

Ahsanullah University of Science and Technology

Department of Electrical and Electronic Engineering

Laboratory Manual

for

Electrical and Electronic Engineering Sessional Course

Student Name :
Student ID :

Course no : EEE 3218

Course Title : Digital Signal Processing (DSP) Laboratory

For the students of the

Department of Electrical and Electronic Engineering

3^{rd} Year, 2^{nd} Semester

First Publish: December 2025

Prepared By:
Md Adnan Quaium
adnan.eee@aust.edu
Department of Electrical and Electronic Engineering
Ahsanullah University of Science and Technology

Editorial Committee:
Dr. Monjur Morshed, *Department of Electrical and Electronic Engineering*
Imran Rahman, *Department of Electrical and Electronic Engineering*
Md Mohi Uddin Khan, *Department of Electrical and Electronic Engineering*
Abdullah Saleh, *Department of Electrical and Electronic Engineering*

Published by:
Department of Electrical and Electronic Engineering
Ahsanullah University of Science and Technology
Dhaka 1208, Bangladesh
Contact: *headeee@aust.edu*

Copyright ©2025 by the Department of Electrical and Electronic Engineering
Ahsanullah University of Science and Technology.
All rights reserved.
No portion of this lab manual may be reproduced in any form without written permission from the publisher or author, except as permitted by *Bangladesh Copyright Act 2023.*

Preface

Welcome to the Digital Signal Processing Laboratory (EEE 3218). This laboratory course is an integral 1.5-credit component of the curriculum for 3rd Year, 2nd Semester Electrical and Electronic Engineering students. It serves as the critical practical complement to the theoretical concepts introduced in the core lecture course, *Digital Signal Processing (EEE 3217)*.

The primary goal of this manual is to facilitate a deep, applied understanding of DSP fundamentals. The experiments are designed to guide students in *investigating, designing, and implementing* digital signal processing concepts through powerful simulation environments like Python, leveraging industry-standard libraries such as NumPy, SciPy, and Matplotlib.

Throughout the term, you will engage with core themes essential to modern signal processing:

— LTI System Analysis: Investigating the time-domain and Z-domain characteristics of Linear Time Invariant systems.
— Spectral Analysis: Utilizing the Discrete Fourier Transform (DFT) and Fast Fourier Transform (FFT) for accurate frequency representation and analysis.
— Digital Filter Design: Mastering the design of both Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters to meet strict frequency domain specifications.
— Real-World Applications: Applying filtering techniques to practical problems such as biomedical signal denoising, audio equalization, and communication system requirements, etc.

This laboratory places a high emphasis on the design-oriented course outcomes (COs). Students are expected to:

(a) Develop Proficiency: Translate mathematical filter equations and theoretical system descriptions into working, verifiable Python code.
(b) Solve Open-Ended Challenges: Design filtering solutions from first principles to address specific requirements in various engineering fields.
(c) Validate and Communicate: Rigorously analyze experimental results, identify sources of error (e.g., phase distortion, instability), and communicate findings effectively through clear reports and plots.

The success of this course relies on your *active participation and independent effort*. We strongly encourage utilizing the interactive Python console for real-time analysis and debugging. Given the dynamic nature of software, students are responsible for ensuring their designated programming environment is correctly set up with all necessary libraries (NumPy, SciPy, Matplotlib, etc.). Please consult the instructor or teaching assistants if you encounter persistent technical difficulties.

This current lab manual, focused on Python implementation, is built upon the foundational structure and conceptual framework established by the original DSP lab manual. We acknowledge the pioneering work of Mr. Bejoy Sikder, former Lecturer in the Department of EEE (AUST) and currently pursuing his Ph.D. at MIT (USA). His demonstrative MATLAB-based lab manual serves as the fundamental pillar for this updated edition, and we pay tribute to his significant contribution.

Please contact Md Adnan Quaium (adnan.eee@aust.edu) if a correction or other revision is required to this manual in order to prevent future students from experiencing the same problems. We welcome your comments, suggestions, and ideas for improving this material. We trust that this practical experience will solidify your understanding of DSP theory and inspire its application in your future engineering career.

We look forward to your engagement and successful completion of the EEE 3218 Laboratory course.

