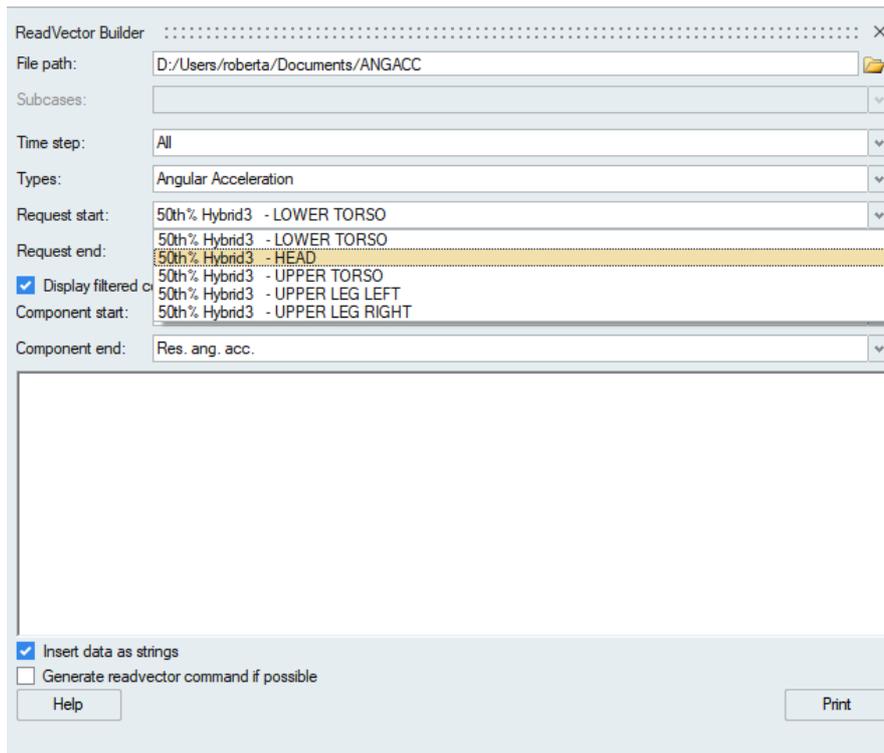


Figure 27. Readvectorbuilder utility demonstrating the list of requests

In this situation, `readcae` is more appropriate because it can explicitly receive as arguments only these 2 requests using a list format, and not necessarily a range format.

```
y = readcae('D:/Users/roberta/Documents/ANGACC', 'Angular
Acceleration', {'50th% Hybrid3 - LOWER TORSO', '50th% Hybrid3
- UPPER TORSO'}, 'Res. ang. acc.', [], 1);
```

3.6. Other CAE/Test Functions

There are also support functions to get specific information from the input file if the user is interested in subcases, data types, requests and components. Not only numeric results can be retrieved, but also a list of subcases, names of the results, number of requests and so on.

<code>getsubcaselist</code> , <code>getsubcasename</code> and <code>getsubcaseindex</code>	→ subcases
<code>getnumtypes</code> , <code>gettypelist</code> , <code>gettypename</code> and <code>gettypeindex</code>	→ data types
<code>getnumreqs</code> , <code>getreqlist</code> , <code>getreqname</code> and <code>getreqindex</code>	→ requests
<code>getnumcomps</code> , <code>getcomplist</code> , <code>getcompname</code> and <code>getcompindex</code>	→ components

These support functions enable the algorithms to query CAE and Test files to be as generic as possible. In cases when entities change their names from file to file, functions like `getsubcaselist` and `getreqname` read the list of subcases and the name of a specific

request, respectively, without the need to hard-code these values in the scripts. As soon as these parameters are queried, they can be passed as variables to the CAE readers.

Exercises

- 1) Use 2 different functions to read an ASCII file (*disptime.ascii*) with displacement over time.
- 2) Plot x, y and z components of angular acceleration of the Lower Torso region from a crash simulation file (*ANGACC*) using `readvector` function to query these components.
- 3) Do the same as in exercise 2) but using `readmultivectors`.
- 4) Compute the resultant angular acceleration and compare it with the resultant one stored in the same file.
- 5) Read Frequency Response Analysis results from an Altair OptiStruct™ file (*FRF_output_OptiStruct.h3d*) and compute the RMS of components magnitude of XX and XY.

4. Data Processing & Preparation

Data acquisition under real conditions can encounter problems that create disturbances, such as trends or spurious spikes, which can arise from unwanted contact with the sensor, unexpected humidity, electromagnetic interferences, amongst others. This noise should be removed to reflect the true behavior of the system.

4.1. Detrending

Detrending a signal is the same as removing a trend from a time series resulting in a certain pattern which is not inherent in the data set to be removed.

The function `detrend` removes the mean or best fit line from a signal, and its inputs are the signal and the method to remove the trend, which could be:

- `constant` to remove the mean:

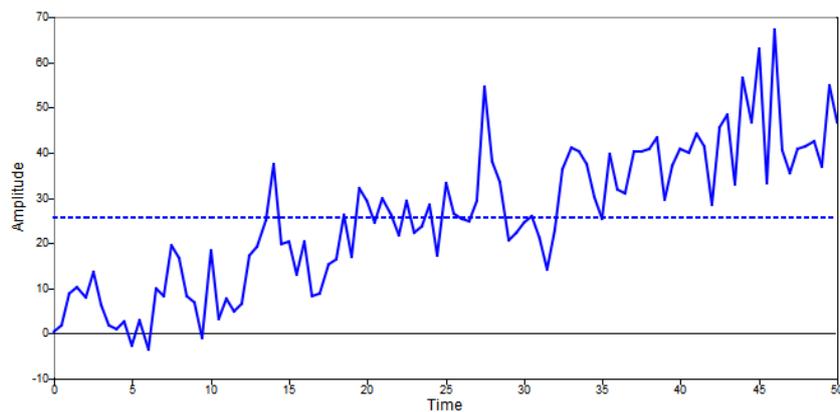


Figure 28. `constant` method to detrend a signal

- `linear` to remove the best fit trend. A linear trend typically demonstrates a systematic increase or decrease in the data over time:

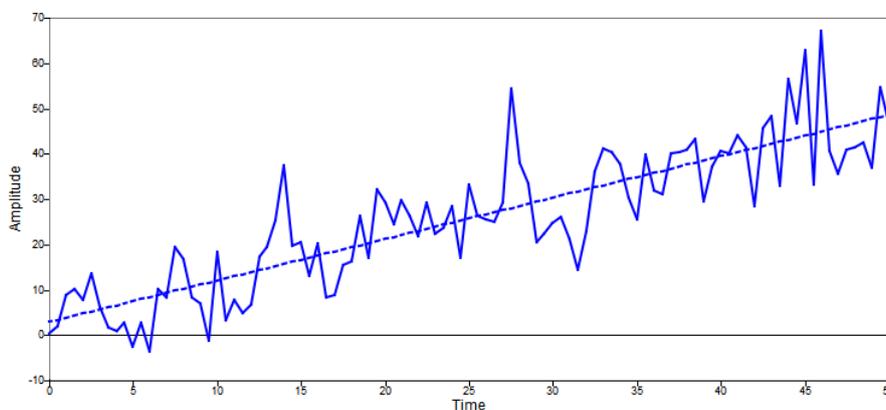


Figure 29. `linear` method to detrend a signal

These two methods essentially use a polynomial approach based on the least squares method:

$$y = a_k x^k + \dots + a_1 x + a_0$$

To find a polynomial equation with the best fit and subtract it from the signal, where `constant` method is equivalent to a polynomial degree of 0, correspondent to the mean, and `linear` is a polynomial degree of 1.

4.2. Spike Detection & Removal

Spikes are extreme changes in measured data that probably do not represent the actual behavior of the system. Spurious spikes could create bias and hence they must be identified and removed. A good practice is always to check whether the identified spikes happen in other channels too or if there is an oscillatory behavior after the peak that might help to qualify it as a true peak.

Median absolute deviation (MAD) is widely used in statistics to assess the variability of a certain data set. Using Math functions such as `interp1` for 1D interpolation and `median` to compute the median value, it is possible to construct a robust MAD method to mathematically detect spikes. For a data set with one variable, the MAD is defined as the median of the absolute deviations from the data's median, that is:

$$MAD = median(|X_i - median(X)|)$$

Where X_i is the i -th sample. This method may be implemented using the Hampel identifier, whose principle is that for each sample, the function computes the median of a window composed of the sample and its surrounding samples.

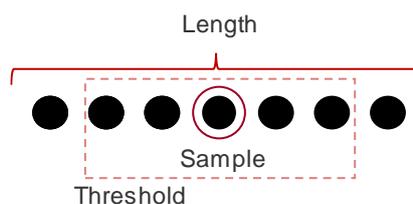


Figure 30. Hampel identifier principle

The number of samples are the size of the window; the threshold is the number of standard deviations away from the median, and if a value is a certain number of MAD away from the median, that value is classified as an outlier; the overlap is the number of samples that are simultaneously in different windows.

The MAD algorithm is described in these steps.

- 1) Determine the number of samples, the threshold and the overlap:

$$N = 7;$$

```
th = 5;
ovlp = 0;
```

- 2) Check whether the number of samples gives an integer division by two or not. This check guarantees that the sample being analyzed is always in the middle of the window by adding an extra sample in its size in case it is not a multiple of 2:

```
if mod(N,2) == 0
    N=N+1;
end
```

- 3) Determine the number of unique samples in each window, that is, the number of samples that will not be taken into account in the next window according to the overlap. It is useful to place an exception handling when overlap is 100%, which would lead to a number of unique samples equal to zero:

```
K = floor((1 - ovlp/100)*N);
if ovlp == 100
    K=1;
end
```

- 4) Compute the number of windows according to the size of the length y:

```
M = floor((length(y)-N)/K)+1;
```

- 5) Initialize variables that will receive the median, MAD, average and corrected signal and initial window, considering the size computed in the previous step for the sake of speeding up the code execution:

```
Me = zeros(M,1);
MAD = zeros(M,1);
ind = zeros(M,1);
y_corrected = y;
n = 1;
```

- 6) From the first window to the last one, compute the median of the window, subtract the median from each value of the window:

```
while n <= M
    i1 = 1+(n-1)*K; %First sample of window
    i2 = i1+N-1; %Last sample of window

    if n == M %Last window
        i2 = length(y);
    end

    v = y(i1:i2); %Samples of window
    ind(n) = (i1+i2)/2; %Average of extreme values
    Me(n) = median(v); %Median of window
end
```

```
MAD(n) = median(abs(v-Me(n))); %MAD
n = n+1; %Next window
```

end

- 7) Linearly interpolate the computed median and MAD values:

```
ini = y(1); %First sample of signal
fini = y(end); %Final sample of signal
Me = [ini;Me;fini];
ind1 = [1;ind;length(y)];

%Interpolation of median and MAD values
v = (1:length(y))';
interp_median = interp1(ind1,Me,v,'linear','extrap');
interp_mad = interp1(ind,MAD,v,'linear','extrap');

idx = find(y>interp_median+th*interp_mad | y<interp_median-
th*interp_mad); %Non-outliers according to the threshold

if length(idx)>= 1
    y_corrected(idx) = interp_median(idx);
end
```

- 8) Use `y_corrected` as the corrected version of `y` signal without spurious spikes:

Using the above configuration in an example of signal with strain over time, several spurious spikes are identified and removed:

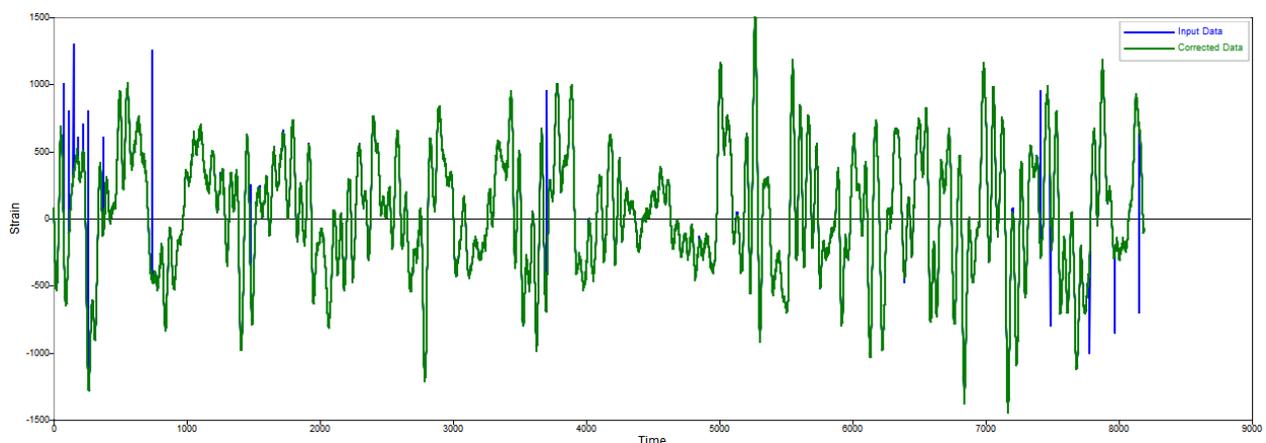


Figure 31. Input data and corrected data without spurious spikes

4.3. Resampling

Typically, the sampling rate is defined as 1 over the amount of time between successive samples, that is, the sampling frequency.

$$f_s = \frac{1}{T_s}$$

Resampling methods allow the user to change the sampling rate of a signal. For example, sample-rate conversion prevents changes of speed in audio when converting from professionally made to consumer equipment. Furthermore, an offset can be specified during the resampling, that is, the sample at which the output data starts.

4.3.1. Upsampling

The `upsample` function increases the sampling rate of the signal:

```
upsample(x, n, p)
```

Where x is the signal to be upsampled, n is the upsample factor and p is the offset. The purpose is to add samples to a signal, while maintaining its original length with respect to time. It will improve both the resolution in the frequency domain and the anti-aliasing filter performance and reduce noise.

As an example, consider the following data set:

```
x = [1 2 3 4 5 6]
```

When its sampling rate is increased by a factor of 2, it increases the sample rate of x by inserting $n - 1$ zeros between samples:

```
y = upsample(x, 2);
y = 1 0 2 0 3 0 4 0 5 0 6 0
```

If a phase offset of 1 is given, it shifts the samples by 1:

```
y = upsample(x, 2, 1);
y = 0 1 0 2 0 3 0 4 0 5 0 6
```

4.3.2. Downsampling

The `downsample` function decreases the sampling rate of the signal:

```
downsample(x, n, p)
```

Where x is the signal to be downsampled, n is the *upsample* factor and p is the offset. The aim is to remove samples from the signal, while maintaining its length with respect to time. It leads to a reduction of resolution in the frequency domain and a compression of the size of data.

In order to decrease the sampling rate of the previous data set by a factor of 2:

$$y = \text{downsample}(x, 2);$$

This gives an output where the first sample and then every n^{th} sample after the first are kept:

$$y = 1 \ 3 \ 5$$

Then an offset of 1 would shift the retained samples by 1:

$$y = \text{downsample}(x, 2, 1);$$

$$y = 2 \ 4 \ 6$$

4.3.3. Resampling to fixed rate

The `resample` function to change the data to a new fixed rate would simply be a combination of `downsample` and `upsample` applied to the same signal.

$$\text{resample} = \text{downsample}(\text{upsample}(x, p), q)$$

Resampling is usually carried out with the aim to interface two systems having different sampling rates. If the ratio of two system's rates turns out to be an integer, decimation (sampling rate is being decreased) or interpolation (sampling rate is being increased) can be used to change the sampling rate. Otherwise, interpolation and decimation must be used together to change the rate. A practical example is the conversion of professionally made audios (usually 48 kHz) to consumer equipment (usually 44.1 kHz).

$$\frac{44100}{48000} = \frac{147}{160}$$

In this case, a rational fraction approximation is used, in other terms to interpolate by a factor of 147 then decimate by a factor of 160:

$$[p, q] = \text{rat}(44100/48100);$$

The outputs are respectively the numerator and the denominator of the rational approximation and may be used to resample the signal to a fixed rate.

4.4. Feature Extraction

Feature extraction refers to the extraction of characteristics about the input signal. Finding peaks within data is one of the most common feature extraction techniques to determine local peaks according to criteria like peak height, distance between peaks and peak width.

The function `findpeaks` locates peaks in a signal, while controlling the identification of peaks with parameters like `MinPeakHeight`, `MinPeakDistance` and `MinPeakWidth`, to respectively determine the minimum peak height, minimum distance between peaks and minimum peak width.

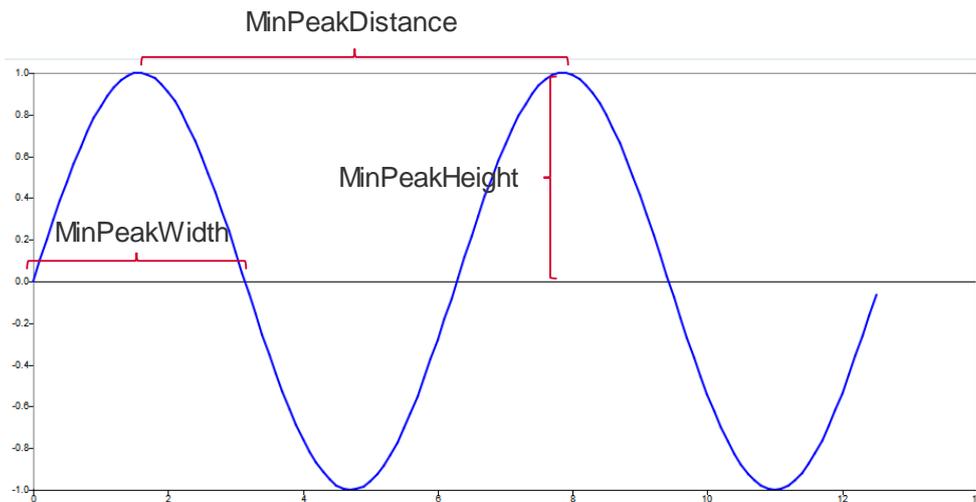


Figure 32. Customizable criteria of `findpeaks` function to find peaks of a signal

As an example, consider an electrocardiogram (ECG) signal:

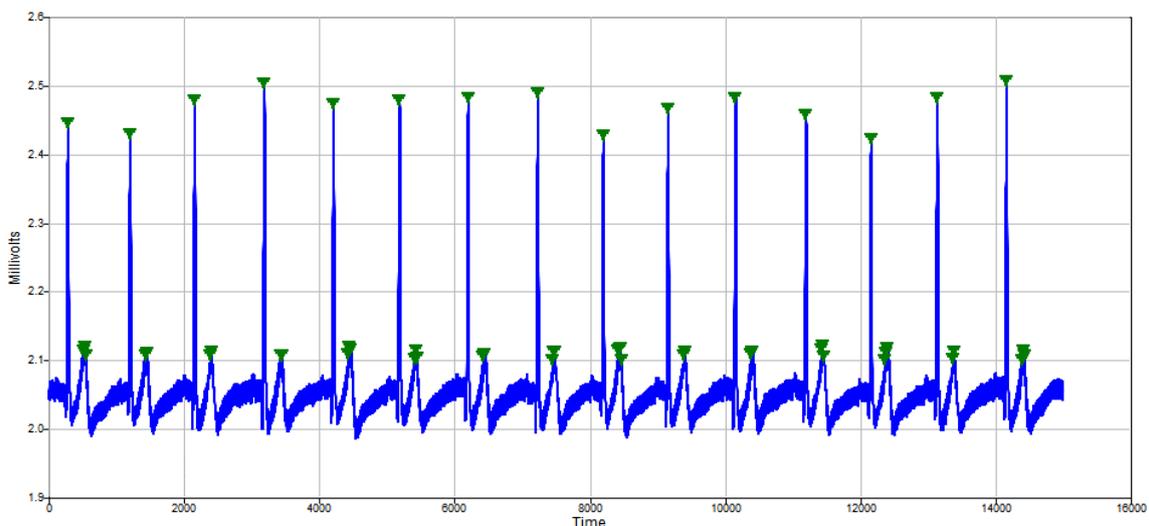


Figure 33. Example of ECG signal

The definition of the minimum peak height value in `findpeaks` function:

```
[pks,idx,extra] = findpeaks(ecg,'MinPeakHeight',2.1);
```

This allows the identification of exclusively two types of waves:

- **Short waves:** indicate the electrical impulse in the upper chambers of the heart.
- **Tall waves:** represent the lower chambers contracting to pump out blood.

It helps to identify cardiovascular anomalies based on the length, width and the time when these waves happen.

The outputs of the function are the peak values (`pks` variable), indices where the peaks were found (`idx` variable) and extra information related to peak widths based on fitted parabolas (`extra` variable).

Exercises

1) Remove the trend from a time series (*Month_vs_sales.csv*) that describes the monthly number of sales of a certain product over 36 months (3 years) using `detrend` and polynomial functions to achieve the same goal. In the end, plot the original and the detrended signals.

2) Identify outliers using the MAD method explained step by step in another time series of sales (*Month_vs_sales2.csv*) and plot the original signal and the one without the spikes. Why is an outlier detected around $t = 10$ even though there are equivalent values in the signal from $t = 20$ onward?

3) What happens if you change the size of the window of the previous exercise from 7 to 1? And from 7 to 15? Why?

4) What happens if you change the threshold of the previous exercise from 5 to 3? Why?

5) Consider the signal whose sampling period is 0.1 second:

$$x(t) = [5 \ 6 \ 4 \ 2 \ 1 \ 7 \ 5 \ -3 \ -4 \ -6 \ -9 \ -12 \ -1 \ 2]$$

What is the output signal if you downsample it by a factor of 3? What are the new sampling period and sampling frequency? What is the new Nyquist frequency after downsampling?

6) If the signal from the previous example has frequency components larger than the new Nyquist frequency:

$$f > \frac{f_s}{2n}$$

Where n is the downsampling factor and f_s is the sampling frequency. In this case, aliasing noise will be introduced into the downsampled data. What should be done to overcome this issue?

7) Consider the same signal in Exercise 5. Upsample it by a factor of 2. What are the new sampling period and sampling frequency? What is the new Nyquist frequency after upsampling?

8) Consider the following signal:

$$x(t) = [15 \ 7 \ 25 \ 8 \ 16 \ 10 \ 6 \ 9 \ 0 \ 22 \ 10 \ 15]$$

Find the local maxima and plot both the signal and the peaks that have been identified. After that, what happens if a criterion is considered to find peaks whose height is greater than 23? What happens if a criterion is considered to find peaks whose distance between each other is 10?

9) Consider the following signal:

$$x(t) = [15 \ 7 \ 25 \ 8 \ 16 \ 10 \ 6 \ 9 \ 0 \ -22 \ -10 \ -5 \ -2 \ -6 \ -8]$$

With positive and negative values. Find the local maxima and plot both the signal and the peaks that have been identified. After that, how can we identify both positive and negative peaks?

5. Time-Domain Analysis

5.1. Time Response

Time response corresponds to the output of a system in response to an input excitation, which is a function of time. It can be:

- **transient**, whereby the response is a reaction of the system to a change from its equilibrium, or
- **steady state**, where the response does not change with time.

An example of transient signal is a chirp, whose frequency increases or decreases with time, and for a steady state signal, sine waves or a composition of different sine waves, whose oscillation over time has a periodic behavior.

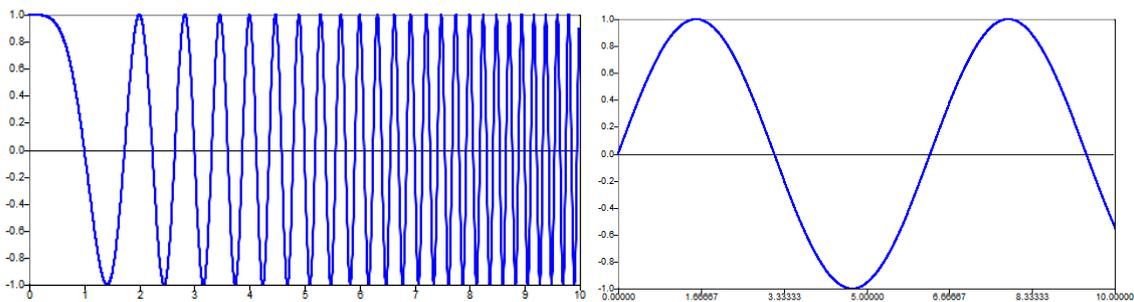


Figure 34. Transient and steady-state signals

Most people are relatively comfortable with the time domain representation as it is more intuitive and more practical compared with the frequency domain. The signal values are known for all real numbers, for the case of continuous time, or at various separate instances in the case of discrete time.

5.2. Continuous Time

The continuous time approach mathematically considers variables as having a specific value for only an infinitesimally short period of time. It means that there are infinite number of points between any two points in time, hence time is viewed as a continuous variable.

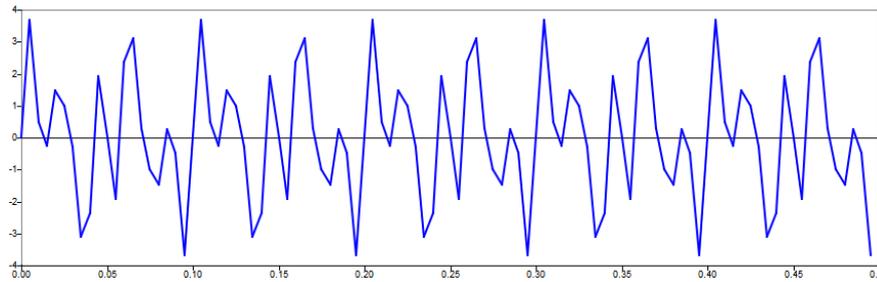


Figure 35. Continuous time-domain signal

One of the most important functions in signal processing is the sine function, which is continuous over time:

$$f(t) = \sin(\omega t), t \in R$$

5.3. Discrete Time

The discrete time approach considers variables as occurring at distinct and finite instants of time; hence time is viewed as a discrete variable. In this case, the signal is a time series consisting of a sequence of quantities.

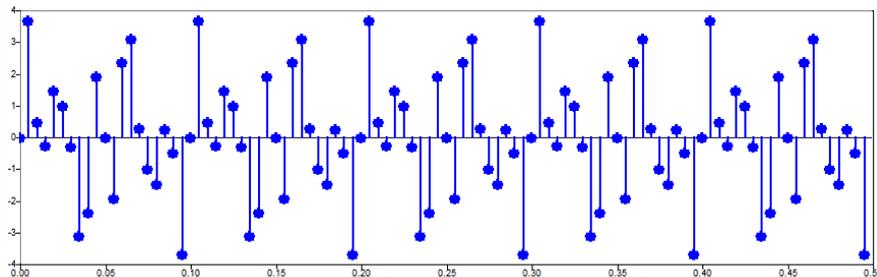


Figure 36. Discrete time-domain signal

There are two bases to describe discrete time signals:

- **Discrete time:** It allows phenomena to be described even when their functions are not known, which is true for most real-life cases, as their representation happens only in a set of instants of time.
- **Discrete amplitude:** It allows the data set to be countable, as each sample takes a specific value and the signal has a finite number of samples.

5.4. Advantages of Discrete over Continuous Time

A common question is whether some power of representation is lost when moving from continuous to discrete time approach. As explained in Chapter 1, the sampling theorem and Nyquist frequency lead to an equivalence of continuous and discrete time representations.

The relationship between continuous and discrete time representations is given by the sampling theorem, which states that the reconstruction equation is:

$$x(t) = \sum_{n=-\infty}^{\infty} x(n) \text{sinc}\left(\frac{t}{T_s} - n\right)$$

Where $x(n)$ is the set of discrete samples, T_s is the sampling period and the sinc function is defined by:

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

Of which the output can be computed with the `sinc` function:

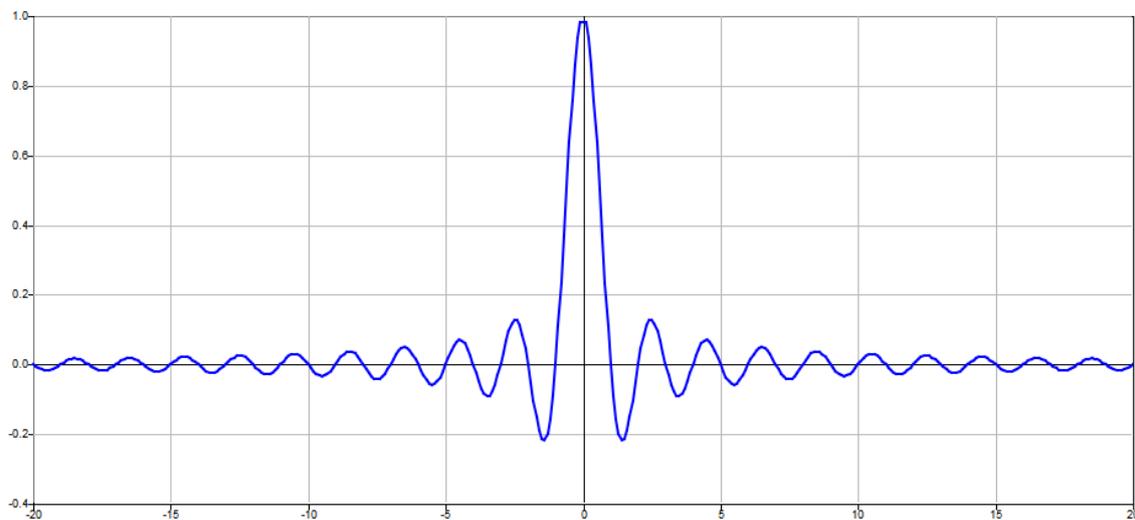


Figure 37. Sinc function

The `sinc` function is known as the sampling function and is symmetric, oscillates from $-\infty$ to ∞ and passes through zero at all positive and negative integers, but at time $t = 0$ it reaches its maximum of 1. This property helps to prevent intersymbol interference, which is a primary cause of quality loss in digital transmission systems.

Copies of `sinc` function are taken and placed at each sample location scaled by the amplitude of the sample – summing all these copies leads to the original continuous time signal.

The discretization of signals brought important enhancements related to the conversion of systems from analog to digital, due to its higher efficiency of compression, transmission and storage. The effect of distortion, noise and interference has less influence on digital signals, the hardware implementation is cheaper, more flexible and highly customizable using programming languages. Digital signals can also be saved and retrieved more conveniently using less space.

5.5. Statistical Features

Statistics and probability are used to characterize signals and the processes that generate them. As one of the main aims of signal processing is to reduce noise and other undesirable components in acquired data, statistical features allows the signal to be measured and classified, which helps identify these spurious components.

5.5.1. Mean

Mean is the statistical term for the average value of a signal. It represents the central tendency of a data set. In some cases, it can be the operating point of a physical system that generates the time series. It is represented by the function `mean` given by the equation for the case of arithmetic mean (option 'a'):

$$\mu = \frac{1}{N} \sum_{n=1}^N x_n$$

Where N is the number of samples and x_n is the n -th sample. Or the geometric mean (option 'g'):

$$\mu = \frac{1}{N} \prod_{n=1}^N x_n$$

which is an average useful for signals that are interpreted according to their product and not by their sum.

5.5.2. Median

Median is the middle value in a set of values arranged in order of size, independent of any distance metric. It is represented by the function `median`.

Consider the following data set:

$$x = [1 \ 3 \ 6 \ 7 \ 10 \ 12 \ 18]$$

Its median can be calculated with the following command:

$$y = \text{median}(x)$$

The output is 7 which is the value in the middle of the set.

5.5.3. Standard Deviation

Standard deviation is a measure of the dispersion or variation of a signal, represented by the `std` function and given by the equation:

$$\sigma = \sqrt{\frac{\sum_{n=1}^N (x_n - \mu)^2}{N - 1}}$$

Where N is the number of samples, μ is the mean and x_n is the n -th sample.

There is also an option to take into account Bessel's correction and compute the standard deviation dividing it by N instead of $N - 1$. The subtraction of 1 is a method to correct the bias in the estimation of population variance.

`std(x)` or `std(x, 0)`: standard deviation with Bessel's correction as the default

`std(x, 1)`: standard deviation without Bessel's correction

As most acquired signals do not show a well-defined peak-to-peak value, but have a random nature instead, standard deviation is a more generalized method. It is an indication of how close the data is to the mean value of the signal.

5.5.4. Root Mean Square

Root mean square (RMS), also known as quadratic mean, is very useful because other measurements, such as mean, do not give much information because the negative values cancel the positive values. Hence the need to measure the magnitude of all samples without considering them positive or negative.

As the power is proportional to the squared signal, this average power corresponds to the mean value of this squared signal. It gives a representation of the average power of the signal, given by:

$$RMS = \sqrt{W} = \sqrt{\frac{\sum_{n=1}^N x_n^2}{N}}$$

Where N is the number of samples and x_n is the n -th sample.

5.5.5. Crest Factor

The crest factor is a ratio of signal's peak value to its RMS value. It is an indication of how extreme the peaks are in a time series.

$$CF = \frac{x_{peak}}{x_{rms}}$$

It could be simply computed using `max` and `rms` functions to get the maximum value of the signal and its RMS, respectively.

$$CF = \text{max}(x) / \text{rms}(x)$$

In the case of a sine wave, it is approximately 1.41, which is the most common crest factor for electrical appliances.

```
x = 0:0.1:2*pi;
y = sin(x);
CF = max(y) / rms(y);
```

When a load has a crest factor of more than this value, the source must supply the peak current desired by the load, otherwise the source voltage will become distorted by the excess peak current.

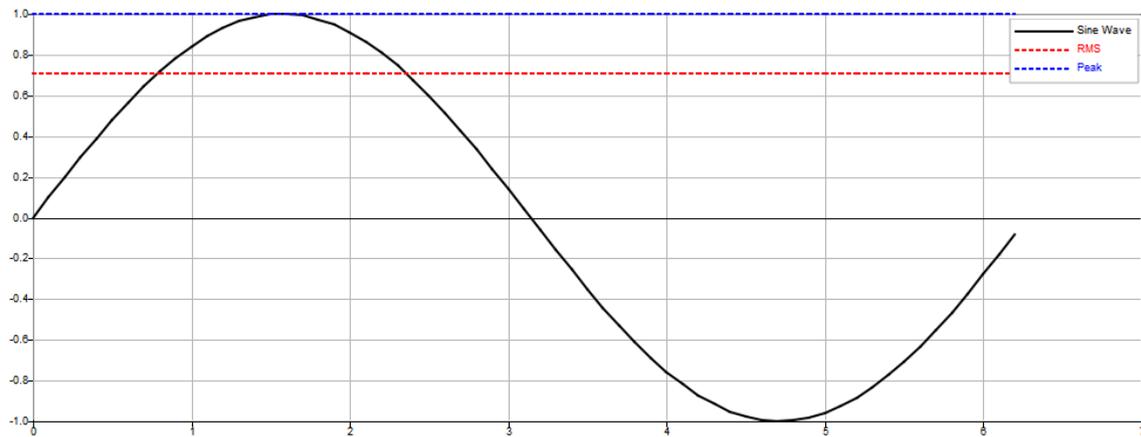


Figure 38. RMS and peak curves of a sine wave, parameters to compute the crest factor

The introduction of an outlier of value equal to 3 increases the crest factor to 3.79, as the peak becomes higher:

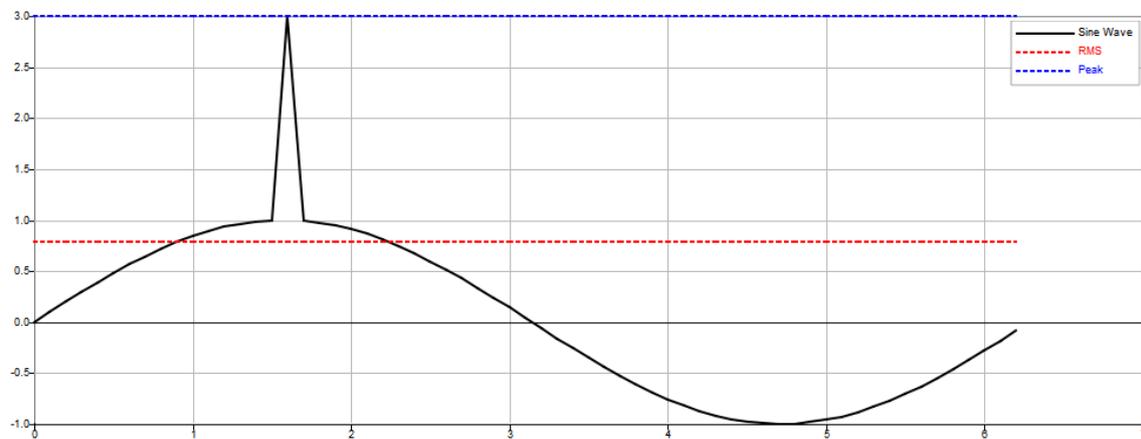


Figure 39. Introduction of an outlier to increase crest factor

Table 2. Sine wave crest factor before and after the introduction of an outlier

Sine Wave	Sine Wave with Outlier
Crest factor = 1.41	Crest factor = 3.79

This outlier was introduced by finding the index of the signal where the peak occurs and replacing its value:

```
[maxy, imax] = max(y);
```

```
y(imax) = 3;
```

5.5.6. Skewness

Skewness represents the degree of distortion from the symmetrical “bell curve” or the normal distribution of a time series. It can be computed using the `skewness` function, which measures the lack of symmetry in data distribution. For a symmetrical distribution, as shown in Figure 38, the skewness value is 0.

A histogram plot obtained using the `hist` function gives a graphical representation of the frequency distribution:

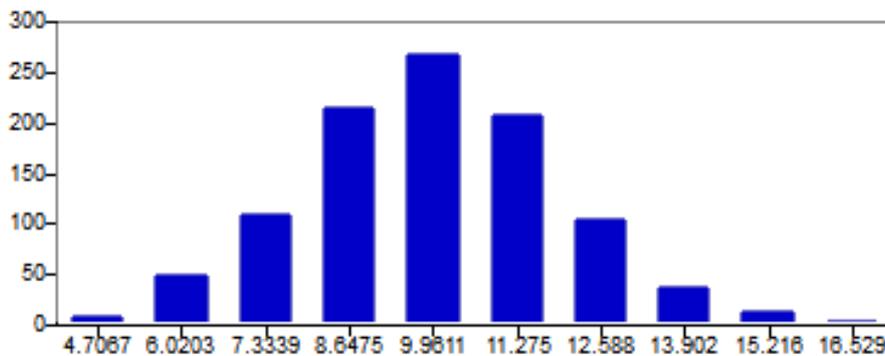


Figure 40. Skewness = 0

The data is divided into class intervals denoted in x -axis. The bins of the data are on the y -axis, where each rectangle represents the numbers of frequencies that lie within that particular class interval.

If the peak of the distribution is located to the left of the average value, it gives a positive skewness in the distribution. In practical terms it means that most samples of the signal are smaller than the average value. The mean of positively skewed data is greater than the median.

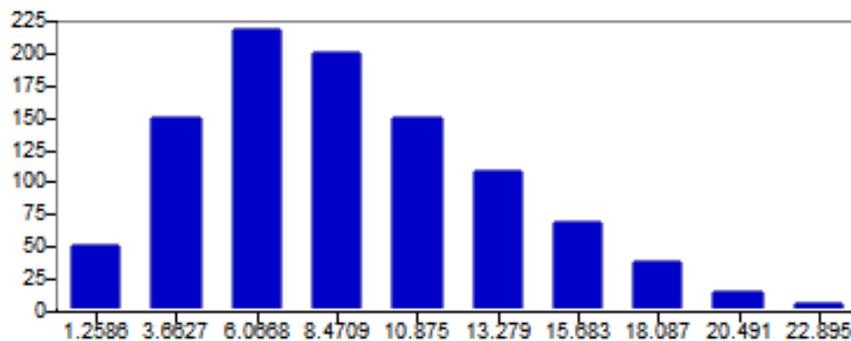


Figure 41. Skewness > 0

On the other hand, if the peak of the distributed data is located to the right of the average value, it has a negative skew meaning that most samples of the signal are greater than the average value. The mean of negatively skewed data is smaller than the median.

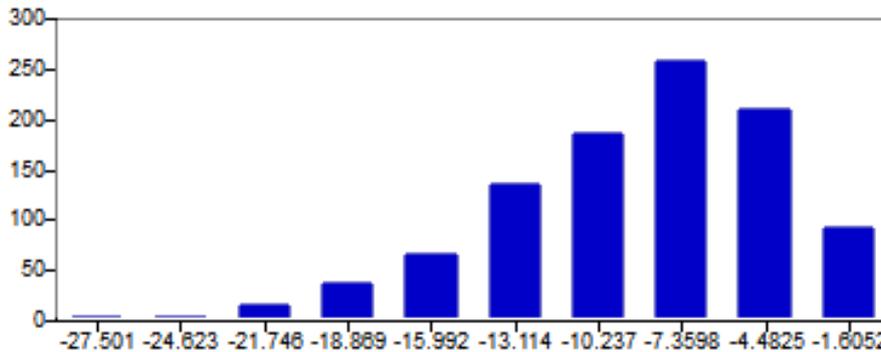


Figure 42. Skewness < 0

Using the previous example, the original sine wave had a skewness equal to 0. Then the introduction of an outlier led the skewness to be greater than 0, which means that most samples of the signal are smaller than the average value. It also confirms that the mean of the data is greater than the median.

Table 3. Sine wave skewness, mean and median before and after the introduction of an outlier

Sine Wave	Sine Wave with Outlier
Skewness = 0	Skewness = 0.716
Mean = 0	Mean = 0.0316
Median = 0	Median = 0

5.5.7. Kurtosis

Kurtosis is a measure of the tails of distribution in a time series. In other words, how flat or peaked it is. High kurtosis in a data set indicates that the signal has sharp peaks or outliers, whereas low kurtosis is the opposite, a signal with low peaks.

