

Digital signal processing laboratory guide

Fehér Áron

CONTENTS

CHAPTER 1.	INTRODUCTION	PAGE 4.
1.1	The aim of the chapter	4
1.2	Overview of Digital Signal Processing in Embedded Systems	4
	History of Digital signal processors — 4 • Importance of DSP in Modern Applications — 6	
1.3	What to Expect in this Guide	6
	Breakdown of the Modules — 7	
1.4	Learning outcomes	8
1.5	Python as a data analysis and signal processing tool	8
	Preparing the environment — 8 • Signal generation and display — 11 • File-based data management — 12	
CHAPTER 2.	SIGNAL ANALYSIS	PAGE 16.
2.1	The aim of the chapter	16
2.2	The Nyquist-Shannon sampling theorem	16
2.3	Signal domains and their uses	17
2.4	Basic modulation techniques	18
CHAPTER 3.	CONVOLUTION AND CORRELATION	PAGE 21.
3.1	The aim of the chapter	21
3.2	Discrete-time convolution	21
	Theoretical introduction — 21 • Implementation — 22	
3.3	Discrete-time cross-correlation	22
	Theoretical introduction — 22 • Implementation — 22	
CHAPTER 4.	THE DISCRETE FOURIER TRANSFORM	PAGE 24.
4.1	The aim of the chapter	24
4.2	Theoretical introduction	24
4.3	Implementation	27
CHAPTER 5.	THE FAST FOURIER TRANSFORM	PAGE 30.
5.1	The aim of the chapter	30
5.2	Theoretical introduction	30
5.3	Implementation	33

CHAPTER 6.	THE MOVING AND EXPONENTIAL AVERAGE FILTERS	PAGE 35.
6.1	The aim of the chapter	35
6.2	Theoretical introduction	35
6.3	Implementation	38
CHAPTER 7.	FINITE IMPULSE RESPONSE FILTER IMPLEMENTATION USING IDTFT	PAGE 40.
7.1	The aim of the chapter	40
7.2	Theoretical introduction	40
	An overview: — 40 • Linear phase and symmetry: — 41 • FIR filter design using iDFT: — 43	
7.3	Implementation	46
CHAPTER 8.	OPTIMAL FIR FILTER DESIGN	PAGE 47.
8.1	The aim of the chapter	47
8.2	Theoretical introduction	47
	Fundamental concepts of optimisation — 47 • Optimal FIR filter design procedures — 48	
8.3	Implementation	50
CHAPTER 9.	INFINITE IMPULSE RESPONSE FILTER DESIGN AND IMPLEMENTATION	PAGE 53.
9.1	The aim of the chapter	53
9.2	Theoretical introduction	53
	Analog filter types — 54 • Discretisation of analog filters — 58	
9.3	Implementation	62
CHAPTER 10.	DECIMATION AND INTERPOLATION	PAGE 63.
10.1	The aim of the chapter	63
10.2	Theoretical introduction	63
	Decimation — 63 • Upsampling and interpolation — 64 • Polyphase FIR filter — 65	
10.3	Implementaion	66
	Decimation — 67 • Interpolation — 69 • Rational resapling — 70	
CHAPTER 11.	ADAPTIVE FIR FILTERS	PAGE 71.
11.1	The aim of the chapter	71
11.2	Theoretical introduction	71
11.3	Implementaion	73
	Classical filtering of artificial signal — 73 • Adaptive FIR implementation — 73 • Adaptive FIR for audio applications — 74	
CHAPTER 12.	ADAPTIVE IIR FILTERS	PAGE 77.
12.1	The aim of the chapter	77
12.2	Theoretical introduction	77

12.3	Implementaion	78
------	---------------	----

CHAPTER 13	THE HILBERT TRANSFORM	PAGE 80.
13.1	The aim of the chapter	80
13.2	Theoretical introduction	80
13.3	Implementaion	81

CHAPTER 14	TIME-FREQUENCY DOMAIN AND WAVELET DENOISING	PAGE 83.
14.1	The aim of the chapter	83
14.2	Theoretical introduction	83
	Time-Frequency domain analysis — 83 • Wavelets — 84 • Continuous and Discrete Wavelet Transform — 85	
14.3	Implementaion	87

Chapter 1

Introduction

1.1 The aim of the chapter

The objective of this chapter is threefold. Firstly, I will briefly introduce the history of digital signal processing. Secondly, to show the modules we will discuss during this course. Thirdly, to give a simple but concise introduction to setting up and using Python as a data-science test environment.

1.2 Overview of Digital Signal Processing in Embedded Systems

Digital Signal Processing (DSP) is a powerful technique for analysing, modifying, and manipulating signals in the digital domain. It finds applications in various industries, such as telecommunications, audio and video processing, biomedical engineering, and control systems. This lab guidebook introduces the fundamental concepts of DSP and its real-time implementation on embedded systems, specifically using the STM32 microcontroller platform.

In today's digital age, embedded systems play a vital role in implementing DSP algorithms for real-time applications. The STM32 microcontroller family, with its robust computational capabilities and rich ecosystem of development tools, provides an excellent platform for deploying these algorithms efficiently.

Throughout the 13 modules in this guide, you will learn not only the theoretical foundations of DSP but also how to implement them practically on an embedded system, using tools such as STM32CubeIDE, CubeMonitor, and Python for signal generation and analysis.

1.2.1 History of Digital signal processors

Digital Signal Processors are specialised microprocessors designed for efficiently performing mathematical operations on signals, such as audio, video, and communication data in real time. Over the last several decades, DSPs have revolutionised various fields, from telecommunications to multimedia processing and embedded systems. The development of DSPs is a story of rapid technological evolution driven by the demand for real-time signal processing in industries like aerospace, defense, automotive, and consumer electronics.

Before we look at the history of these processors, we should specify what is so special about these in contrast to general-purpose processors. Digital signal processors typically have a Harvard architecture [2, 1], which allows them to fetch new instructions in parallel with data operations. In addition, they use Very Long Instruction Words (VLIW) to specify multiple instructions that are executed in parallel, for instance, a memory read and a multiply-accumulate (MAC). The MAC unit is typically optimised for fixed-point or floating-point arithmetic operations and features saturation and rounding logic. Direct Memory Access (DMA) controllers have automatic looping and demultiplexing, circular buffering, modulo, and bit-reverse addressing.