Course Outline

1. **Title:** Digital Signal Processing Lab
2. **Code:** EEE 3218
3. **Credit hours:** 1.5
4. **Level:** Year 3, Semester 2
5. **Faculty:** Engineering
6. **Department:** Electrical and Electronic Engineering (EEE)
7. **Programme:** B.Sc. in Electrical and Electronic Engineering
8. **Synopsis from the approved curriculum:** Laboratory experiment based on the theory and the concept learnt in EEE 1109.
9. **Type of course:** Core
10. **Prerequisite(s):** no prerequisite course(s)
11. **Themes to be covered:** Laboratory experiments based on the theories and concepts learnt in the course "Digital Signal Processing" (EEE 3217).
12. **Course outcomes:** By the end of this course, students are expected to
    a) Investigate LTI systems and digital signal processing concepts through laboratory experiments and simulations.
    b) Design and implement open-ended DSP experiments to address specified requirements or challenges in different engineering and scientific fields.
    c) Communicate experimental results effectively through technical documentation and presentations.
13. **Reference materials:**
    a) **Text Book:**
        i. *Digital Signal Processing - Illustration Using Python* by S. Esakkirajan, T. Veerakumar and B.N. Subudhi. (Springer)
    b) **Reference Books:**
        i. *Digital Signal Processing* by J.G. Proakis and D.G. Manolakis. (Pearson)
        ii. *Essentials of Digital Signal Processing* by B.P. Lathi and R. Green. (Cambridge University Press)
        iii. *Discrete-Time Signal Processing* by A.V. Oppenheim and R.W. Schafer. (Pearson)
    c) **Websites:**
        i. https://jackschaedler.github.io/circles-sines-signals/
        ii. https://www.dspguide.com/
        iii. https://greenteapress.com/wp/think-dsp/
    d) **Handout (if any):**
        i. none
    e) **Others:**
        i. none

Course Assessment

1. **Online and offline task:** A combination of online and offline assignments may be provided for assessing the performance.
2. **Mode of evaluation**: Individual code implementation and evaluation will be conducted in the form of lab test, presentation, quiz and/or viva. Late submissions are to be highly discouraged and will be penalized.
3. **Semester project:** An individual project will be assigned to the student before the end of the semester. Students will be provided a time period for completing and submitting the project. Students will be asked to attend a viva on their respective projects.
   a) **Basic Requirements:** Clear understanding of all the lab experiments.
   b) **Presentation Requirement:** Yes.
   c) **Demonstration Requirement:** Depending on the semester project, demonstration will be conducted by respective lab teachers.
   d) **Reporting Requirement:** A brief report along with the codes must be submitted.
4. **Percentages of assessment methods:**

Method	**Percentage (%)**
Lab Performance and Attendance	10
Lab Report	10
Lab Test 01	20
Lab Test 02	20
Semester Project	40
Total Marks	**100**

5. **Administrative Policy of the Laboratory:**
   a) Students must perform class assessment tasks individually without help of others.
   b) Plagiarism is strictly forbidden and will be dealt with punishment.
   c) Every student must bring the lab manual book in the lab.
   d) Lab classes cannot be switched without valid reason and without informing the teachers.