It may be calculated as a combination of `mean` and `std` functions.

$$\text{mean}((y - \text{mean}(y)) .^4) / \text{std}(y) .^4$$

Regarding the denominations, the excess kurtosis is defined as kurtosis minus 3. The 3 distinct descriptions are described; see also Figure 43.

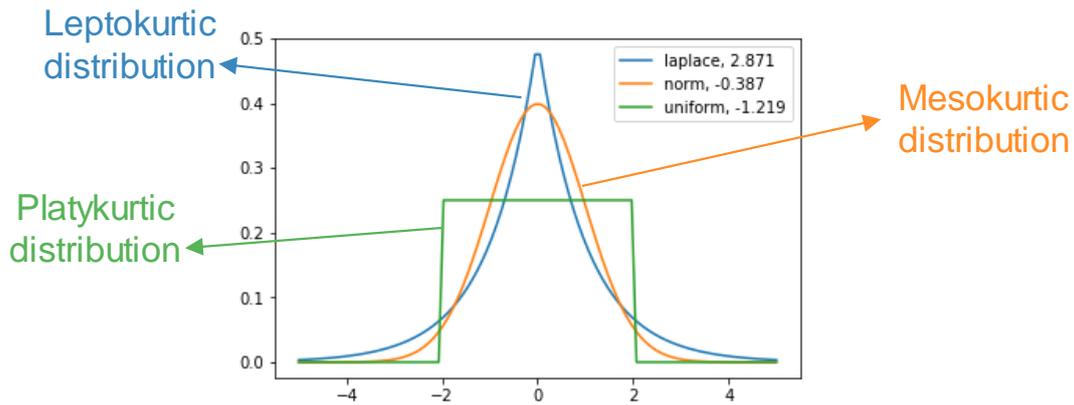


Figure 44. Kurtosis denominations

In the definition of excess kurtosis, its value for a normal distribution is zero and it characterizes a **mesokurtic** distribution. In Figure 41, the kurtosis is close to zero because it was calculated based on a random discrete dataset with normal distribution. The Laplace distribution has a higher kurtosis and therefore exhibits a sharper tail, whose denomination is **leptokurtic** distribution. A uniform distribution, which has negative excess kurtosis and the thinnest tail of these three probability distributions, is called a **platykurtic** distribution.

Using the same example of the sine wave and its copy with an outlier, kurtosis became greater than in the original signal, indicating that this new data set has sharper peaks than before.

Table 4. Sine wave kurtosis before and after the introduction of an outlier

Sine Wave	Sine Wave with Outlier
Kurtosis = 1.457	Kurtosis = 3.959

5.6. Autocorrelation, Cross-Correlation and Convolution

5.6.1. Autocorrelation

Autocorrelation is the correlation of a signal with a delayed copy of itself as a function of delay. Some applications which aim to find repeated events include the presence of periodicity obscured by noise, musical beats to determine tempo, and to identify a missing fundamental frequency in a signal implied by its harmonic frequencies. Its formal description is:

$$(f * f)(\tau) = \int_a^b f^*(t)f(t + \tau)dt$$

Where f is a function, namely signal, and the integral is the product of its conjugate and the same function shifted by τ .

As an example, consider a sine wave that oscillates from 0 to 4π in order to visualize the repetition of the period, shown in Figure 42, created with these simple commands:

```
x = 0:0.1:4*pi;
y = sin(x);
```

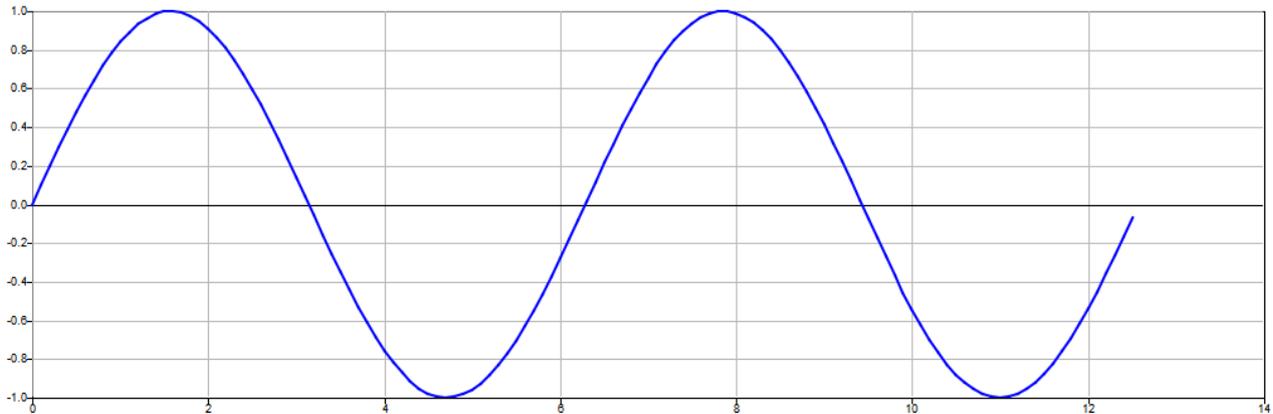


Figure 45. Sine wave from 0 to 4π

Use the `xcorr` function to estimate both autocorrelation and cross-correlation depending on the number of inputs given. Giving only 1 input means that the autocorrelation will be calculated.

```
Ryy = xcorr(y);
```

Then, it is time to create the lag vector, where lag is essentially delay, as we shift the time series.

```
lags = -x(end):0.1:x(end);
```

By plotting lag versus autocorrelation, it is noticeable that the origin of the function is in the middle of the data set.

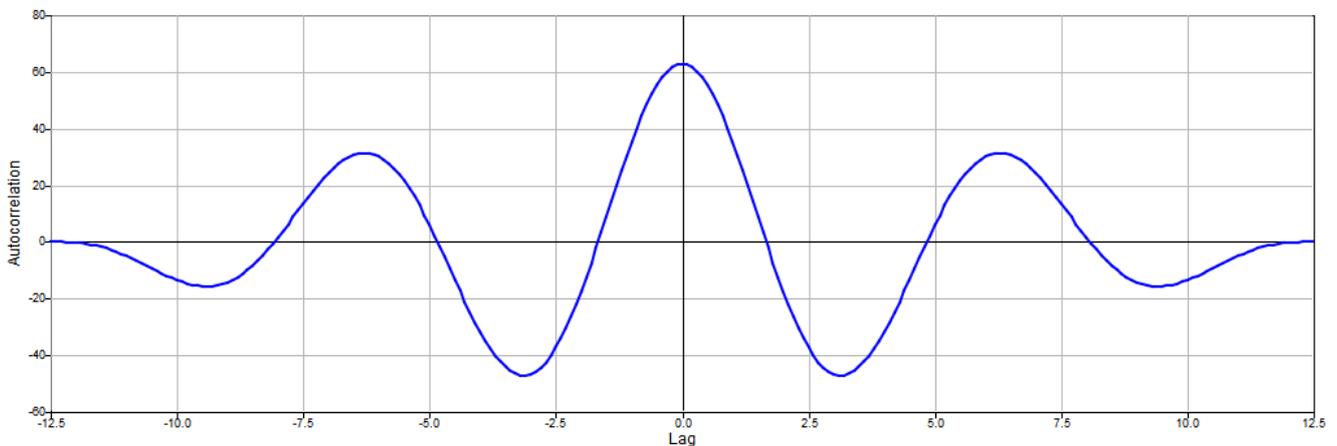


Figure 46. Lag versus autocorrelation

For any time series, there is perfect correlation at lag = 0 because of comparing the same values with each other. As the signal starts to shift, the reduction of correlation, and the next peak of the autocorrelation function occurs at the time period of the signal, i.e. for a sine wave, its period is 2π where the peaks occur.

5.6.2. Cross-Correlation

Cross-correlation is a measure of similarity of two time series as a function of the displacement of one relative to the other. For example, to identify similarity between multichannel signals and to measure differences between signals, such as delays. Its formal description is:

$$(f * g)(\tau) = \int_a^b f^*(t)g(t + \tau)dt$$

Where f and g are two functions, namely signals, and the integral has the product of the conjugate of one function and the other function shifted by τ .

Using the sine wave of the previous example to create a phase-lagged sine wave, Figure 44 shows that these signals have the same frequency but the second one has a different phase of $\pi/2$ radians.

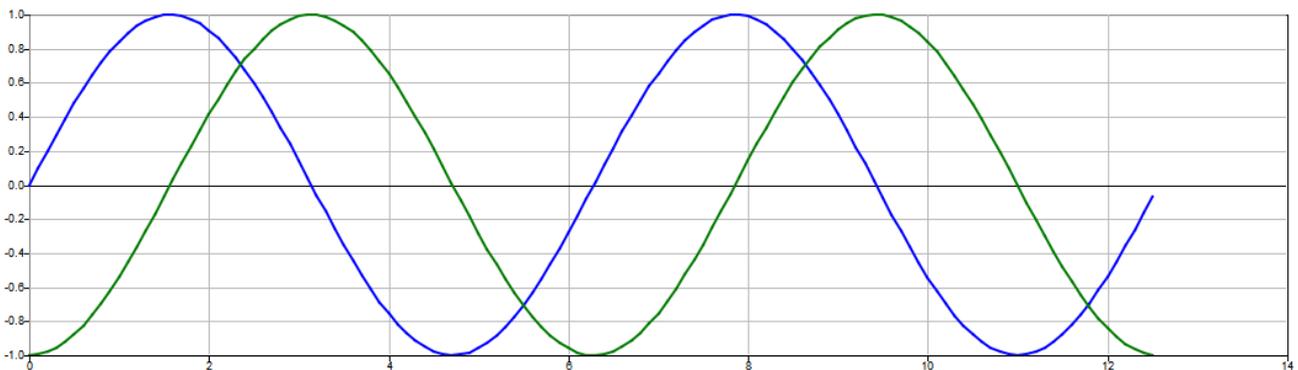


Figure 47. Sine wave from 0 to 4π and its phase-lagged copy

Now using the same `xcorr` function, giving 2 arguments means that the 2 signals will be compared in order to estimate the cross-correlation.

$$R_{xy} = \text{xcorr}(y, y2);$$

After creating the lag vector as explained previously, the same principle can be used to understand the cross-correlation between the 2 signals.

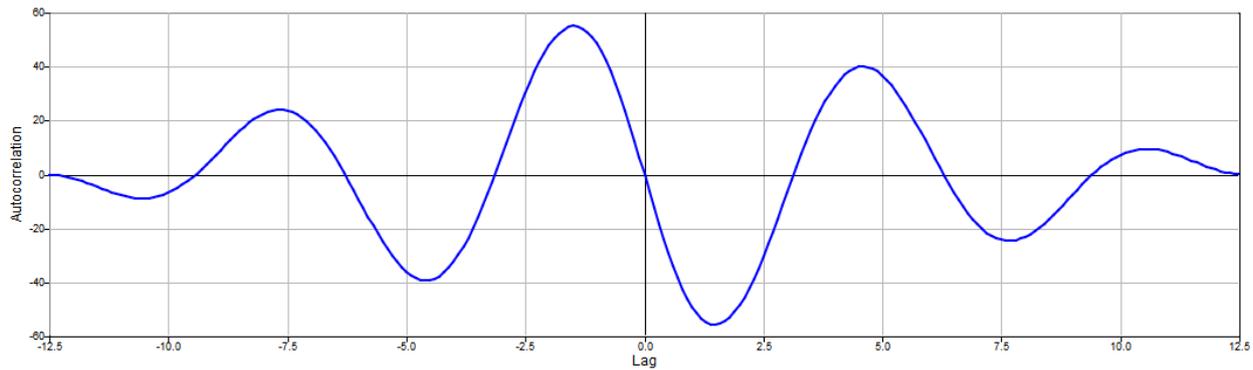


Figure 48. Lag versus cross-correlation

As a phase lag of $\pi/2$ has been added to the second sine wave, Figure 45 shows that their correlation is maximum at $-\pi/2$ and $\pi/2$ instead of 0 like in the previous example. The next peaks are multiples of π indicating other points where the signals correlate strongly.

5.6.3. Convolution

Convolution combines two time series in order to produce a third time series. It is an integral that expresses how the shape of one is modified by the other by showing the amount of overlap of one signal as it is shifted over another signal. Its formal description is:

$$(f * g)(\tau) = \int_a^b f(t)g(t - \tau)dt$$

Where f and g are two functions, namely signals, and the integral has the product of one function and the other after it has been reversed and shifted by τ .

The convolution vector between two other vectors of size n is computed with the `conv` function, having a size of $2n - 1$ and each element is computed as:

Element 1: $f(1) * g(1)$

Element 2: $f(1) * g(2) + f(2) * g(1)$

Element 3: $f(1) * g(3) + f(2) * g(2) + f(3) * g(1)$

...

Element n : $f(1) * g(n) + f(2) * g(n-1) + \dots + f(n) * g(1)$

...

Last element $2n - 1$: $f(n) * g(n)$

Exercises

1) Consider the following data set:

$$x(t) = [12 \ 15 \ 51 \ 58 \ 22]$$

Determine the skewness of the data and its meaning, as well as the value and the type of kurtosis.

2) Using the same data set in Exercise 1, compute its maximum value, RMS value and crest factor. What happens with these parameters if the 4th element (58) is changed to 100?

3) Consider the function of a half-rectified sine wave:

$$f(t) = \begin{cases} \sin(t), & \sin(t) \geq 0 \\ 0, & \sin(t) < 0 \end{cases}$$

Plot the function and calculate its crest factor from $t = 0$ to 4π . Then, introduce an outlier where the function has its maximum value, plot it again and recalculate the crest factor.

4) Based on the function in Exercise 3, what is the skewness before and after the introduction of the outlier? What does it mean in practical terms?

5) Consider the signals:

$$x(t) = [8 \ 3 \ 7 \ 2 \ 4 \ 6]$$

$$y(t) = [0 \ 8 \ 3 \ 7 \ 2 \ 4]$$

Compute the cross-correlation of them. What does the output of the cross-correlation function mean?

6) Consider the signal:

$$x(t) = [8 \ 3 \ 2 \ 1 \ 5 \ 8 \ 3 \ 2 \ 1 \ 4 \ 8 \ 3 \ 2 \ 1 \ 7 \ 8 \ 3 \ 2 \ 1]$$

Compute its autocorrelation. What does the output of the autocorrelation function mean?

7) Consider the signals:

$$f(t) = [1 \ 5 \ 0 \ 1 \ -2]$$

$$g(t) = [3 \ 4]$$

Compute the convolution of them. What does the output of the convolution function mean?

6. Fourier Analysis

6.1. Frequency Domain

The time domain and frequency domain are two ways to visualize signals. As explained in Chapter 5, time domain analysis gives the behavior of the signal over time, whereas frequency-domain refers to the amplitude versus frequency of each sine wave within the spectrum.

Time-domain analysis considers the amplitude of data over a period of time, such as displacement, velocity and acceleration to characterize a vibration problem. However, these signals contain relevant information and certain kinds of noise that are difficult or even impossible to detect when the analysis is performed only in the time-domain.

Other characteristics and anomalies of the signal are visible in the frequency-domain, which breaks the signal into its correspondent sine waves. Every waveform can be expressed as the sum of sine waves with different amplitudes, phases, and frequencies. Therefore, it is necessary to transform the data from the time-domain into a series of discrete sine waves in the frequency-domain.

Consider the following signal in time domain:

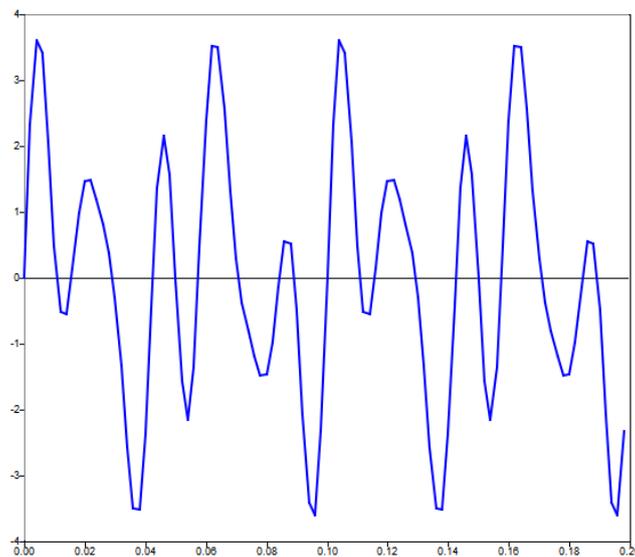


Figure 49. Time-domain signal

It may be decomposed in its constituent sine waves:

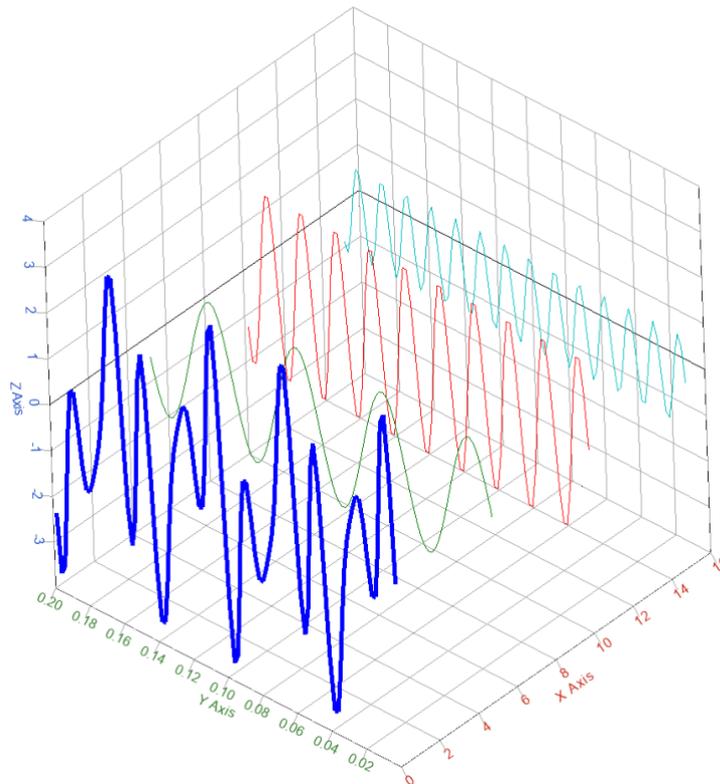


Figure 50. Time-domain signal (in blue) is composed of multiple sine waves

The signal comprises 3 different sine waves:

$$x_1(t) = 1.5\sin(2\pi * 20t)$$

$$x_2(t) = 2\sin(2\pi * 50t)$$

$$x_3(t) = \sin(2\pi * 70t)$$

The target of frequency-domain analysis is to identify the most predominant frequencies of the signal.

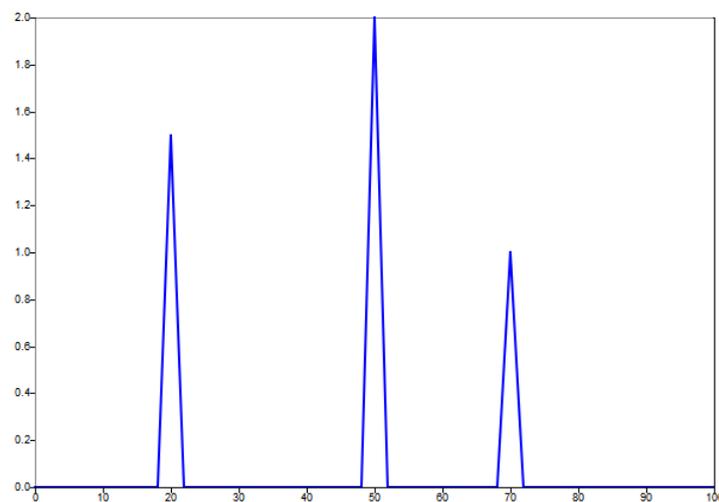


Figure 51. Frequency-domain analysis, with 3 peaks representing the predominant frequencies of the signal

6.2. Fourier Series

Fourier analysis is the representation of periodic waveforms in terms of trigonometric functions, enabling the decomposition of continuous functions into its essential sine waves. A function is considered periodic if $f(t + T) = f(t)$, where T is the period.

It is based on Fourier series, which is an expansion of a periodic function as an infinite sum of sines and cosines. As a periodic function, it has energy at specific frequencies, and the distribution of energy among these frequencies determines the shape of the wave.

Considering a periodic function $f(t)$ with period T , the Fourier series is given by:

$$f(t) = \frac{1}{2} a_0 + \sum_{n=1}^{\infty} a_n \cos\left(\frac{2\pi n t}{T}\right) + \sum_{n=1}^{\infty} b_n \sin\left(\frac{2\pi n t}{T}\right)$$

Where $n = 1, 2, 3, \dots$ and the coefficients are:

$$a_0 = \frac{2}{T} \int_{-\pi}^{\pi} f(t) dt$$

$$a_n = \frac{2}{T} \int_{-\pi}^{\pi} f(t) \cos\left(\frac{2\pi n t}{T}\right) dt$$

$$b_n = \frac{2}{T} \int_{-\pi}^{\pi} f(t) \sin\left(\frac{2\pi n t}{T}\right) dx$$

These coefficients will tell how much each frequency contributes to the function and they mathematically mean that they are projections of the function into the basis (cosine or sine according to the coefficient) as the frequency increases.

The same expression of Fourier series could be written with sine and cosine transforms instead, using Euler's formula and its derived expressions, as explained in Chapter 2:

$$e^{ix} = \cos(x) + i\sin(x)$$

$$\cos(x) = \frac{1}{2} e^{ix} + \frac{1}{2} e^{-ix}$$

$$\sin(x) = -\frac{1}{2} e^{ix} + \frac{1}{2} e^{-ix}$$

Substituting them in the Fourier series:

$$f(t) = \sum_{n=-\infty}^{\infty} A_n e^{\frac{i2\pi n t}{T}}$$

Where $n = 1, 2, 3, \dots$ and the coefficient is:

$$A_n = \begin{cases} \frac{1}{2}a_n - \frac{1}{2}ib_n & n \geq 1 \\ \frac{1}{2}a_0 & n = 0 \\ \frac{1}{2}a_n + \frac{1}{2}ib_n & n \leq -1 \end{cases}$$

For a function with period T , the signal repeats in $1T, 2T, 3T, \dots, nT$ and therefore the respective frequencies of sines and cosines are $\frac{2\pi}{T}, \frac{4\pi}{T}, \frac{6\pi}{T}, \dots, \frac{2\pi n}{T}$, i.e. they are multiples of the period T and fundamental frequency $\frac{2\pi}{T}$. Therefore the n -th frequency is called the n -th harmonic of the signal.

As an example, consider a square wave with period $T = 1$:

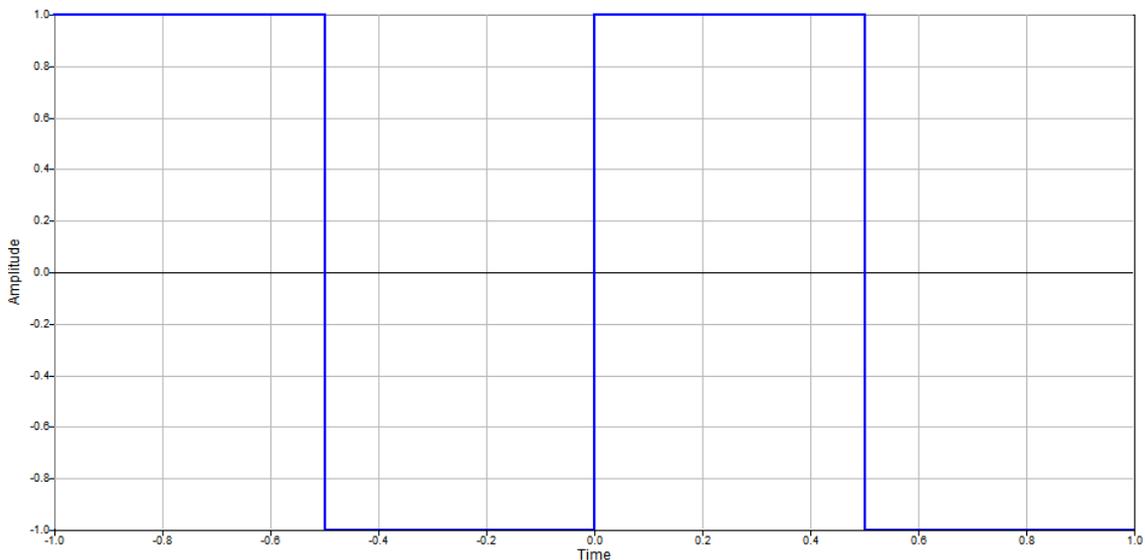


Figure 52. Square wave

To compute its Fourier series, it is necessary to obtain the coefficients and then apply them in the series formula. Using the previous equations, consider 5 terms of the Fourier series, i.e. 5 harmonics, which are sinusoids whose frequency is an integer multiple of the fundamental frequency:

```
N = 5;
T = 1;
t = linspace(-1, 1, 10000);
f = 0*t;

for n=-N:1:N
    if (n==0)
```

```

        continue;

    end;

    A_n = (1/(pi*i*n))*(1-exp(-pi*i*n/T));

    f_n = A_n*exp(2*pi*i*n*t/T);

    f = f + f_n;

end

```

Using t and f variables, the time vector and the correspondent function values, the plot is generated:

```
plot(t, f);
```

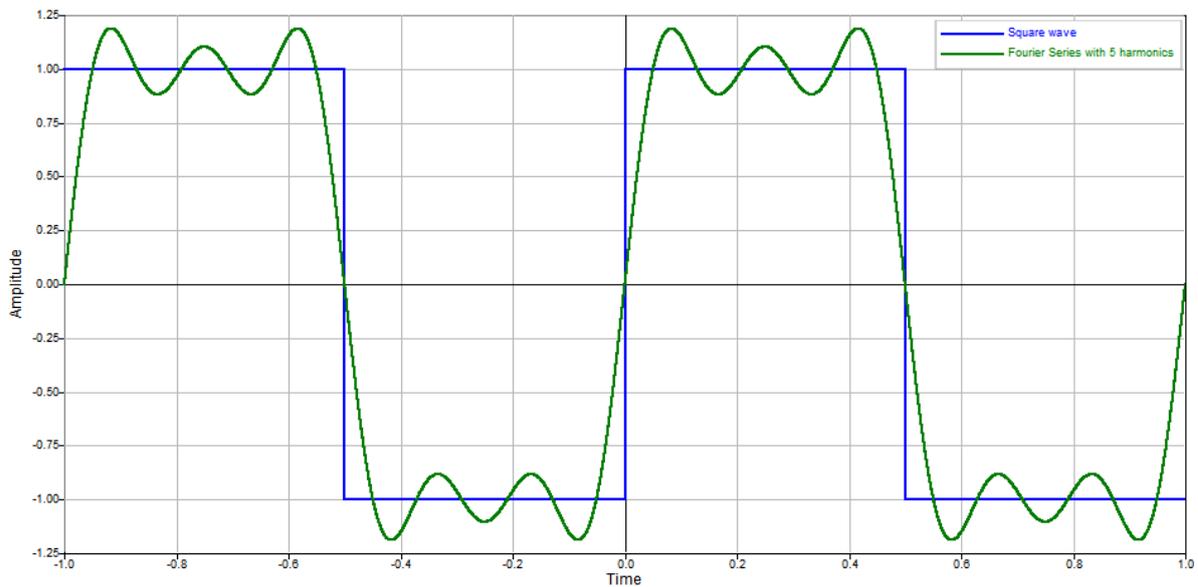


Figure 53. Square wave and respective Fourier series with 5 harmonics

As the number of terms increases, namely harmonics, the Fourier series approach the square wave form.

For $N = 25$:

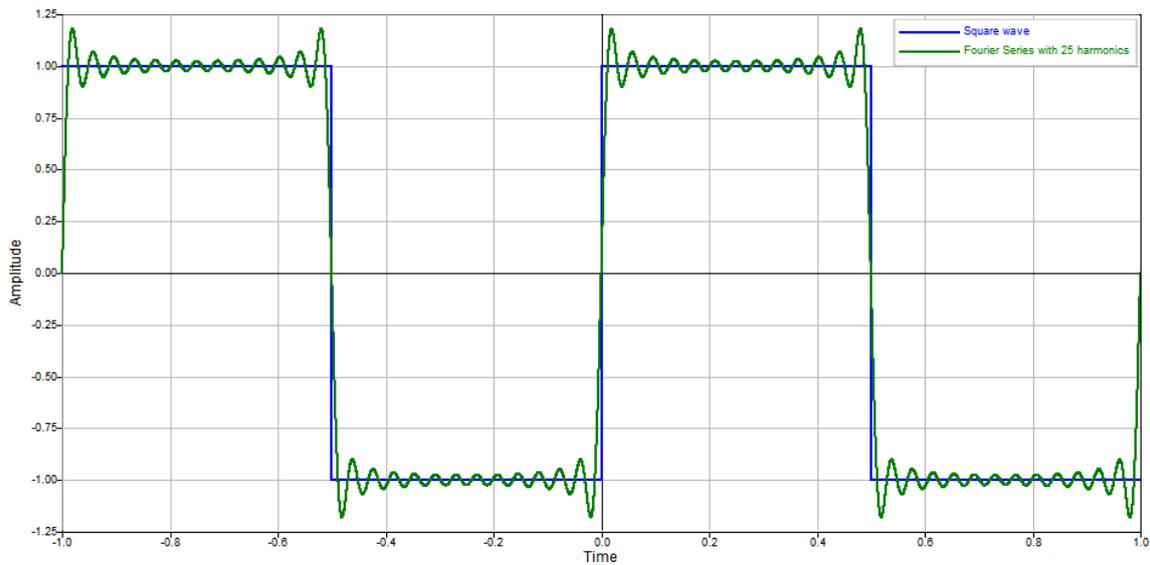


Figure 54. Square wave and respective Fourier series with 25 harmonics

For $N = 99$:

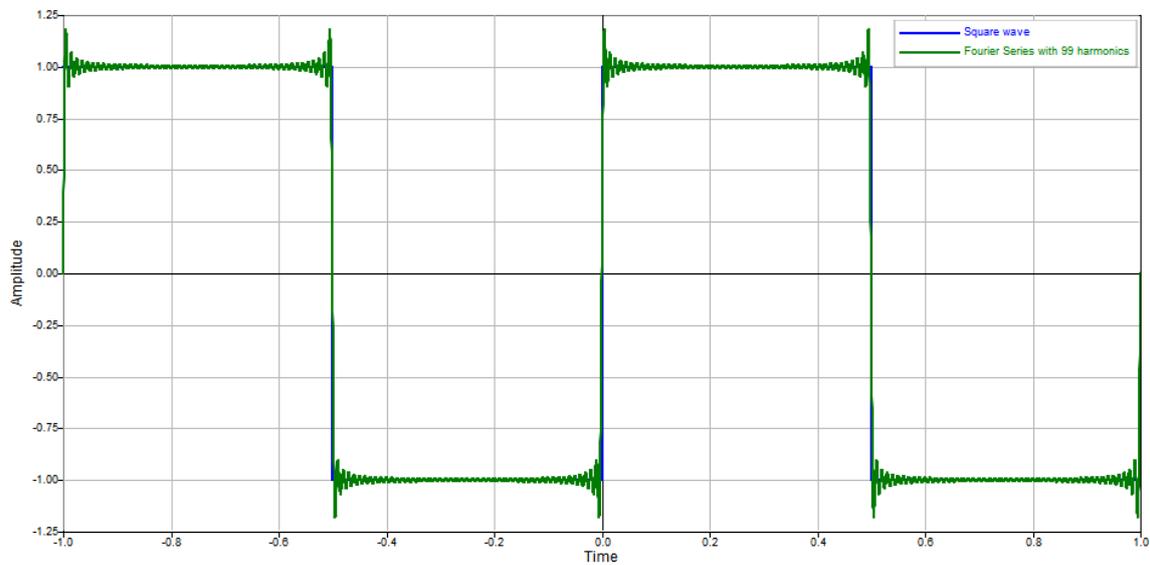


Figure 55. Square wave and respective Fourier series with 99 harmonics

For a Fourier series, the more harmonics taken into account then the more accurate the approximation of the wave function becomes. Hence, the aim of Fourier analysis is to calculate the Fourier coefficients for the largest possible value of harmonics. The spikes near the discontinuities that do not reduce in amplitude regardless of the number of harmonics considered may be explained by the Gibbs phenomenon.

In some problems, the Fourier coefficients a_0 , a_n or b_n become zero after integration. Finding these null coefficients is time consuming but can be avoided by considering the concept of even and odd functions because these coefficients can be predicted without performing the integration.

6.2.1. Simplification for Even Functions

A function is said to be even if $f(-t) = f(t)$ for all values of t . For such functions, b_n is equal to zero and therefore only a_0 and a_n should be calculated. The Fourier series becomes:

$$f(t) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty} a_n \cos\left(\frac{2\pi nt}{T}\right)$$

6.2.2. Simplification for Odd Functions

A function is said to be odd if $f(-t) = -f(t)$ for all values of t . For such functions, a_0 and a_n are equal to zero and therefore only b_n should be calculated. The Fourier series becomes:

$$f(t) = \sum_{n=1}^{\infty} b_n \sin\left(\frac{2\pi nt}{T}\right)$$

6.3. Gibbs Phenomenon

The Gibbs phenomenon is a peculiar behavior of overshoots and undershoots of a Fourier series near its discontinuities.

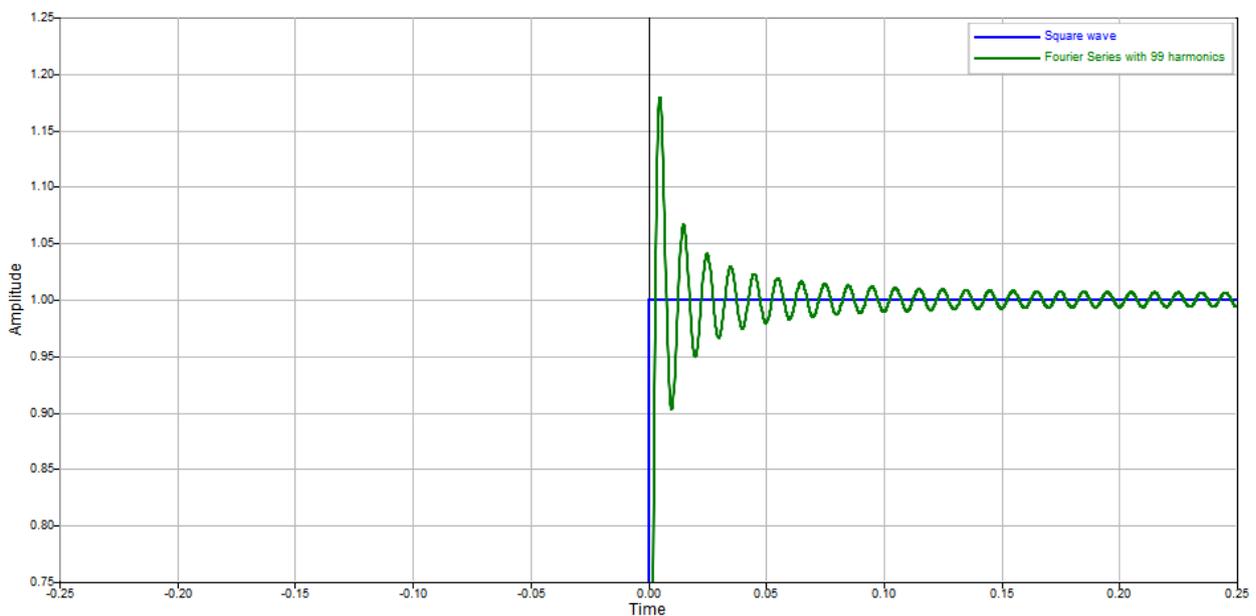


Figure 56. Gibbs phenomenon

This occurrence converges to a finite value and reduces in width and energy as the number of summed harmonics grows, but it does not decrease in amplitude. This convergence value is about 9% of the discontinuity and its demonstration is beyond of scope of this book.

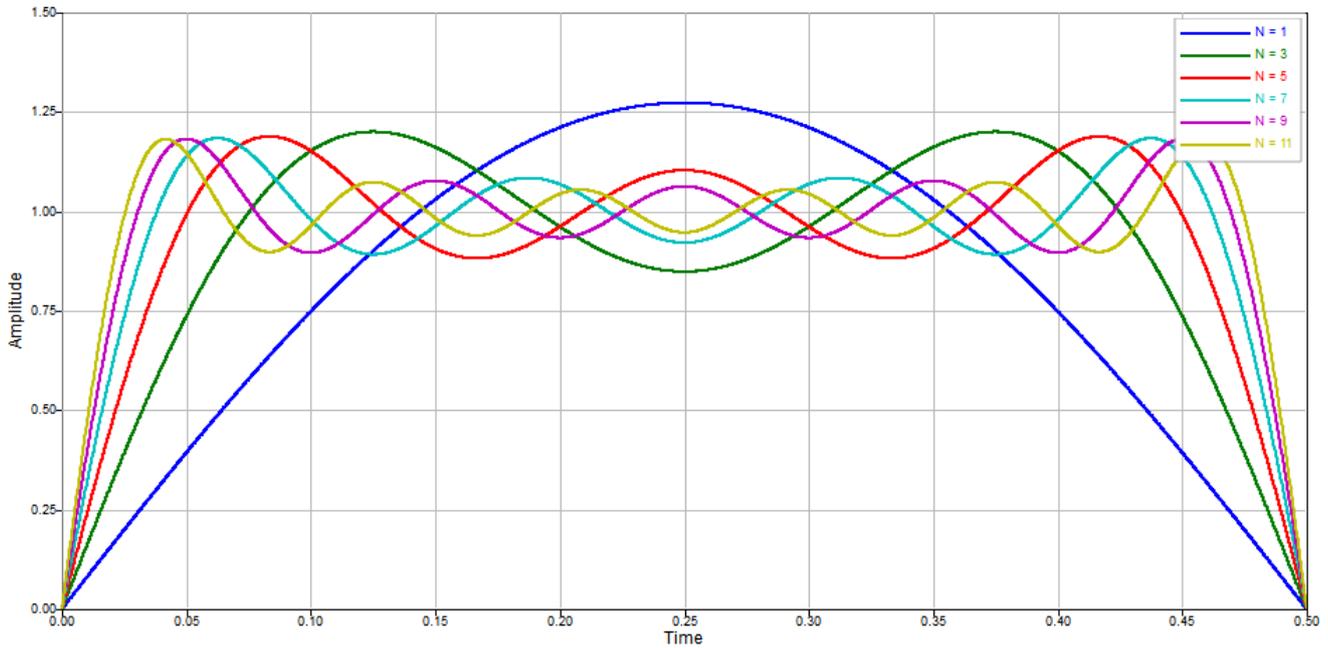


Figure 57. Gibbs phenomenon according to the number of harmonics

Any signal with a sudden discontinuity exhibits the Gibbs phenomenon because this abrupt change and its correspondent infinite frequency content are difficult to capture with acquisition systems having finite frequency capabilities. The truncation of frequency content causes a time domain ringing artifact on the signal, which is an error characterized as a spurious output near sharp transitions in a signal.

6.4. Discrete Fourier Transform (DFT)

6.4.1. Definition

DFT is the Fourier analysis tool for finite-length signals. Since digital systems cannot work with continuous-time signals, real-life applications take samples of the signal to be analyzed instead of the original signal. It is defined by the equation that transforms the signal into discrete frequencies based on the length of the data set (N):

$$X(k) = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N}$$

Where x_n is a sequence of complex numbers; $X(k)$ is another sequence of complex numbers, with the frequency representation of the time-domain signal. The same expression could also be written with sine and cosine transforms instead, using Euler's formula, but this form is more cumbersome and less compact because it requires two separate transforms.

DFT is useful to reveal periodicity of the input signal as well as the relative power of any periodic components, namely frequencies. For example, consider the Dirac delta function, which is equal to zero everywhere except for zero:

$$\delta(n) = \begin{cases} 1, & n = 0 \\ 0, & \text{otherwise} \end{cases}$$

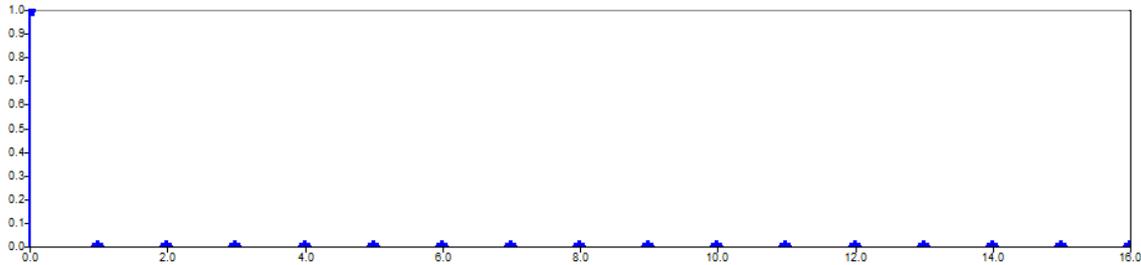


Figure 58. Delta function plot

It is seen that each $X(k)$ has N computations and there are N x_n to be computed, meaning that the total computational complexity is N^2 .