The concept of Digital Signal Processing can be traced back to the 1960s and early 1970s when advancements in digital electronics and microprocessor technologies paved the way for processing signals in the digital domain. Initially, signal processing tasks were performed using analogue circuits, but digital computing began to emerge as a more flexible and accurate approach.

Digital Signal Processors

The advent of digital computers made it possible to process signals numerically, but early general-purpose computers lacked the speed and specialised architecture necessary for real-time signal processing. The processing power of general-purpose microprocessors limited the earliest attempts at DSP. The need for high-speed, real-time computations in applications like radar, telecommunications, and audio led to the development of dedicated processors.

To meet the need for high-speed calculations, specialised hardware that could handle operations like convolution, filtering, and Fourier transforms more efficiently than general-purpose CPUs were introduced. The first steps toward dedicated DSP hardware involved customising microprocessor architectures to handle tasks such as MAC more efficiently.

The 1980s marked the true beginning of the Digital Signal Processor industry, with two key players emerging: Texas Instruments (TI) and Analog Devices (ADI). These companies laid the groundwork for modern DSP technology and are still industry leaders.

1982, TI introduced the TMS32010, widely regarded as the first commercially successful DSP. The TMS32010 featured a 16-bit architecture and could perform a MAC operation in a single instruction cycle, a critical feature for signal processing tasks.

The success of the TMS32010 led to a family of DSPs that evolved significantly over the years. In the 1990s, TI released the TMS320C25, which introduced enhancements like floating-point arithmetic, improving the precision and flexibility of DSP applications. Later, TI expanded its DSP offerings with the TMS320C6000 series, incorporating VLIW architectures to increase performance.

ADI quickly followed TI's lead and established itself as another major player in the DSP industry. In 1986, ADI launched the ADSP-2100, which featured a dedicated hardware multiplier and a Harvard architecture that separated program and data memory for faster access. This design became a hallmark of DSP architecture, emphasising the importance of parallelism and efficient data handling.

SHARC (Super Harvard Architecture Computer): Introduced in the early 1990s, the SHARC family is one of ADI's most famous DSP lines. The SHARC DSPs were known for their powerful floating-point processing capabilities, which are ideal for high-performance audio, imaging, and video applications. The SHARC architecture focused on scalability, supporting multiple parallel operations that significantly boosted processing speeds.

Blackfin: Later, ADI introduced the Blackfin family, which integrated both DSP and microcontroller functions in a single chip, creating a hybrid processor. This combination proved effective in embedded systems, such as automotive systems and mobile devices, where real-time signal processing needed to be coupled with control logic.

Digital Signal Controllers

Digital signal controllers are hybrid devices that combine the features of a traditional DSP and an MCU. A DSC aims to provide real-time signal processing capabilities and general-purpose control functions in embedded systems.

Beyond Texas Instruments and Analog Devices, companies like Microchip and xMOS entered the DSP market, each contributing unique innovations to DSP technology.

In the early 2000s, Microchip introduced the dsPIC series, combining DSP capabilities with a traditional microcontroller architecture. The dsPIC family aimed to provide embedded systems designers with a versatile platform to handle control and signal processing tasks.

dsPIC30F: The first generation of dsPIC processors offered features such as dedicated DSP instructions, hardware MAC units, and support for fixed-point arithmetic. These features enabled designers to implement digital filters, FFTs, and control algorithms for real-time applications like motor control and power conversion.

The dsPIC architecture found widespread use in applications requiring both signal processing and control, such as automotive systems (e.g., anti-lock braking systems), audio processing (e.g., equalisers), and industrial automation.

In the mid-2000s, xMOS introduced the xCORE architecture, a unique approach to real-time DSP. xCORE is a multi-core microcontroller with deterministic timing and parallel processing capabilities, which is particularly well-suited for audio and control applications requiring tight timing guarantees.

The xCORE architecture allows multiple cores to operate independently from each other while sharing access to peripherals. This parallelism, along with its deterministic execution, makes xMOS processors ideal for real-time audio processing, voice recognition systems, and industrial control.

While dedicated DSP chips dominated the field, general-purpose microprocessors also started incorporating DSP-like capabilities. One of the most significant developments in this area was the introduction of DSP extensions and floating-point support in ARM processors.

In the early 2010s, ARM introduced DSP extensions in its Cortex-M microcontrollers, which are widely used in embedded systems. These DSP extensions added specialised instructions for tasks like MAC operations, saturated arithmetic, and single-cycle multiply-accumulate, making ARM processors competitive for signal processing tasks.

Some ARM processors, such as the Cortex-M4F and Cortex-M7F, feature built-in floating-point units (FPUs, in the case of M4F FPUs, are single precision, while M7F incorporate double precision FPUs), allowing them to handle more complex mathematical operations. This advancement made ARM processors suitable for audio processing, motor control, and other real-time DSP applications that previously required dedicated DSP chips.

ARM also introduced hardware filter accelerators that offload the computation of digital filters (e.g., FIR and IIR) from the main processor core, further enhancing embedded systems' real-time signal processing capabilities without the need for separate DSP hardware.

Digital Signal Processing cores in Reconfigurable Logic

As Field-Programmable Gate Arrays (FPGAs) became more powerful, they began incorporating DSP cores to offer flexible and customisable DSP solutions. FPGAs are reconfigurable devices that can be programmed to implement specific hardware functions, including DSP algorithms.

In the late 1990s and early 2000s, FPGA manufacturers like Xilinx and Altera (now part of Intel) introduced dedicated DSP blocks in their FPGA architectures. These DSP blocks are hardware units designed to perform high-speed MAC operations, making FPGAs highly suitable for DSP tasks such as filtering, modulation, and video processing.

Xilinx DSP48: Xilinx introduced the DSP48 block, a dedicated slice that can perform a wide range of arithmetic functions, including multiply-accumulate, which is critical for DSP algorithms. These blocks allow FPGAs to handle massive parallelism, making them ideal for high-throughput signal processing applications.

Altera DSP Blocks: Altera also introduced DSP blocks in its Stratix and Arria FPGA families, offering similar capabilities for high-speed arithmetic operations in signal processing.