If DFT is applied:

$$X(k) = \sum_{n=0}^{N-1} \delta(n) e^{-i2\pi kn/N}$$

Using the commands applied to a variable called `delta`, representing the delta function:

```
N = length(delta);
W = -j*2*pi/N;
W_mat = repmat(W,N,N);
k = (0:N-1)';
k_mat = repmat(k,1,N);
n = 0:N-1;
n_mat = repmat(n,N,1);
Xk = exp(W_mat.*k_mat.*n) * delta';
```

Leads to the following result:

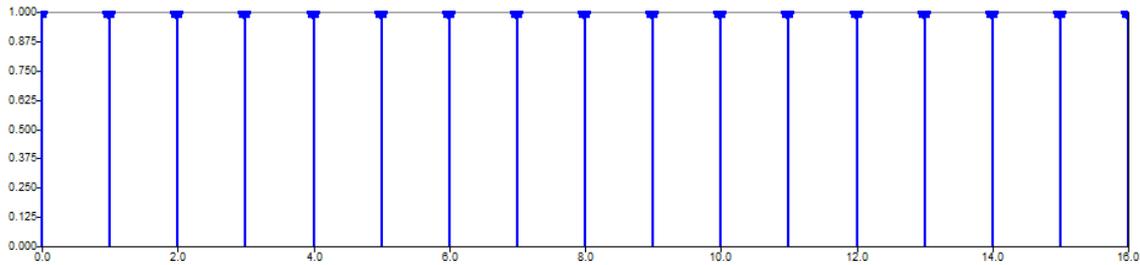


Figure 59. DFT of Delta function

It means that the Delta function contains all frequencies over the possible range of frequencies with equal contribution to the signal.

Considering now another example having a sinusoidal function:

$$f(t) = 5 \cos\left(\frac{2\pi t}{32}\right)$$

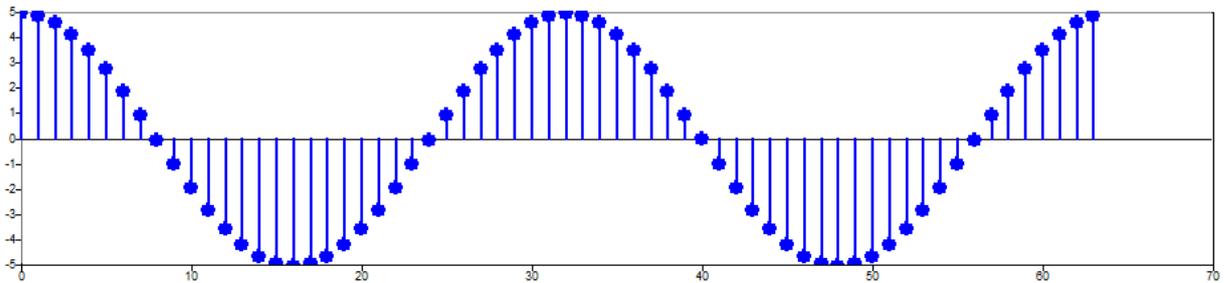


Figure 60. Example of sinusoidal function

Using the trigonometric relations explained previously:

$$f(t) = 5 \cos\left(\frac{2\pi t}{32}\right) = 5 \left(\frac{1}{2} e^{\frac{2\pi t i}{32}} + \frac{1}{2} e^{-\frac{2\pi t i}{32}} \right) = \frac{5}{2} \left(e^{\frac{2\pi t i}{32}} + e^{-\frac{2\pi t i}{32}} \right)$$

As the second term in the exponential is negative, it can be converted to positive according to the trigonometric circle:

$$-\frac{2\pi}{32} + 2\pi = \frac{62\pi}{32}$$

$$f(t) = \frac{5}{2} \left(e^{\frac{2\pi t i}{32}} + e^{\frac{62\pi t i}{32}} \right)$$

The original signal is expressed as a sum of 2 Fourier basis vectors. If DFT is computed with the same commands as for the Delta function, it gives:

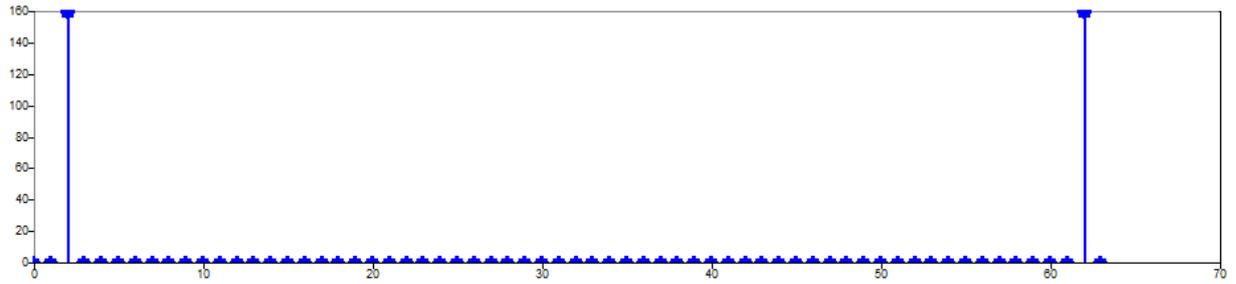


Figure 61. DFT of sinusoidal function (real part)

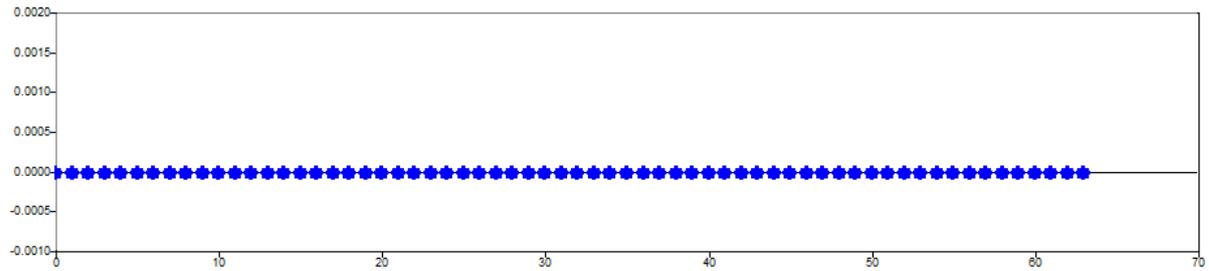


Figure 62. DFT of sinusoidal function (imaginary part)

In fact, only 2 values are non-zero in the real part of the DFT that correspond to the strongest frequencies present in the signal. With respect to the imaginary part, all values are zero, as expected because the considered function was even.

In the case where a phase is added to the original sinusoidal function:

$$f(t) = 5 \cos\left(\frac{2\pi t}{32} + \frac{\pi}{6}\right)$$

The real part has 2 non-zero values at the same frequencies as before, but smaller amplitude, and the imaginary part now has 2 non-zero values too. As the function is not considered to be even after the phase has been added, it explains why the imaginary part is no longer completely null.

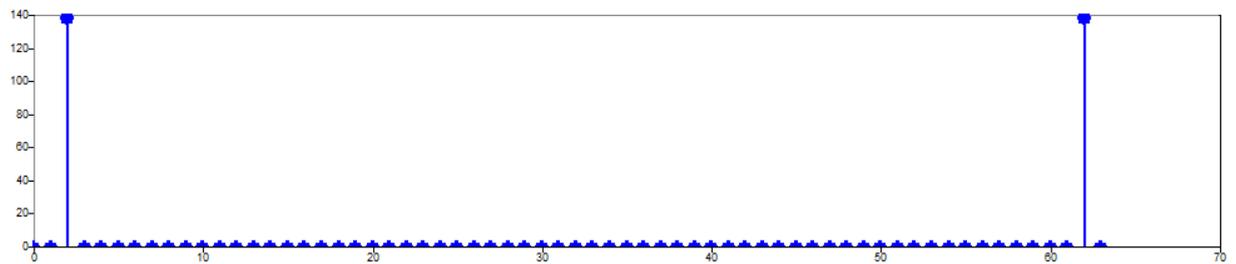


Figure 63. DFT of sinusoidal function with non-zero phase (real part)

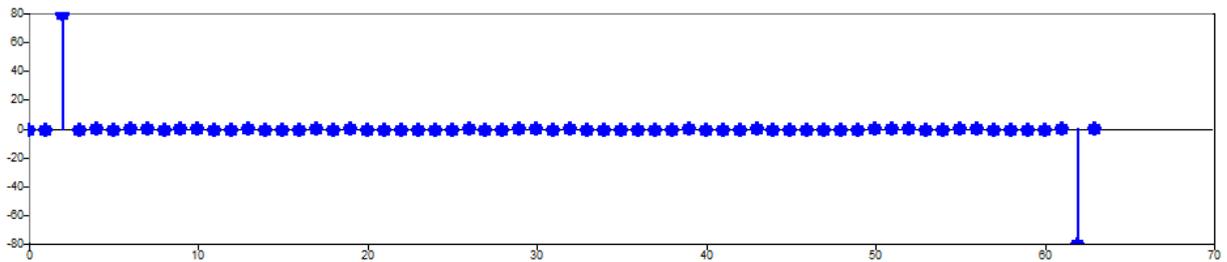


Figure 64. DFT of sinusoidal function with non-zero phase (imaginary part)

- **magnitude** represents the relative presence of a certain sinusoid in the function, i.e. the relative power of the signal at different frequencies.
- **phase** determines how the sinusoids are aligned with respect to one another, i.e. the phase angle of the signal at different frequencies.

Whenever the phase angle φ of the exponential function in the Fourier transform is larger than π , 2π is subtracted and φ stays between $-\pi$ and π .

It is noticeable that DFT output has conjugate complex symmetry, which means that it is symmetric with respect to the real part and symmetric in magnitude with respect to the imaginary part, but opposite in sign.

6.5. Fast Fourier Transform (FFT)

6.5.1. Definition

Fast Fourier Transform (FFT) is one of the most useful and popular algorithms in signal processing. The underlying FFT process is DFT (see: 6.4.1 for equations), but with much more reduced computational complexity, which drops from N^2 to $N \log_2(N)$. The FFT algorithm is also significantly faster than DFT as the length of the signal increases.

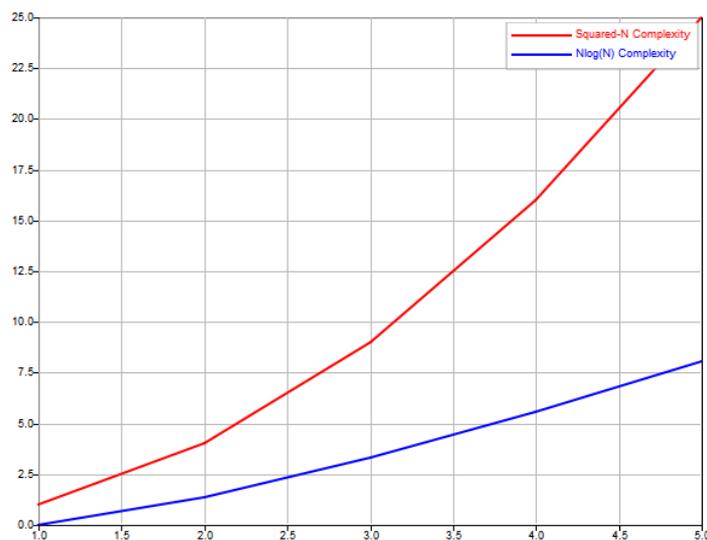


Figure 65. Comparison of computational complexity between FFT and DFT

Table 5. Comparison of number of operations between DFT and FFT according to the number of samples

Number of Samples	10^2	10^4	10^6
Number of Operations (DFT)	10^4	10^8	10^{12}
Number of Operations (FFT)	~ 664	$\sim 1.3 \cdot 10^5$	$\sim 2 \cdot 10^7$

Periodicity is key because the term $-\frac{i2\pi kn}{N}$ is considered to be periodic. Trigonometric relationships reduce the complexity of calculation because many terms give the same value that was calculated before:

$$k * 0 = k * 4 = k * 8 \dots$$

$$k * 1 = k * 5 = k * 9 \dots$$

The most important algorithms used for FFT computation, such as Cooley-Tukey, exploit this symmetry to reduce the number of operations by separating the DFT in two sums: one with points in even positions, the other with points in odd positions.

$$\begin{cases} 2n & \text{if even} \\ 2n + 1 & \text{if odd} \end{cases}$$

With n ranging from 1 to $N/2-1$. Substituting the two sums into the DFT equation gives:

$$X(k) = \sum_{n=0}^{N-1} x_n e^{-\frac{i2\pi kn}{N}} = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} e^{-\frac{i2\pi k(2n)}{N}} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} e^{-\frac{i2\pi k(2n+1)}{N}}$$

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} e^{-\frac{i2\pi kn}{N/2}} + e^{-\frac{i2\pi k}{N}} \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} e^{-\frac{i2\pi kn}{N/2}}$$

$$X(k) = X_k^{even} + e^{-\frac{i2\pi k}{N}} X_k^{odd}$$

Where X_k^{even} is the k -th component of the Fourier transform comprising even components of the original signal; X_k^{odd} comprises odd components. Each of the two transforms is periodic having period $N/2$.

The same concept in a visual representation, considering an 8-point signal:

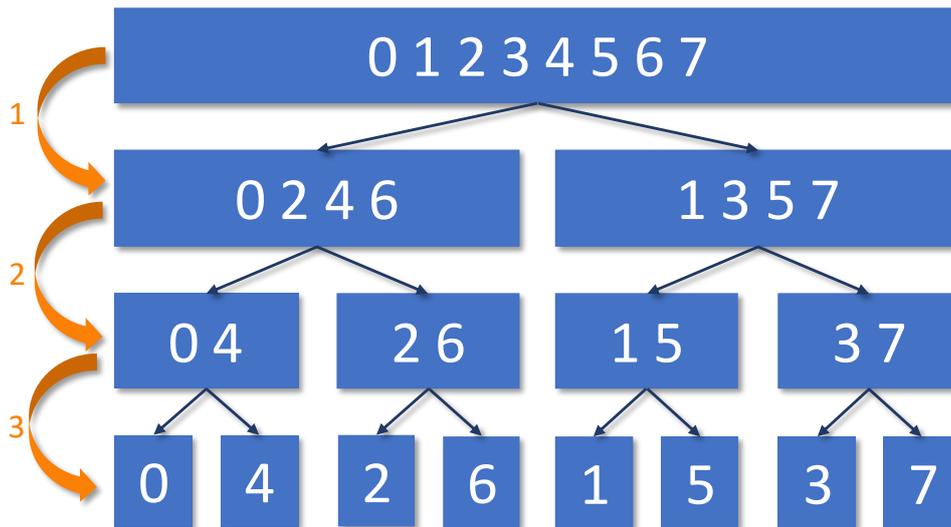


Figure 66. Visual representation of FFT algorithm

The decomposition represents a reordering of the samples and is carried out until there are N signals with 1 point each. The number of steps to achieve such a condition is $\log_2 N$, in this case $\log_2 8 = 3$.

As the frequency spectrum of a 1-point signal is equal to itself, the final step in the FFT process is to combine the N frequency spectrum points in the exact reverse order of the time domain decomposition, leading to a total computational complexity of $N \log_2(N)$, as explained before, which is $8 * 3 = 24$ in this example with an 8-point signal.

6.5.2. FFT Function

The function `fft` gives the frequency-domain representation of the input signal. Some other arguments may also be given, such as the size of FFT (which is the length of the input vector by default) and the dimension on which to operate.

```
f = fft(signal, nFFT, dim)
```

- If the length of the input vector is not the default, the signal vector is either truncated or extended with zeros to the specified length; known as zero padding. The effect on the frequency spectrum is to insert additional samples symmetrically with respect to the Nyquist frequency.
- The N frequencies associated with the outputs are spaced in increments of fs/N , where fs is the sampling frequency. This ratio is the frequency resolution, as explained previously.
- The `fft` output is returned as double-sided, ranging from $-F_{max}$ to F_{max} . Most real-life applications only use the positive half of the frequency spectrum, known as single-sided, as the spectrum is symmetrical, thus the negative frequency information is redundant. To

discard half the signal, consider only half the points plus 1 and account for the power of the discarded part multiplying the final spectrum by 2:

```
P1 = P2 (1:L/2+1) ;
P1 (2:end-1) = 2*P1 (2:end-1) ;
```

Consider a signal composed of 3 different sine waves with amplitudes of 1.5, 2 and 1, and frequencies 20, 50 and 70 Hz, respectively:

```
X1 = 1.5*sin(2*pi*20*t) ;
X2 = 2*sin(2*pi*50*t) ;
X3 = 1*sin(2*pi*70*t) ;
X = X1 + X2 + X3 ;
```

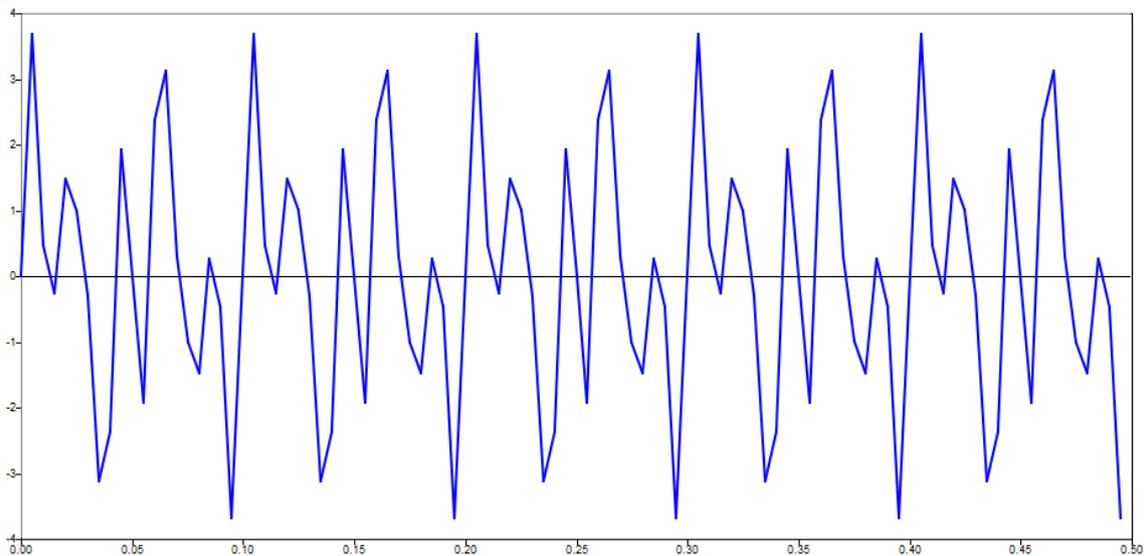


Figure 67. Example of signal made of 3 sine waves

With a sampling frequency of 200 Hz and length of signal of 100:

```
Fs = 200 ;
L = 100 ;
T = 1/Fs ;
t = (0:L-1) *T ;
```

The FFT can be computed, its negative half discarded and then plotted:

```
Y = fft(X) ;
P2 = abs(Y/L) ;
```

```

P1 = P2 (1:L/2+1) ;

P1 (2:end-1) = 2*P1 (2:end-1) ;

f = Fs*(0:(L/2))/L;

plot(f,P1);

```

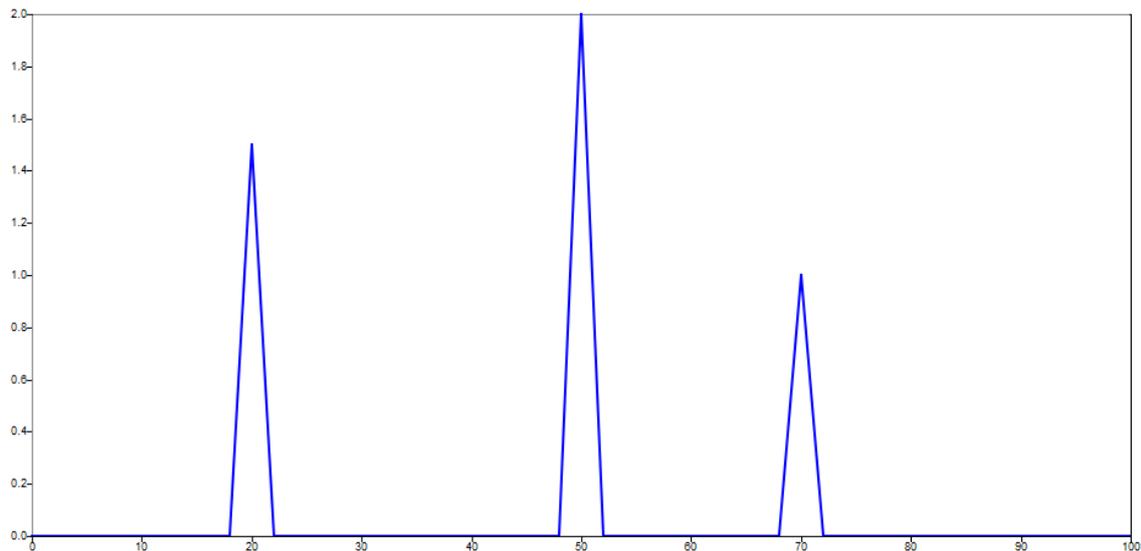


Figure 68. FFT plot of the signal

Here, 3 frequencies are clearly identified with their respective amplitude:

- Peak equal to 1.5 and frequency equal to 20 Hz
- Peak equal to 2 and frequency equal to 50 Hz
- Peak equal to 1 and frequency equal to 70 Hz

6.5.3. Symmetry Considerations for FFT

Regarding symmetry of FFT results, described as a double-sided spectrum, it is important to remember that even functions can be written as a sum of cosines, whereas odd functions can be written as a sum of sines; see: 6.2. It has practical implications on the results because even functions have even transforms, whereas odd functions have odd transforms.

Consider $\cos(t)$ and $\sin(t)$ as examples of even and odd functions, respectively. Using a sampling frequency of 100 Hz and constructing two signals with length of 100 based on cosine and sine waves with frequency equals to 20 Hz:

```

Fs = 100;

T = 1/Fs;

L = 100;

t = (0:L-1)*T;

```

```
y_even = cos(2*pi*30*t);  
y_odd = sin(2*pi*30*t);
```

It is possible to use the `fft` function to visualize the frequency domain representation of both signals `y_even` and `y_odd`, folding the spectrum and creating the respective frequency vector to visualize a plot with an axis containing positive and negative values (double-sided spectrum). This double-sided spectrum is obtained using the `fftshift` function, which reorganizes the outputs of the `fft` function by shifting the frequency spectrum to position its zero component in the center of the output:

```
Y_even = fft(y_even);  
Y_odd = fft(y_odd);  
f = linspace(-Fs/2, Fs/2, L);  
Y_even = fftshift(Y_even);  
Y_odd = fftshift(Y_odd);
```

For the **even function**, plotting the respective real and imaginary parts by fixing the y-axis limit gives a better understanding:

```
subplot(2,1,1);  
plot(f, real(Y_even));  
title('Cosine');  
ylim([-50 50]);  
subplot(2,1,2);  
plot(f, imag(Y_even));  
ylim([-50 50]);  
set(gca, 'ytick', 6);
```

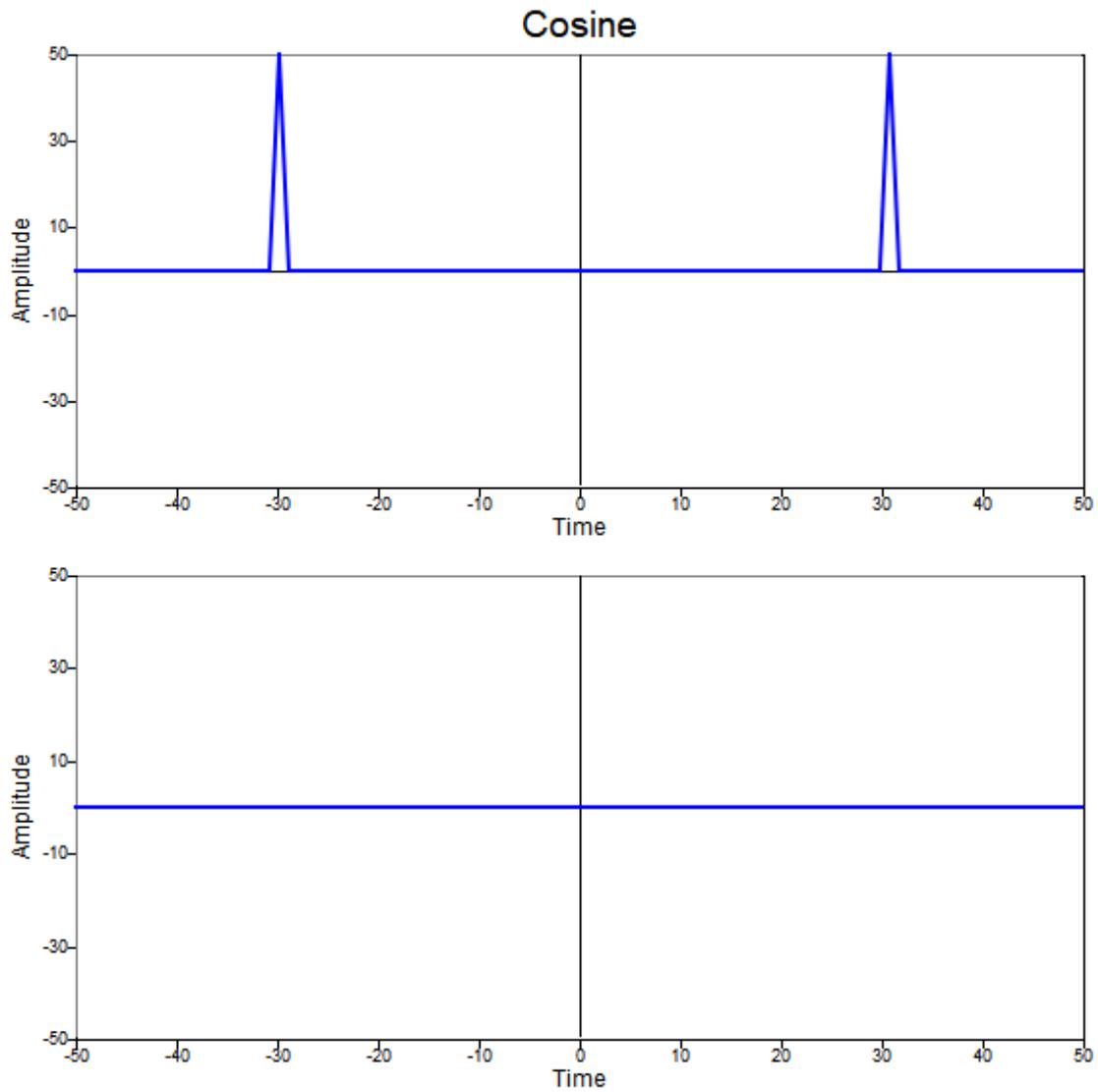


Figure 69. Double-sided FFT result of even function with positive frequency with its respective real and imaginary parts plotted

The same values are obtained when a negative frequency is used:

$$y_{\text{even}} = \cos(2 * \pi * (-30) * t);$$

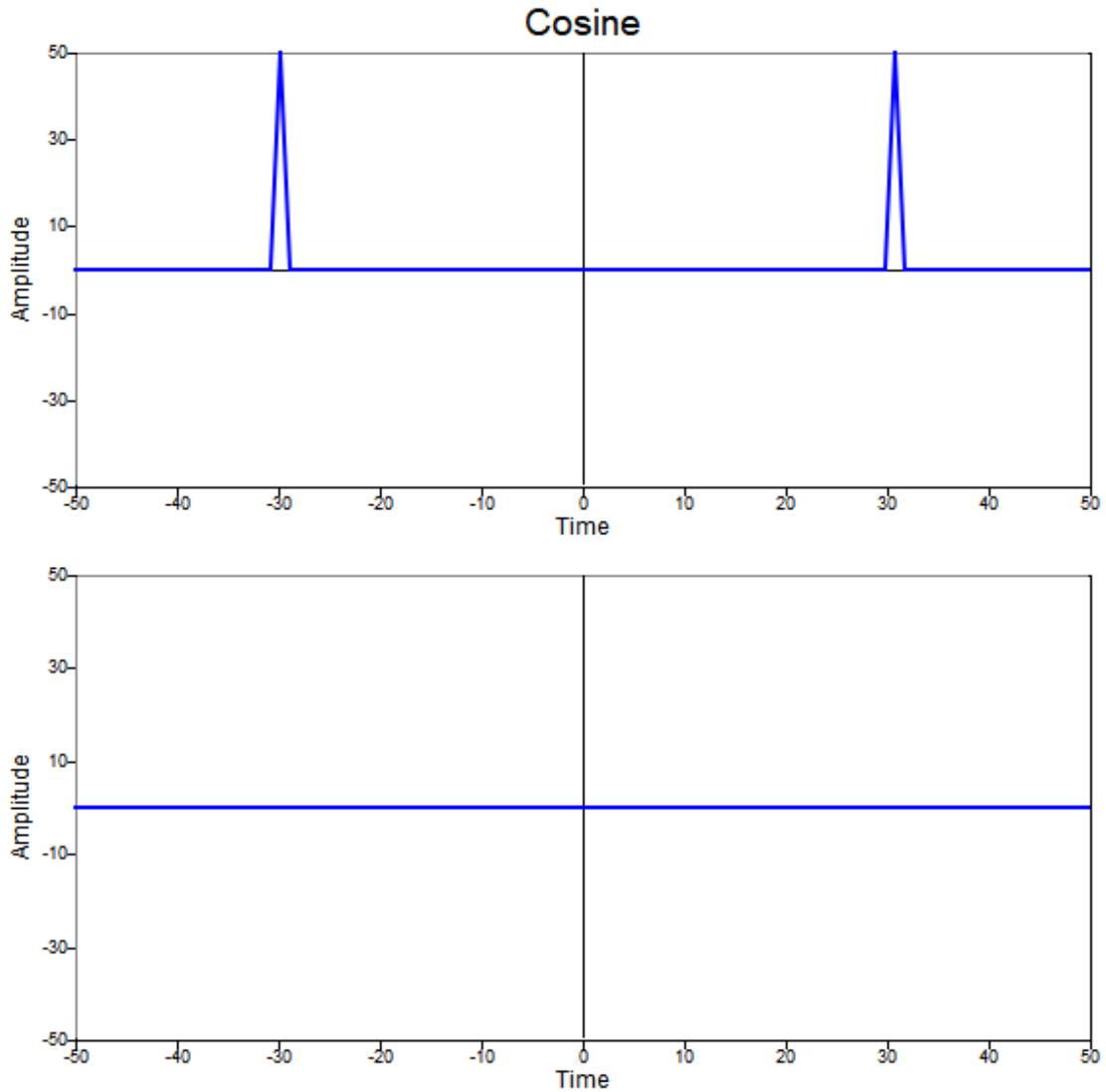


Figure 70. Double-sided FFT result of even function with negative frequency with its respective real and imaginary parts plotted

Verify that the frequency domain representation of the even function has the characteristics:

- 2 non-zero values in the real part with same magnitude.
- Only zero values in imaginary part (see: 6.2.1).

Therefore, the transform of an even function is also even:

- The FFT of an even function is real.
- The FFT of an even function is even.

For the **odd function**, plotting the respective real and imaginary parts, fixing the y axis:

```
subplot(2,1,2);
plot(f, real(Y_odd));
title('Sine');
```

```

ylim([-50 50]);

set(gca, 'ytick', 6);

subplot(2,1,2);

plot(f, imag(Y_odd));

ylim([-50 50]);

set(gca, 'ytick', 6);

```

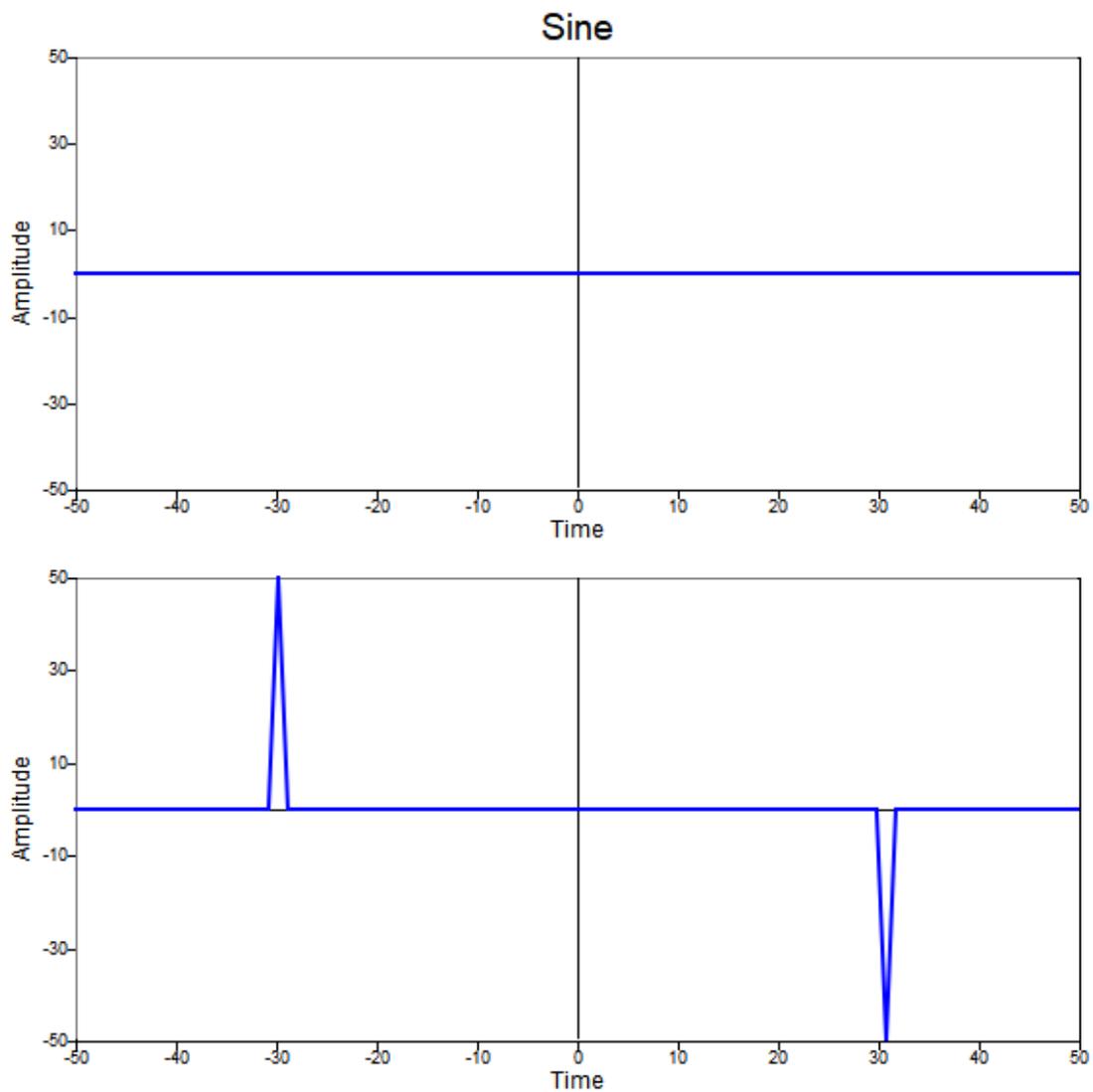


Figure 71. Double-sided FFT result of odd function with positive frequency with its respective real and imaginary parts plotted

The same values but conjugate are obtained when a negative frequency is used:

```

y_odd = sin(2*pi*(-30)*t);

```

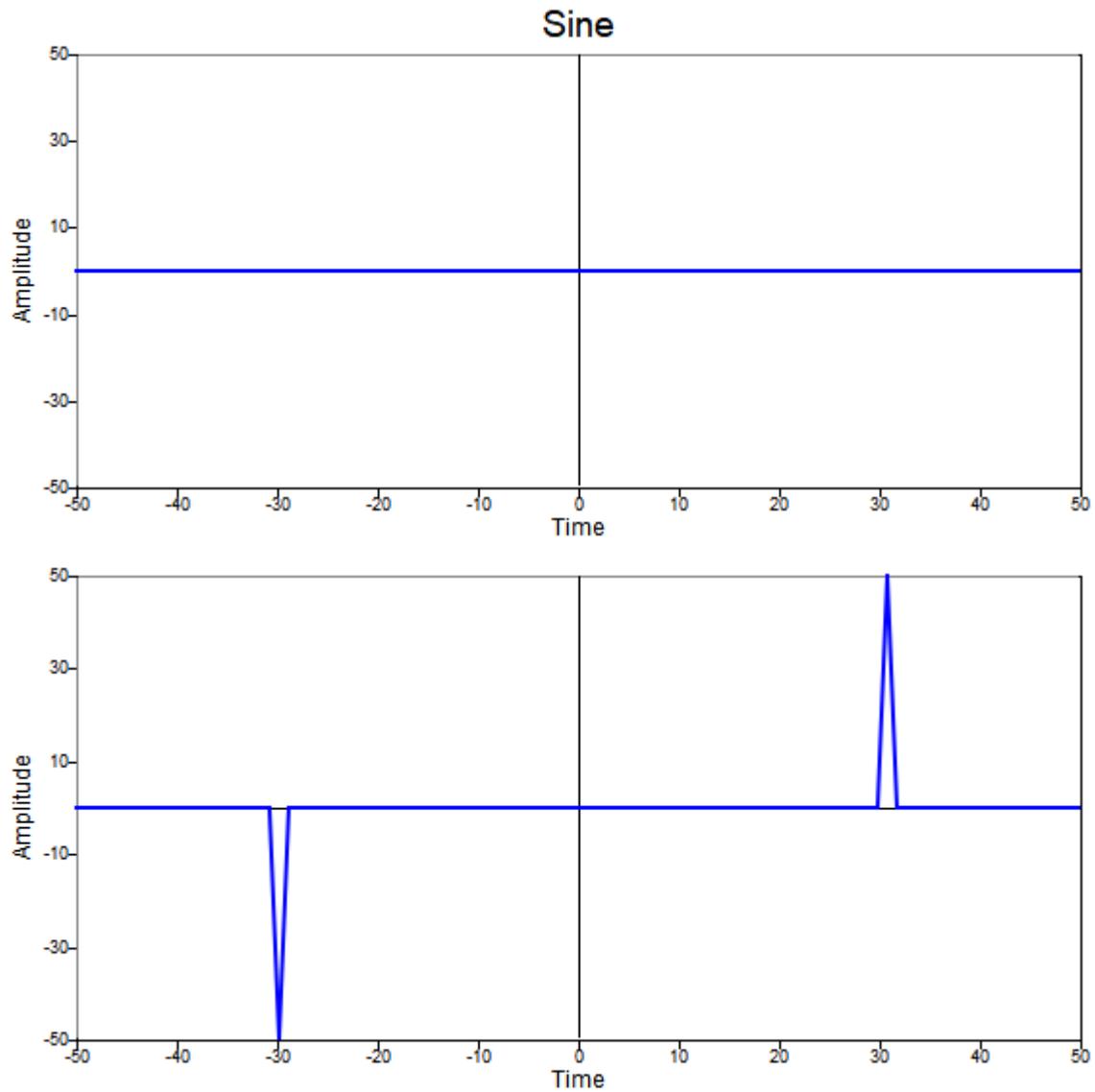


Figure 72. Double-sided FFT result of odd function with negative frequency with its respective real and imaginary parts plotted

Verify that the frequency domain representation of the odd function has the characteristics:

- Only zero values in the real part (see: 6.2.2).
- 2 non-zero values in imaginary part, with same magnitude but different signs.

Therefore, the transform of an odd function is also odd:

- The FFT of an odd function is imaginary.
- The FFT of an odd function is odd.

In summary of these 4 symmetries, the Fourier transform of a real function is:

- Real part symmetry (even): $real\{X(-k)\} = real\{X(k)\}$
- Imaginary part antisymmetric (odd): $imag\{X(-k)\} = -imag\{X(k)\}$
- Magnitude symmetry (even): $|X(-k)| = |X(k)|$
- Phase antisymmetric (odd): $\angle X(-k) = -\angle X(k)$

In other words, if a signal $X(k)$ is real, then its spectrum is said to be Hermitian, i.e. conjugate symmetric. The definitions stated for even and odd symmetries are similar to the real-valued functions but with complex conjugation.

6.5.4. Impacts of Frequency Resolution on the FFT Result

In practical terms, increasing the frequency resolution will decrease the number of frequency bins, leading to a larger frequency difference between each; as explained in Chapter 2. One way to accomplish it is to decrease the length of the signal while keeping the sampling frequency constant. With $L = 90$ in the previous example:

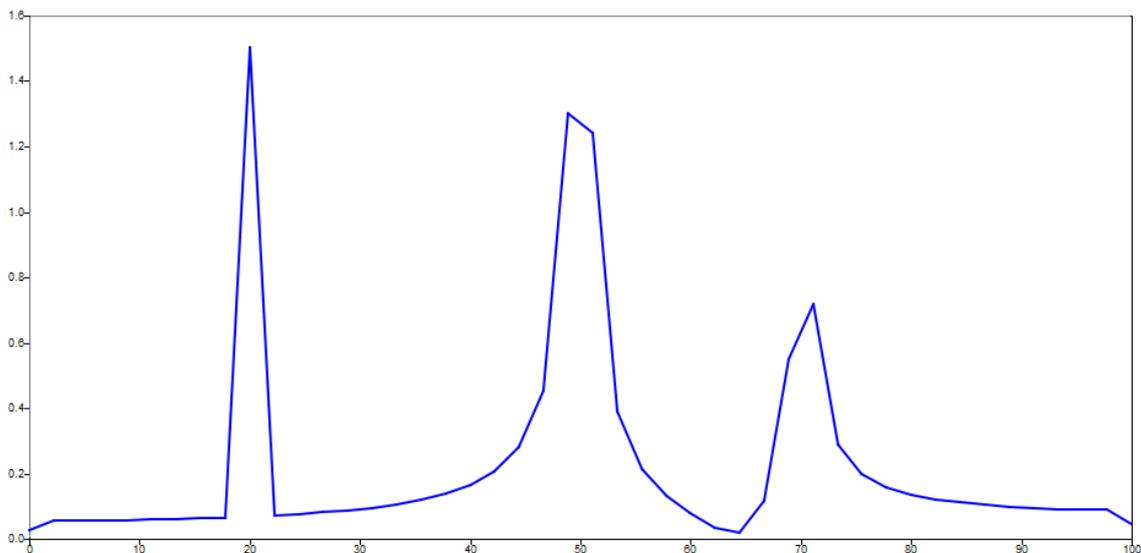


Figure 73. FFT plot of the signal with higher frequency resolution

When looking at the new peaks in the frequency spectrum, it is noticeable that one of the signal frequencies is approximately between 48 Hz and 51 Hz because both had a large magnitude. Likewise, the third peak is approximately between 69 Hz and 71 Hz. Sine waves whose frequencies are not integer multiples of the frequency resolution will experience this kind of leakage, which will be discussed in the next chapters.