1.2.2 Importance of DSP in Modern Applications

Digital Signal Processing is everywhere! It is the backbone of numerous technologies we interact with daily. Here are some real-world examples and applications where DSP plays a critical role:

- **Telecommunications:** In mobile phones, DSP is used for data compression, modulation, error detection and correction, and more.
- **Audio and Music Processing:** DSP enables efficient sound processing from noise cancellation in headphones to audio compression in music files (MP3).
- **Image Processing:** DSP is used for tasks like filtering, image enhancement, and feature extraction in cameras and medical imaging.
- **Control Systems:** DSP algorithms, such as filtering and feedback control, are used in robotics, automotive systems, and industrial automation.

By the end of this course, you will have the skills to implement DSP techniques directly onto embedded hardware and understand how to interface with real-world systems.

1.3 What to Expect in this Guide

This guidebook consists of 13 practical, hands-on modules, each building upon the previous one to cover various essential DSP concepts. The lab sessions in this guide have been carefully designed to complement the theoretical content covered in lectures. By engaging in hands-on exercises, students will gain practical experience in implementing DSP algorithms, analysing signals, and solving real-world problems using Python. Each lab focuses on a specific aspect of DSP, such as frequency domain analysis and synthesis, filtering techniques, signal reconstruction and advanced time-frequency analysis methods.

1.3.1 Breakdown of the Modules

- **Module 1 - Introduction:** In this introductory module, students will get an introduction to Python as the main tool for DSP, covering essential packages (Numpy, SciPy, Matplotlib, Pandas, etc.) and Jupyter Notebooks. Activities involved: installing packages, generating basic signals, plotting these signals, saving and reading data to/from files.
- **Module 2 - Basic signal analysis:** In this module, students will understand how signals can be represented in both time and frequency domains and explore the sampling theorem. Activities involved: generating sampled signals, analysing aliasing effects, and visualising the spectrum of the signals. The focus will be on understanding the trade-offs between time and frequency representation.
- **Module 3 - Convolution and correlation:** Students will explore the two fundamental operations in DSP: convolution (used for filtering) and correlation (used for signal matching). Activities involved: implementing and visualising these core algorithms in Python and applying these techniques to signal smoothing and pattern detection.
- **Module 4 - Discrete Fourier Transform:** This module delves into the Discrete Fourier Transform (DFT) and its inverse, covering its theoretical basis and practical applications. Activities involved: Gaining a deep understanding of DFT/IDFT by implementing them from scratch; studying the mathematical foundations and the role of transforming signals from the time domain to the frequency domain and back.
- **Module 5 - Fast Fourier Transform:** Building on the DFT, students will learn how to implement the Fast Fourier Transform (FFT) and the inverse (IFFT) algorithm on STM32. The FFT is an optimised version of the DFT that allows efficient computation, which is crucial for real-time applications. Activities involved: FFT implementation, performance analysis, and comparison with built-in methods.
- **Module 6 - Moving Average and Exponential Average Filters:** This module introduces simple smoothing techniques - the moving average filter and the exponential average filter. Activities: Implement moving average and exponential average filters in Python. Compare the effectiveness of each filter in reducing noise in signals and explore their use in real-time applications.
- **Module 7 - FIR filter design and implementation:** Students will design Finite Impulse Response (FIR) filters using Python, applying windowing and optimal methods by reusing the functions from Module 4. Activities: Use windowing methods (e.g., Hamming, Blackman) to design FIR filters and visualise their frequency response. Implement these filters to clean up noisy signals and analyse the impact of filter length, symmetry and tap parity.
- **Module 8 - Optimal FIR filter design:** Students will design Finite Impulse Response (FIR) filters using Python, applying windowing and optimal methods by reusing the functions from Module 4. Activities: Use Python libraries to design filters that meet specific performance criteria (e.g., minimal ripple, stopband attenuation). Explore the trade-offs between filter complexity and performance.
- **Module 9 - IIR Filter Design and Implementation:** This module covers the design and implementation of Infinite Impulse Response (IIR) filters. Students will use Python, focusing on computational efficiency and stability. Activities: Design and implement IIR filters using Python. Analyse their frequency response and stability. Compare IIR filters to FIR filters in terms of performance and computation efficiency.
- **Module 10 - Adaptive FIR Filters and Noise Cancelling:** This module focuses on adaptive FIR filters, where filter coefficients are updated mid-filtering using a predefined error measure optimisation. Activities: Implement an adaptive FIR filter using the L1, L2, and Linf norm-based algorithms. Apply the filter to noise-cancellation scenarios and explore real-time applications of adaptive filtering.
- **Module 11 - Adaptive IIR Filters and System Identification:** Students will build and tune an adaptive IIR filter on a constant and variable parameter system. This module explains how adaptive algorithms are used to model unknown systems based on their input-output behaviour, e.g. echo cancellation or reverb modelling. Activities: Implement an adaptive IIR filter in Python and use it to model unknown systems. Compare the performance of adaptive FIR and IIR filters for system identification tasks.

- **Module 12 - Decimation, Interpolation, Polyphase filtering:** In this module, students will explore techniques for downsampling and upsampling of signals. They will implement these operations in Python, learning how they are used in multirate DSP systems. Activities: Implement decimation and interpolation algorithms in Python. Explore polyphase filtering techniques to improve the efficiency of multi-rate processing systems.
- **Module 13 - The Hilbert Transform:** This module introduces the Hilbert Transform, a technique used to generate the analytic signal, providing information about the signal's envelope and instantaneous phase. Students will implement the Hilbert Transform in Python and analyse its applications in communication systems. Activities: Implement the Hilbert Transform in Python, visualise the analytic signal, and extract the amplitude envelope and instantaneous frequency from modulated signals.
- **Module 14 - Wavelet Transform:** In the final module, students will learn about Wavelet Transforms and how they are used for denoising signals. They will implement wavelet-based denoising and scalograms, exploring the applications in removing noise while preserving essential signal features. Activities: Implement the Continuous and Discrete Wavelet Transform (CWT/DWT) in Python. Use wavelets to analyse signals with non-stationary characteristics and apply wavelet-based denoising techniques.