The same situation would happen if the frequency resolution was slightly decreased with an increase of the length of the signal. There is a delicate trade-off between leakage suppression and spectral resolution, either to increase or to decrease it.

6.5.5. Impacts of Sampling Frequency on the FFT Result

According to the Nyquist theorem mentioned in Chapter 2, the sampling frequency should be greater than 2 times the maximum frequency contained in the waveform. In this case, the maximum frequency is 70 Hz, therefore the sampling frequency should be at least 140 Hz.

If the sampling frequency is less than 140 Hz, this value is insufficient to capture each peak of the signal and aliasing will occur, which is a false representation of the signal spectrum.

With $F_s = 100$:

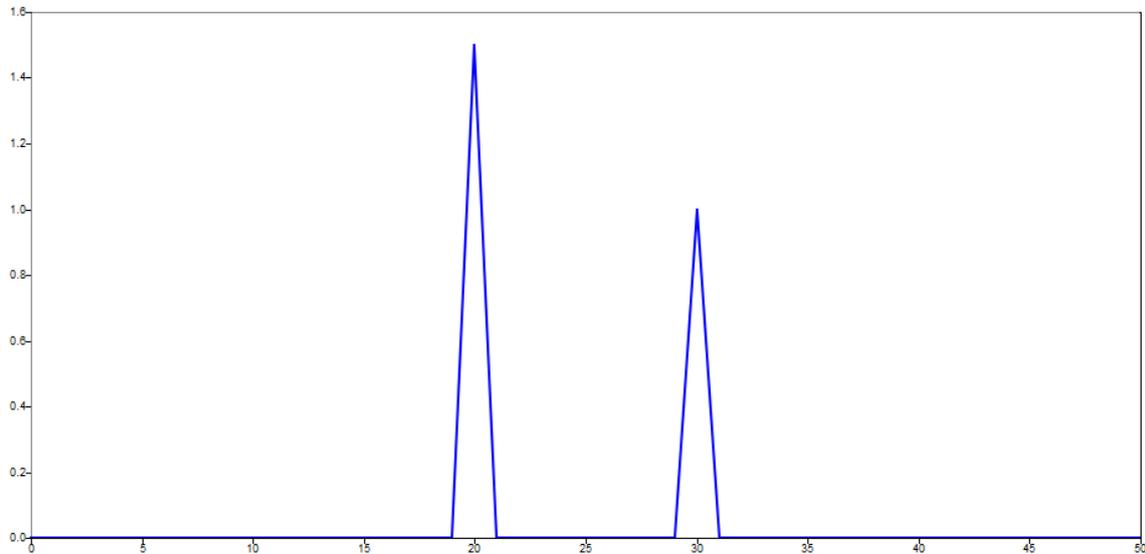


Figure 74. FFT plot of the signal with smaller sampling frequency

Whereas, if the sampling frequency is greater than 140 Hz, there are more than enough samples to capture the true spectrum of the signal.

6.6. Inverse Fast Fourier Transform

It is possible to get the signal back from the frequency-domain into the time-domain using an Inverse Fast Fourier Transform (IFFT). It is simply a transformation from the frequency-domain to the time-domain using the same principle: the underlying process of Inverse FFT is Inverse DFT.

The algorithm also has the same advantages of FFT over DFT, as explained previously. The DFT formulation is simply rewritten isolating x_n terms to use the inverse DFT to navigate from one domain to the other according to the problem to be solved:

$$X(k) = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N} \quad \leftrightarrow \quad x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{i2\pi kn/N}$$

As seen in this equation, the synthesis formula (moving from the frequency to the time domain) is a linear combination of the basic functions, scaled by the coefficients found in the analysis formula (term $1/N$). Using the Delta function, explained previously, the outcome of the DFT calculation can be computed:

```

N = length(Xk);
W = -j*2*pi/N;
W_mat = repmat(W,N,N);
k = (0:N-1)';
k_mat = repmat(k,1,N);
n = 0:N-1;
n_mat = repmat(n,N,1);
delta_IDFT = (1/N)*exp(W_mat.*k_mat.*n)*Xk;

```

The result is:

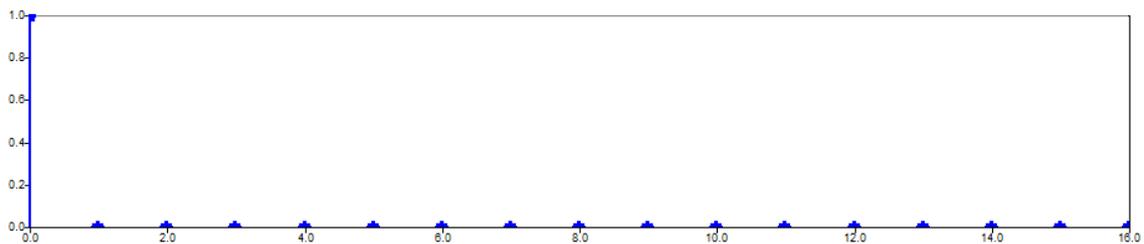


Figure 75. Delta function obtained from Inverse DFT

The function `ifft` gives the signal-domain representation of the input signal, along with some other arguments, such as the size of the FFT (which is the length of the input vector by default) and the dimension on which to operate. If the length of the input vector is not the default, the signal vector is either truncated or extended with zeros to the specified length, exactly like the `fft` function:

```
signal = ifft(f,nFFT,dim)
```

Considering the output of the `fft` function of a sinusoidal waveform comprising 3 frequencies, the time-domain signal can be computed with `ifft` again and then plotted:

```

X_ifft = ifft(Y);
scatter(t,X_ifft,'r');

```

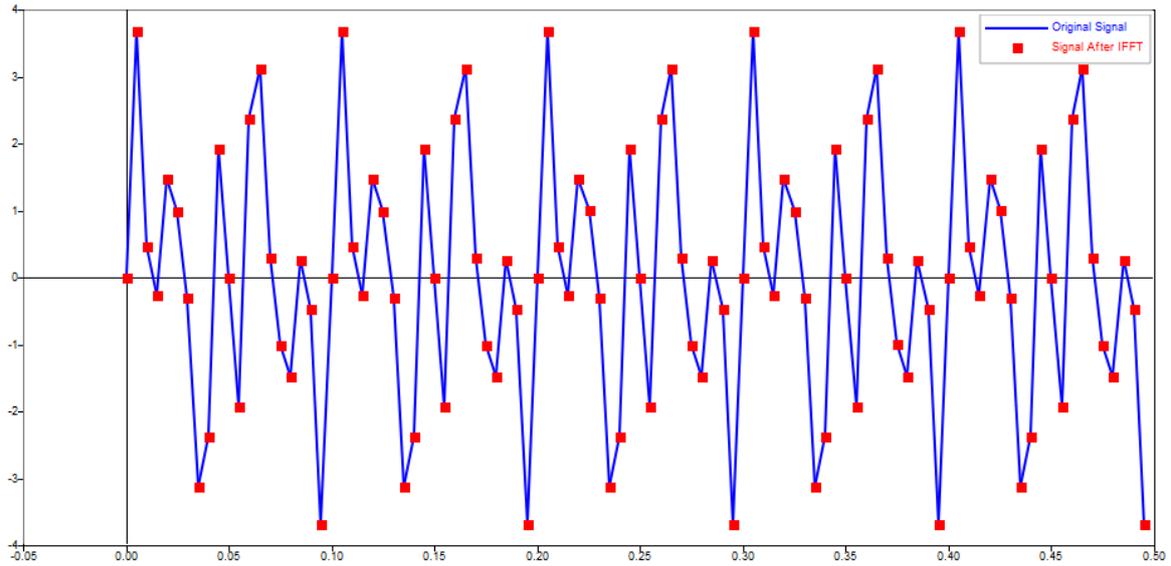


Figure 76. IFFT plot of the signal in overlay with the original one

6.7. Leakage

Spectral leakage is a phenomenon which causes the frequency content of a bin to spread into adjacent frequency bins. This is an unwanted effect since it corrupts the frequency analysis of a signal.

Consider a cosine wave at 12.5 Hz and measured for 1 s :

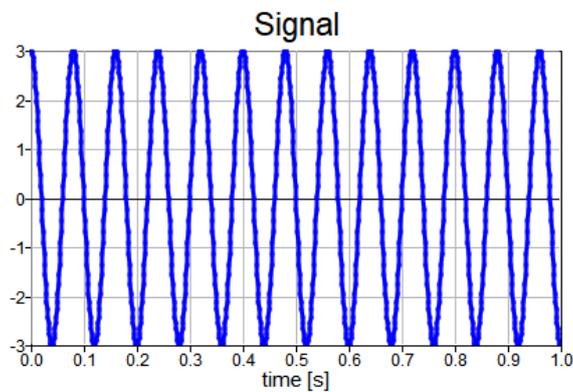


Figure 77. 12.5 Hz cosine wave

An untrained eye, may expect the FFT result to appear as:

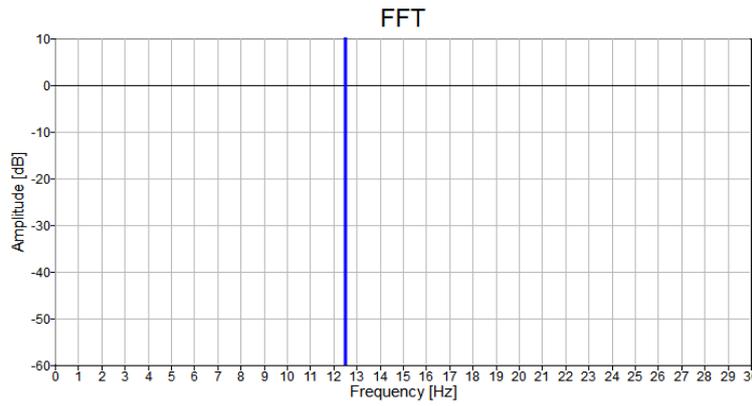


Figure 78. Expected FFT result of a 12.5 Hz cosine wave

Whereas, the FFT algorithm returns the single-sided DFT drawn in red:

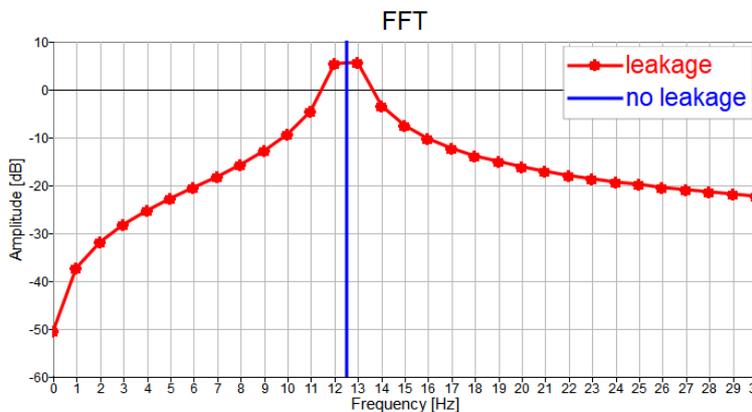
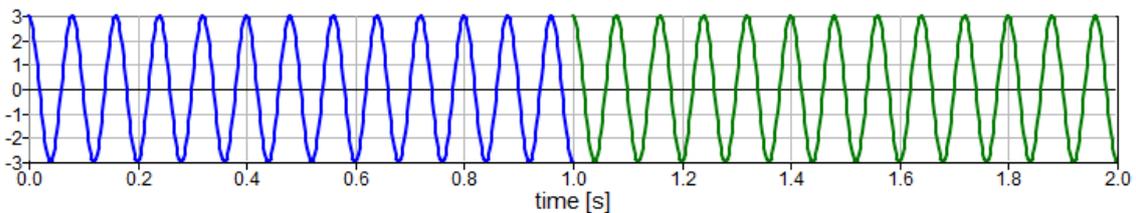


Figure 79. Expected (no leakage) and actual (with leakage) output of FFT of a 12.5 Hz cosine wave

It can be seen that the frequency content of the signal, represented by the blue line, spreads into adjacent frequency bins due to leakage.

The reason why leakage is occurring can be explained by keeping in mind the math involved in the DFT. Considering the DFT as a change of basis (see: Annex 1), it means that the signal is represented as a linear combination of harmonic functions. Hence the signal is forced to be periodic over the acquisition time. Extending by periodicity the previous cosine wave:



The transition from one period to the other, i.e. from the blue curve to the green one, is sharp. And since the sharp discontinuity has to be represented – or better approximated – with continuous harmonic functions, it explains why leakage occurs.

These continuous harmonic functions are also characterized by frequencies that are integer multiples of the frequency resolution. In this example, the frequency resolution is 1 Hz while the frequency of the cosine wave is 12.5 Hz. It follows that in the DFT basis there is not the most appropriate frequency bin to correctly represent the frequency content of the signal, hence leakage occurs.

It can be readily proven that, for a signal made up of a single frequency, non-periodicity over the acquisition time leads to the lack of the proper frequency bin in the DFT, and vice versa.

To see what happens with a signal containing more frequency components:

```

Fs = 500; %Hz
t = 0:1/Fs:1;
f1 = 5.3; %Hz
f2 = 10; %Hz
s1 = 2*sin(2*pi*f1*t);
s2 = 4*sin(2*pi*f2*t);
s = s1 + s2;

```

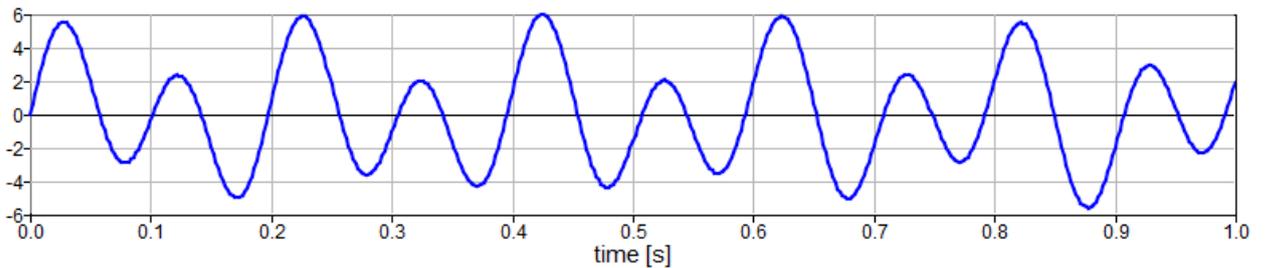


Figure 80. Signal composed by multiple sine waves

When extended by periodicity, gives the following signal:

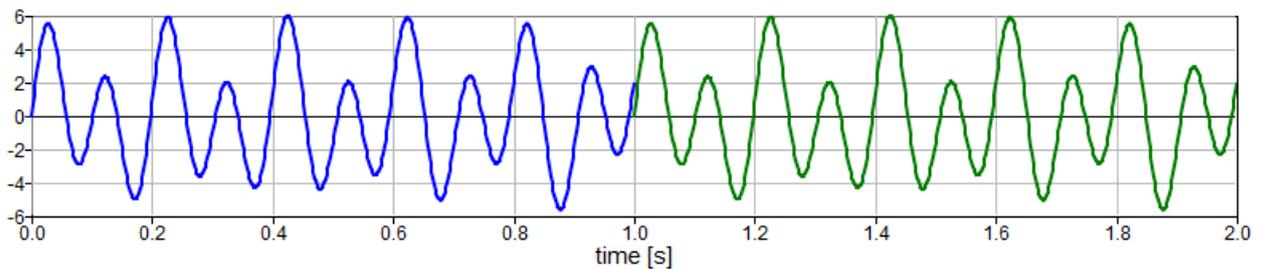


Figure 81. Signal composed by multiple sine waves and extended by periodicity

The signal is forced to be periodic, so a sharp discontinuity occurs in this case as well, thus some leakage will occur.

Computing the FFT:

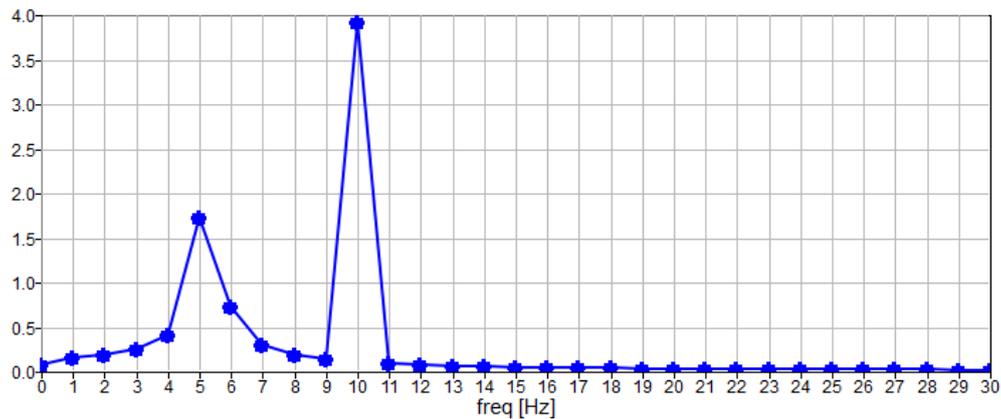


Figure 82. FFT of signal of Figure 81

It can be seen that the leakage has a greater effect on the lower frequency of the signal, at 5.3 Hz. Since the frequency resolution is 1 Hz, there is not the proper frequency bin to represent that lower frequency. However, there is a frequency bin at 10 Hz, which explains why the second peak is less affected by leakage.

At this point, it should be clear what is causing leakage. But, how can it be reduced?

6.8. Windowing

Windowing is a signal processing technique used to reduce the undesired effect of leakage. This technique consists of multiplying the time domain signal by a window function, which is generally:

- Zero-valued outside of some chosen interval, which is the acquisition time.
- Normally symmetric around the middle, where the maximum is located.
- Usually tapering off from the middle.

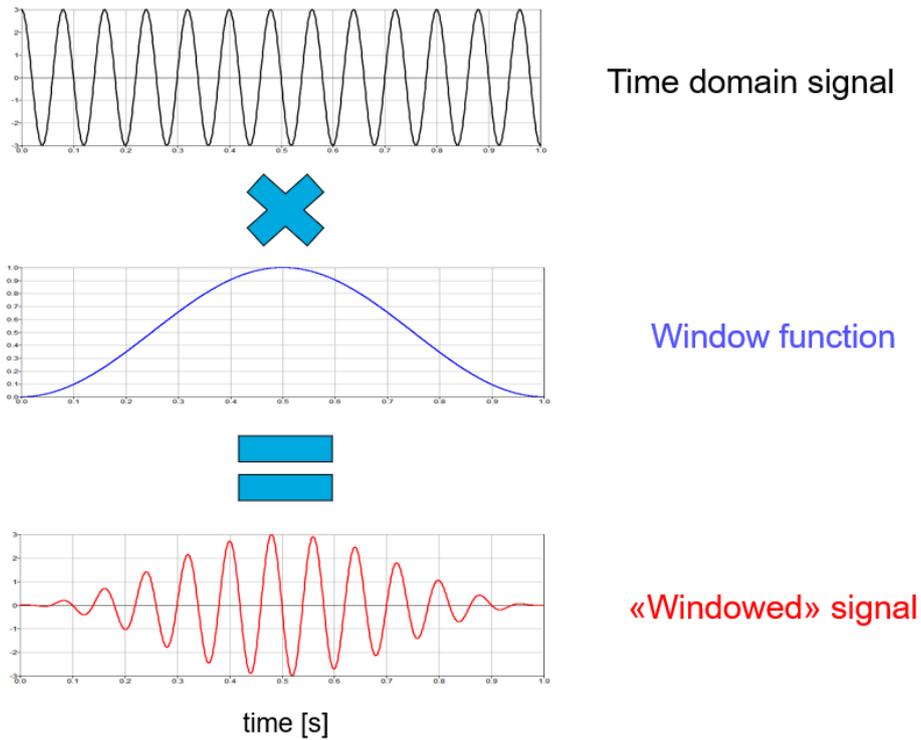


Figure 83. Representation of effect of windowing of a signal

Hence, the aim of the windowing is to modify the original signal to obtain a periodic signal and smooth out the sharp discontinuities, which can arise from the finiteness of the acquisition time. In this way the leakage is reduced.

Consider a signal windowed with a Hann window:

```

Fs = 1000;
t = 0:1/Fs:1;
f = 12.5; %Hz
s = 3*cos(2*pi*f*t);
w = hann(length(s), 'periodic');
s_win = s .* w;

```

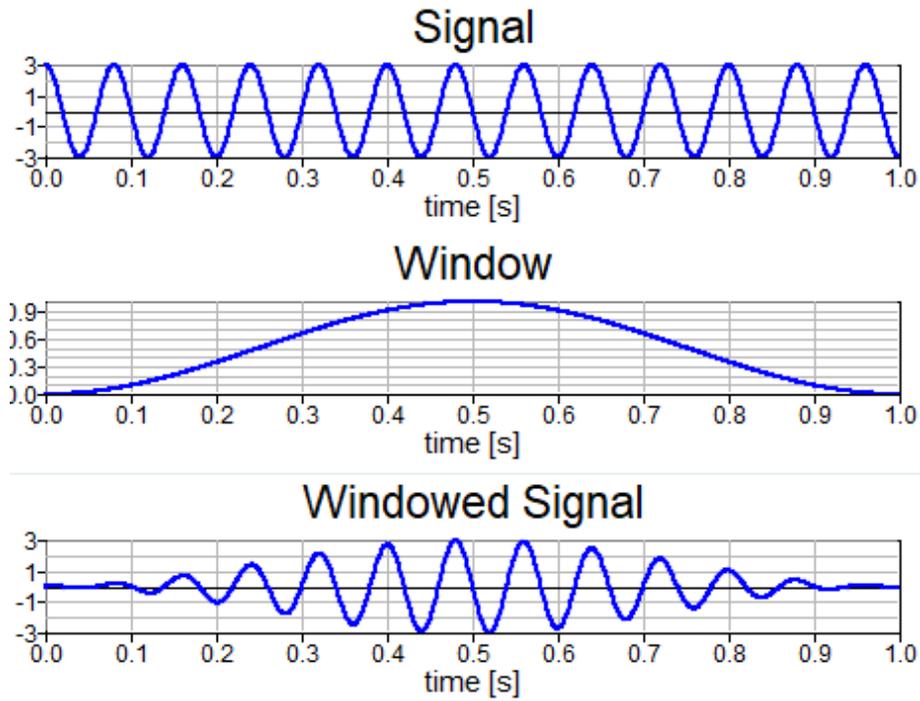


Figure 84. Original and windowed signal using Hann window

By periodic extension, the window has smoothed out the sharp discontinuity that arose due to the finiteness of the acquisition time.

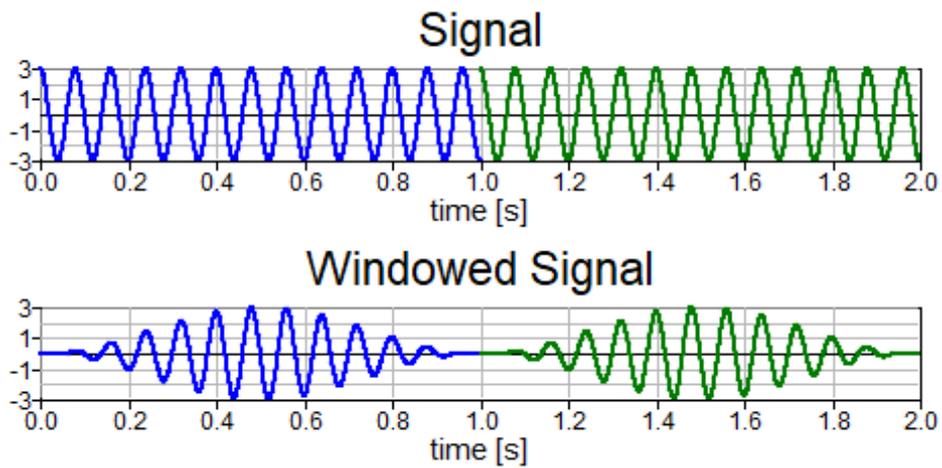


Figure 85. Smoothing of discontinuity due to windowing

Computing the FFT:

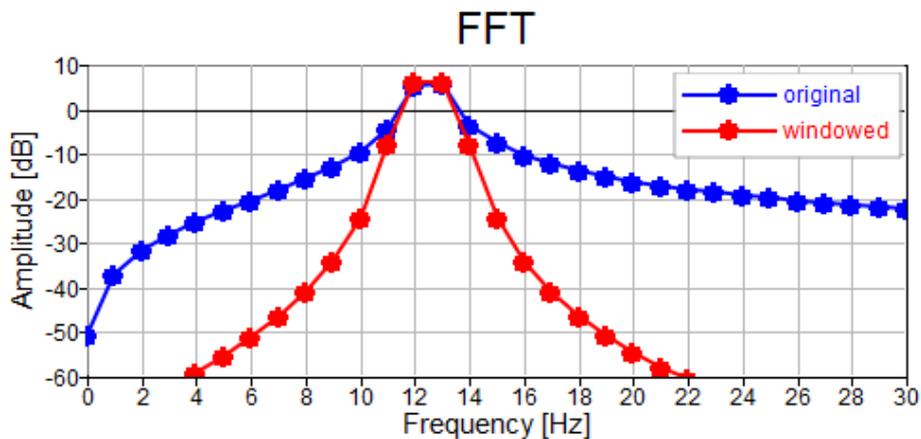


Figure 86. FFT of original and windowed signals

The spreading of the frequency content into adjacent bins has been reduced.

In literature, there are many window functions cited of which some are already available in Altair Compose. But are there any criteria to select the proper window?

The frequency response of the window can be used to choose the most suitable one.

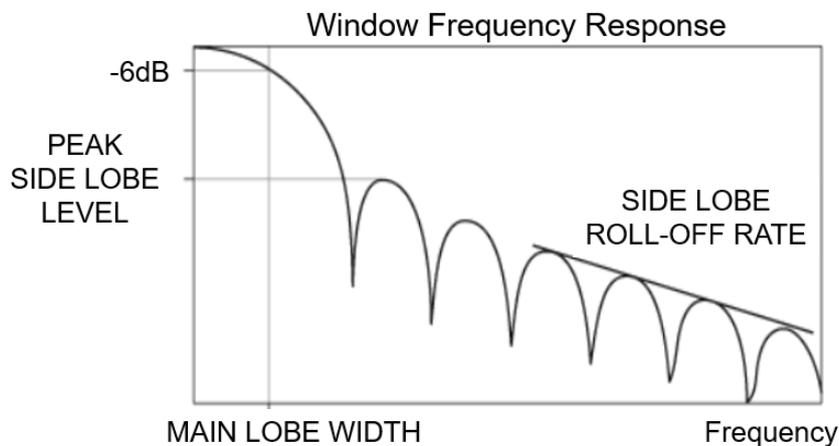


Figure 87. Window frequency response [4]

The frequency response is characterized by parameters such as:

- main lobe width,
- side lobe roll-off rate,
- peak side lobe level.

The main lobe width regulates the spectral resolution and the amplitude accuracy. A narrower main lobe means better spectral resolution. A wider one means a smaller scalloping loss (which are not discussed here).

The side lobe roll-off rate and the peak side lobe level regulate the leakage. A small peak side lobe level and a high side lobe roll-off rate mean a greater leakage reduction.

The choice of window is usually driven by a trade-off between these parameters.

Before moving on with other frequency analysis, it is useful to also consider:

- By not applying any window, it is equivalent to using a rectangular window.
- With windowing, the effect of leakage related to the sharp discontinuities is reduced but the frequency resolution is not improved.
- Using windows, the signal is changed in the time domain, therefore the DFT also changes. It can be proven that the effect of the modifications introduced is smaller with respect to the leakage of the unwindowed signal. There are also correction factors that can be used; explained further in Chapter 7
- When using built-in windows in Altair Compose, it is necessary to specify the second input argument as 'periodic'

6.9. Spectrogram

A spectrogram is a visual representation of the spectrum of frequencies of a signal as it varies over time. Since it condenses in a single representation information about time and frequency, it is extremely powerful in the analysis of non-stationary signals. The typical fields of application are speech processing, seismology, and vibration analysis.

A simple example of a non-stationary signal is a chirp signal in which its frequency changes linearly over time:

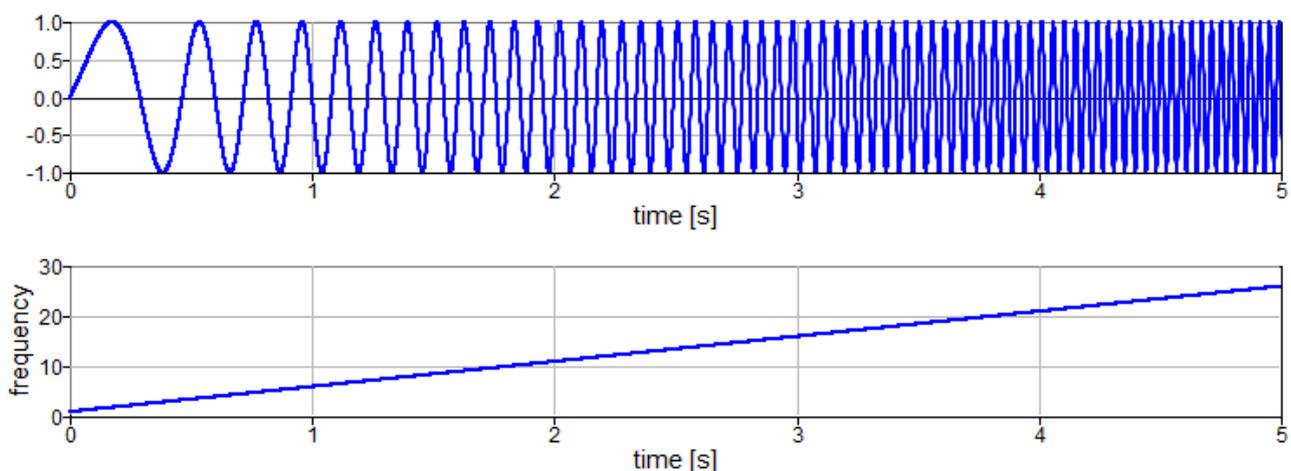


Figure 88. Chirp signal

Its spectrogram, shown in Figure 85, shows the linear increase of frequency over time is:

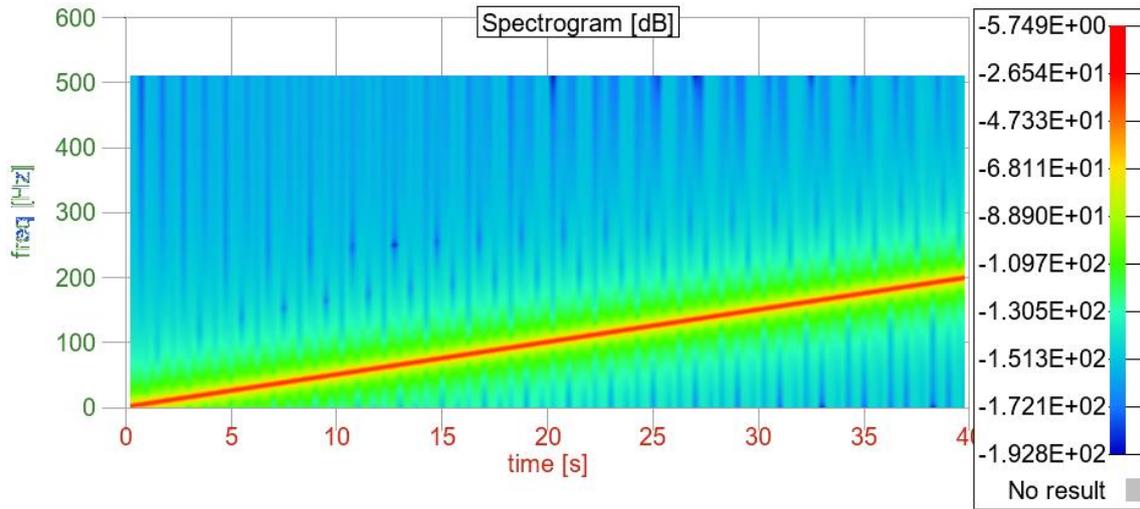


Figure 89. Spectrogram of signal of Chirp (from Figure 88)

For a mechanical system excited by a rotating unbalanced mass, its spectrum shows angular speed changes linearly with time; see Figure 87.

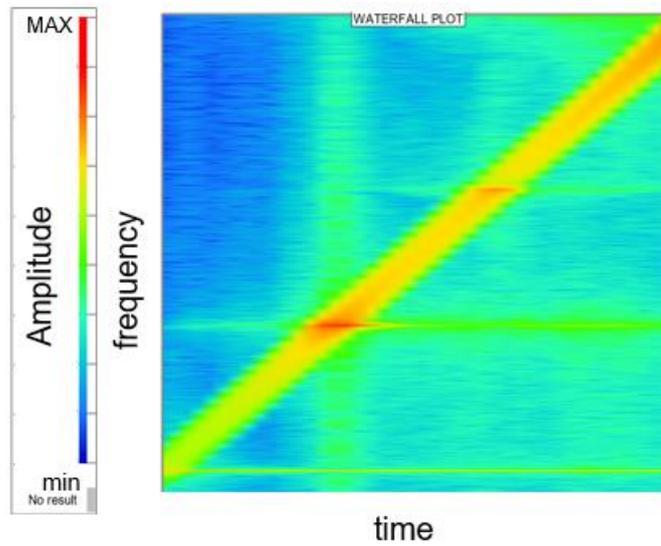


Figure 90. Spectrogram of a mechanical system excited by a rotating unbalanced mass

In this case, it also exhibits the dependency of the exciting force on frequency, along with the resonance frequencies of the mechanical system, which appear as short horizontal lines in Figure 88:

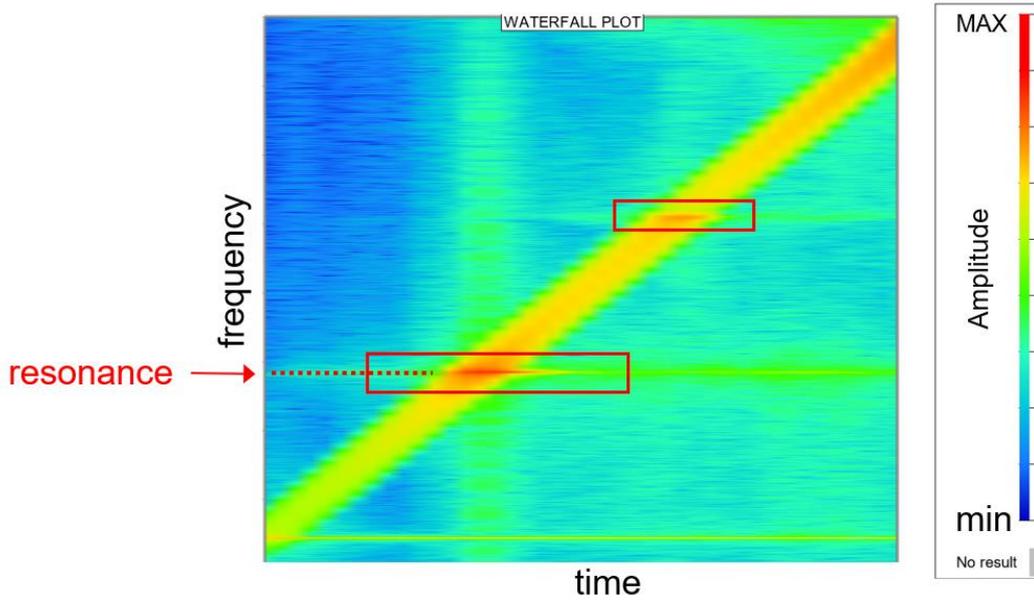
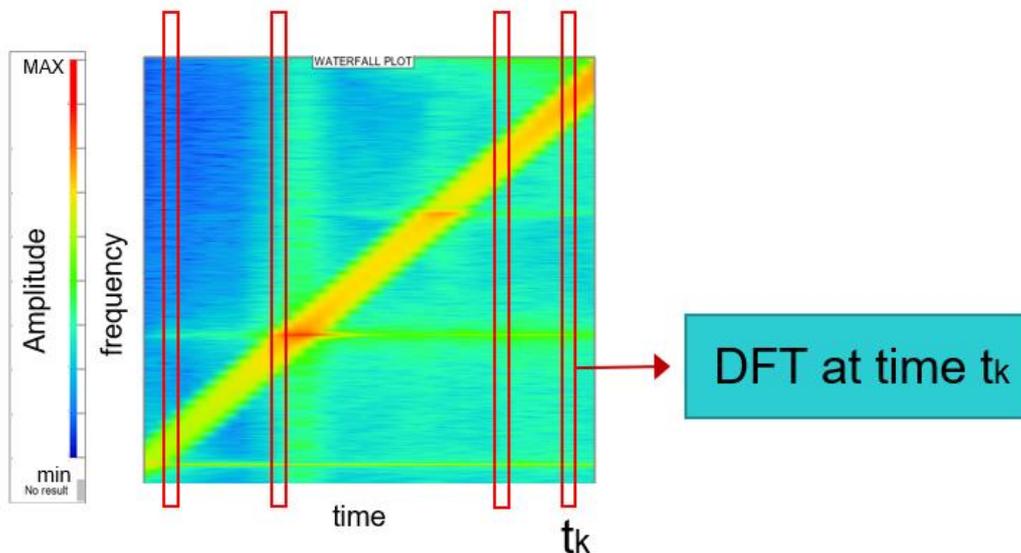


Figure 91. Resonance Frequencies

The spectrogram can be understood as sequential DFT calculations horizontally stacked along the x -axis (time). The time-domain signal is divided into short segments of equal length. Then the DFT is applied to each one to provide a local overview of the content across the frequency, as it changes over time.



This is how it is obtained by leveraging the short-time Fourier transform (STFT), given that:

- The frequency vector f , comprising the frequency bins used for the DFT.
- The time vector t , comprising the time instants at which the DFT is computed., and
- The DFTs $Amp1$, a matrix containing the DFT of each time instant.

The spectrogram can be drawn in Altair Compose using this script:

```
figure ();
[f, t_mean] = meshgrid(f, t);
h = surf(t, f, Ampl);
set(h, 'meshlines', 'off');
view(2);
colorbar;
ylabel('freq [Hz]');
xlabel('time [s]');
title('Spectrogram');
```

6.10. Short-time Fourier Transform (STFT)

The short-time Fourier transform (STFT) is a Fourier-related transform used to determine the frequency of a signal as it changes over time. In practice, a long signal is divided into short segments, hence its name “short-time”. Then the Fourier transform is applied to each segment:

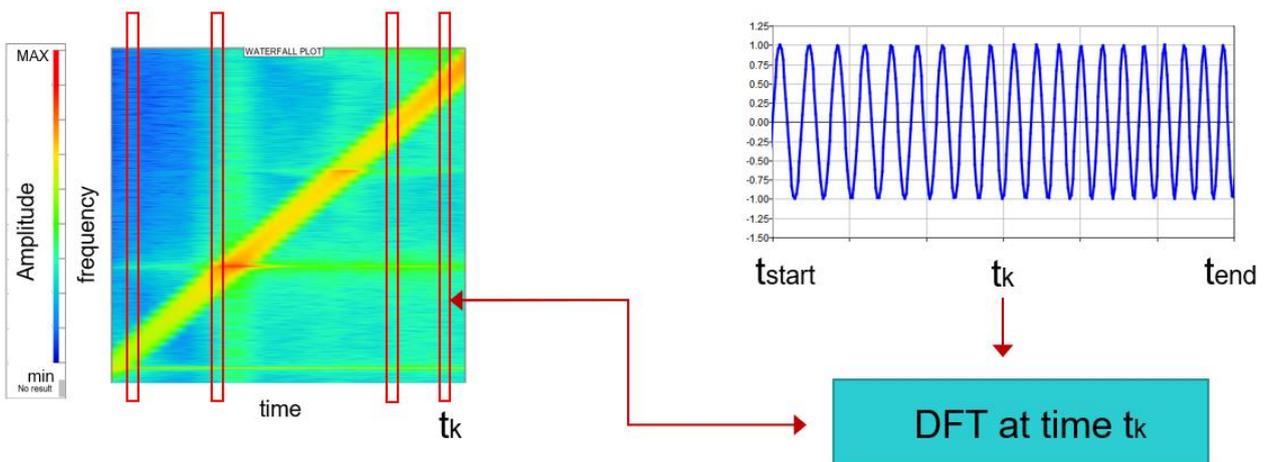


Figure 92. STFT representation

Defining the algorithm is straightforward and gives the basis to compute spectrograms:

- 1) The signal is divided into short overlapping segments.
- 2) Windowing is applied on each segment to reduce leakage.
- 3) The DFT is computed for each segment leveraging the FFT algorithm.
- 4) The DFTs are stacked all together to build the spectrogram.

To implement the algorithm some choices must be made, such as: length of the segments, amount of overlap, window function.

The choice of the window function is determined by the type of application, which requires some window properties; see 6.8. Whereas the length of the segments and the amount of overlap are explained further here.

Selecting the length of the segment should consider:

- Not too long, so that the signal can be considered quasi-stationary along the segment.
- Not too short, so that the frequency resolution (y -axis resolution) is good enough.

In fact, the time resolution (x -axis) and the frequency resolution (y -axis) depend on the segment length. It becomes a trade-off between longer segments, meaning better frequency resolution and poorer time resolution and shorter segments giving better time resolution and poorer frequency resolution; as explained in 2.6.