1.4 Learning outcomes

By the end of these 14 modules, students will:

- Gain a comprehensive understanding of time and frequency domain signal processing techniques.
- Develop proficiency in implementing various filters and transforms on in Python.
- Learn advanced signal processing techniques such as adaptive filtering, system identification, and wavelet transforms.
- These modules will prepare students for working in industries requiring real-time signal processing, such as telecommunications, audio engineering, and embedded systems.

1.5 Python as a data analysis and signal processing tool

We begin the first steps to set up our Python environment for signal analysis and processing. We only need a program execution platform (your preferred terminal on Linux, Command Prompt or Power Shell on Windows). We will download the Python interpreter, create a virtual environment, and install the required modules, including the Jupyter Notebook, which we will use as a browser-based IDE and markup platform. Finally, we will learn some basic use cases needed in the upcoming exercises.

1.5.1 Preparing the environment

Open a Terminal and check your Python version using the command `Python - -version`. If it does not return a version or is lower than 3.11, please download the latest stable version from [the official site](#). When installing, you can leave the default install location, but be sure to add the executable to the system path

After the installation, the previous command should display the version if this is your first Python installation. Otherwise, the system will only display the default version. In this case, use the `py -0p` instruction to list all available versions, including their paths.

Next, update the pip package manager using and install the dependency and project manager globally

Running `uv init - -python 3.13` initialises our new project with a specified python version; then we can check the created `toml` file with

Next, install all the required modules using a single-line command

- **numpy:** This is a fundamental package for scientific computations. It includes multidimensional array objects, array operations, tools for linear algebra, statistical operations, etc. You can read more about it on [this documentation](#).

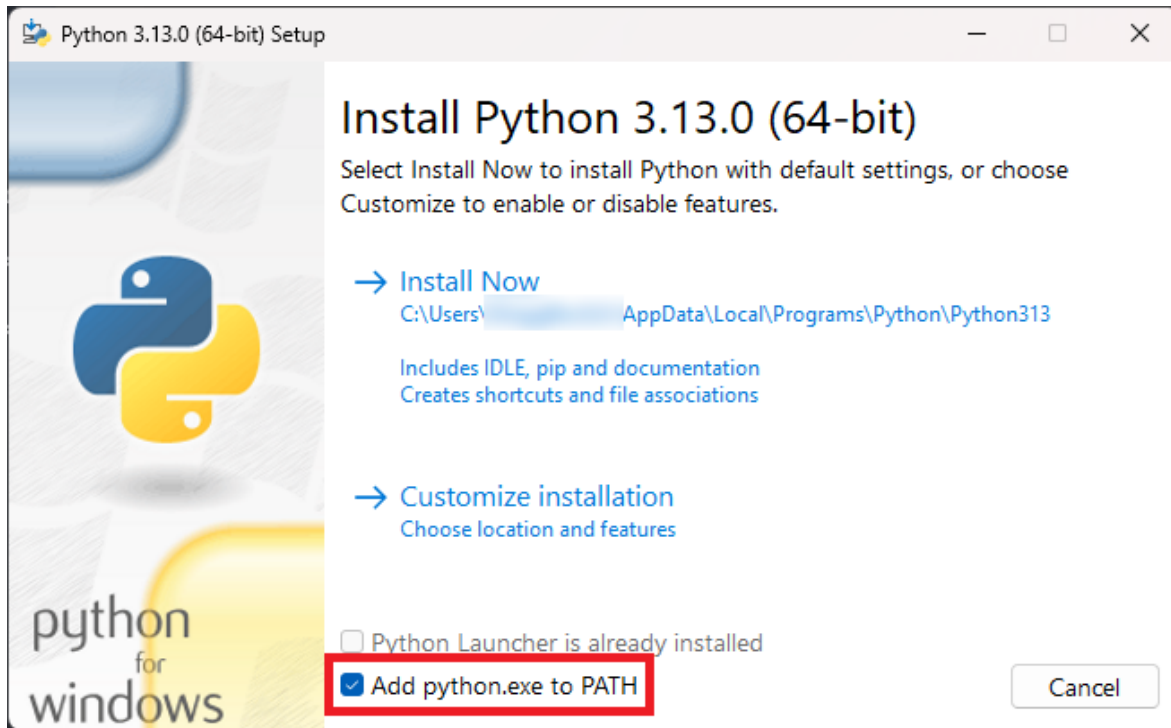



Figure 1.1: Python installation settings

```
P:\Path\To\Executable\python.exe -m pip install --upgrade pip
P:\Path\To\Executable\python.exe python.exe -m pip install uv
```

Figure 1.2: Global update and UV installation

- **scipy**: Provides algorithms for optimisation, integration, interpolation, eigenvalue problems, algebraic equations, differential equations, statistics and many other classes of problems, all written in low-level, optimised languages like Fortran, C, and C++. You can read more about it on [this documentation](#).
- **matplotlib**: This is one of many options for a comprehensive library for creating static, animated and interactive visualisations in Python. [Matplotlib](#) makes easy and hard things possible. Some alternatives are Plotly, Seaborn, Ggplot, and PyQtGraph.
- **pandas**: This is a fast, powerful, flexible and easy-to-use open-source data analysis and manipulation tool built on top of the Python programming language. Check the [documentation](#) for more information.
- **jupyter**: The latest web-based interactive development environment for notebooks, code, and data. Its flexible interface allows users to configure and arrange workflows in data science, scientific computing, computational journalism, and machine learning. A modular design invites extensions to expand and enrich functionality. Check it out [here](#).

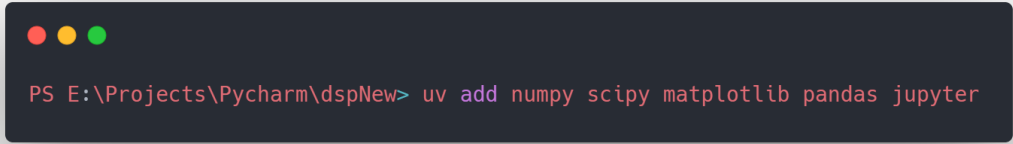


```

PS E:\Projects\Pycharm\dspNew> cat pyproject.toml
[project]
name = "dspnew"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">

```

Figure 1.3: New project setup



```

PS E:\Projects\Pycharm\dspNew> uv add numpy scipy matplotlib pandas jupyter

```


Figure 1.4: Basic module installation

If you check your *toml* file again, every module should be listed inside the dependencies. Executing a single Python script is done using *uv run name.py*. When we created the project, a *hello.py* file was already made; try executing the script.