An overlap is necessary because windowing is applied to each segment. The overlap is used to avoid those portions of the signal where window coefficients are small and get neglected.

Again, there is a trade-off to decide how much overlap must be used. By increasing the overlap ratio, the number of segments increase, hence the time resolution increases while the frequency resolution is unchanged. If the overlap is too high, the hypothesis of uncorrelation between successive segments is lost.

If the time domain signal is to be rebuilt from its spectrogram (an operation similar to an Inverse Fourier Transform), the amount of overlap is determined by the type of window deployed.

An Altair Compose script which implements the short-time Fourier transform (STFT) algorithm is:

```

function [Ampl, f, t_mean] = ShortTimeFourierTransform (t,...
s, SegmentLength, OverlapRatio, WindowType)

    [segments, t_segments, Fr] =...
    GetOverlappingSegments (t, s, SegmentLength, OverlapRatio);
    t_mean = GetTimeMeanValues (t_segments);
    segments = Windowing (segments, WindowType);
    [Ampl, f] = FFTeval (segments, Fr);
    MyPlot (t_mean, f, Ampl);

end

```

```

%% ===== UTILITY FUNCTIONS =====

function [segments, t_segments, Fr] =
GetOverlappingSegments(t,... s,SegmentLength,OverlapRatio)

    n = floor(length(s)/SegmentLength);
    overlap = floor(SegmentLength * OverlapRatio);
                %number of overlapping points

    nOverlapSegments = floor((n-OverlapRatio) / ...
        (1-OverlapRatio)); %number of overlapping segments

    segments = zeros(nOverlapSegments, SegmentLength);
    t_segments = zeros(nOverlapSegments, 2);

    overlap = floor(SegmentLength * OverlapRatio);
                %number of overlapping points

    idx_start = 1;
    idx_end = SegmentLength;

    Fr = 1/(t(idx_end) - t(idx_start)); %Frequency resolution

    for ii = 1:nOverlapSegments
        segments(ii,:) = s(idx_start:idx_end);
        t_segments(ii,:) = [t(idx_start), t(idx_end)];
        ii = ii + 1;
        idx_start = idx_end - overlap + 1;
        idx_end = idx_start + (SegmentLength - 1);
    end

end

function t_mean = GetTimeMeanValues(t_segments)

    t_mean = mean(t_segments,2);

end

```

```

function s = Windowing(s,WindowType)

    switch WindowType

    case 'hann'

        windowLength = size(s,2);

        window = hann(windowLength, 'periodic')';

    end

    s = s .* repmat(window, size(s,1),1);

end

function [Ampl, f] = FFTeval(s, Fr)

    DFT = fft(s(:,1:end-1), [], 2);

    P2 = abs(DFT) ./ size(DFT, 2);

    P1 = P2(:, 1:ceil(length(P2)/2)); %folding

    P1(:, 2:end-1) = P1(:, 2:end-1)*2; %single-sided FFT

    Ampl = mag2db(P1); %[dB]

    f = 0:Fr:(length(Ampl)-1)*Fr;

end

function MyPlot(t_mean, f, Ampl)

    figure();

    [f, t_mean] = meshgrid(f, t_mean);

    h = surf(t_mean, f, Ampl);

    set(h, 'meshlines', 'off');

    view(2);

    colorbar;

    ylabel('freq [Hz]');

    xlabel('time [s]');

    title('Spectrogram [dB]');

end

```

Exercises

- 1) Find the Fourier series of a sawtooth wave defined with the interval $[-1,1]$ and having period of 1, considering 5 harmonics. What happens when 25 harmonics are considered?
- 2) Why can Fourier series only represent periodic signals?

- 3) The ideal square wave only has components of odd-integer harmonic frequencies, which means that the coefficients a_0 and a_n are equal to zero. Therefore the square wave function may be written as:

$$f(x) = \frac{4}{\pi} \sum_{n=1}^N \frac{1}{n} \sin\left(\frac{\pi n x}{L}\right)$$

Compute its Fourier series for $N = 5$.

- 4) Verify if the following sets of vectors are a basis for \mathbb{R}^3 :

$$4.1) \left\{ \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix}, \begin{bmatrix} -2 \\ 1 \\ 4 \end{bmatrix} \right\}$$

$$4.2) \left\{ \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}, \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} \right\}$$

Hint: Different strategies may be used, such as `rank`, `linsolve` and `det` functions.

- 5) Create 3 signals with unitary amplitude, 200 points and natural frequencies of 30 Hz, 45 Hz and 60 Hz, respectively. Sum them, then compute and plot the Fast Fourier Transform of the resultant signal.
- 6) Repeat exercise 5 using a single-sided spectrum instead of a double-sided one.
- 7) What happens to the spectrum in exercise 6 if the sampling frequency is modified to 100 Hz?
- 8) Plot the spectrogram of a chirp signal leveraging the short-time Fourier transform (STFT) algorithm; given in 6.10.
- 9) Implement additional window types in the STFT algorithm given in 6.10.
- 10) Create a sinusoidal signal with 32 samples and frequency of 15 Hz, compute its Fast Fourier Transform (FFT). Then, check its absolute normalized output with a stem plot and the respective frequency bins. Although the signal has only one predominant frequency (15 Hz), why does the plot have 2 important peaks?
- 11) What can be done to reduce the leakage identified in exercise 10?

7. Power Spectral Density

7.1. Definition

As its name implies, a power spectral density (PSD) describes how the power of a signal is distributed in each frequency band. Usually it is used to characterize non-deterministic processes, such as broadband noise, artificial road profile, among others.

7.2. Parseval Theorem

The Parseval Theorem states that the sum – or integral – of the square of a function is equal to the sum of the square of its Fourier transform. This means that the total energy of a signal can be computed equivalently by:

1. Summing power-per-sample across time
2. Summing spectral power across frequency

Translating into mathematical equations for digital signals gives:

$$\sum_{n=0}^{N-1} |x(n)|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X(k)|^2$$

Applying this theorem provides the mathematical formula to compute the power spectral density.

7.3. Windows Correction Factor

Parseval Theorem can be applied to compute the correction factors for signals when windowing is deployed.

When windowing is applied (see: 6.8), the signal is changed in time domain. Hence, its energy, and therefore power change too. Correction factors can be applied so that signal energy stays constant.

Given the vector of window coefficients $w(n)$ and the signal $x(n)$, the correction factor CF can be computed as:

$$\sum_{n=0}^{N-1} |x(n)|^2 = \sum_{n=0}^{N-1} |CFx(n)w(n)|^2$$

$$CF = \sqrt{\frac{\sum_{n=0}^{N-1} |x(n)|^2}{\sum_{n=0}^{N-1} |CFx(n)w(n)|^2}}$$

7.4. PSD from DFT

The right-hand side of the equation (see: 7.3) shows that the power associated to each frequency component, known as spectral power, is given by:

$$\frac{1}{N}|X(k)|^2.$$

Dividing the spectral power by the length of the frequency range ($[-\frac{F_s}{2}, \frac{F_s}{2}]$), the power spectral density (SPD) is obtained:

$$PSD(k) = \frac{1}{F_s N}|X(k)|^2$$

This can be folded into a single-sided PSD; as done in 6.5 for the DFT.

Sometimes, only the single-sided and normalized DFT is provided. In this case, skipping some mathematical steps, the single-sided power spectral density can be computed by:

$$PSD(k) = \frac{1}{2F_r}|X^*(k)|^2 \text{ if } k > 0, \quad PSD(k) = \frac{1}{F_r}|X^*(k)|^2 \text{ if } k = 0$$

Where $X^*(k)$ are the coefficients of the single-sided and normalized DFT.

Consider a simple sine wave:

```

Fs = 1000; %[Hz]
t = 0:1/Fs:1;
f = 10;
s0 = 5;
s = s0 + 4*sin(2*pi*f*t);
figure();
plot(t,s);
grid on;
xlabel('time [s]');
```

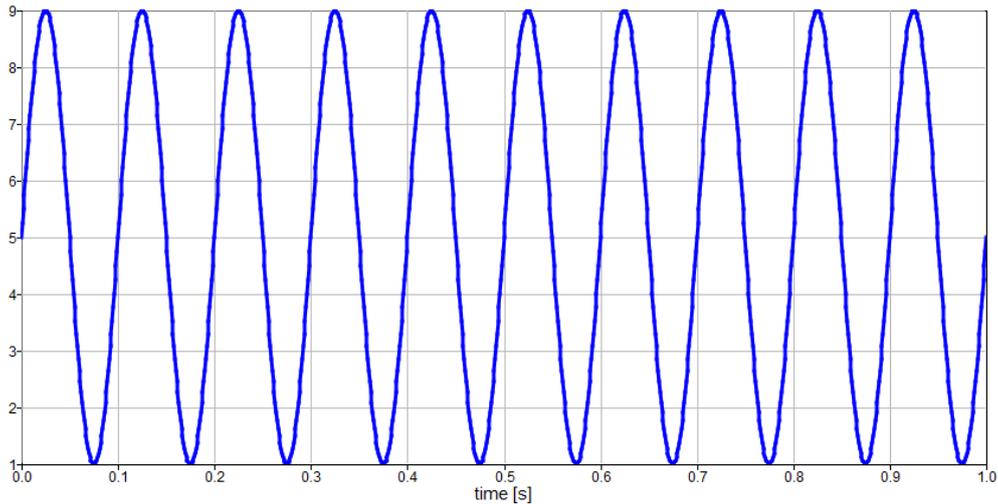


Figure 93. Sine wave

The power spectral density (PSD) can be computed applying the first formula:

```
DFT = fft(s(1:end-1));
DFT = abs(DFT);
N = length(DFT);
PSD = 1/(Fs*N) * DFT.^2;
```

At this point, a simple check can be performed to compute the total energy of the signal in both time and frequency domains:

```
sum(PSD*Fs)

sum(s(1:end-1).^2)
```

Both correctly return:

```
ans = 33000
```

Finally, the double-sided PSD can be folded into a single sided one:

```
PSD = PSD(1:ceil(length(PSD)/2));
PSD(2:end-1) = PSD(2:end-1)*2; %single-sided PSD
```

```
Fr = 1/t(end);
f = 0:Fr:(length(PSD)-1)*Fr;
```

```
figure();
plot(f, PSD, 'bo-');
hold on; grid on;
xlabel('freq [Hz]');
ylabel('PSD');
xlim([0, 30]);
```

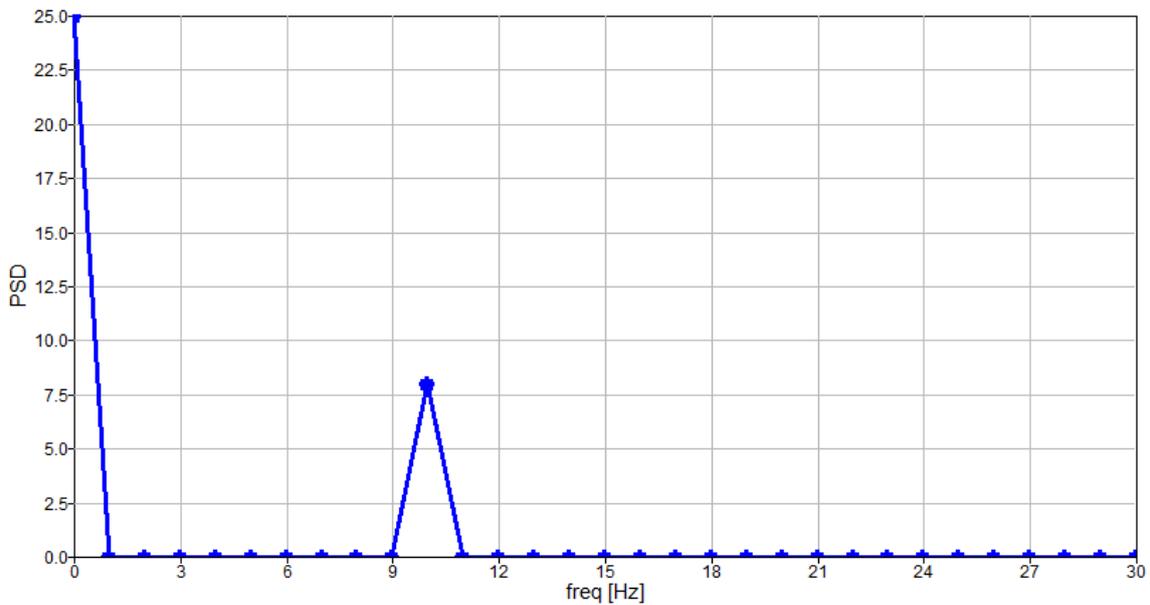


Figure 94. Single-sided PSD

As expected, there are only two frequency components with peaks above zero:

- 0 Hz, since the mean value of the signal is different from 0.
- 10 Hz, which is the frequency of the sine wave.

7.5. Welch's Method

The power spectral density is used to represent random and ergodic processes; see 7.1. A typical algorithm to compute the power spectral density for these kinds of processes involves the Welch's method.

Often only one (or a few) realization of the random process is available. However, to obtain a good estimate of the PSD of a random process, more realizations are needed. Here, the gap is filled by the Welch's method whose pseudo-code is:

1. Split the signal into overlapping segments; where each segment can be considered as a realization of the random process.
2. Apply windows on each segment to reduce leakage.
3. Compute the DFT of each segment; applying for example the FFT algorithm.
4. Compute the PSD of each segment; using one of the formulas defined in 7.4.
5. Average all the PSDs together.

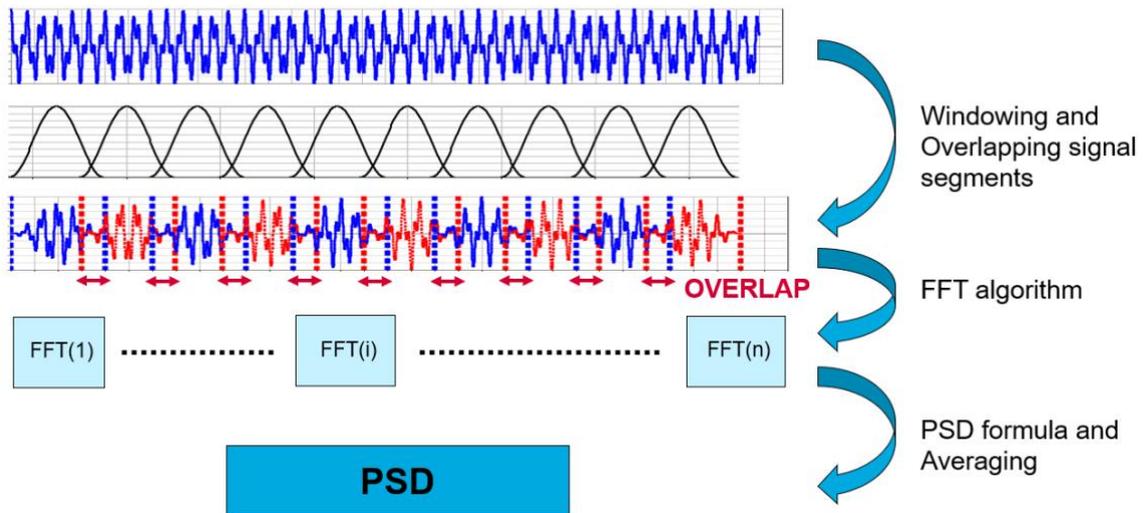


Figure 95. Realization scheme for PSD

Each segment should be overlapped because every segment is being windowed and the portion of the signal where the windows coefficients are small are neglected; as explained in 6.10. It means that:

1. The signal is not used efficiently for the PSD computation.
2. Information contained in certain locations of the signal might be lost, i.e. at the beginning and at the end of each segment.

Hence the importance of introducing an overlap between segments.

A disadvantage of this algorithm is the frequency resolution, determined by the time length of the signal (in this case the time length of the segments), is reduced.

In Altair Compose, the Welch's method is obtained with the `pwelch` function.

Now, consider a signal with noise:

```

Fs = 1000; %Hz
t = 0:1/Fs:10;
s0 = 5;
f1 = 10;
s1 = 4*sin(2*pi*f1*t);
f2 = 5;
s2 = 8*sin(2*pi*f2*t);
f3 = 20;
s3 = 3*sin(2*pi*f3*t);
n = 5*rand(size(t));
s = s0 + s1 + s2 + s3 + n;

```

```
figure();
plot(t,s);
grid on;
xlabel('time [s]');
```

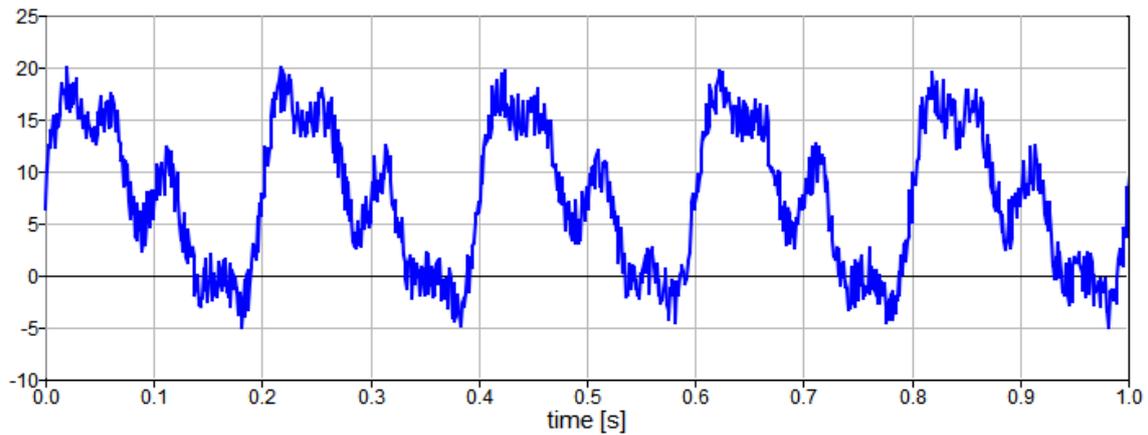


Figure 96. A portion of the signal with noise

The single-sided PSD can be computed from the DFT as:

```
DFT = fft(s(1:end-1));
DFT = abs(DFT);

N = length(DFT);
PSD = 1/(Fs*N) * DFT.^2; %double-sided PSD
PSD = PSD(1:ceil(length(PSD)/2)); %folding
PSD(2:end-1) = PSD(2:end-1)*2; %single-sided PSD
Fr = 1/t(end);
f = 0:Fr:(length(PSD)-1)*Fr;
```

Or applying the Welch's method:

```
WindowLength = (length(s)-1)/10; %% 10 segments
window = ones(1,WindowLength); %% rectangular window
Overlap = round(WindowLength/2); %% 50%
nfft = WindowLength;
range = 'onesided'; %% single-sided PSD
[Pxx,frq]=pwelch(s(1:end-1),window,Overlap,nfft,Fs,range);
```

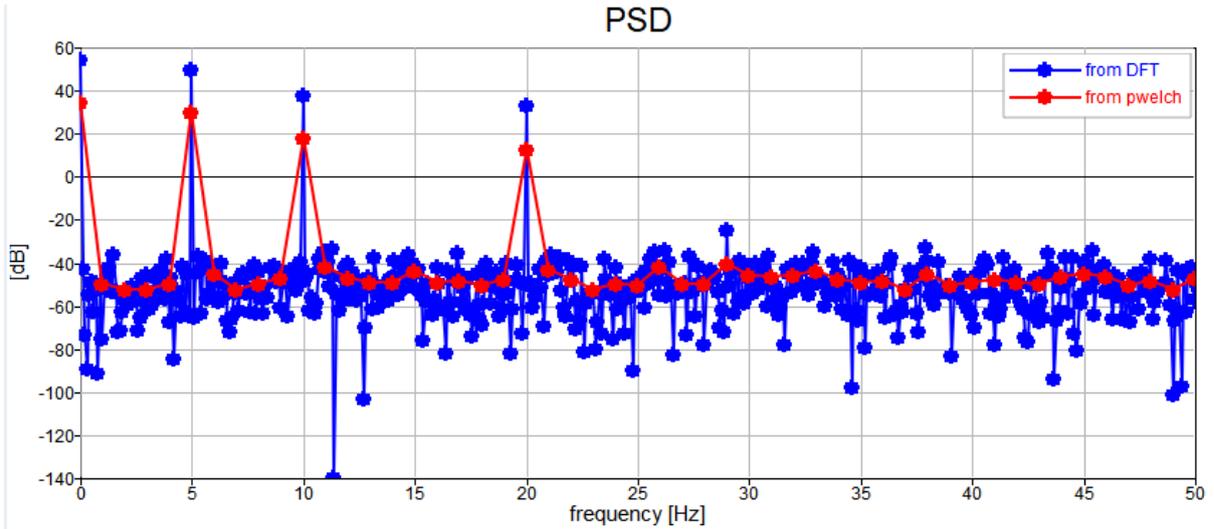


Figure 97. PSD calculation using 2 different approaches

To understand the reason for these differences, the noise from the signal is removed and the PSDs are re-computed and re-plotted using a regular scale:

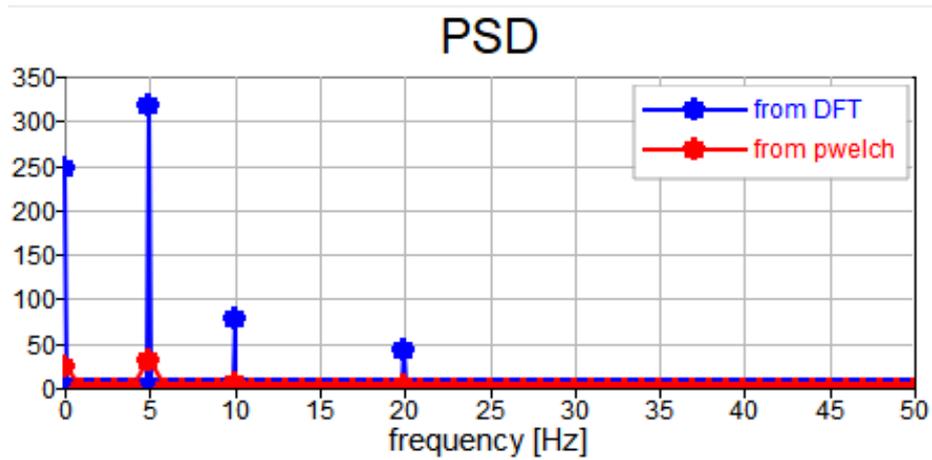


Figure 98. Denoised signal and respective PSD using 2 different approaches

Since they are still different and cannot be attributed to noise, it can be explained in two ways:

Compute the total energy of the signal for both PSDs:

```
sum (PSD*Fs)
```

This returns:

```
ans = 695000, for the PSD computed from the DFT
```

```
ans = 69500, for the PSD computed using pwelch
```

This is correct because the original signal was divided into 10 segments. Remembering how energy was defined (see: 7.2), then it is correct that the peaks do have different heights.

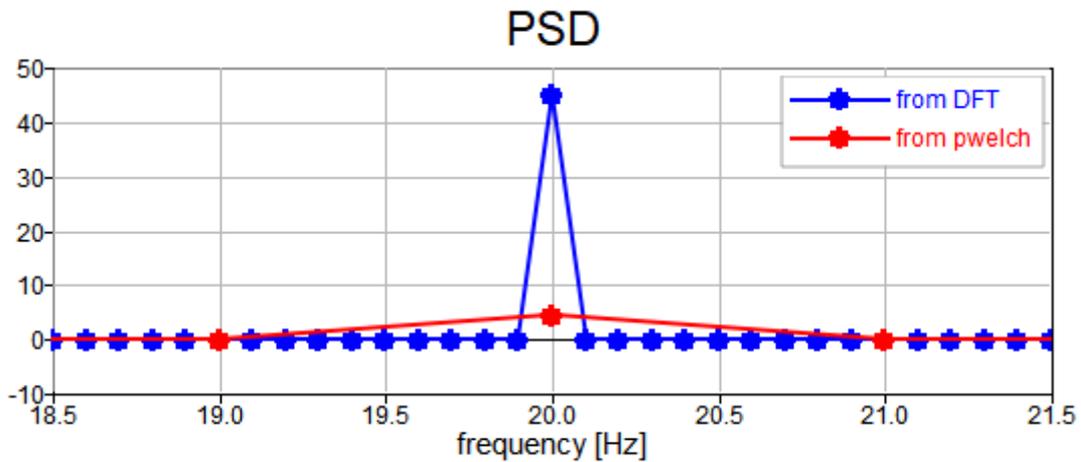


Figure 99. PSD amplitude using 2 different approaches

The spectral power, determined from the area under the peaks, can be verified to be the same. The peak becomes lower as the frequency resolution gets bigger. Hence the power is spread into a wider frequency range.

To show these properties, a rectangular window was selected (equivalent to not applying a window); the signal was divided into a number of segments, which does not introduce any leakage. Of course, this is not possible when the frequency content of a signal is not known a priori. In this case, a good practice is to divide the signal into 8 segments, use a Hamming window, and set a 50% overlap.

7.6. Creation of Time-History from PSD

In random analysis, such as random vibration analysis, the input is represented by a random process, which is characterized by a certain power spectral density curve. However, the solver or to the test bench needs an input signal in the time domain, i.e. a realization of that random process.

Here, the algorithm to move from the PSD curve to its time realization is discussed.

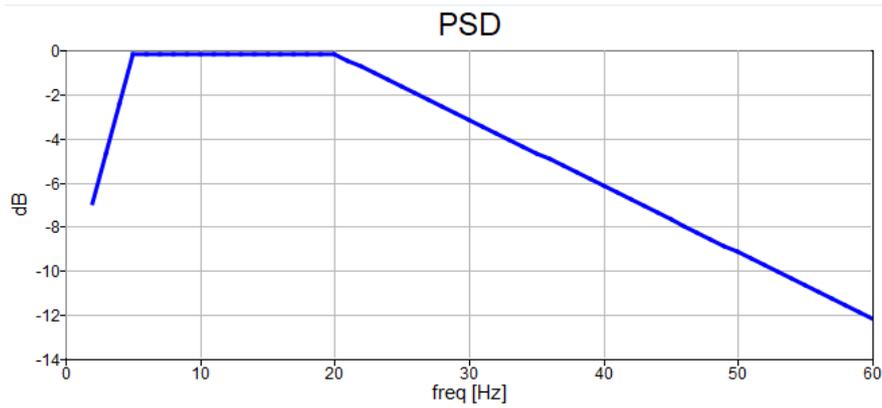


Figure 100. Example of single-sided PSD

The algorithm comprises 7 steps.

1. Define the time length of the time series data:

From the time length, the frequency resolution is computed as:

$$F_r = \frac{1}{t_{final}}$$

2. Define the frequency range $[f_0, f_{MAX}]$

Here, the most general case is tackled, which means that f_0 and f_{MAX} do not coincide with the starting and ending frequency of the PSD:

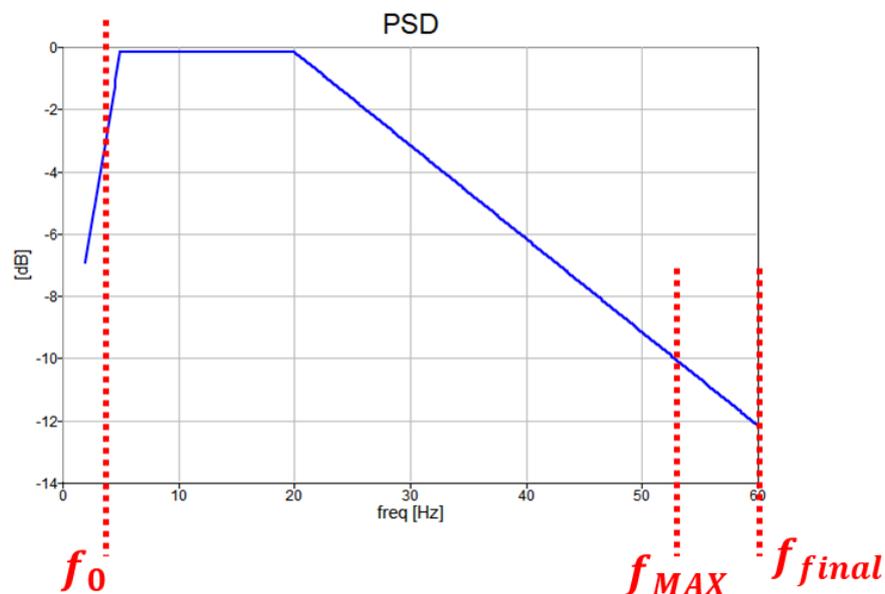


Figure 101. Initial and final frequencies different from those of the PSD

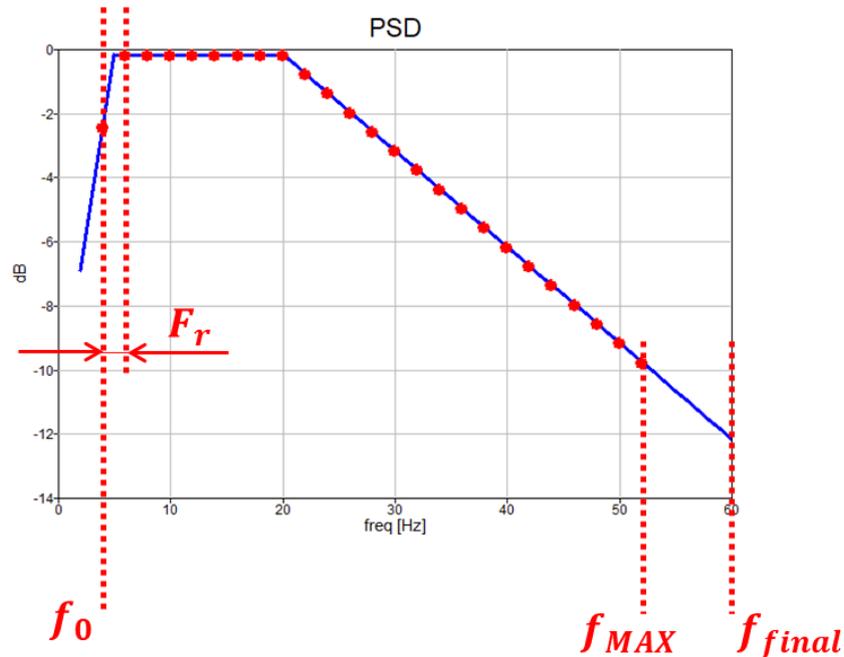
Applying the Nyquist Theorem (see: 2.5), the sampling frequency can be computed as:

$$F_s = 2f_{final}$$

Knowing the sampling frequency, the time array is defined as:

$$t = 0 : \frac{1}{F_s} : t_{final}$$

3. Sample the PSD curve in the selected range $[f_0, f_{MAX}]$, accordingly to F_r



The frequency vector is built as:

$$f = 0 : F_r : f_{final}$$

And outside the range $[f_0, f_{MAX}]$ the PSD values are set to 0.

4. Compute the single-sided and normalized spectrum ($X^*(k)$)

In order to move from the PSD to the DFT, reverse the formula defined in 7.4. Skipping the mathematical steps gives:

$$|X^*(k)| = \sqrt{2F_r PSD(k)} \quad \text{if } k > 0$$

$$|X^*(k)| = \sqrt{F_r PSD(k)} \quad \text{if } k = 0$$

At this point, the amplitude of the single-sided DFT is known but the information about the phase, which is necessary to perform the IFFT algorithm, is not contained in the PSD. This leads to the next step.

5. Generate random phase $\varphi^*(k)$

At each amplitude value $X^*(k)$, associate a random phase $\varphi^*(k)$. This is why the PSD is representative of the random process. From the same PSD curve, infinite realizations of the same process can be generated.

6. Retrieve the double-sided DFT coefficients (real/imag format)

The IFFT algorithm typically requires the coefficients of the double-sided DFT in the Real/Imaginary representation. In this step, the single sided spectrum is unfolded, the Euler formula is applied, and some scaling is added.

7. Compute the inverse DFT to generate the signal in time domain

$$signal(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{j2\pi kn/N}$$

The IFFT algorithm is applied.

Figure 99 shows one realization of the random process described by the PSD.

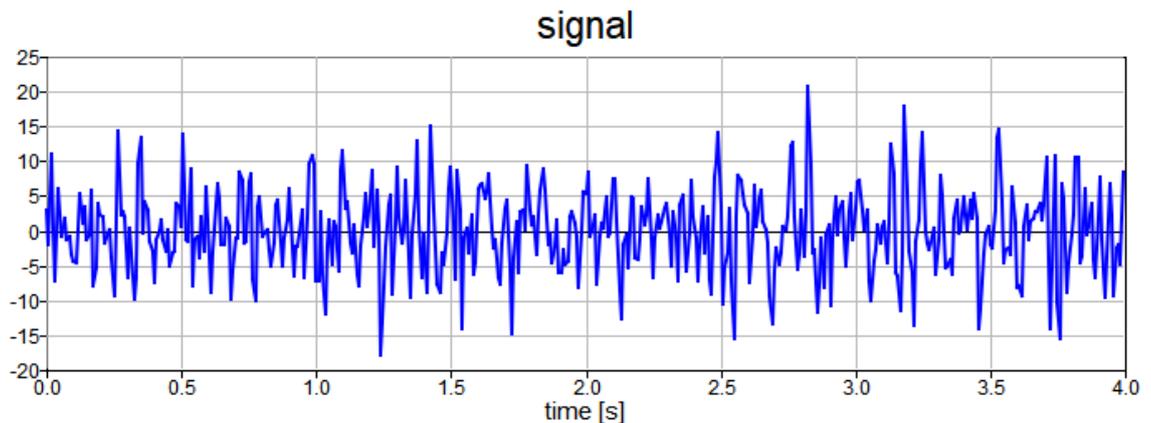


Figure 102- Time signal resulting from the PSD

Exercises

- 1) Can `pwelch` algorithm return the same PSD computed from the DFT?
- 2) Why is the `pwelch` algorithm used?
- 3) Develop an Altair Compose script to generate a realization of the random process described by the PSD stored in the “PSDcurve.mat” file.

8. Filtering

8.1. Definition

Filtering is a signal processing technique that aims to remove or modify certain components of a signal. Filtering can be achieved by electronic circuits or within software.

8.2. Filter Classification

8.2.1. Digital and Analog Filters

Filters are generally classified as:

- **Digital**, which are systems that perform mathematical operations on a sampled, discrete-time signal.

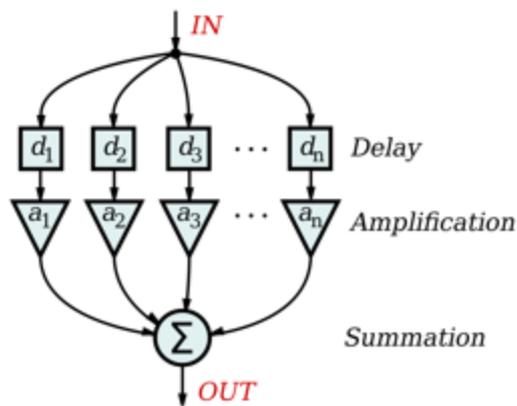


Figure 103. Digital filter representation

- **Analog**, comprising circuits made of analog components, such as resistors, capacitors, inductors. Working with continuous analog signals, they are less flexible than digital filters, but they are faster and have a wider dynamical range.

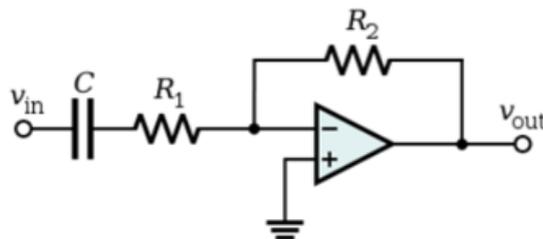


Figure 104. Analog filter representation [5]

Owing to increasing performance of micro-processors, digital filters are now replacing analog filters in many applications. However, there remain applications where analog filters are preferred because they outperform digital filters or where digital filters are not suitable at all.

8.2.2. Task Classification

Filters can be classified by the task they perform. Four main groups can be identified, based on which range of frequencies they block or allow to pass:

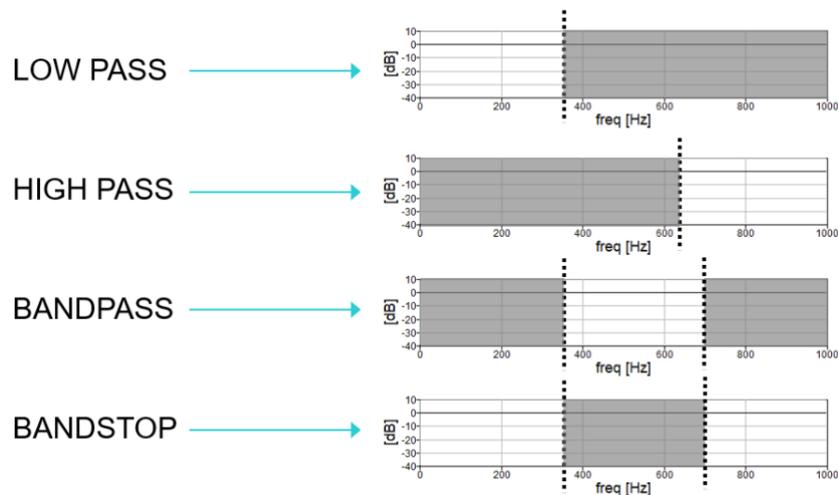


Figure 105. Tasks classification of filters

- **Low pass** filters block frequencies above the cut-off frequency; see also 8.4. It could be used to remove high frequency electric noise.
- **High pass** filters block frequencies below the cut-off frequency. It can be applied to remove unwanted sound below the audible range or remove sensor drift.
- **Bandpass** filters pass only those frequencies included in a certain range. For example, in transmitters it is used to limit the bandwidth of the output signal to that band allocated for the transmission.
- **Band-stop** filters stops those frequencies included in a certain range. It can be used to remove specific noise coming from other equipment.

8.3. Mathematical Formulation

Filters can be considered as systems with Inputs and Outputs:



Figure 106. Schematic representation of filtering process

Hence, the mathematical formulation used to describe filters is the same as dynamical systems but is not the same for digital and analog filters.

Digital filters are described in the discrete time domain using finite difference equations:

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

Which through the Z-transform can be turned into algebraic equations in the variable z , and rearranged into a transfer function:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^M b_k z^{-k}}{\sum_{k=0}^N a_k z^{-k}}, \quad a_0 = 1$$

From the transfer function, the frequency response function of the filter can be obtained:

$$z = e^{j\omega T} \rightarrow H(e^{j\omega T})$$

Analog filters are described in the continuous time domain using differential equations:

$$\begin{aligned} a_n \frac{dy^{(n)}(t)}{dt^n} + a_{n-1} \frac{dy^{(n-1)}(t)}{dt^{n-1}} + \dots + a_1 \dot{y}(t) + a_0 y(t) \\ = b_m \frac{dx^{(m)}(t)}{dt^m} + b_{m-1} \frac{dx^{(m-1)}(t)}{dt^{m-1}} + \dots + b_1 \dot{x}(t) + b_0 x(t) \end{aligned}$$

Which, similarly, can be turned into algebraic equations in the variable s by applying the Laplace Transform, and eventually can be rearranged into a transfer function:

$$H(s) = \frac{Y(s)}{X(s)} = \frac{b_m s^m + b_{m-1} s^{m-1} + \dots + b_1 s^1 + b_0}{a_n s^n + a_{n-1} s^{n-1} + \dots + a_1 s^1 + a_0}, \quad m < n$$

Here again, from the transfer function the frequency response function of the filter can be obtained:

$$s = j\omega \rightarrow H(j\omega)$$

The frequency response of the filter is used to define the filter specifications.

8.4. Filter Specifications

For brevity, only the frequency response of a lowpass filter will be described. Although this can be extended to the other filter types, such as highpass, bandstop, passband.

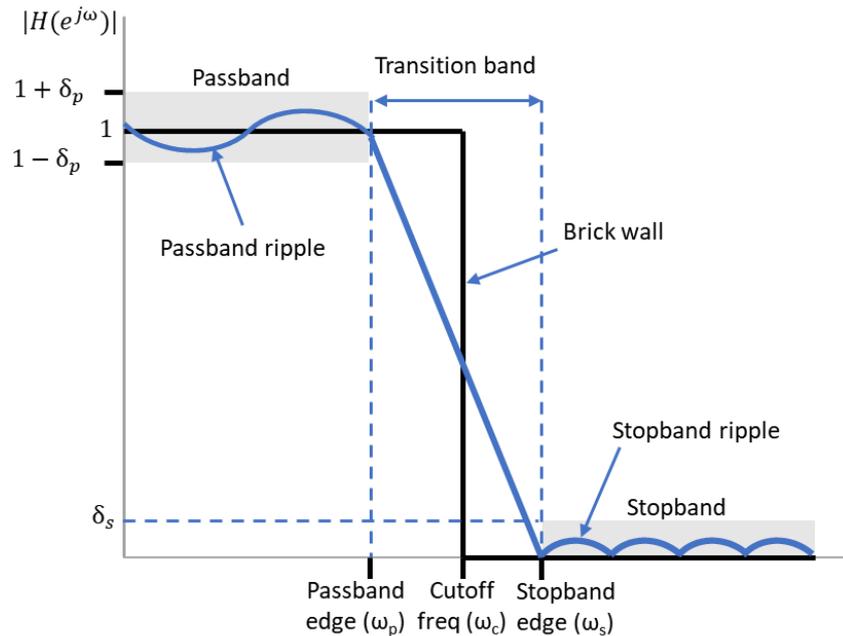


Figure 107. Filter specification representation

In Figure 104, the black curve represents an ideal low pass filter. It passes the frequencies below the cut-off frequencies but stops any frequencies above. It is an ideal filter because:

- The amplitude is exactly 1 in the whole passband
- The amplitude is exactly 0 in the whole stopband
- The transition band appears as a single vertical line located at the cut-off frequency

Unfortunately, the ideal lowpass filter cannot be implemented in practical applications because its impulse response, i.e. the response of the system to an impulse at $t = 0$, is an infinitely long time. As most signals in real-life problems are finite, the ideal lowpass filter is not feasible in such cases.