Finally, let's open our web-based ide by executing *uv run jupyter notebook*. This should open a new window in the default browser. Create a new Jupyter notebook and name it *1_dsp_lab_intro*, paste a test print and execute it:

Let's have a final look at our project structure.

- UV gave us a project that is directly usable with git versioning - we have the *.gitignore* that tells which files/folders should be ignored from versioning. In our case, the local cache (*__pycache__*) directory, any compiled intermediary or executable files (**.py[oc]* refers to *.pyo* and *.pyc* compiled and optimised binaries), any debug or distribution files (*build/*, *dist/*), any downloaded or generated module wheels (*wheels/*), the metadata from python packages and dependencies (**.egg-info*). Since UV automatically builds the environment, the local *.venv* is also ignored.
- The *.python-version* file shows the version of the used Python interpreter.
- We have the *hello.py* Python script that we won't use, so you can delete it.
- The *pyproject.toml* that shows the name and project version, here you can also add a description for this project. This file contains the associated readme file **required** for a properly usable GIT project. The Python interpreter version and the dependencies.



```
import sys

print(f"The python kernel version is {sys.version}")
```

Figure 1.5: Testing the newly created Jupyter Notebook

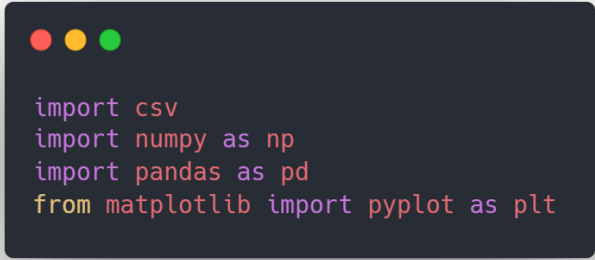
- *Readme.md* is a markdown-formatted text file commonly included in software projects, repositories, and other documentation folders. It serves as an introduction to the project and provides essential information for users and contributors. We will leave it empty, but anytime you create a project, you are advised to create a readme with an introduction, usage and installation instructions, documentation for (potential) contributors, license information, and project dependencies.

1.5.2 Signal generation and display

We have a project setup ready for data analysis and processing. Whenever you want to open this project and run the Jupyter Notebook, UV will activate the virtual environment with all the defined modules. The usual workflow is to import all the used modules in the first section and then have one section for each task/exercise.

Import the required dependencies and generate a 10Hz sine wave with 5000 Hz sampling frequency:

We will use *numpy*, *matplotlib.pyplot*, *csv* - a core module that is already in the interpreter, and *pandas*. Execute



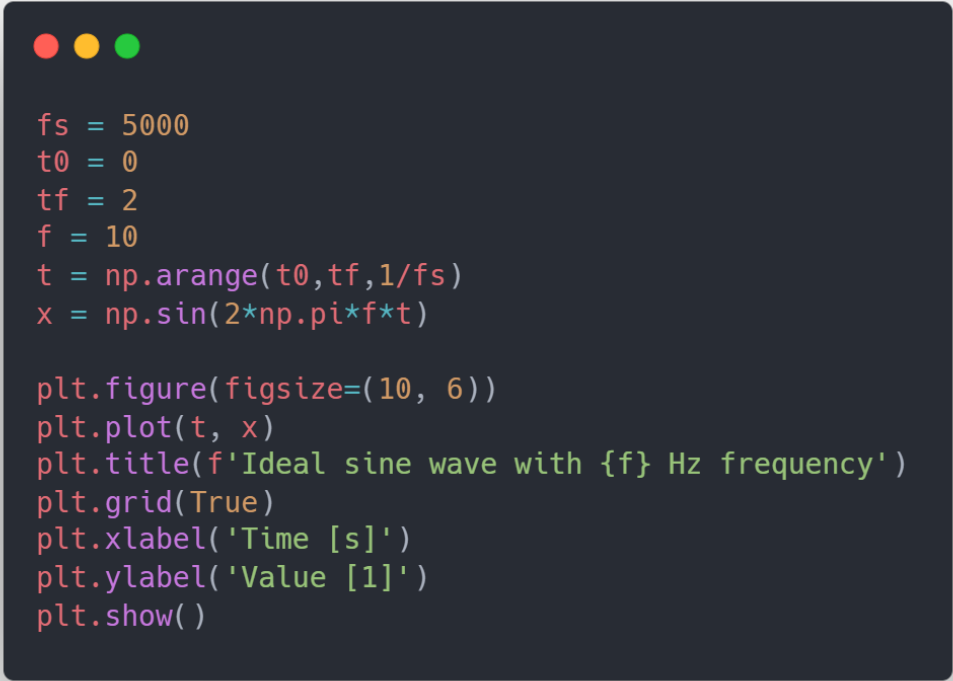
```
import csv
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
```

Figure 1.6: The required modules

it, then generate the sine wave and plot the result

Exercises:

1. Using this knowledge, generate a composite wave containing three ideal sine waves: a 1Hz, a 10Hz and a



```

fs = 5000
t0 = 0
tf = 2
f = 10
t = np.arange(t0,tf,1/fs)
x = np.sin(2*np.pi*f*t)

plt.figure(figsize=(10, 6))
plt.plot(t, x)
plt.title(f'Ideal sine wave with {f} Hz frequency')
plt.grid(True)
plt.xlabel('Time [s]')
plt.ylabel('Value [1]')
plt.show()

```

Figure 1.7: The required modules

100Hz component, with amplitudes 0.6, 0.3, 0.1 respectively. Plot each signal and the composite one on the same plot pane.

2. Find a way to generate a square wave with a fundamental frequency of 10Hz.
3. Generate a sine wave with unit amplitude, where the frequency increases linearly from 1Hz to 100Hz.

1.5.3 File-based data management

In the previous section, we generated the signals and plotted them, but sometimes, we would want to save the generated signal in a user or machine-readable data format. Or we would want to open a file given to us. This file could be a simple text file, a comma-separated file (.csv), JSON, etc. We will remain at the current task's simple .csv files.

Python has a core module that can handle them. Let's use the previously generated composite wave as an example and save it. Usually, one opens the file, reads or writes the data, and closes it. We could do this, but there is a more elegant way: Here, the *with ... as ...* structure is used, which terminates and closes the procedure when we exit the script structure. This can be used for file operation or connection-based data transmissions (like serial port messages or TCP/IP communication). In this case, t is the time array, and z is the composite wave.

Now open the file, read back the values and plot them

Here, I had to read and discard the header line manually. There are more capable tools to manage data files, like Pandas. The same reading procedure using pandas is:

Exercises:

1. Download a WAV file containing your favourite music. Find a way to open it and plot 1s data.
2. Download a recorded ECG signal from the [physionet](#) database. Open it and plot the record using pandas.

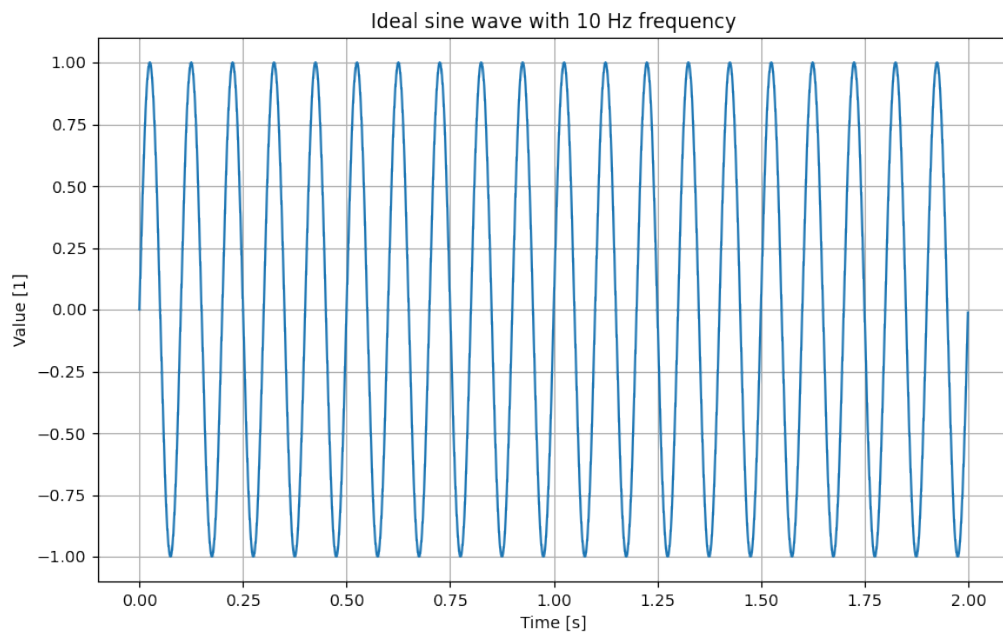


Figure 1.8: The result

```
with open('data.csv', 'w', newline='') as csvfile:
    file_writer = csv.writer(csvfile, delimiter=',')
    file_writer.writerow(['Time', 'Value'])
    for idx in range(len(t)):
        file_writer.writerow([t[idx], z[idx]])
```

Figure 1.9: Write data to CSV using the core module



```
data = []

with open('data.csv', 'r') as csvfile:
    file_reader = csv.reader(csvfile, delimiter=',')

    header = next(file_reader)
    print("Header:", header)

    for row in file_reader:
        data.append([float(row[0]), float(row[1])])

plt.figure(figsize=(10, 6))
data = np.array(data)
plt.plot(data[:,0], data[:,1], label='Summed signal')
plt.title('Composite wave')
plt.grid(True)
plt.xlabel('Time [s]')
plt.ylabel('Value [1]')
plt.legend()
plt.show()
```

Figure 1.10: Read data from CSV using the core module

```
df = pd.read_csv('data.csv')

print("Header:", df.columns.tolist())

data_array = df.to_numpy()

print(f"Shape of data: {data_array.shape}")

plt.figure(figsize=(10, 6))
data = np.array(data)
plt.plot(data_array[:,0], data_array[:,1], label='Summed signal')
plt.title('Composite wave')
plt.grid(True)
plt.xlabel('Time [s]')
plt.ylabel('Value [1]')
plt.legend()
plt.show()
```

Figure 1.11: Read data from CSV using Pandas

Chapter 2

Signal analysis

2.1 The aim of the chapter

This chapter aims to explore the fundamental concepts of signal sampling and analysis, which are pivotal in the field of digital signal processing. This chapter will provide a comprehensive understanding of how continuous-time signals can be converted into discrete-time signals through sampling, and how these sampled signals can be analysed. The practical applications of these concepts in various domains, such as communications, audio processing, and biomedical engineering, will also be discussed.

2.2 The Nyquist-Shannon sampling theorem

The Nyquist-Shannon sampling theorem, a cornerstone in digital signal processing, states that a continuous-time signal can be perfectly reconstructed from its samples if the sampling frequency is at least twice the maximum frequency present in the signal. This critical rate is known as the Nyquist rate. Mathematically, if f_{\max} is the highest frequency of the signal, the sampling frequency f_s must satisfy $f_s \geq 2f_{\max}$.

Consider a continuous-time, band-limited signal $x(t)$ with a maximum frequency component f_{\max} . The sampling process is multiplying the original signal with an impulse train $\delta_{\Delta t}(t)$, where Δt is the sampling period. This gives us the sampled signal $x_s(t)$:

$$x_s(t) = x(t) \cdot \delta_{\Delta t}(t) = \sum_{n=-\infty}^{\infty} x(n\Delta t) \delta(t - n\Delta t) = \sum_{n=-\infty}^{\infty} x[n] \delta(t - n\Delta t)$$

The Fourier transform of $x_s(t)$, denoted as $\hat{x}_s(\omega)$, can be obtained using the convolution property of the Fourier transform

$$\hat{x}_s(\omega) = \frac{1}{2\pi} \hat{x}(\omega) * \hat{\delta}_{\Delta t}(\omega),$$

and the fact that the Fourier transform of an impulse train is also an impulse train:

$$\hat{\delta}_{\Delta t}(\omega) = \frac{2\pi}{\Delta t} \sum_{n=-\infty}^{\infty} \delta\left(\omega - \frac{2\pi n}{\Delta t}\right),$$

The proof of this is left as an exercise for the reader.