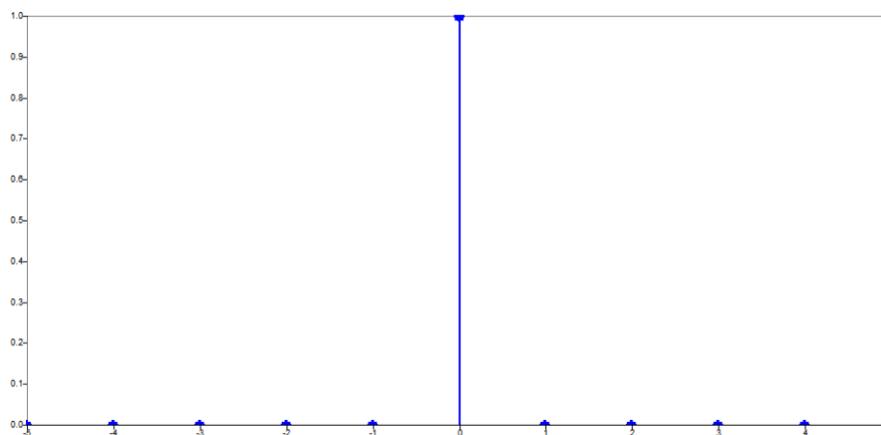


Figure 108. Impulse signal

It is also non-causal, which means that it does not perform the same operation at all times and therefore cannot be shifted, as explained in 8.6.1.

The blue curve in Figure 107 represents a real filter, where ripples might appear both in the passband and stopband. The transition band is no more a vertical line but spans for a certain range of frequencies.

In terms of the amplitude of the frequency response, the design space of the filter is defined by the:

- cut-off frequency,
- width of the transition band,
- tolerance on the ripples.

Additional design requirements can be specified through the phase of the frequency response. In fact, the delay introduced by the filter can be obtained from the phase. Mathematically, the delay is computed as:

$$\text{delay}(\omega) = \tau(\omega) = \frac{\arg(H(e^{j\omega}))}{\omega} = \frac{\varphi(\omega)}{\omega}$$

Where φ is the phase; ω is the frequency.

The delay is a function of frequency. It means that if the signal is made up of different frequencies, each frequency component is delayed by a different quantity and the signal is distorted. But if the phase φ is linear with frequency:

$$\varphi(\omega) = \alpha\omega \quad \rightarrow \quad \tau(\omega) = \frac{\alpha\omega}{\omega} = \alpha$$

The delay becomes constant for all frequencies resulting in no frequency distortion. There are filters that can be designed to have a linear phase.

8.5. Filter Stability

Like any dynamic system, filters can become unstable. Since the mathematical formulation is the same of dynamic systems, a stability check is performed through the poles of the filter transfer function.

A digital filter is asymptotically stable if all its poles are included in the unit circle in the complex plane:

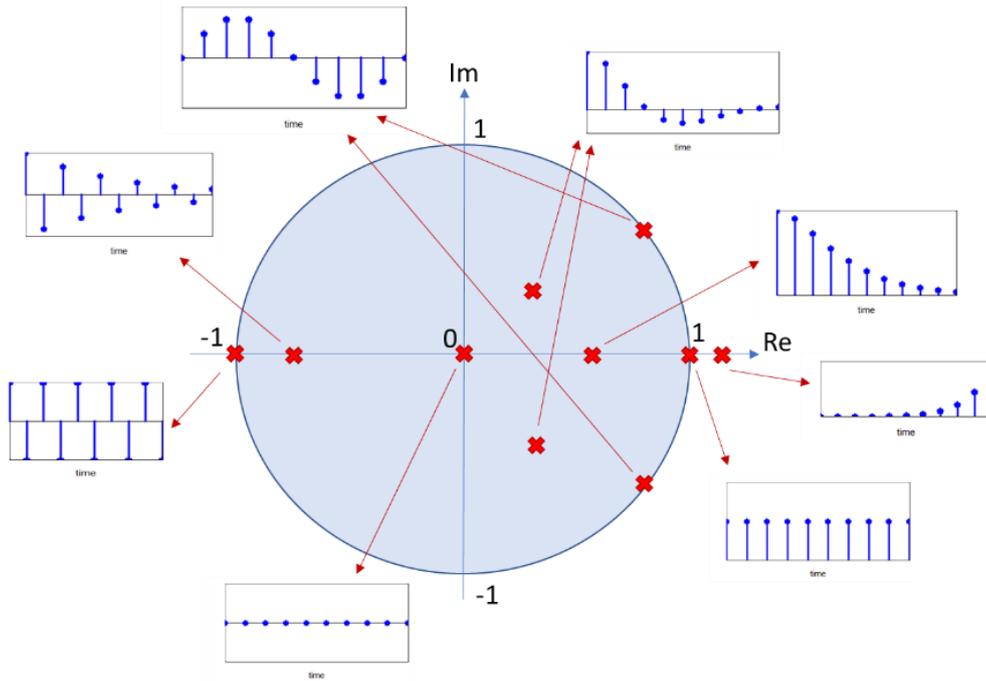


Figure 109. Digital Filter Stability

An analog filter is asymptotically stable, if all its poles are in the left-hand side of the complex plane:

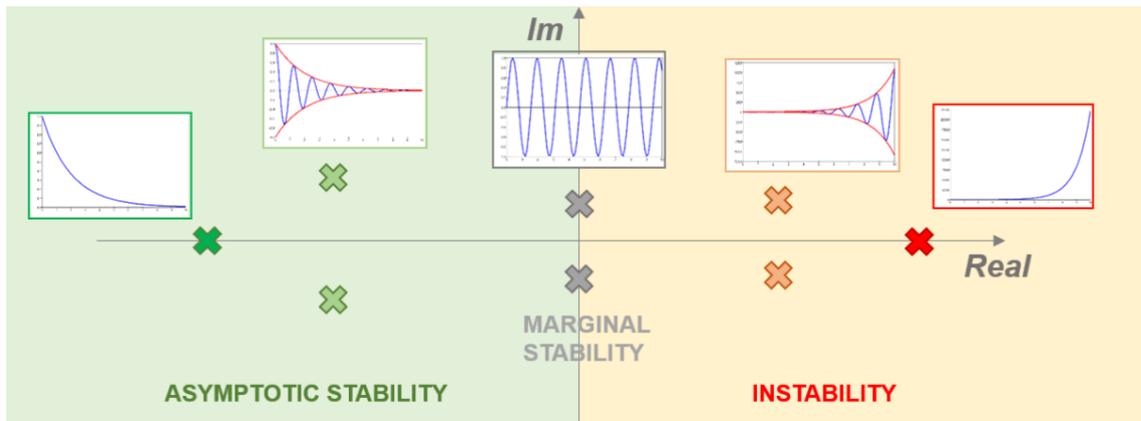


Figure 110. Asymptotic stability versus instability

8.6. Digital Filters

Digital filters are software processes that runs on microprocessors. Hence, they are flexible and adaptive like anything that can be programmed. Depending on their application scenario, they can be described as causal and non-causal, (see: 8.6.1), and also divided into:

- Finite impulse response filters (FIR); see 8.6.2.
- Infinite impulse response filters (IIR); see 8.6.3.

8.6.1. Causal and Non-Causal

In a real-time or system simulation where future signal data is not available, the filter is **causal**. It removes the frequencies it is designed for but introduces some delay. This delay depends on the phase of the frequency response of the filter:



Figure 111. Causal filter representation

For offline or post-processing, past, current and future values of the signal are known, which means that a **non-causal** filter can be applied that does not introduce any delay. In practice, non-causal filters are realized by filtering the signal forward and backward with a causal filter. In this case the phase of the frequency response can be ignored when designing the filter:



Figure 112. Non-causal filter representation

8.6.2. Finite Impulse Response (FIR) Filters

The finite impulse response filters are characterized by a feed-forward structure, as shown in Figure 110:

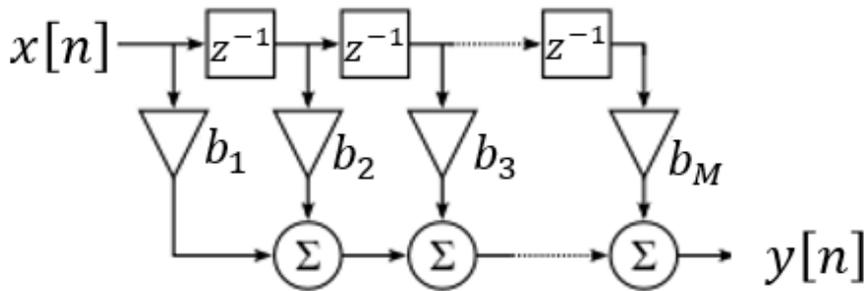


Figure 113. FIR filter representation

The output of these filters only depends on the input.

As there are no more *a* coefficients, their mathematical formulation becomes:

$$y[n] = \sum_{k=0}^M b_k x[n - k]$$

Consequently, FIR filters are described only by the b coefficients, which also define their impulse response:

$$h[n] = \begin{cases} b_n, & 0 \leq n \leq M \\ 0, & \text{otherwise} \end{cases}$$

Since the impulse response goes to 0 after a certain number of time steps, they are called finite impulse response filters.

Applying the Z-transform (see: 8.3), their transfer function is obtained:

$$H(z) = \frac{Y(z)}{X(z)} = \sum_{k=0}^M b_k z^{-k}$$

The transfer function denominator is equal to 1, which means there are no poles. Hence, they are always asymptotically stable.

8.6.3. Infinite Impulse Response (IIR) Filters

Infinite impulse response filters, in addition to a feed-forward structure, have also a feedback one:

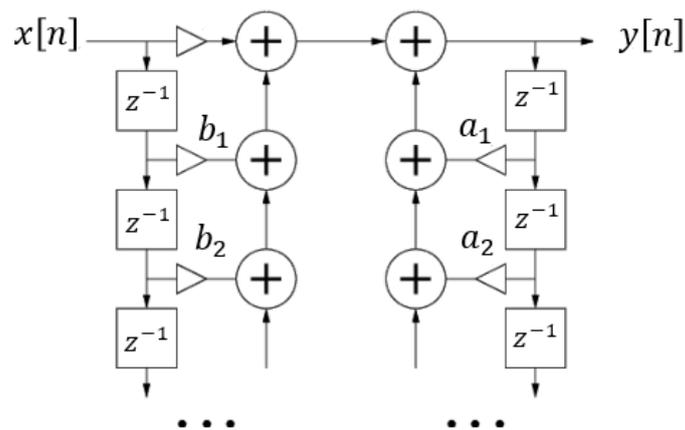


Figure 114. IIR filter representation

Here, the output also depends on its previous values.

As they are described by a and b coefficients:

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

which also defines their impulse response:

$$\text{e.g. first order filter, } h[n] = \begin{cases} (a_1)^n b_0, & n \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

Since the impulse response does not become exactly zero after a certain number of time steps, they are called infinite impulse response filters.

As before, applying the Z-transform, the transfer function is obtained:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^M b_k z^{-k}}{\sum_{k=0}^N a_k z^{-k}}, \quad a_0 = 1$$

Here, the transfer function denominator differs from a constant, which means that these filters might become unstable, based on the location of their poles; as described in 8.5.

8.6.4. Comparison between FIR and IIR

The choice of using FIR or IIR is strongly dependent on the application; as shown in Table 6.

Table 6. FIR versus IIR filters

	FIR	IIR
ADVANTAGES	<ul style="list-style-type: none"> • Stability: no feedback, no poles • Linear phase: no phase distortion • Arbitrary frequency response 	<ul style="list-style-type: none"> • Low latency: lower number of coefficients • Low computational and memory requirement • Analog equivalent
DISADVANTAGES	<ul style="list-style-type: none"> • No analog equivalent • Higher number of coefficients • Higher latency: higher number of coefficients • High computational and memory requirement 	<ul style="list-style-type: none"> • Non-linear phase characteristics • Stability: feedback, poles

8.7. Design Methods

When it comes to practical filter design, for both FIR and IIR types some different design methods are qualitatively described here.

- **FIR** filters can be designed using:
 - Window method, where suitable windows are chosen to smooth out the unfeasible sharp frequency response of the ideal filter. The filter properties are then determined by those of the window.
 - Optimization, where filter coefficients are optimally selected to fit a certain frequency response.
- **IIR** filters can be designed using:
 - Analog Prototyping, where well-known classical filters, e.g. Butterworth or Chebyshev, are converted from the continuous to discrete domains.
 - Optimization, where filter coefficients are optimally selected to fit a certain frequency response.

As shown in Figure 112, Altair Compose has built-in functions for all the methods mentioned:

FIR FILTERS

- **Window method** → fir1
- **Optimization** → firls

IIR FILTERS

- **Analog Prototyping** → besself, butter, cheby1, cheby2, ellip
- **Optimization** → invfreqz



Figure 115. Functions for each type of filter

8.7.1. Analog Prototyping - Design Functions

Some parameters are needed to design the analog prototype, such as the filter order and the frequency range. These parameters can be derived based on the design requirements specified in Figure 107, and there are several functions available, such as `butterd`, `cheblord`, `cheb2ord` and `ellipord` functions. Their syntaxes are:

$$[n, Wn] = \text{butterd}(Wp, Ws, Rp, Rs, \text{domain})$$

$$[n, Wn] = \text{cheblord}(Wp, Ws, Rp, Rs, \text{domain})$$

$$[n, Wn] = \text{cheb2ord}(Wp, Ws, Rp, Rs, \text{domain})$$

$$[n, Wn] = \text{ellipord}(Wp, Ws, Rp, Rs, \text{domain})$$

Where:

- Wp : a scalar specifying the passband frequency of a low or high pass filter, or a two-element vector specifying the passband frequencies of a bandpass or bandstop filter
- Ws : a scalar specifying the stopband frequency of a low or high pass filter, or a two-element vector specifying the stopband frequencies of a bandpass or bandstop filter

Therefore, the transition band corresponds to the difference between Wp and Ws . For a digital filter the values, expressed in Hz, are normalized relative to the Nyquist frequency, such as:

$$Wp = \text{cutoff_freq} / (Fs/2)$$

Where `cutoff_freq` is the desired cut-off frequency and the Nyquist frequency is half the sampling frequency F_s .

Regarding the other parameters:

- R_p : passband ripple, which is the maximum attenuation in decibels
- R_s : stopband ripple (or attenuation), which is the minimum attenuation in decibels
- domain: 'z' for digital or 's' for analog.

Note: Here, the focus is on the design of digital filters, based on their analogue prototypes. Hence, the correct option is to use 'z'. These functions can also be used for applications with analog filters, selecting the 's' option. This also applies to functions in 0 and 8.7.3, which can be used for both analog and digital filters.

Elliptic and Chebyshev-based filters have constant ripple across their passbands, whereas Butterworth-based filters have no ripple. Typically the passband ripple R_p is $20 \log_{10}(1 - \delta_p)$ and the stopband ripple R_s is $-20 \log_{10}(\delta_s)$, where these equations are derived from the definition of a decibel:

$$dB = 20 \log_{10} \left(\frac{A_0(f)}{A(f)} \right)$$

Where $A_0(f)$ is the amplitude before filtering at a specific frequency; $A(f)$ is the amplitude after the filtering process.

$$20 \log_{10} \left(\frac{A_0(f)}{A(f)} \right) = 20 \log_{10} \left(\frac{1 - \delta_p}{1} \right) = 20 \log_{10}(1 - \delta_p) - 20 \log_{10} 1 = \mathbf{20 \log_{10}(1 - \delta_p)}$$

$$20 \log_{10} \left(\frac{A_0(f)}{A(f)} \right) = 20 \log_{10} \left(\frac{1}{\delta_s} \right) = 20 \log_{10} 1 - 20 \log_{10}(\delta_s) = \mathbf{-20 \log_{10}(\delta_s)}$$

8.7.2. Analog Prototyping - Functions for Filter Creation

The outputs of such design functions are:

- n: lowest filter order that meets the input requirements
- W_n : normalized frequency input for which the requirement is met

These outputs can now be used to create the respective filters:

`buttord` → `[b, a] = butter(n, Wp, band, domain)`

`cheblord` → `[b, a] = cheby1(n, Rp, Wp, band, domain)`

`cheb2ord` → `[b, a] = cheby2(n, Rs, Ws, band, domain)`

`ellipord` → `[b, a] = ellip(n, Rp, Rs, Wp, band, domain)`

Finally, with the outputs of such functions, i.e. numerator b and denominator a the filter's polynomial coefficients, the signal may be filtered.

8.7.3. Filter Functions

A signal can be filtered with functions such as `filter` and `filtfilt`, using the numerator and denominator determined by the filter creation functions:

- `filter`: Filters a signal using either infinite impulse response (IIR) or finite impulse response (FIR); as explained in 8.6.2 and 8.6.3. The main feature is that it only filters the signal once, forward in time, causing a phase distortion. Usually it adds different delays at different frequencies.

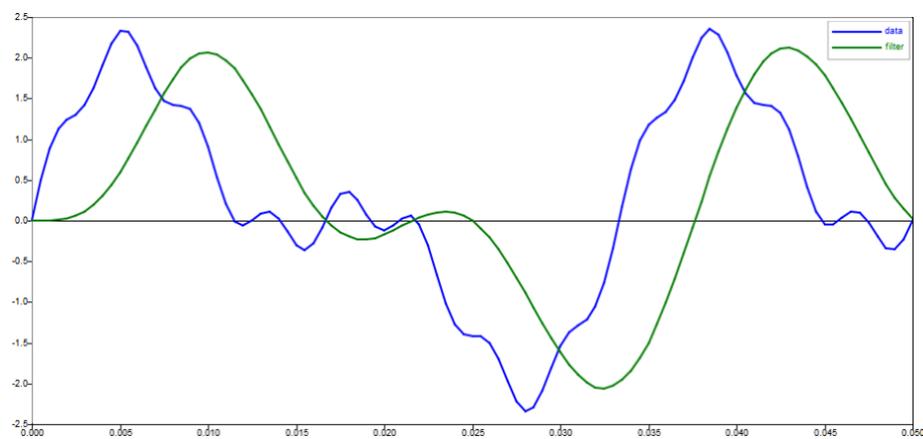


Figure 116. Output of `filter` function, with phase distortion

- `filtfilt`: Filters the signal forwards and then backwards, compensating for end effects and therefore correcting the phase distortion. The backward filtering process requires the signal to be known in future instants, thus it cannot be used in real-time applications. It implements non-causal filtering; as explained in 8.6.1.

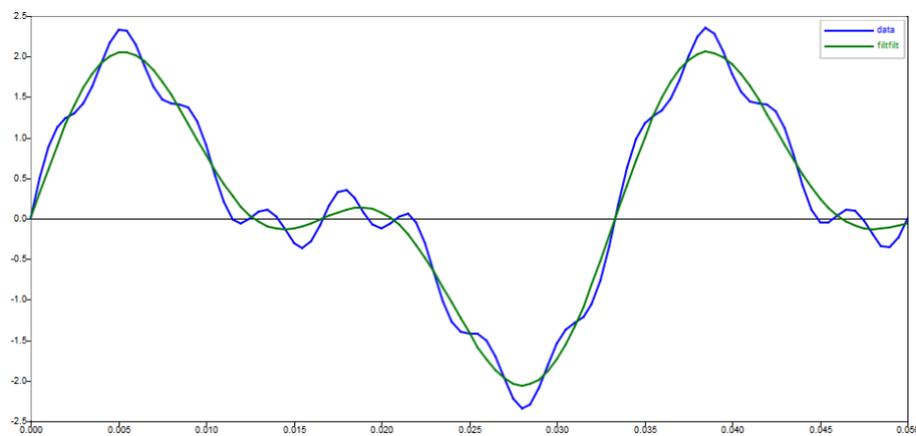


Figure 117. Output of `filtfilt` function, without phase distortion

Exercises

- 1) Design an IIR Butterworth low pass filter (cutoff frequency at 10 Hz), given the signal:

$$s = A_1 \sin(2\pi f_1 t) + A_2 \sin(2\pi f_2 t) + A_3 \sin(2\pi f_3 t)$$

Where:

- $A_1 = 10, f_1 = 5$ [Hz]
 - $A_2 = 6, f_2 = 15$ [Hz]
 - $A_3 = 1, f_3 = 30$ [Hz]
- 2) With the designed IIR filter, implement causal and non-causal filtering.
 - 3) For the same signal, design and deploy a FIR low pass filter.
 - 4) For the same signal, design and deploy high pass filters (both FIR and IIR). These filters must stop f_1 and f_2 frequencies.
 - 5) For the same signal, design and deploy band stop filters (both FIR and IIR). These filters must stop f_2 frequency. Can phase distortion be detected in causal filtering?

References

- [1] Lai, Edmung. “*Practical Digital Signal Processing for Engineers and Technicians*”. Elsevier, 2003.
- [2] Prandoni, Paolo; Vetterli, Martin. “*Digital Signal Processing Training Material*”. École Polytechnique Fédérale de Lausanne. Available at <https://www.coursera.org/learn/dsp1>.
- [3] Toledo, Marcelo. Workshop: Introduction to Accelerated Durability Tests using Altair Compose®(2019)
- [4] National Instruments website. Available at <https://www.ni.com/pt-br.html>.
- [5] Analogic Tips website. Available at <https://www.analogictips.com/>.
- [6] Smith, Steven W. “*The Scientist and Engineer's Guide to Digital Signal Processing*”. Available at <https://www.dspguide.com/>.

Annex

1. DFT as a Change of Basis

In linear algebra, a sequence of vectors from a vector space is said to be **linearly dependent** if there are scalars different from zero such that:

$$a_1 \mathbf{v}_1 + a_2 \mathbf{v}_2 + \cdots + a_n \mathbf{v}_n = \mathbf{0}$$

One may represent any vector in a vector space as a sum of linearly independent vectors, that is, vectors whose only scalars that satisfy the equation above are necessarily equal to zero. This representation is known as a change of basis and is carried out by taking the dot product of the vector with each of the linearly independent basis vectors. Each dot product corresponds to the amount of that vector necessary to the change of basis. For example, consider the change of basis of the vector \mathbf{v} described below:

$$\mathbf{v} = \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix}$$

To the basis U in \mathbb{R}^3 :

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}; \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}; \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The solution is obtained through a system of equations:

$$\begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix} = a_1 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + a_2 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + a_3 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$1 = a_1 * 1 + a_2 * 0 + a_3 * 0 \therefore \mathbf{a}_1 = \mathbf{1}$$

$$-1 = a_1 * 0 + a_2 * 1 + a_3 * 0 \therefore \mathbf{a}_2 = -\mathbf{1}$$

$$2 = a_1 * 0 + a_2 * 0 + a_3 * 1 \therefore \mathbf{a}_3 = \mathbf{2}$$

Therefore, the transformation matrix is:

$$[\mathbf{v}]_U = \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix}$$

And the same vector \mathbf{v} could also be described in another basis V in \mathbb{R}^3 :

$$\begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}; \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix}; \begin{bmatrix} -2 \\ 1 \\ 4 \end{bmatrix}$$

Solving the system of equations as described gives the following coefficients:

$$a_1 = 19$$

$$a_2 = -5$$

$$a_3 = 4$$

Therefore, the transformation matrix in this case is:

$$[v]_V = \begin{bmatrix} 19 \\ -5 \\ 4 \end{bmatrix}$$

Which means that although the vector is the same in both bases, it has a different representation in each one of them.

Considering the discrete Fourier transform (DFT), each step of the analysis formula (transform from the time to frequency-domain) is a sum of products of the input signal and a complex exponential, where these complex exponentials are in essence the basis vectors and the signal is the vector subjected to the change of basis. As seen in the Fourier series, the coefficients are obtained by projecting the function into this new basis.

2. Data Handling Strategies

2.1. Vectorization

Altair Compose® is designed to optimize operations using vectors and matrices, which is called vectorization, one of the best practices in open-matrix language (OML). It is a strategy to replace loop-based tasks focused on scalar operations for vector/matrix operations, that is, the algorithm changes from executing a single value at a time to a set of values all at once.

There are several advantages of using vectorization whenever is feasible:

- **Higher performance:** vectorized operations run much faster than loop-based ones.
- **Better organization:** vectorized commands are shorter than loop-based ones and therefore the scripts become more concise and more organized.
- **Less errors:** as vectorization involves using as many inbuilt functions as possible, it also reduces the susceptibility to errors in the algorithms, as these functions were tested, validated and optimized in terms of code execution.

As an example, consider the usage of a user-defined sum function and the inbuilt `sum` function to be used in a random vector of 100 000 elements. The functions `tic` and `toc` are used to get the estimated time to run the script. Firstly, the user-defined sum function:

```
x = randn(100000,1);
```

```
tic

mysum = 0;

for ii = 1:length(x)

    mysum = mysum + x(ii);

end

toc
```

There is no perceptible difference of performance (elapsed time is 0.005 seconds) in comparison with the inbuilt function:

```
tic

mysum = sum(x);

toc
```

The difference starts to escalate as the number of operations grow. If a vector of 10 million elements is now used in both cases, the user-defined function takes 4.3 seconds to run while the inbuilt `sum` function takes only 0.01 second, which is almost 400 times faster. On top of the higher performance, the inbuilt function gives a much smaller algorithm.

Another example using the element-wise matrix multiplication of 2 vectors, each one with 10 million elements, with a user-defined function:

```
x = randn(10000000,1);

y = randn(10000000,1);

tic

for ii = 1:length(x)

    z(ii) = x(ii)*y(ii);

end

toc
```

And vectorized operations:

```
tic

z = x.*y;

toc
```

The user-defined function takes approximately 7.4 seconds to run while the vectorized multiplication takes only 0.05 seconds, which is more than 160 times faster.

Even conditional statements may be vectorized. Using logical operators vectorizes conditional statements that would normally require a loop.

As an example, consider a certain vector whose values greater than 3 must be queried:

```
data = [1 5 3 6 9 2 0 3];
```

Instead of looping through each element to store it in case it meets the requirement

```
count = 1;
for ii = 1:length(data)
    if data(ii) > 3
        newdata(count) = data(ii);
        count = count + 1;
    end
end
```

A vectorized conditional statement could be used to speed up the execution:

```
newdata = data(data > 3);
```

2.2. Multidimensional Matrices

OML supports multidimensional matrices and their algebraic operations. A multidimensional matrix contains more than the conventional 2 dimensions (rows and columns) and it is useful to store data that represents several entities in a single variable, for example:

- **First dimension:** time steps
- **Second dimension:** accelerations
- **Third dimension:** channels

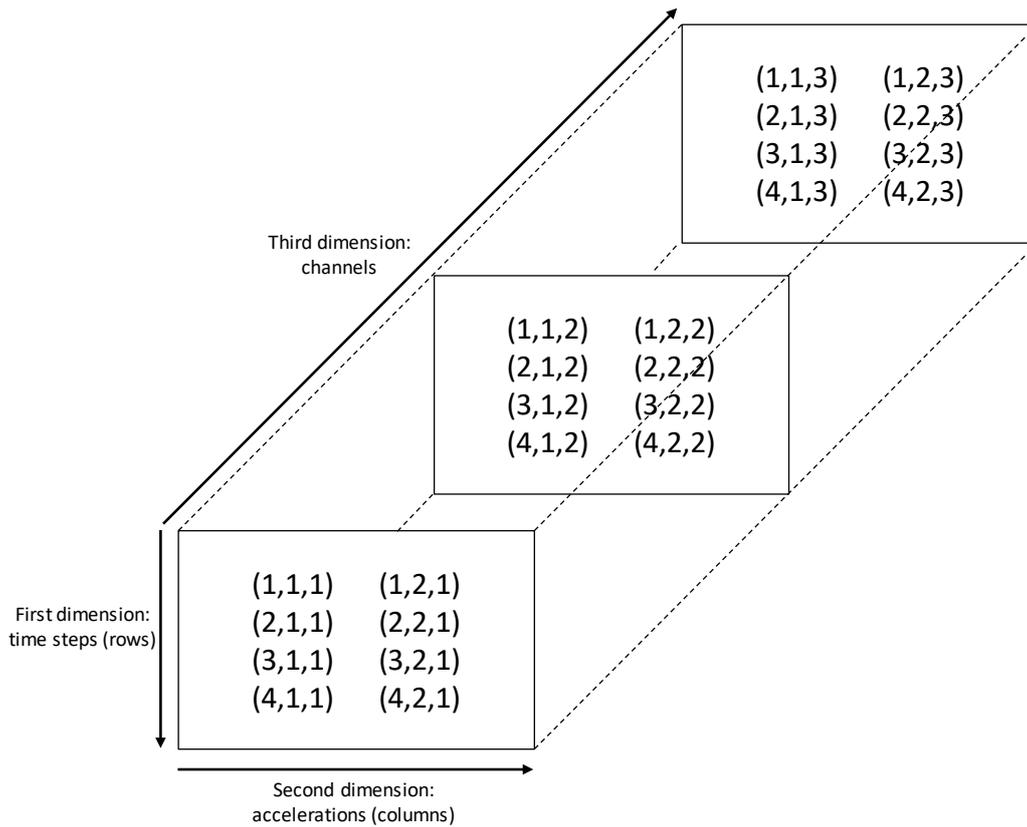


Figure 118. Representation of a multidimensional matrix and its indices

There are several ways to create a multidimensional matrix:

- Create a 2D matrix and append extra dimensions using “:” operators


```
mat = [1 2 3; 4 5 6];
mat(:, :, 2) = [7 8 9; 10, 11, 12];
```
- Concatenate matrices in a specific dimension with `cat` function


```
mat = [1 2 3; 4 5 6];
mat = cat(3, mat, [7 8 9; 10, 11, 12]);
```
- Create an N -dimensional matrix with functions like `zeros` and `ones` and populate it


```
mat = zeros(2, 3, 2);
mat = ones(2, 3, 2);
```

2.3. Sparse Matrices

Sparse matrices are those in which most elements are equal to zero. The number of zero-valued elements divided by the total number of elements is called the sparsity of the matrix. A matrix is considered sparse when its sparsity is greater than 0.5.

```

1 0 2 0 0 0
0 0 0 1 0 4
0 1 0 1 0 0
0 0 0 0 3 0
0 0 0 0 0 0
1 0 0 2 0 0
0 0 1 0 0 0
0 3 0 0 0 0
0 0 0 0 0 1

```

When stored in the conventional representation, both memory and computational time are wasted uselessly to manipulate and process such matrices, as zero-valued elements need the same amount of memory as any other matrix element. On the other hand, explicitly using functions such as `sparse` to create a sparse matrix saves a significant amount of memory and time to handle with these variables. In this case, OML stores only the nonzero elements and their respective indices.

As an example, consider the storage of the matrix shown above, where only specific rows and columns have non-zero values:

```

row = [1, 1, 2, 2, 3, 3, 4, 6, 6, 7, 8, 9];
col = [1, 3, 4, 6, 2, 4, 5, 1, 4, 3, 2, 6];

```

These values are:

```

vals = [1, 2, 1, 4, 1, 1, 3, 1, 2, 1, 3, 1];

```

The creation of such sparse matrix is carried out with `sparse` function.

```

s = sparse(row, col, vals)

```

The output is printed in the command window only for the non-zero values

```

s = sparse [9 x 6], nnz = 12
[1,1] 1
[6,1] 1
[3,2] 1
[8,2] 3
[1,3] 2
[7,3] 1
[2,4] 1
[3,4] 1
[6,4] 2
[4,5] 3
[2,6] 4
[9,6] 1

```

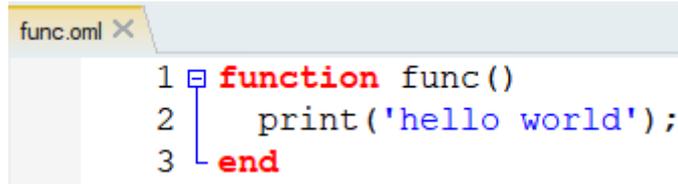
Where `nnz` is the number of non-zero elements and it may also be retrieved using `nnz` function.

Another function to create a sparse matrix is `speye`, which creates a sparse identity matrix of a specified size.

3. Other Best Practices

3.1. Organizational Best Practices

- Keep frequent procedures in separated functions/files and call them wherever it is necessary. It reduces errors related to version control and typos



```
func.OML x
1 function func()
2   print('hello world');
3 end
```

Example of function

- Use separate folders for each project in order to keep related files together
- Use modular programming to split the code into simple and cohesive functions instead of having a long and unique block

3.2. Performance Best Practices

- Although OML does not need variables to be declared upfront, pre-allocating them when their size is known speeds up execution. Functions that serve for this purpose are `zeros`, `ones`, `eye`, `diag`, `rand` and `repmat`. An example without pre-allocation:

```
x = 0;

for k = 2:1000000
    x(k) = x(k-1) + 1;
end
```

Leads to an execution approximately 65 times slower than:

```
x = zeros(1,1000000);

for k = 2:100000
    x(k) = x(k-1) + 1;
end
```

- When looping through a matrix, loop down columns to access memory, that is, use of indices. Consider the following 3x3 matrix:

$$\begin{array}{l} [(1,1), (1,2), (1,3)] \\ [(2,1), (2,2), (2,3)] \\ [(3,1), (3,2), (3,3)] \end{array} \Rightarrow \begin{array}{l} [1, 4, 7] \\ [2, 5, 8] \\ [3, 6, 9] \end{array}$$

The left matrix contains the subscript tuples for each matrix element, while the right matrix shows the linear indices for the same matrix. The linear index traverses dimension 1 (rows), then dimension 2 (columns), then dimension 3 (pages), etc. until it has numbered all of the elements. Indices are faster than subscripts and are preferable where the data set is very large.

- Place independent operations outside loops
- Create new variables if data type changes

Exercises - Solutions

Chapter 3

1) Considering *disptime.ascii* to be stored in the same folder of the File Browser, as it is a delimited file, the first function to read it is `dlmread`.

```
signal1 = dlmread('disptime.ascii', ' ');
```

Using the CAE readers, `readcae` does the same task.

```
signal2 = readcae('disptime.ascii', 1, 1, [], [], 1)
```

The difference is that due to the way that the file is written, a spurious singular dimension is stored that may be removed with the `squeeze` function. It may also be transposed with `'` operator to have time steps in rows and not in columns:

```
signal2 = squeeze(signal2)';
```

2) Considering *ANGACC* to be stored in the same folder of the File Browser, there must be a `readvector` command for each component:

```
x = readvector('ANGACC', 'Angular Acceleration', '50th% Hybrid3  
- LOWER TORSO', 'X-comp. ang. acc.');
```

```
y = readvector('ANGACC', 'Angular Acceleration', '50th% Hybrid3  
- LOWER TORSO', 'Y-comp. ang. acc.');
```

```
z = readvector('ANGACC', 'Angular Acceleration', '50th% Hybrid3  
- LOWER TORSO', 'Z-comp. ang. acc.');
```

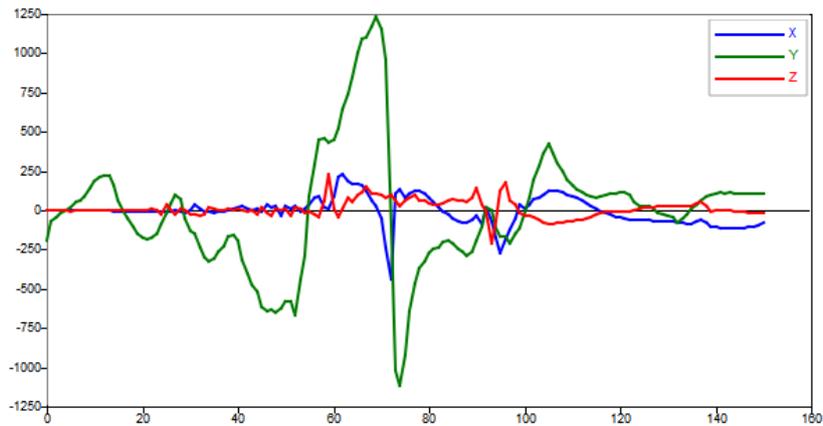
The time step list to be used as X data of the plot is retrieved with `gettimesteplist` function.

```
t = gettimesteplist('ANGACC', 'Angular Acceleration', '50th%  
Hybrid3 - LOWER TORSO', 'Z-comp. ang. acc.');
```

And finally, the plot with the 3 pairs of curves and a legend to identify each one of them:

```
plot(t, x, t, y, t, z);  
  
({'X', 'Y', 'Z'});
```

```
legend({'X','Y','Z'});
```



3) Instead of multiple calls of `readvector` function, a single `readmultivectors` queries the 3 components, knowing that the initial request is the same as the final one (Lower Torso), the initial component is X and the final is Z.

```
comps = readmultivectors('ANGACC','Angular  
Acceleration','50th% Hybrid3 - LOWER TORSO','50th% Hybrid3  
- LOWER TORSO','X-comp. ang. acc.','Z-comp. ang.  
acc.','All');
```

A spurious singular dimension is stored that may be removed with `squeeze` function.

```
comps = squeeze(comps);
```

Each component can be stored in a different variable according to the index of `comps` variable.

```
x = comps(1,:);
```

```
y = comps(2,:);
```

```
z = comps(3,:);
```

To plot using the same commands of 2.

4) The resultant is defined by:

$$resultant = \sqrt{x^2 + y^2 + z^2}$$

Therefore, in OML:

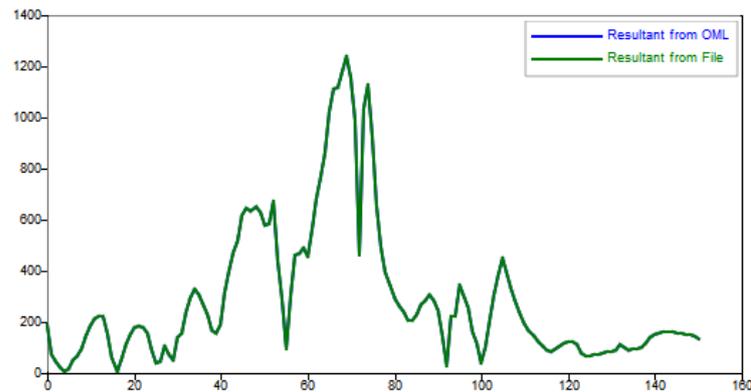
```
resOML = sqrt(x.^2 + y.^2 + z.^2);
```

The resultant from the file queried with readvector function is:

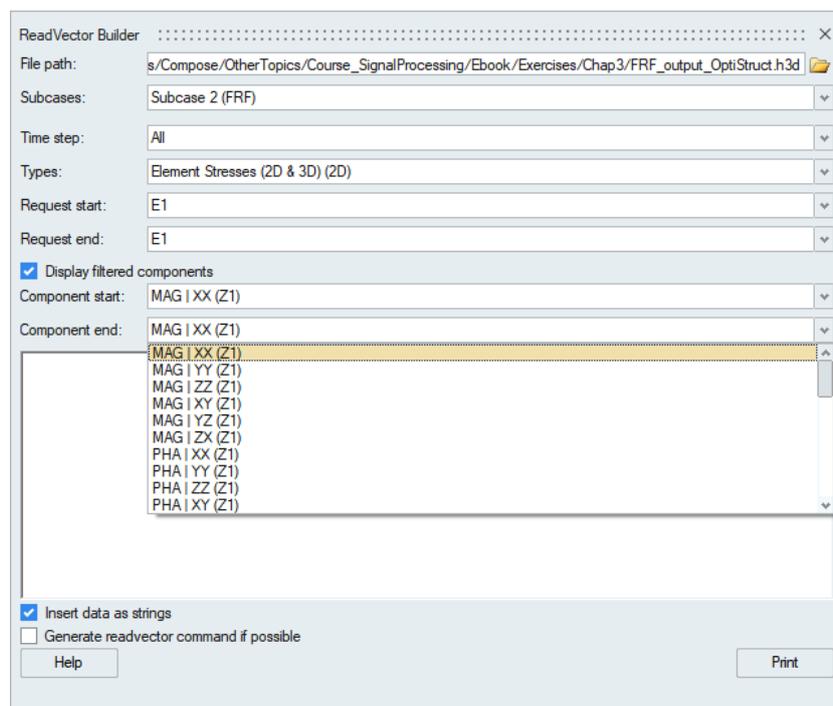
```
res = readvector('ANGACC', 'Angular Acceleration', '50th% Hybrid3  
- LOWER TORSO', 'Res. ang. acc.')
```

If both variables are plotted, it confirms that they are the same:

```
plot(t, resOML, t, res);  
  
legend({'Resultant from OML', 'Resultant from File'});
```



5) As seen with readvectorbuilder, these components are not sequential and therefore readcae is more appropriate, passing a list of components instead of a range.



Considering *FRF_output_OptiStruct.h3d* to be stored in the same folder of the File Browser, `readcae` reads all data of these components accessing the indices within the file:

```
comps = readcae('FRF_output_OptiStruct.h3d', 2, 2, [], [2, 5], [], 1);
```

Where:

- 2: second subcase
- 2: second result type
- []: all elements
- 2 and 5: indices of components
- []: all time steps

The RMS stresses of XX and XY components are:

```
RMS_XX = rms(squeeze(comps(:,1,:)));
```

```
RMS_XY = rms(squeeze(comps(:,2,:)));
```

Each variable is a 500-element vector, where each value corresponds to the RMS value of a specific element over all time steps.

Chapter 4

1) Considering *Month_vs_Sales.csv* to be stored in the same folder of the File Browser, as it is a delimited file, the most efficient function to read it is `dlmread`

```
signal = dlmread('Month_vs_Sales.csv', ',');
```

The headers should be discarded and it is convenient to store each vector in a different variable

```
signal = signal(2:end, :);
```

```
t = signal(:, 1);
```

```
sales = signal(:, 2);
```

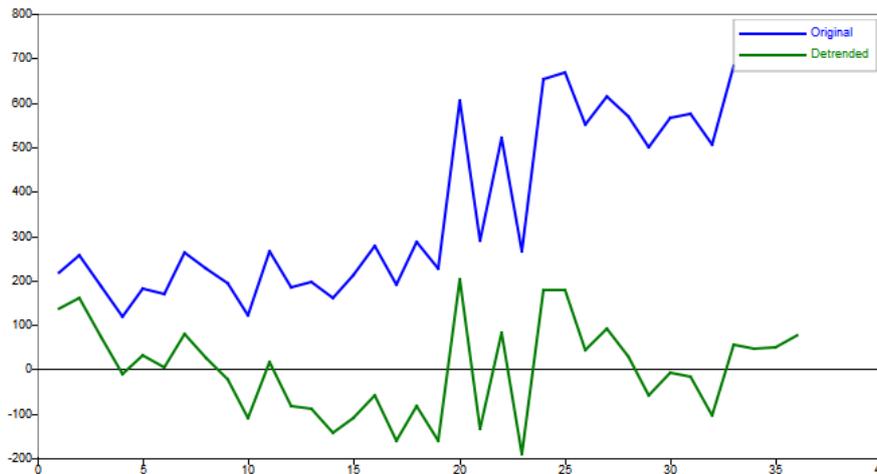
To detrend the data set, use the `detrend` function. As the data shows an increasing trend, 'linear' is the most appropriate method:

```
sales_detrend = detrend(sales, 'linear');
```

Then plot both curves by giving the 2 pairs of variables:

```
plot(t, sales, t, sales_detrend);
```

```
legend({'Original','Detrended'});
```



As the detrend function with 'linear' method is nothing more than a polynomial fit of degree equal to 1, polynomial functions achieve the same goal in order to detrend the data set.

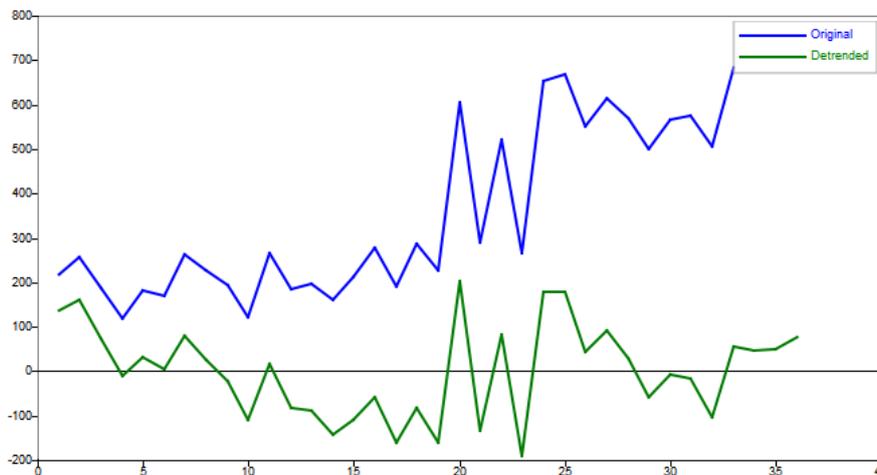
The polynomial coefficients are determined by `polyfit` function and the polynomial values are computed with `polyval` function.

```
p = polyfit(t,sales,1);
sales_poly = polyval(p,t);
```

As detrending the signal means that a trend is subtracted, simply perform a subtraction between the original values and those calculated with `polyval` function.

```
sales_detrend_poly = sales - sales_poly;
```

The result is the same.



2) Considering *Month_vs_Sales2.csv* to be stored in the same folder of the File Browser, as it is a delimited file, the most efficient function to read it is `dlmread`

```
signal = dlmread('Month_vs_Sales2.csv', ',');
```

Discard the headers and it is convenient to store each vector in a different variable

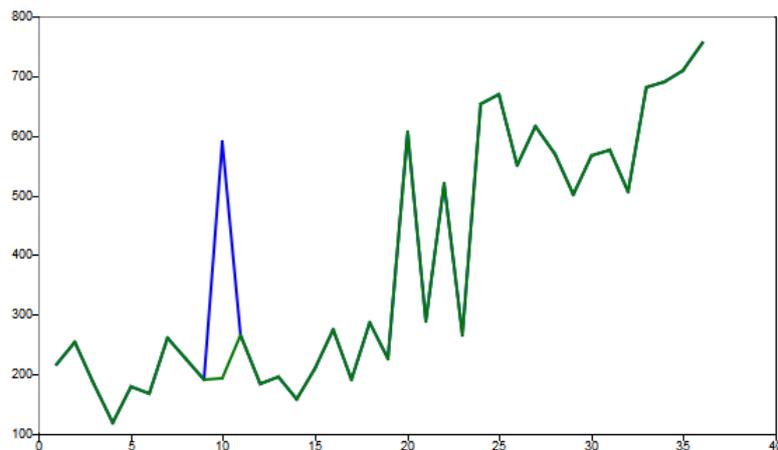
```
signal = signal(2:end, :);
t = signal(:, 1);
sales = signal(:, 2);
```

To detect and remove outliers, the user-defined `mad` function is used:

```
[interp_median, interp_mad, sales_corrected] = mad(sales);
```

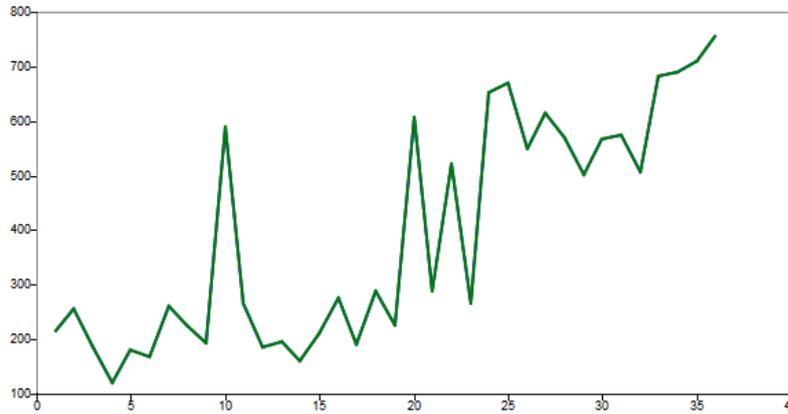
Then plot both curves by giving the 2 pairs of variables:

```
plot(t, sales, t, sales_corrected);
```

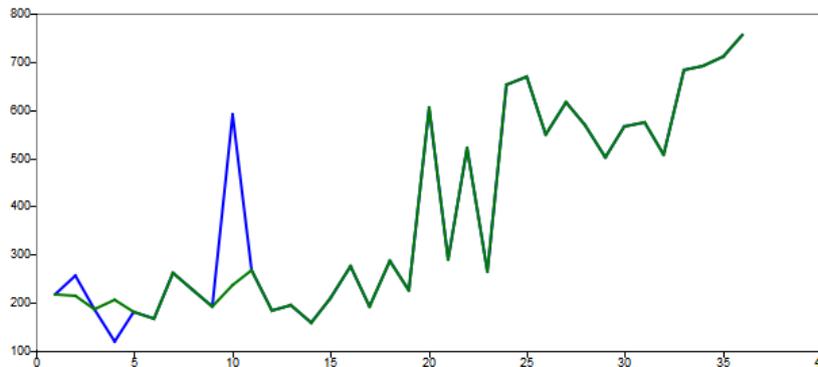


Although there are equivalent values over the time series, the point around $t = 10$ is considered to be an outlier because this method considers the window and its surrounding samples to compute the median of such window and to determine whether a point is a spike or not.

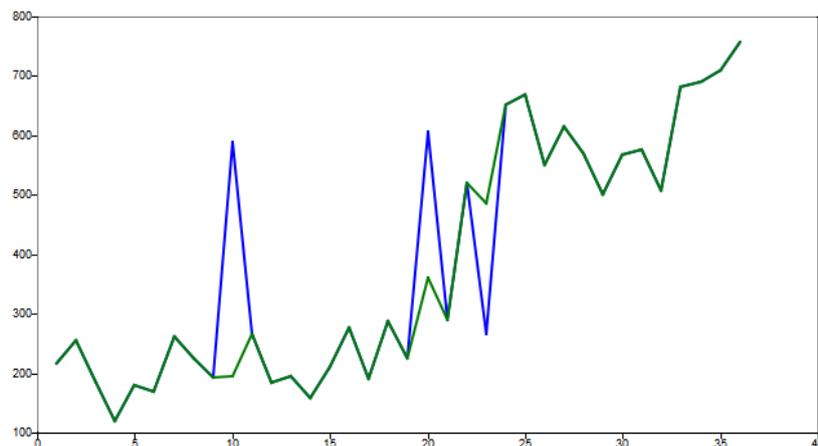
3) Changing the size of the window to 1, no outliers are detected because the window is purely composed of 1 sample and therefore there is no variability in data



Changing the size of the window to 15 gives more outliers. As the number of surrounding samples increased, the median of the window was modified after considering these new samples. Hence a new dispersion criterion results in identification of new outliers.



4) The threshold is the number of standard deviations away from the median, and if a value is a certain number of MAD away from the median, that value is classified as an outlier. As the number of standard deviations decreases from 5 to 3, there is less dispersion in the data and therefore it become stricter with regard to the identification of outliers. That is why other spikes are not considered as this value decreases:



5) The signal can be declared as:

```
x = [5 6 4 2 1 7 5 -3 -4 -6 -9 -12 -1 2];
```

And downsampled with the command:

```
x_dsampl e = downsample(x, 3);
```

Whose `x_dsampl e` variable is:

```
5 2 5 -6 -1
```

The new sampling period is the original sampling period times the downsampling factor, hence:

```
ts = 0.1;
new_ts = ts*3;
```

Which is equal to 0.3 seconds. And sampling frequency is the inverse of sampling period:

```
new_fs = 1/new_ts;
```

Which is equal to 3.33 Hz. The new Nyquist frequency is half the sampling frequency:

```
nyquist_fs = new_fs/2;
```

Which is equal to 1.66 Hz. This indicates that after downsampling the signal by a factor of n , the new Nyquist frequency is decreased n times.

6) To avoid aliasing, the original signal must be processed in a lowpass filter before downsampling, to ensure that the maximum frequency of the filter signal is smaller than $\frac{f_s}{2n}$. The normalized stop frequency edge should be:

$$\Omega_{stop} = \frac{2\pi f_s}{2n} T = \frac{\pi}{n} \text{radians}$$

As in the previous example the downsampling factor was equal to 3:

```
n = 3;
f_stop = pi/n;
```

Which is equal to 1.047 radians.

7) The signal can be declared as:

```
x = [5 6 4 2 1 7 5 -3 -4 -6 -9 -12 -1 2];
```

And upsampled with the command:

```
x_usample = upsample(x,2);
```

Whose `x_usample` variable is:

```
5 0 6 0 4 0 2 0 1 0 7 0 5 0 -3 0 -4 0 -6 0 -9 0 -12 0 -1 0 2 0
```

The new sampling period is the original sampling period divided by the downsampling factor, hence:

```
ts = 0.1;
new_ts = ts/2;
```

Which is equal to 0.05 seconds. And sampling frequency is the inverse of sampling period:

```
new_fs = 1/new_ts;
```

Which is equal to 20 Hz. The new Nyquist frequency is half the sampling frequency:

```
nyquist_fs = new_fs/2;
```

Which is equal to 10 Hz. This indicates that after upsampling the signal by a factor of n , the new Nyquist frequency is increased n times.

8) The signal can be declared as:

```
x = [15 7 25 8 16 10 6 9 3 0 22 10 15];
```

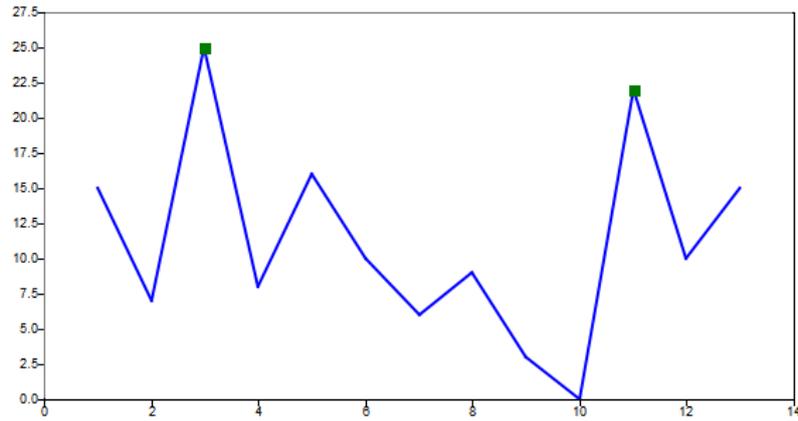
And `findpeaks` function detects the local maxima:

```
[pks,idx,extra] = findpeaks(x);
```

Where the `pks` variable has the values of the peaks and `idx` variable has the indices where they occur. In order to plot the signal and the peaks that have been identified:

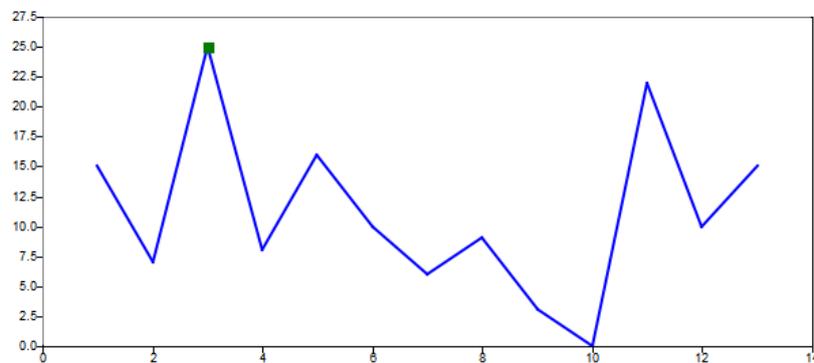
```
plot(x);
hold on;
scatter(idx,pks);
```

The plot is:



In the case where a criterion is added to consider only peaks above 23, one less peak is identified:

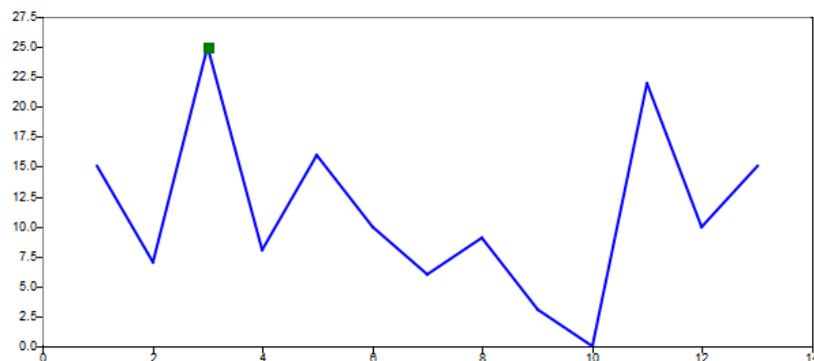
```
[pks,idx,extra] = findpeaks(x,'MinPeakHeight',23);
```



The second peak has a value of 22 and therefore it is not considered according to this criterion.

In the case where only peaks whose distance between each other is 10 are considered, one less peak will also identified:

```
pks,idx,extra] = findpeaks(x,'MinPeakDistance',10)
```



The distance from the first peak (at position = 3) to the second one (at position = 11) is not greater than 10. Therefore, the second peak is no longer considered according to this criterion.

9) The signal can be declared as:

```
x = [15 7 25 8 16 10 6 9 0 -22 -10 -5 -2 -6 -8];
```

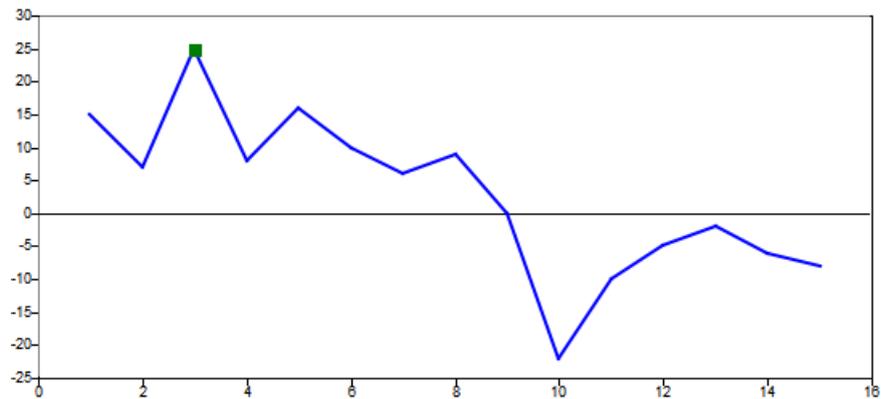
And findpeaks function detects the local maxima:

```
[pks,idx,extra] = findpeaks(x);
```

Where pks variable has the values of the peaks and idx variable has the indices where they occur. In order to plot the signal and the peaks that have been identified:

```
plot(x);
hold on;
scatter(idx,pks);
```

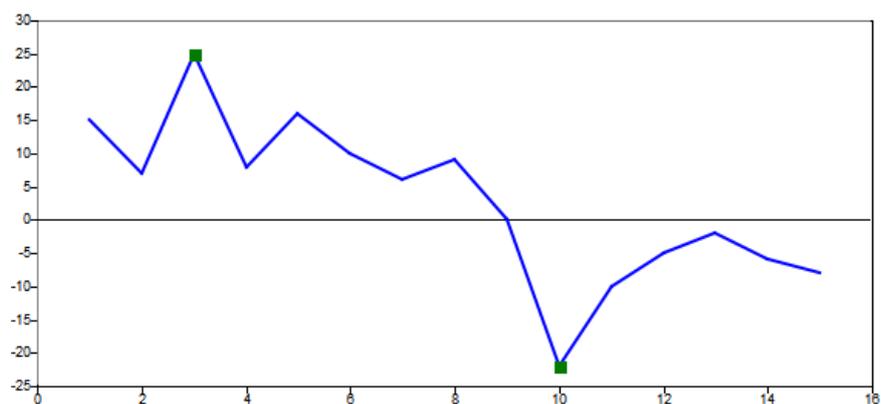
Whose plot is:



As expected, negative peaks are not identified and in order to catch them, the parameter 'DoubleSided' must be used:

```
[pks,idx,extra] = findpeaks(x,'DoubleSided');
```

Which indicates that data has positive and negative values, and that both positive and negative peaks are desired.



When the parameter 'DoubleSided' is used, the minimum peak height is relative to the mean, which is:

```
x_mean = mean(x);
```

And is equal to 2.867.

Chapter 5

1) The signal can be declared as:

```
x = [12 15 51 58 22];
```

The skewness may be calculated with:

```
sk_x = skewness(x)
```

As skewness is approximately equal to 0.3645, it is positive and hence the data has a positively skewed distribution.

And kurtosis:

```
krt_x = mean((x - mean(x)).^4) / std(x).^4
```

```
excess_krt = krt_x - 3;
```

Kurtosis is approximately 0.8255. As the excess kurtosis is -2.1745, it is negative and therefore a platykurtic distribution.

2) Using max function and rms function:

```
max_x = max(x)
```

```
rms_x = rms(x)
```

A maximum and RMS values of X, 58 and 36.9269 respectively, are obtained that can be used to compute crest factor:

```
CF = max_x / rms_x
```

Which is equal to 1.5706.

When the 4th element is changed from 58 to 100:

```
x = [12 15 51 100 22]
```

The same parameters (maximum, RMS and crest factor) now become equal to 100, 51.8729 and 1.9277, respectively. Although the RMS value increased, a higher peak increases the ratio, leading to a higher crest factor.

3) The function is:

```
f = @(t) sin(t) .* (sin(t) >= 0) + 0 * (sin(t) < 0);
```

And its values are:

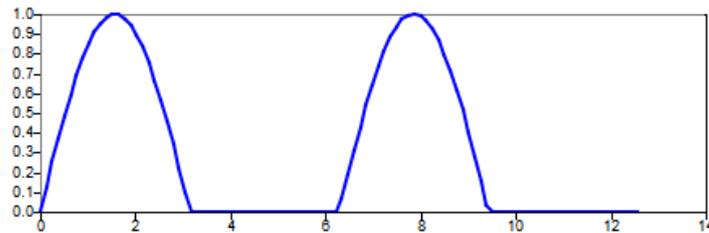
```
t = linspace(0, 4*pi);
```

```
fun = f(t);
```

Then plot function with t and fun variables is:

```
plot(t, fun);
```

Generates the graph:



The crest factor is computed by:

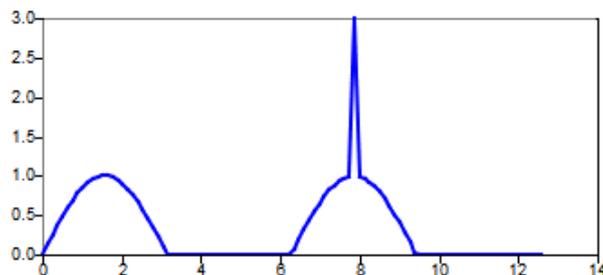
```
CF = max(fun) / rms(fun);
```

Whose value is 2.0098. When an outlier is introduced, replacing the peak value of 1 for 3, for example:

```
[maxval, idxmax] = max(fun);
```

```
fun(idxmax) = 3;
```

Naturally the crest factor increases, reaching the value of approximately 5.2422:



4) The skewness is computed with the command:

```
sk_fun = skewness(fun)
```

Its value before the outlier was 0.6781, which is greater than 0 and means that most samples of the signal are smaller than the average value. In fact, it also confirms that the mean of the data is greater than the median:

```
mean_fun = mean(fun)
```

```
median_fun = median(fun)
```

Whose values are 0.3151 and 0, respectively. After the introduction of the outlier, it becomes 2.1646 and the difference between the mean and median increases, becoming 0.3351 and 0, respectively.

5) After declaring the signals:

```
x = [8 3 7 2 4 6];
```

```
y = [0 8 3 7 2 4];
```

`xcorr` function calculates the cross-correlation when 2 inputs are given:

```
Rxy = xcorr(x, y);
```

Whose output is a 1 x 11 vector:

```
Rxy = [Matrix] 1 x 11
```

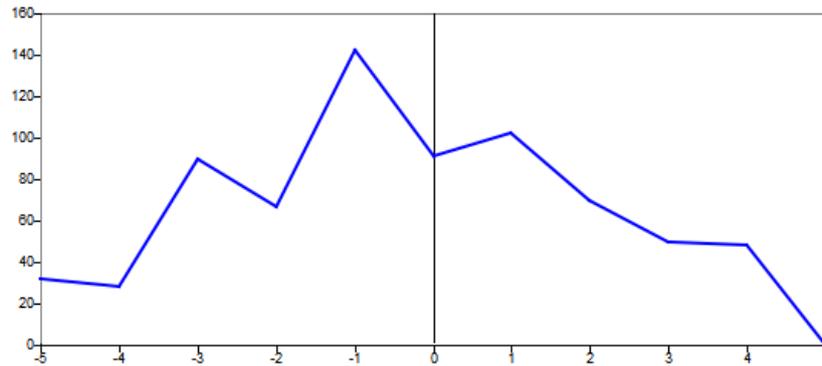
```
32 28 90 67 142 91 102 70 50 48 0
```

For an N -dimensional input, the `xcorr` function returns a $(2N - 1)$ vector. In this case, N is 6 and therefore the output has $2 * 6 - 1$ elements, which is equal to 11. This output is an estimate of correlation between the variable x and shifted/lagged copies of the variable y . Computing the lags and plotting them with the cross-correlation values:

```
lags = -length(x)+1:length(x)-1;
```

```
plot(lags, Rxy);
```

The highest similarity occurs at lag = -1:



Which means that a delayed copy of y in 1 unit is the most similar to x , which can be verified visually:

$$\begin{array}{r}
 x = [8 \ 3 \ 7 \ 2 \ 4 \ 6] \\
 \quad \downarrow \downarrow \downarrow \downarrow \\
 y = [0 \ 8 \ 3 \ 7 \ 2 \ 4]
 \end{array}$$

6) After declaring the signal:

```
x = [8 3 2 1 5 8 3 2 1 4 8 3 2 1 7 8 3 2 1];
```

`xcorr` function calculates the autocorrelation when only 1 input is given:

```
Rxx = xcorr(x);
```

Whose output is a 1 x 37 vector:

```
Rxx = [Matrix] 1 x 37 Row[1] Columns[1:25]
```

```
8 19 32 78 93 66 75 111 191 161 113 122 179 282
240 169 180 272 402 272 180 169 240 282 179
```

```
Row[1] Columns[26:37]
```

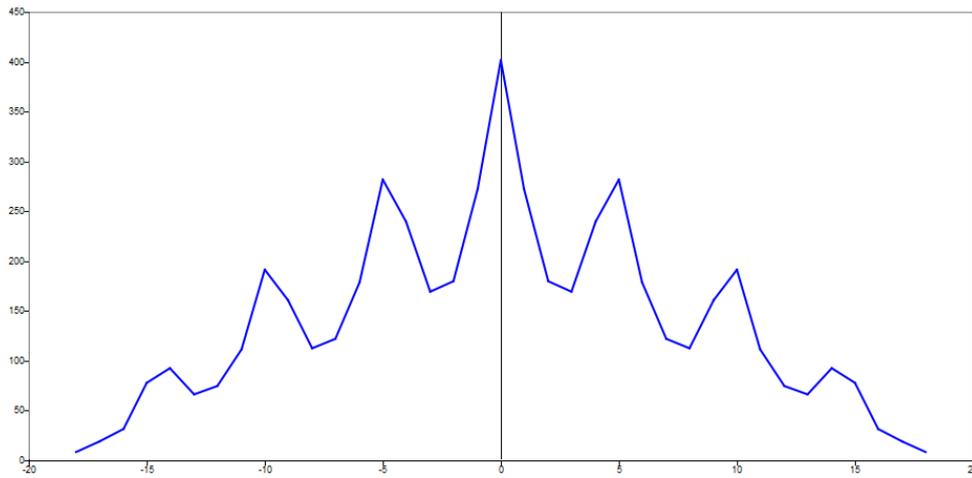
```
122 113 161 191 111 75 66 93 78 32 19 8
```

Computing the lags and plotting them with the cross-correlation values:

```
lags = -length(x)+1:length(x)-1;
```

```
plot(lags,Rxx);
```

The highest similarity occurs at lag = -0:



This is expected because the same values are compared with each other. As the signal starts to shift, a reduction in correlation occurs and the next peak of the autocorrelation function occurs when the delay is equal to 5, 10 and 15, because the signal has a pattern of repetition in every 5 samples:

[8 3 2 1 5 8 3 2 1 4 8 3 2 1 7 8 3 2 1]

Which are the exact lags where the autocorrelation becomes stronger.

7) After declaring the signals:

```
f = [1 5 0 1 -2];
```

```
g = [3 4];
```

`conv` function calculates the convolution:

```
y = conv(f,g);
```

Whose output is:

```
y = [Matrix] 1 x 6
     3  19  20  3  -2  -8
```

The meaning of the output depends on the input:

- If f and g are vectors, the output is the amount of overlap of g as it is shifted over f
- If f and g are vectors of polynomial coefficients, the output is equivalent to the multiplication of these two polynomials.

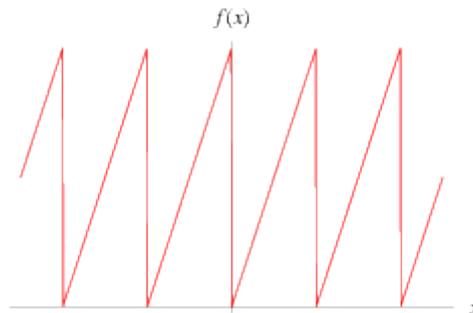
Chapter 6

1) For the square wave (shown in example at 6.2), its simplified formulas:

$$f(t) = \sum_{n=-N}^N A_n e^{\frac{2\pi i n t}{T}}$$

$$A_n = \int_0^1 f(t) e^{-\frac{2\pi i n t}{T}} dt$$

Must be adapted for the sawtooth wave, knowing that $T = 1$:



$$\text{At } t = 0: f(t) = 0$$

$$\text{At } t = 1: f(t) = 1$$

Therefore, from 0 to 1, the function is equal to $f(t) = t$ and the coefficient A_n becomes:

$$A_n = \int_0^1 t e^{-\frac{2\pi i n t}{T}} dt = \frac{T e^{-\frac{2i\pi n}{T}} (2i\pi n - T e^{\frac{2i\pi n}{T}} + T)}{4\pi^2 n^2}$$

Adapting the A_n variable in the algorithm in example at 6.2, considering 5 harmonics:

```

N = 5;

T = 1;

t = linspace(-1, 1, 10000);

f = 0*t;

for n=-N:1:N

    if (n==0)

        continue;

    end

```

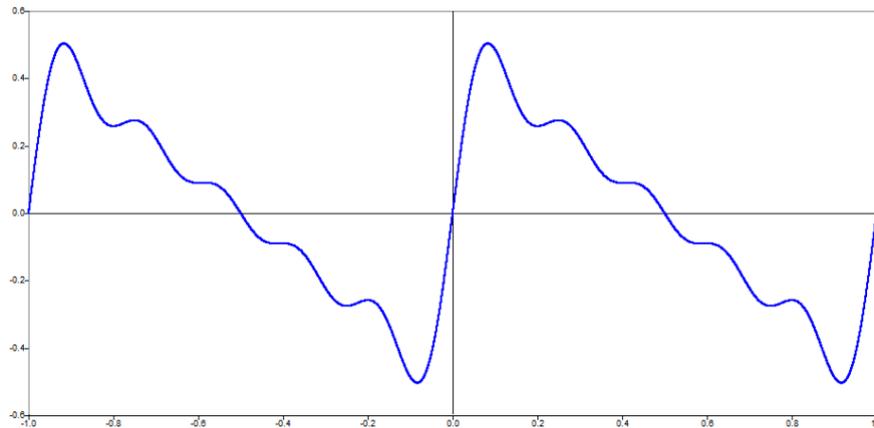
$$A_n = \frac{(T \cdot \exp(-2 \cdot \pi \cdot i \cdot n / T) \cdot (-2 \cdot i \cdot \pi \cdot n / T - T \cdot \exp(2 \cdot i \cdot \pi \cdot n / T) + T))}{(4 \cdot \pi^2 \cdot n^2)}$$

$$f_n = A_n \cdot \exp(2 \cdot \pi \cdot i \cdot n \cdot t / T);$$

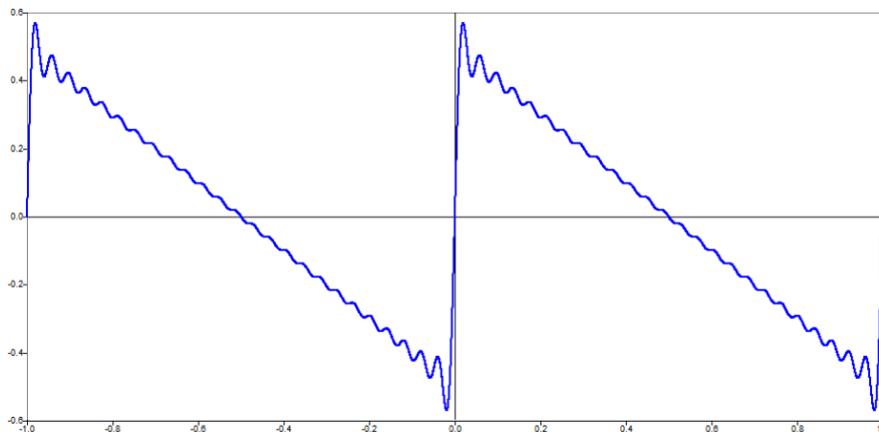
$$f = f + f_n;$$

end

Plot the result with `plot(t, f)` command:



When $N = 25$, i.e. the number of harmonics to be considered is 25:



The approximation becomes more accurate with respect to the wave function reconstructed by the Fourier series.

2) The Fourier series is the representation of signals in terms of sine and cosine waves, that is, a sum of harmonically related sinusoids, which are periodic by nature.

3) Considering a square wave with period $L = 1$ and computing it from 0 to 10:

```

L = 1;

x = 0:0.1:20;

f = zeros(size(x));

N = 5;

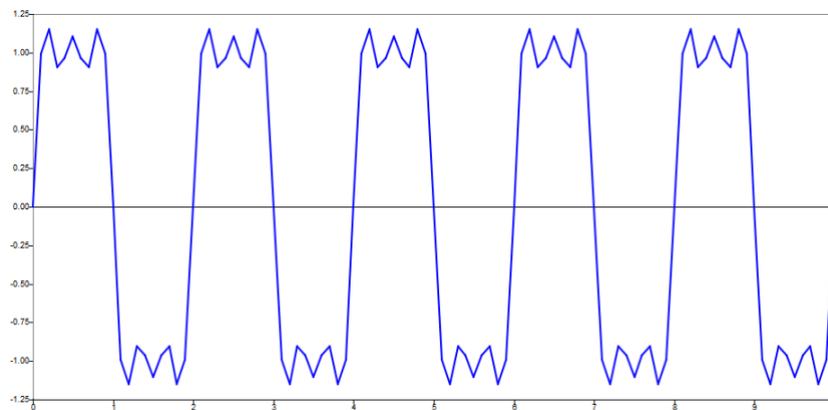
for n=1:2:N

    f = f + (4/pi)*(1/n)*sin(pi*n*x/L);

end

```

Plot the results with `plot(x, f)` command:



4) It is necessary to determine if the vectors are linearly independent, which would characterize them as a basis. Then the system with the form $Ax = B$ must be solved in order to determine the coefficients of the vector such that $a_1v_1 + a_2v_2 + \dots + a_nv_n = \mathbf{0}$

4.1) After declaring the vectors (namely A in system form) and the result (namely B):

```
A1 = [1 0 -1; 2 1 -1; -2 1 4]';
```

```
B = [0 0 0]';
```

The `rank` function obtains the dimension of the vector space generated by its columns, which corresponds to the maximal number of linearly independent columns.

```
R1 = rank(A1)
```

As this result is equal to 3, it means that there are 3 linearly independent columns and therefore the vectors form a basis.

Another way to solve it is using the `linsolve` function, which solves a linear system of equations:

```
X1 = linsolve(A1,B)
```

As the result is:

```
X1 = [Matrix] 3 x 1
0
0
0
```

The only solution is the trivial one and therefore the vectors form a basis.

4.2) Using the same principles described for 4.1:

```
A2 = [1 2 3; 4 5 6; 7 8 9]';
B = [0 0 0]';
R2 = rank(A2)
```

As this result is equal to 2, it means that we have 2 linearly independent columns and therefore the vectors do not form a basis.

Another way to solve it is using `linsolve` function, which solves a linear system of equations:

```
X2 = linsolve(A2,B)
```

In this case, there is a warning:

```
Warning: singular matrix divisor (to within
machine precision) in argument 1 in call to
function linsolve at line number 17 in file
Ex4.com1
```

Which means that the matrix is not invertible and has a determinant equal to zero. In fact, with `det` function the determinant of this square matrix is zero can be verify:

```
det(A2)
```

The determinant of a square matrix is zero if and only when the column vectors are linearly dependent.

5) Considering the Nyquist theorem, the sampling frequency to perform the FFT must be at least twice the highest frequency in the signal, which is 60 in this case. Therefore, the sampling

frequency should be 120 Hz or more. Let's consider 200 Hz to create a time vector with 200 points:

```

Fs = 200;

L = 200;

T = 1/Fs;

t = (0:L-1)*T;

```

The signals may now be created and summed:

```

X1 = sin(2*pi*30*t);

X2 = sin(2*pi*45*t);

X3 = sin(2*pi*60*t);

X = X1 + X2 + X3;

```

Finally, compute the FFT, normalize it according to the length of the signal and plot it:

```

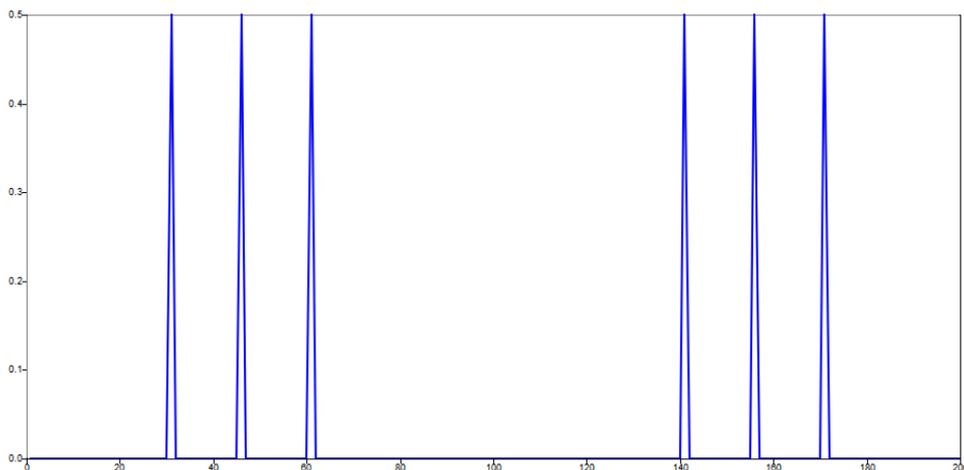
Y = fft(X);

Y = abs(Y/L);

plot(Y);

```

The result is:



It is possible to verify that the most predominant frequencies are 30 Hz, 45 Hz and 60 Hz, as expected, and the respective double-sided spectrum.

6) Reuse the same commands from exercise 5 to compute the double-sided normalized spectrum. Then, consider only half the signal and multiply it by 2 to preserve the amplitude:

```

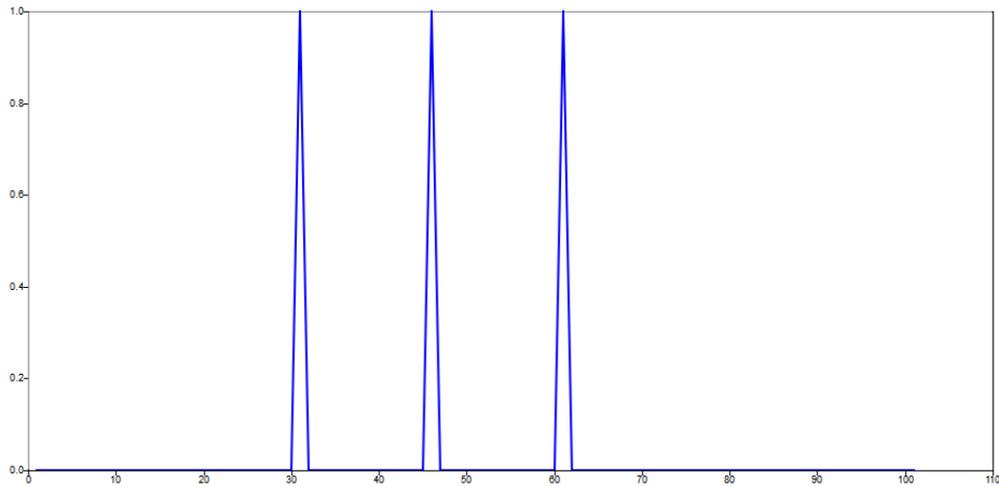
Y_single = Y(1:L/2+1);

Y_single(2:end-1) = 2*Y_single(2:end-1);

plot(Y_single);

```

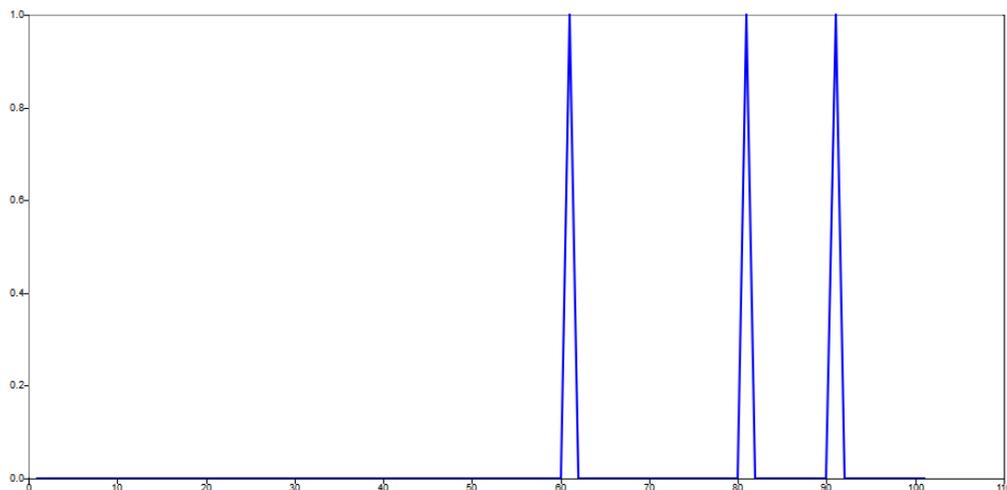
The result is:



It is possible to verify that the most predominant frequencies are 30 Hz, 45 Hz and 60 Hz, as expected.