Finally, we have

$$\begin{aligned} \hat{x}_s(\omega) &= \frac{1}{2\pi} \hat{x}(\omega) * \hat{\delta}_{\Delta t}(\omega) = \frac{1}{2\pi} \hat{x}(\omega) * \frac{2\pi}{\Delta t} \sum_{n=-\infty}^{\infty} \delta\left(\omega - \frac{2\pi n}{\Delta t}\right) \\ &= \frac{1}{\Delta t} \hat{x}(\omega) * \sum_{n=-\infty}^{\infty} \delta\left(\omega - \frac{2\pi n}{\Delta t}\right) = \frac{1}{\Delta t} \sum_{n=-\infty}^{\infty} \hat{x}\left(\omega - \frac{2\pi n}{\Delta t}\right). \end{aligned}$$

Graphically, this means that after sampling, the magnitude of the spectral content is attenuated by $1/\Delta t$, and there are copied images at 2π intervals:

Exercise:

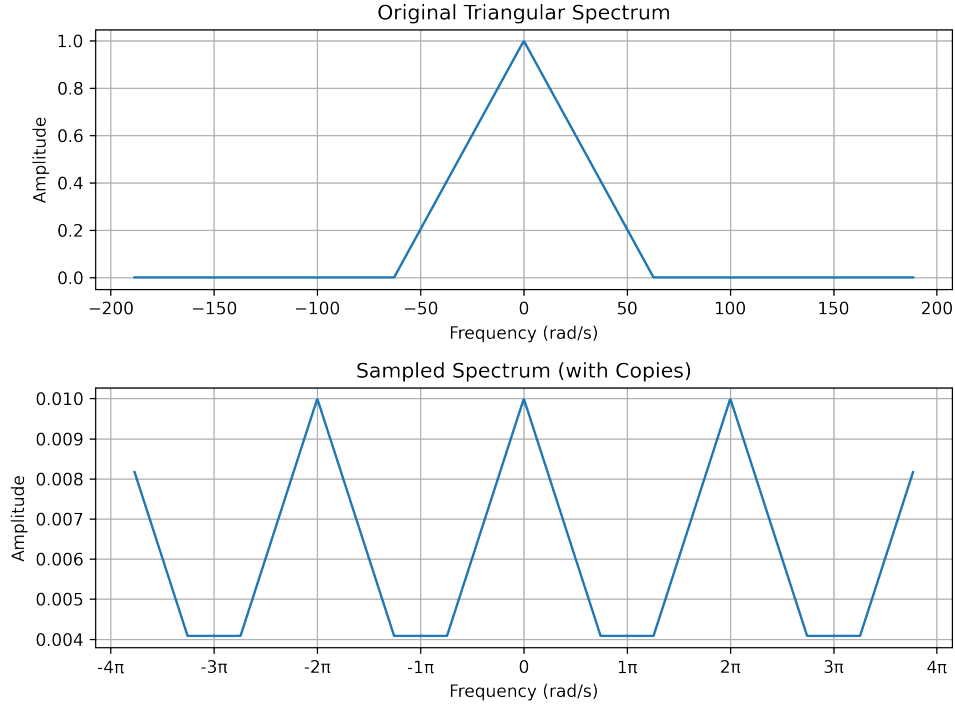


Figure 2.1: Sampling effect on a continuous-time signal.

1. Generate a sine wave with a frequency of 5Hz and a duration of 3s.
2. Sample the wave at varying rates (e.g., 50Hz, 20Hz, 10Hz, 8Hz, 4Hz, and 1Hz).
3. Plot the original and sampled signals to observe differences.
4. What happens when the sampling rate is below the Nyquist limit?
5. Can you identify aliasing effects?

2.3 Signal domains and their uses

Signals can be represented in various domains, offering unique insights and tools for analysis and processing. The primary domains include:

- **Time Domain:** The most intuitive representation, showing signal amplitude as a function of time. It is helpful for analysing temporal characteristics such as amplitude, duration, and waveform shape.
- **Frequency Domain:** By applying the Fourier transform, signals are represented in terms of their frequency components. This domain is crucial for identifying frequency content, filtering, and spectral analysis.
- **Laplace and Z Domains:** These domains extend the analysis to include complex frequency representations, providing system stability and control tools. The Laplace transform is used for continuous-time systems, while the Z transform applies to discrete-time systems.

Regarding the analysis of signals, the Frequency domain is more commonly used. The Fourier transform explicitly maps the temporal domain into the frequency domain. Many practical tasks, like filtering, modulation, and spectrum analysis, naturally involve frequency interpretation, making the Fourier transform a direct and intuitive tool. While the Z Transform underpins many design and analysis methods for discrete systems, it is often used as an intermediate step, with the results frequently interpreted in the frequency domain.

Let two functions $x, \hat{x} \in \mathcal{L}^2(\mathbb{R})$, where \hat{x} is the Fourier transform of x . \hat{x} is a complex-valued function that can be defined as:

$$\hat{x} = \Re(\hat{x}) + j\Im(\hat{x}) = ae^{j\varphi},$$

```

fig = plt.figure(figsize=(10, 8), layout='constrained')
axs = fig.subplot_mosaic([["signal", "signal"], ["magnitude", "log_magnitude"], ["phase", "angle"]])

axs["signal"].set_title("Signal")
axs["signal"].plot(t, x, color='C0')
axs["signal"].set_xlabel("Time [s]")
axs["signal"].set_ylabel("Amplitude [1]")

axs["magnitude"].set_title("Magnitude Spectrum")
axs["magnitude"].magnitude_spectrum(x, Fs=fs, color='C1')

axs["log_magnitude"].set_title("Log. Magnitude Spectrum")
axs["log_magnitude"].magnitude_spectrum(x, Fs=fs, scale='dB', color='C1')

axs["phase"].set_title("Phase Spectrum ")
axs["phase"].phase_spectrum(x, Fs=fs, color='C2')

axs["angle"].set_title("Angle Spectrum")
axs["angle"].angle_spectrum(x, Fs=fs, color='C2')

plt.show()