7) If the sampling frequency is less than 120 Hz, this value is insufficient to capture each peak of the signal and aliasing occurs, according to Nyquist's theorem, and the result will be a false representation of the signal spectrum:

Using $F_s = 100$:



8) First of all, define the chirp signal:

```

Fs = 1024;

t = 0:1/Fs:40;

f0 = 1;

c = 5; % chirpness [Hz/s]

s = sin(2*pi*(c/2*t.^2 + f0*t));

```

And the commands for plotting:

```

figure;

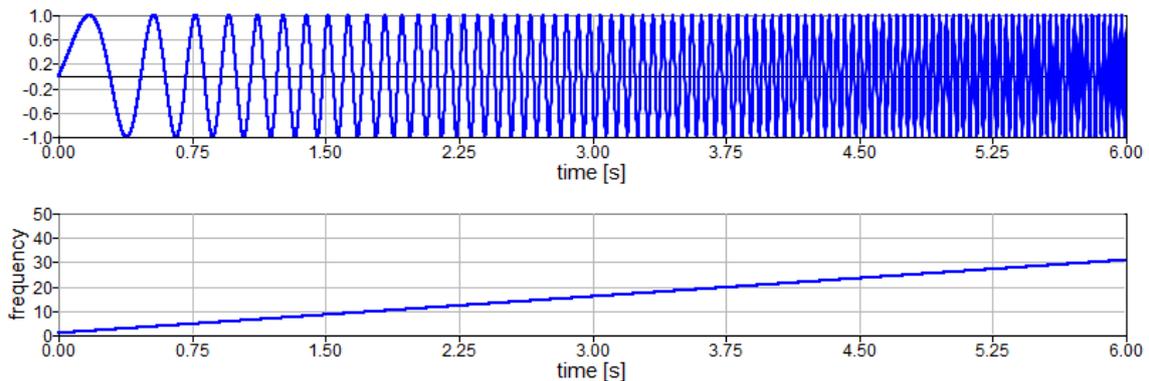
subplot(2,1,1); plot(t,s);

grid on; xlabel('time [s]');

subplot(2,1,2); plot(t,c*t + f0);

grid on; xlabel('time [s]'); ylabel('frequency');

```



Then, define the STFT parameters:

```

SegmentLength = floor(length(s)/80);

OverlapRatio = 0.5;

WindowType = 'hann';

```

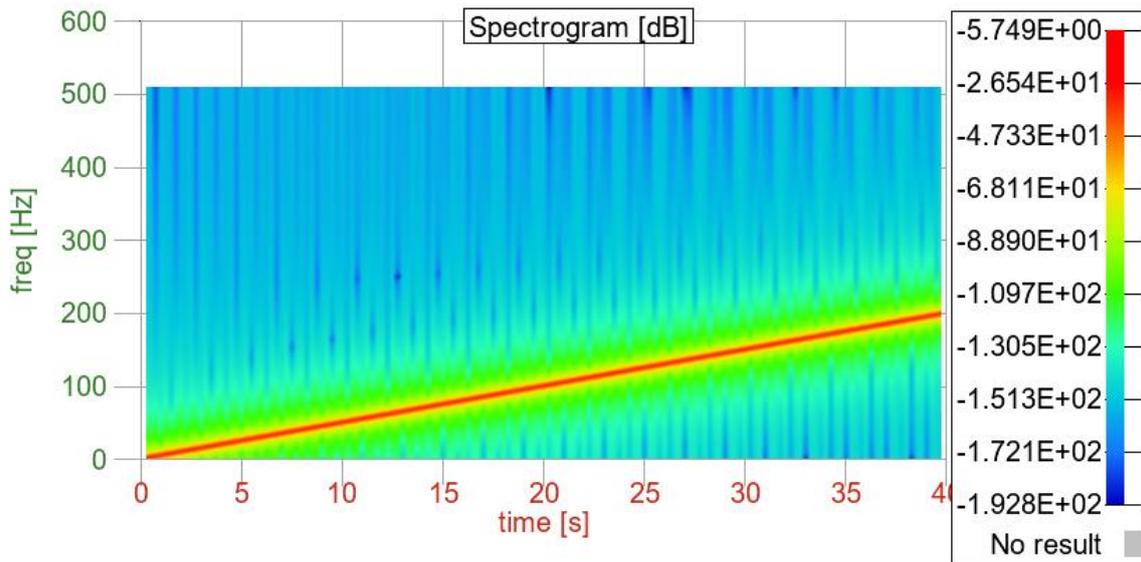
Finally, apply the STFT algorithm, as provided in 6.10:

```

[Ampl,f,t_mean]= ShortTimeFourierTransform(t,s,...

SegmentLength,OverlapRatio,WindowType);

```



Explore changes to the STFT parameters, then re-plot the spectrogram.

9) Modify the function `Windowing` of the algorithm; as provided in 6.10:

```
function s = Windowing(s, WindowType)

    switch WindowType
    case 'hann'
        windowLength = size(s, 2);
        window = hann(windowLength, 'periodic')';

    case 'blackman'
        windowLength = size(s, 2);
        window = blackman(windowLength, 'periodic')';

    case 'rectangular'
        windowLength = size(s, 2);
        window = ones(1, windowLength);

    end

    s = s .* repmat(window, size(s, 1), 1);
end
```

10) Declare the sampling frequency, signal length and create the respective time and frequency vectors based on these parameters:

```
Fs = 64;
```

```
L = 32;
```

```
T = 1/Fs;
t = (0:L-1)*T;
f = [0:Fs/L:Fs-Fs/L];
```

Then, create the signal:

```
X = sin(2*pi*15*t);
```

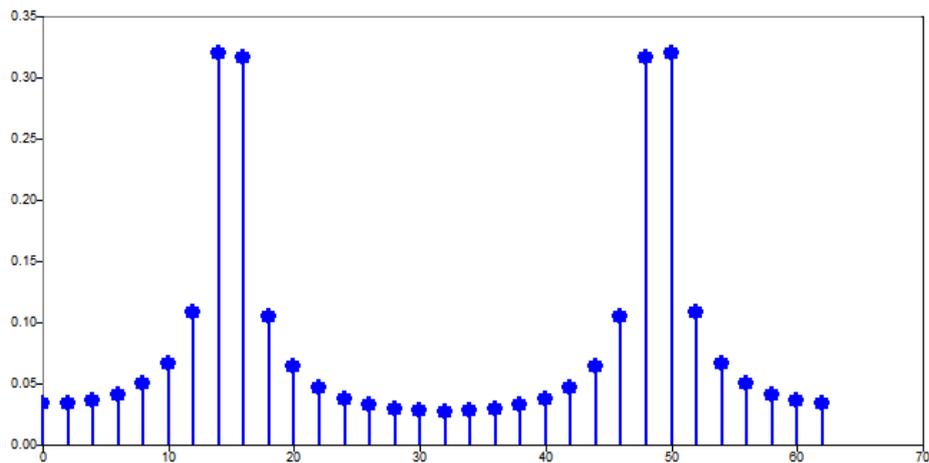
Calculate its FFT:

```
Y = fft(X);
```

Plot the normalized output with `stem` function:

```
stem(f, abs(Y/L));
```

There are 2 prominent frequencies in the plot:

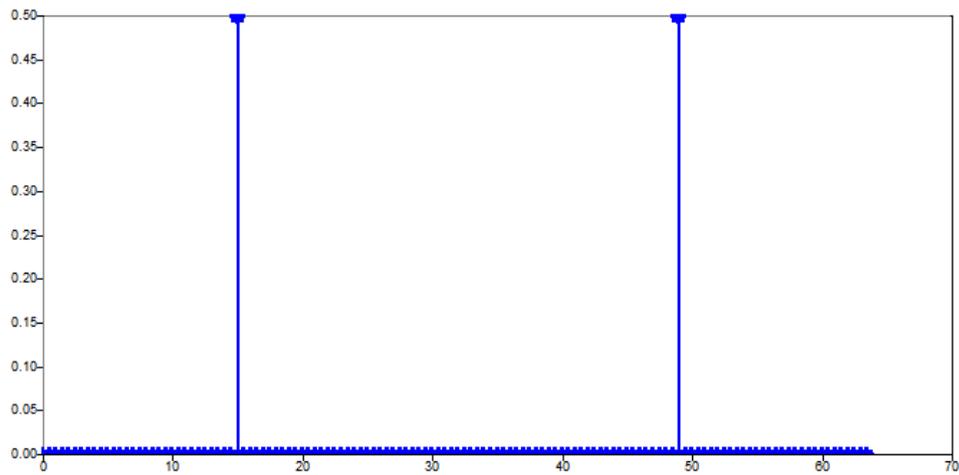


By verifying the frequency vector, the bins do not have the value of 15 Hz, which is the actual predominant frequency of the signal:

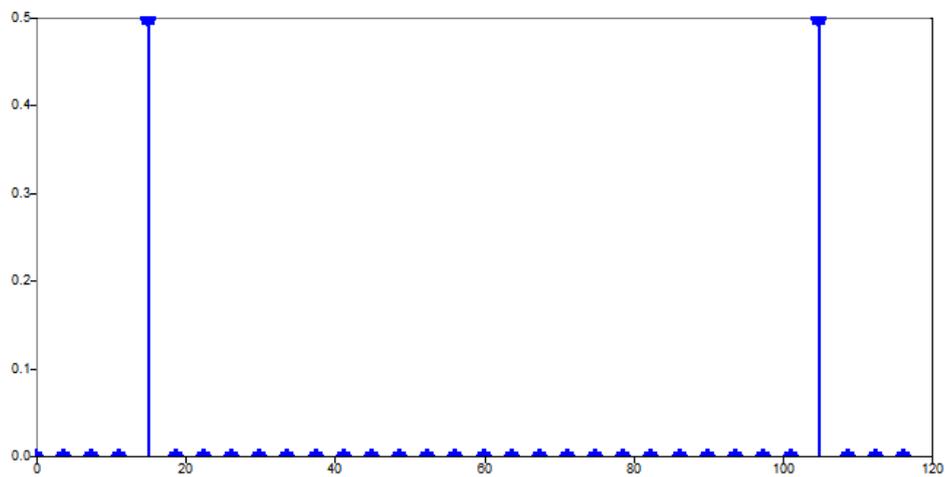
```
f = [Matrix] 1 x 32
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
40 42 44 46 48 50 52 54 56 58 60 62
```

It indicates that the frequency resolution is not appropriate for this signal, leading to leakage.

11) There are different strategies to reduce the leakage of the previous example, such as using a signal length that is capable of capturing the appropriate frequency bins, like $L = 128$:



Or, use a sampling frequency that achieves the same goal, such as $F_s = 120$:



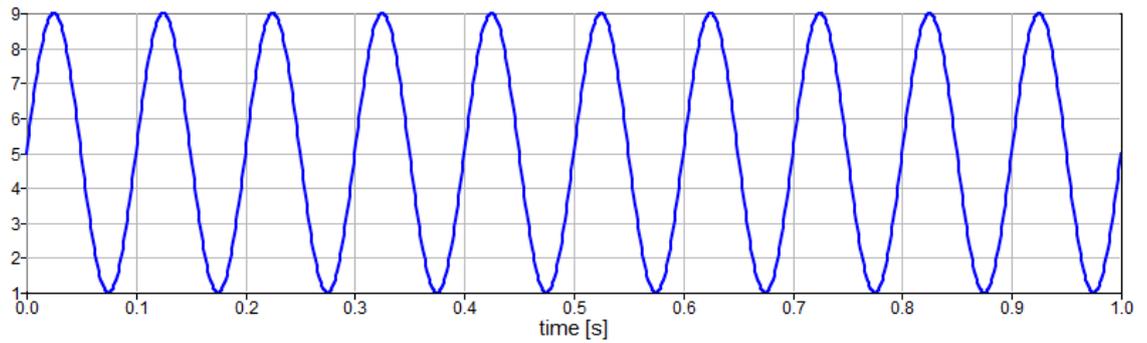
Chapter 7

1) Define the signal:

```

Fs = 1000; %Hz
t = 0:1/Fs:1;
f = 10;
s0 = 5;
s = s0 + 4*sin(2*pi*f*t);
figure();
plot(t,s); grid on; xlabel('time [s]');

```



Then compute the PSD from DFT:

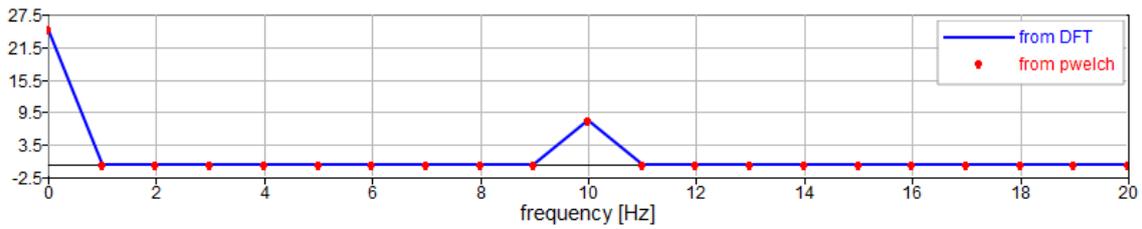
```
DFT = fft(s(1:end-1));
DFT = abs(DFT);
N = length(DFT);
PSD = 1/(Fs*N) * DFT.^2; %double-sided PSD
PSD = PSD(1:ceil(length(PSD)/2)); %folding
PSD(2:end-1) = PSD(2:end-1)*2; %single-sided PSD
Fr = 1/t(end);
f = 0:Fr:(length(PSD)-1)*Fr;
```

Afterwards, compute the PSD leveraging pwelch algorithm with the proper parameters:

```
WindowLength = (length(s)-1); %% only 1 segment!
window = ones(1,WindowLength); %% rectangular window
Overlap = 0; %% no overlap!
nfft = WindowLength;
range = 'onesided'; %% single-sided PSD
[Pxx,frq]=pwelch(s(1:end-1),window,Overlap,nfft,Fs,range);
```

Finally, plot both curves:

```
figure()
plot(f,PSD,'b-',frq,Pxx,'r. ');
grid on; xlim([0,20]); xlabel('frequency [Hz]');
legend({'from DFT','from pwelch'});
```

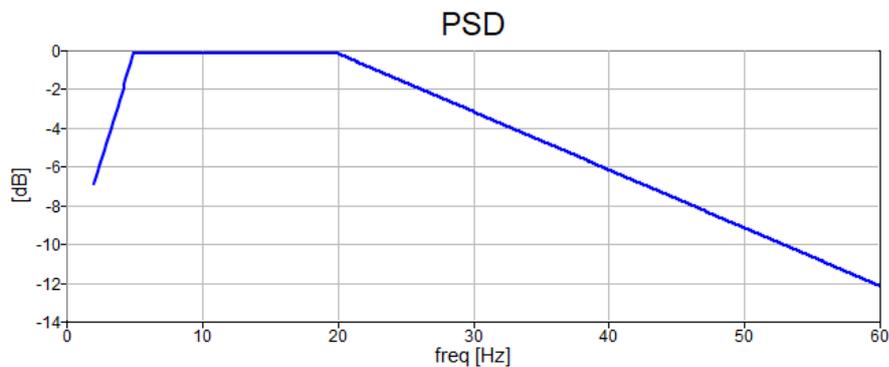


This exercise aims to help the reader become familiar with the `pwelch` algorithm. However, applying the `pwelch` algorithm with this choice of parameters is meaningless.

2) The `pwelch` algorithm is used because it gives a more robust estimation of the PSD.

3) Load and plot the PSD curve:

```
load('.\PSDcurve.mat');
figure(1);
plot(f_curve, PSDcurve_dB);
xlabel('freq [Hz]'); ylabel('[dB]');
title('PSD'); hold on; grid on;
```



Then, set the final time and the frequency range. Let's also sample the PSD curve:

```
%% =====SET FINAL TIME

t_fin = 4;

Fr = 1/t_fin;

%% =====FREQUENCY RANGE

F0 = 4; %Hz
```

```

Fmax = 50; %Hz

Fs = 2*f_curve(end);

t = 0:1/Fs:t_fin;

t = t(1:end-1);

%% =====Sampling PSD

f = 0:Fr:Fs/2;

PSD = interp1(f_curve, PSDcurve_dB, f);

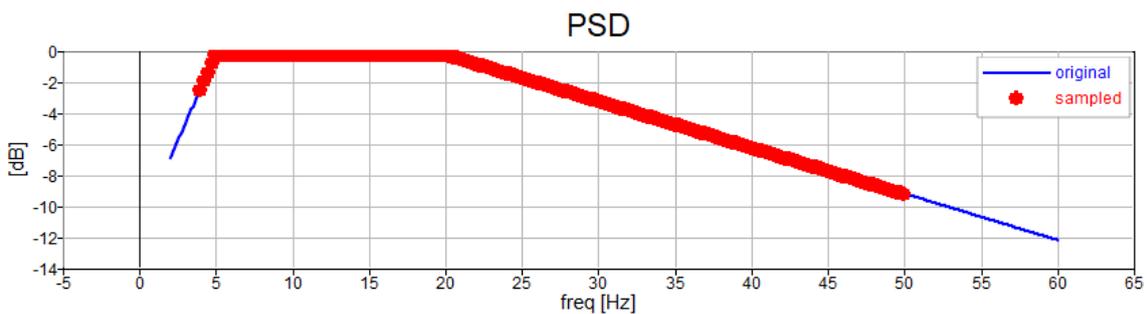
PSD(f<F0) = -inf;

PSD(f>Fmax) = -inf;

plot(f,PSD,'ro');

legend('original','sampled');

```



Retrieve the single-sided and normalized DFT, and add the random phase:

```

%% =====Single Sided and Normalized DFT

Plamp1 = zeros(size(PSD));

PSD = db2mag(PSD);

Plamp1(1) = sqrt(Fr * PSD(1));

Plamp1(2:end) = sqrt(2 * Fr * PSD(2:end));

%% =====Random Phase

Plphase = rand(1,numel(Plamp1))*(2*pi);

```

Move from the single-sided normalized DFT in the amplitude/phase representation to the double-sided DFT in the real/imaginary representation:

```

Plamp1(2:end-1) = Plamp1(2:end-1)/2;

Plreal = Plamp1 .* cos(P1phase);
Plimag = Plamp1 .* sin(P1phase);

P2 = [Plreal + i*Plimag, ...

      conj(flip(Plreal(2:end-1) + i*Plimag(2:end-1)))];

DFT = numel(P2) * P2;

```

Finally, use the `ifft` algorithm to obtain a realization of the random process described by the PSD:

```

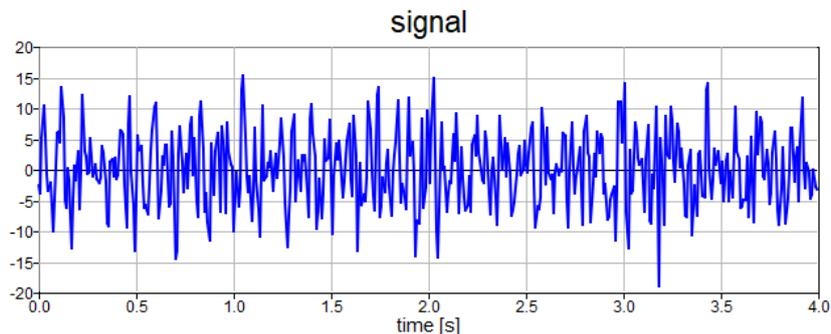
s = ifft(DFT);

figure(2);

plot(t,s);

grid on; xlabel('time [s]'); title('signal');

```



A check can be performed, computing the PSD of the just obtained signal:

```

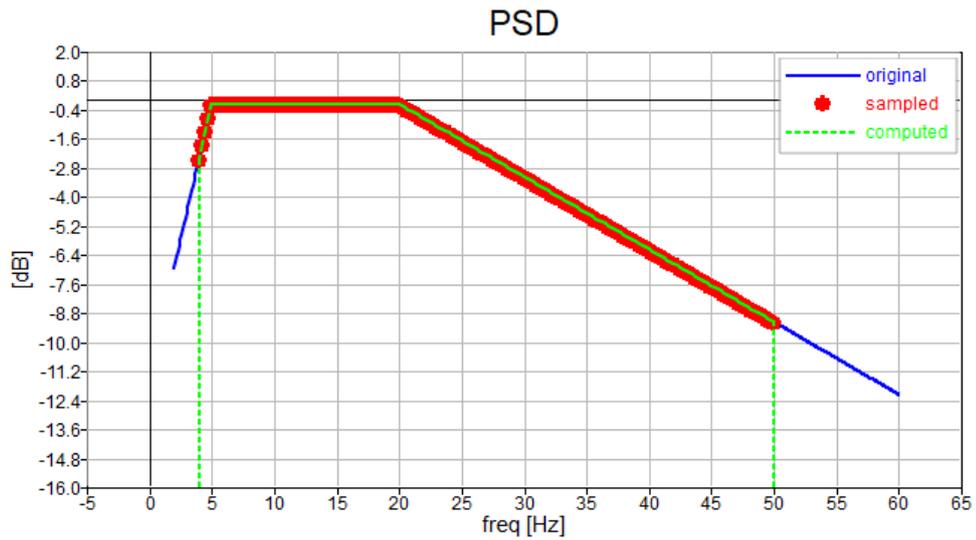
[Pxx,F] = pwelch(s,ones(1,length(s)),0,length(s),Fs);

figure(1);

plot(F,mag2db(Pxx),'g:');

ylim([-16,2]); legend('original','sampled','computed');

```



Chapter 8

1) Define the signal:

```

Fs = 256;

N = 2*Fs;

t = 0:1/Fs:N*1/Fs;

f1 = 5; % [Hz]

f2 = 15; % [Hz]

f3 = 30; % [Hz]

s1 = 10*sin(2*pi*f1*t);

s2 = 6*sin(2*pi*f2*t);

s3 = 1*sin(2*pi*f3*t);

s = s1 + s2 + s3;

```

Then plot it:

```

figure(1);

subplot(4,1,1);

plot(t,s1); grid on; xlabel('time [s]'); title('s1');

subplot(4,1,2);

plot(t,s2); grid on; xlabel('time [s]'); title('s2');

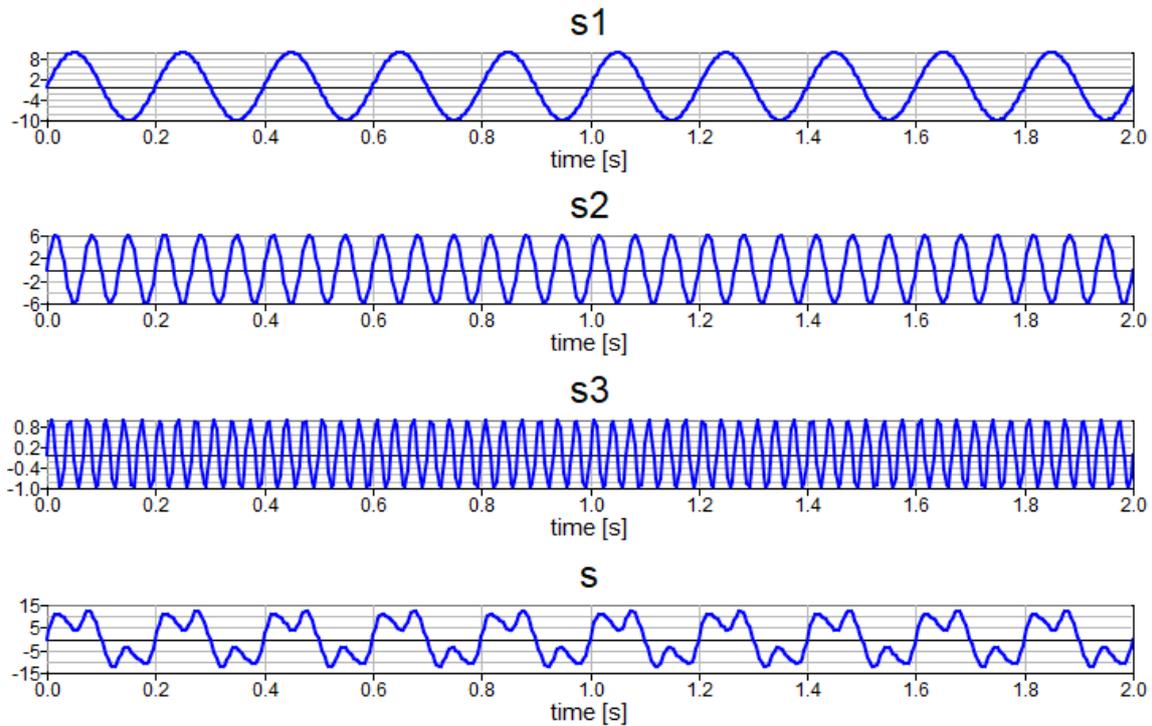
```

```
subplot(4,1,3);

plot(t,s3); grid on; xlabel('time [s]'); title('s3');

subplot(4,1,4);

plot(t,s); grid on; xlabel('time [s]'); title('s');
```



No additional requirements are provided, such maximum and minimum attenuation respectively for the passband and the stopband. Hence there is no need to deploy `buttord` function to determine the order of the filter. The order of the Butterworth filter is arbitrarily set to 6.

Design the Butterworth low pass filter:

```
n_iir = 6;

f_cut = 10; %[Hz]

[b,a] = butter(n_iir, f_cut / (Fs/2), 'low');
```

Compute and visualize its frequency response:

```
f = 0:1/t(end):Fs/2;

h = freqz(b,a,f,Fs);

Ampl_iir = abs(h);

Phase_iir = unwrap(atan2(imag(h),real(h)));
```

```

figure();

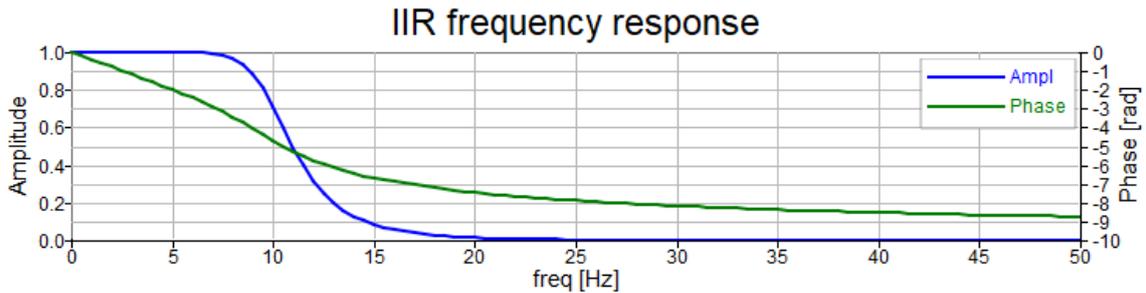
[ax h1 h2] = plotyy(f,Ampl_iir,f,Phase_iir);

xlabel('freq [Hz]');

ylabel(ax(1), 'Amplitude'); ylabel(ax(2), 'Phase [rad]');

legend({'Ampl','Phase'}); title('IIR frequency response');

```



2) Implement causal filtering applying the *filter* function:

```
s_filt = filter(b,a,s);
```

And non-causal filtering, utilizing the *filtfilt* function:

```
s_filtfilt = filtfilt(b,a,s);
```

Then plot everything:

```

figure();

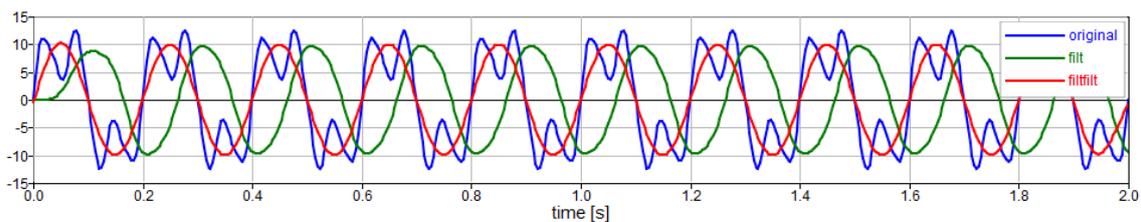
plot(t,s); grid on; hold on; xlabel('time [s]');

plot(t,s_filt);

plot(t,s_filtfilt);

legend({'original','filt','filtfilt'});

```



It shows the delay introduced by causal filtering.

3) Usually the order of the FIR required to perform similarly to an IIR filter is much higher. Let's set it to 150, and design the FIR filter through the window method.

Pick a Hamming window and apply the `fir1` function to design the FIR filter:

```
n_fir = 150;

f_cut = 10; %[Hz]

w = hamming(n_fir + 1, 'symmetric');

b = fir1(n_fir, f_cut / (Fs/2), 'low', w);
```

Compute and visualize its frequency response:

```
f = 0:1/t(end):Fs/2;

h = freqz(b,1,f,Fs);

Ampl_fir = abs(h);

Phase_fir = unwrap(atan2(imag(h),real(h)));

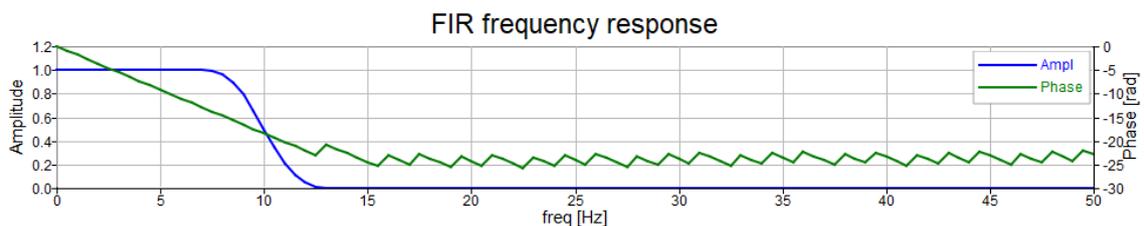
figure();

[ax h1 h2] = plotyy(f,Ampl_fir,f,Phase_fir);

xlabel('freq [Hz]');

ylabel(ax(1), 'Amplitude'); ylabel(ax(2), 'Phase [rad]');

legend({'Ampl','Phase'}); title('FIR frequency response');
```



The phase is linear in the passband (which was not for the IIR filter). Hence, no phase distortion should be introduced (this will be seen in exercise 5).

Apply the filter (both causal and non-causal filtering):

```
s_filt = filter(b,1,s);

s_filtfilt = filtfilt(b,1,s);
```

```

figure();

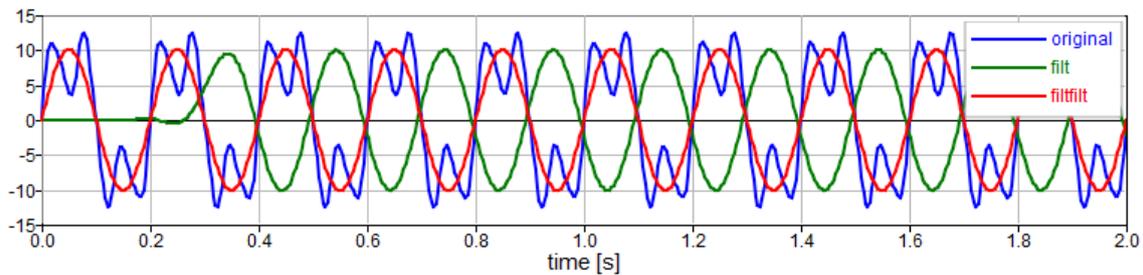
plot(t,s); grid on; hold on; xlabel('time [s]');

plot(t,s_filt);

plot(t,s_filtfilt);

legend({'original','filt','filtfilt'});

```



This time, the delay introduced by causal filtering is much bigger because the filter order is also much higher. Non-causal filtering does not introduce any delay.

4) The procedure is the same as the previous exercises. Only a few lines of code have to be changed.

For the IIR Butterworth high pass filter design:

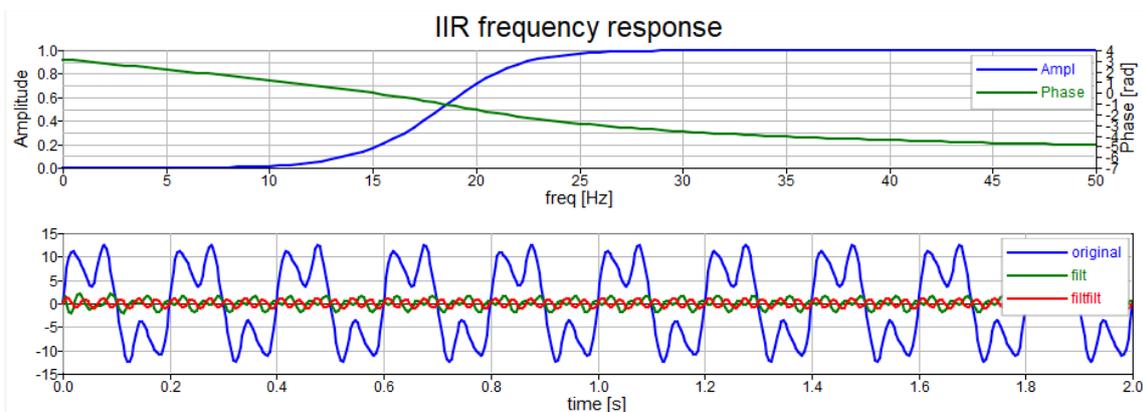
```

n_iir = 6;

f_cut = 20; % [Hz]

[b,a] = butter(n_iir, f_cut / (Fs/2), 'high');

```



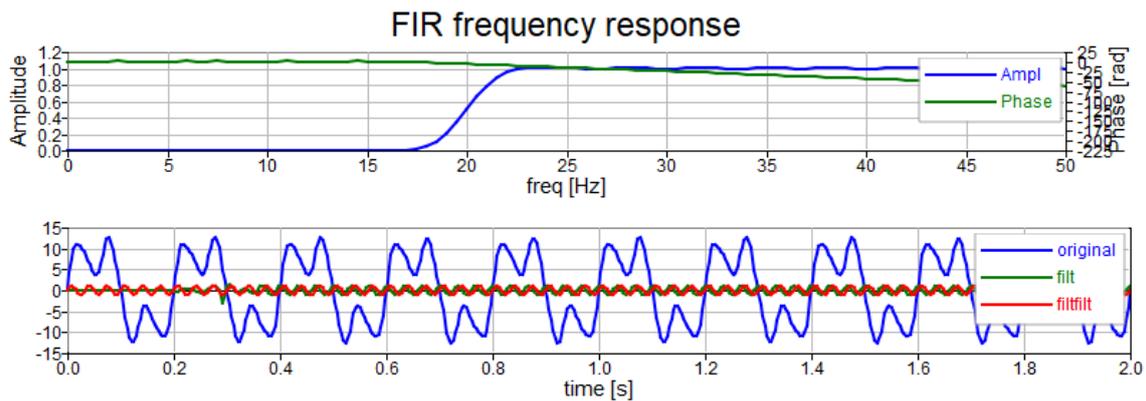
For the FIR high pass filter design:

```
n_fir = 150;

f_cut = 20; %[Hz]

w = hamming(n_fir + 1, 'symmetric');

b = fir1(n_fir, f_cut / (Fs/2), 'high', w);
```

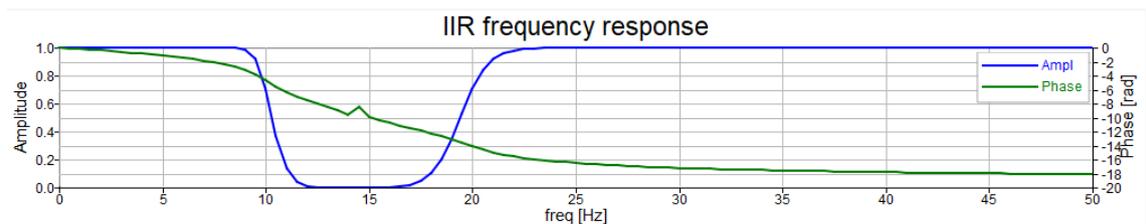


5) The IIR stop-band Butterworth filter:

```
n_iir = 6;

f_stop = [10,20]; %[Hz]

[b,a] = butter(n_iir, f_cut / (Fs/2), 'stop');
```



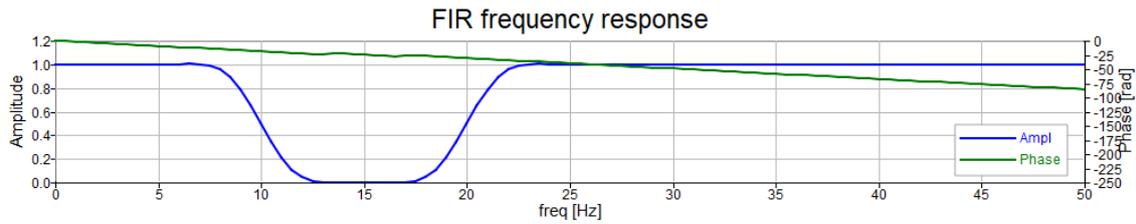
The FIR stopband filter:

```
n_fir = 150;

f_stop = [10,20]; %[Hz]

w = hamming(n_fir + 1, 'symmetric');

b = fir1(n_fir, f_cut / (Fs/2), 'stop', w);
```



Now, implement causal and non-causal filtering with the IIR filter:

```
s_filt = filter(b,a,s);

s_filtfilt = filtfilt(b,a,s);

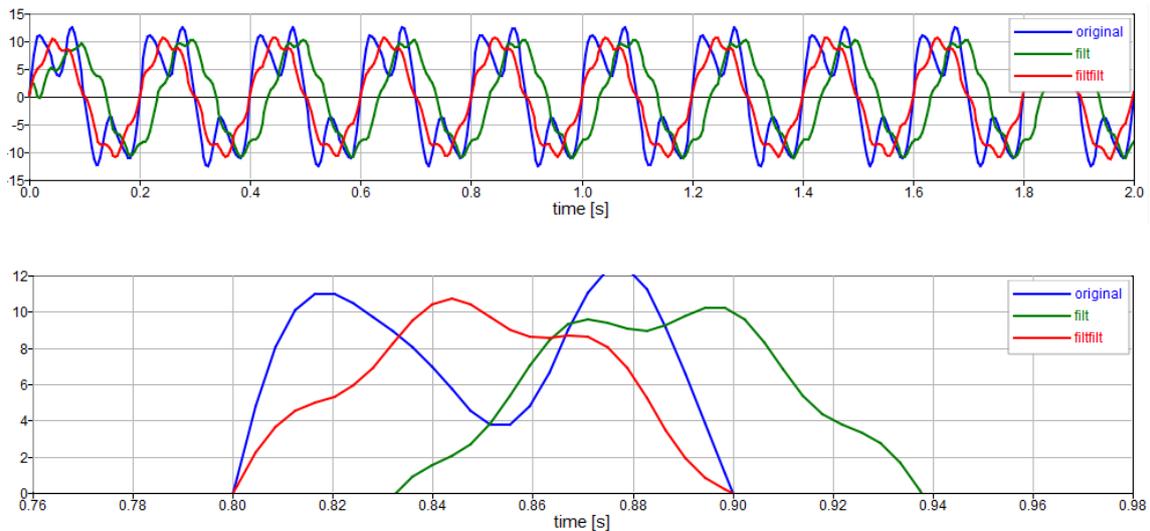
figure();

plot(t,s); grid on; hold on; xlabel('time [s]');

plot(t,s_filt);

plot(t,s_filtfilt);

legend({'original', 'filt', 'filtfilt'});
```



The green curve (causal filtering) is delayed with respect to the red curve (non-causal filtering). It also has a different shape due to phase distortion. Phase distortion is introduced because the phase of the IIR filter is not linear.

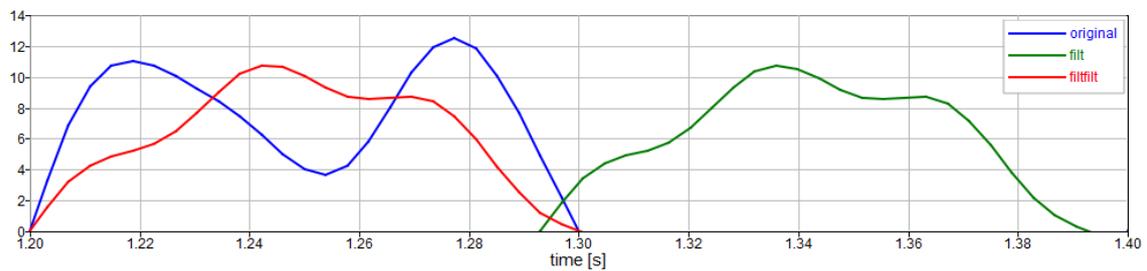
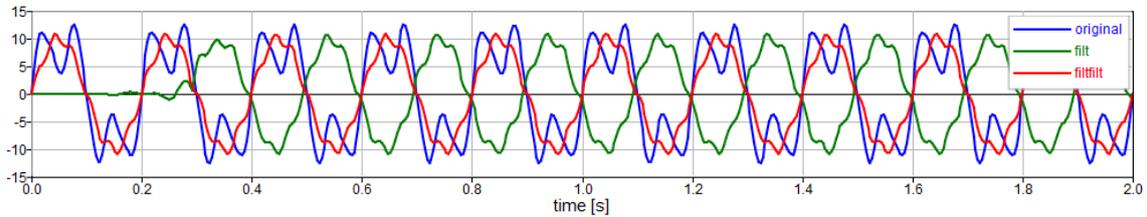
Instead, implement causal and non-causal filtering with the FIR filter:

```
s_filt = filter(b,1,s);

s_filtfilt = filtfilt(b,1,s);

figure();
```

```
plot(t,s); grid on; hold on; xlabel('time [s]');  
  
plot(t,s_filt);  
  
plot(t,s_filtfilt);  
  
legend({'original','filt','filtfilt'});
```



The green curve (causal filtering) is delayed with respect to the red curve (non-causal filtering), but it has the same shape. Hence there is no phase distortion, since the phase of the FIR filter is linear.

Disclaimer

Every effort has been made to keep the book free from technical as well as other mistakes. However, publishers and authors will not be responsible for loss, damage in any form and consequences arising directly or indirectly from the use of this book.

Trademarks

Altair Compose® is a registered trademark of Altair Engineering Inc. MATLAB® is a registered trademark of The MathWorks, Inc. Microsoft® Excel is a product name of Microsoft Corporation in the United States and other countries.

All Python releases are Open Source. Python software and documentation are licensed under the PSF License Agreement (<https://docs.python.org/3/license.html>). Octave is free software under the GNU General Public License (<https://www.gnu.org/software/octave/>).

All other product names, brands, trademarks and registered trademarks are property of their respective owners.

© 2020 **Altair Engineering, Inc.** All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, or translated to another language without the written permission of Altair Engineering, Inc. To obtain this permission, write to the attention Altair Engineering legal department at: 1820 Big Beaver, Troy, Michigan, USA, or call +1-248-6142400.