```

Figure 2.2: Spectrum plot example snippet.

where $a, \varphi \in C^2(\mathbb{R})$ are the magnitude and phase of the Fourier transform, defined as

$$a = \sqrt{\Re^2(\hat{x}) + \Im^2(\hat{x})} \quad \varphi = \text{atan} \frac{\Im(\hat{x})}{\Re(\hat{x})}.$$

Sometimes the magnitude is represented in logarithmic scale, i.e. $a_{\log} = 20 \log_{10} a$, which has [dB] units, as opposed to the linear a that is unitless. The phase can be represented as a continuous function spanning the real numbers or wrapped onto the interval $[-\pi, \pi)$. Since we did not develop our own Fourier transform methods, we will use the built-in functions from *matplotlib*. The *magnitude_spectrum* plots the magnitude of a temporal signal, the *phase_spectrum* is the unwrapped phase, while the *angle_spectrum* is the wrapped one.

The following code snippet shows the use of these functions in a nice contained figure:

Exercise:

1. Create a signal combining frequencies of 10Hz, 100Hz, and 250Hz.
2. Plot the signal in the time domain.
3. Compute and plot the magnitude and phase spectra using.
4. Compute the spectra for a discrete-time Dirac delta.
5. Compute the spectra for a discrete-time Heaviside step.

2.4 Basic modulation techniques

Modulation is the process of varying one or more properties of a carrier signal, such as amplitude, frequency, or phase, to transmit information. Basic modulation techniques include:

- **Amplitude Modulation (AM):** Varies the amplitude of the carrier signal by the message signal. Widely used in radio broadcasting.
- **Frequency Modulation (FM):** Varies the carrier signal frequency based on the message signal. It is known for its robustness against noise and is used in FM radio.
- **Phase Modulation (PM):** Alters the carrier signal phase according to the message signal. Used in digital communication systems. This section will explore these techniques in detail, highlighting their principles, applications, and comparative advantages.

```

carrier_freq = 10000 # Carrier frequency in Hz
bit_rate = 1000 # Bit rate (how many bits per second)
amplitude = 1.0 # Amplitude of the carrier
fs = 100000 # Sampling frequency in Hz
t = np.arange(0, 0.01, 1/fs) # Time vector for 10 ms

# Binary data (modulating signal) - let's assume some random bits for ASK
data_bits = np.random.randint(0, 2, int(bit_rate * len(t) / fs)) # Generating random 0s and 1s
bit_duration = int(fs / bit_rate) # Samples per bit

# Expanding the data bits to match the time vector
modulating_signal = np.repeat(data_bits, bit_duration)

# Carrier signal
carrier = amplitude * np.cos(2 * np.pi * carrier_freq * t)

# Amplitude Shift Keyed (ASK) signal
ask_signal = modulating_signal[:len(t)] * carrier

```

Figure 2.3: ASK modulation snippet.

This section aims to test and study different modulation techniques and check which part of the known spectra contains valuable information in each case. **Amplitude modulation (AM):** Consider the modulated signal as

$$y = (1 + \alpha x_m)x_c$$

, where $\alpha > 0$ is the modulation index, x_m is the modulating signal, and x_c is the carrier.

Exercise:

1. Create an amplitude-modulated signal where the carrier and the modulator are cosine waves. The modulating frequency is 1kHz , the carrier frequency is 10kHz , and the modulation index is 0.5.
2. Plot the temporal and spectral characteristics as before.
3. What do you see in the magnitude spectrum? Explain it.

Amplitude shift keying (ASK): The ASK modulation is an amplitude modulation, where the modulating signal changes according to a binary pattern. The following script shows a simple ASK modulation:

Exercise:

1. Create the ASK signal shown above.
2. Plot the temporal and spectral characteristics as before.
3. What do you see in the magnitude spectrum? Explain it.

Frequency modulation (FM):

In the FM technique, the information is encoded in the frequency change of the modulating signal. Consider the modulated signal as

$$y = \cos(2\pi f_c t + \alpha \sin(2\pi f_m t))$$

, where $\alpha > 0$ is the modulation index, f_m is the modulating frequency, and f_c is the carrier frequency. Note that the modulating signal can be any periodic signal; it's not required to be a sine wave.

Exercise:

1. Create an FM signal with 10kHz carrier frequency, 200Hz modulation frequency. Make the modulation index 5.
2. Plot the temporal and spectral characteristics as before.

```

# Parameters for FSK
carrier_freq_0 = 8000 # Frequency for bit 0 in Hz
carrier_freq_1 = 12000 # Frequency for bit 1 in Hz
bit_rate = 1000 # Bit rate (how many bits per second)
fs = 100000 # Sampling frequency in Hz
t = np.arange(0, 0.01, 1/fs) # Time vector for 10 ms

# Binary data (modulating signal) - let's assume some random bits for FSK
data_bits = np.random.randint(0, 2, int(bit_rate * len(t) / fs)) # Generating random 0s and 1s
bit_duration = int(fs / bit_rate) # Samples per bit

# Expanding the data bits to match the time vector
modulating_signal = np.repeat(data_bits, bit_duration)

# FSK signal generation
fsk_signal = np.array([
    np.cos(2 * np.pi * carrier_freq_0 * t[i]) if modulating_signal[i] == 0 else np.cos(2 * np.pi *
    carrier_freq_1 * t[i])
    for i in range(len(t))
])

```

Figure 2.4: FSK modulation snippet.

3. What do you see in the magnitude spectrum? Explain it.

Frequency shift keying (FSK):

Just as in the case of ASK, there is a possibility to modulate a carrier with digital data, i.e. using two frequency values representing the binary 0 and 1. The following script shows a simple FSK modulation:

Exercise:

1. Create the FSK signal shown above.
2. Plot the temporal and spectral characteristics as before.
3. What do you see in the magnitude spectrum? Explain it.

In the case of shift keying modulations, you might've seen some strange artefacts in the frequency domain. Try smoothing the state changes using a Gaussian filter (making GASK or GFSK modulation).

Exercise:

1. Download and open a music file. What is the sampling frequency? Why is it that value? What can you see on the spectra?
2. Open the provided CSV that contains an ECG signal. Plot the spectra and analyse the signal.