Contents

Laboratory 1. Introduction to the DSP Lab . . . 1
1.1. Objective . . . 1
1.2. Digital Signal Processing . . . 1
1.3. Advantages and Applications of Digital Signal Processing . . . 2
1.4. DSP Nomenclatures and Topics . . . 2
1.4.1. Time Domain Processing . . . 2
1.4.2. Frequency Transforms . . . 3
1.4.3. Frequency Domain Processing . . . 3
1.5. An Overview of Using Python in DSP Lab . . . 3
1.5.1. Spyder, the IDE of our choice . . . 4
1.6. Basics of Python . . . 5
1.6.1. The `print()` function . . . 5
1.6.2. Variables and Data Types . . . 6
1.6.3. Lists and Arrays . . . 6
1.6.4. Loops . . . 6
1.6.5. Functions . . . 7
1.6.6. Importance of Indentation in Python . . . 8
1.7. Generation of Continuous-Time Signals . . . 8
1.7.1. Generation of Sinusoidal Signal . . . 8
1.7.2. Sinusoidal Signal with Different Phase . . . 9
1.7.3. Generation of Three-Phase Sinusoidal Signal . . . 10
Post-laboratory Problems . . . 11
Laboratory 2. Sampling and Quantization of Signals . . . 13
2.1. Objective . . . 13
2.2. Analog to Digital Conversion . . . 13
2.3. Sampling . . . 13
2.3.1. Violation of Sampling Theorem . . . 14
2.3.2. Sampling in Time Domain . . . 15
2.3.3. Aliasing in Time Domain . . . 16
2.4. Quantization . . . 18
2.4.1. Mid-Tread Quantizer . . . 19
2.4.2. Quantization Levels . . . 20
2.5. Signal Reconstruction . . . 22
Post-laboratory Problems . . . 25
Laboratory 3. Generation and Operation on Discrete-Time Sequence . . . 27
3.1. Objective . . . 27
3.2. Generation of Discrete-Time Signals . . . 27
3.2.1. Generation of the Unit Impulse Sequence . . . 27
3.2.2. Generation of the Unit Step Sequence . . . 28
3.2.3. Generation of the Unit Ramp Signal . . . 30
3.3. Mathematical Operation on Discrete-Time Signals . . . 31
3.3.1. Amplitude scaling . . . 31
3.3.2. Amplitude Shifting . . . 31
3.3.3. Product of Two Signals . . . 33
3.3.4. Signal Addition . . . 34
3.3.5. Time Scaling Operation . . . 36
3.3.6. Time Shifting Operation . . . 37
3.3.7. Time Reversal Operation . . . 38

3.4. Convolution . . . 40
3.4.1. Properties of Convolution . . . 42
3.5. Correlation . . . 43
Post-laboratory Problems . . . 45
Laboratory 4. Transforms and Discrete-Time Systems . . . 47
4.1. Objective . . . 47
4.2. Discrete Fourier Transform (DFT) . . . 47
4.2.1. Computation of DFT of a sequence using DFT matrix . . . 48
4.2.2. Computation of Inverse DFT through DFT matrix . . . 48
4.2.3. Computation of N-point DFT of a sequence and reconstruct the sequence from inverse DFT 49
4.3. Fast Fourier Transform (FFT) . . . 50
4.3.1. Computation of N-point FFT of a sequence and reconstruct the sequence from inverse FFT 50
4.3.2. Computation of the frequency spectrum of a *sinc* function . . . 51
4.3.3. Convolution using DFT . . . 53
4.3.4. Observe modulation and demodulation of a message signal in frequency domain using FFT . 55
4.4. Z-Transformation . . . 58
4.4.1. Z-Transform of Unit Sample and Unit Step Sequences using `lcapy` package . . . 59
4.4.2. Z-Transform of $x[n] = e^{jn}u[n]$, and $x[n] = cos(n)$, and $x[n] = \frac{1}{2}^n u[n]$ . . . 59
4.4.3. Inverse Z-Transform of $X(z) = z^{-1}$, and $X(z) = \frac{1}{1-z^{-1}}$ using `lcapy` package . . . 60
4.4.4. Inverse Z-Transform using partial fraction expansion . . . 60
4.5. Discrete-Time System . . . 61
4.5.1. Difference Equation Representation of Discrete-Time Linear Time-Invariant System . . . 61
4.5.2. Impulse Response of Discrete-Time System . . . 62
4.5.3. Pole-Zero Plot of Discrete-Time System . . . 64
4.5.4. Frequency Response of a Discrete-Time System . . . 65
Post-laboratory Problems . . . 68
Laboratory 5. Digital Filter Design (Part - 1) . . . 69
5.1. Objective . . . 69
5.2. Digital Filters . . . 69
5.3. Design Specification of a Practical Filter . . . 69
5.4. Classification of Digital Filters . . . 70
5.4.1. Classification by Structure (Impulse Response) . . . 70
5.4.2. Classification by Design/Response Type . . . 70
5.4.3. FIR and IIR Filters . . . 71
5.5. Characteristics of The Traditional Digital Filter . . . 72
5.5.1. First-Order IIR Filter . . . 72
5.5.2. Moving Average filter . . . 74
5.5.3. Digital Resonator . . . 76
5.6. FIR Filter Design *(using Window Method)* . . . 78
Post-laboratory Problems . . . 85
Laboratory 6. Digital Filter Design (Part - 2) . . . 87
6.1. Objective . . . 87
6.2. IIR Filters . . . 87
6.3. FIR vs IIR - Which one to Choose? . . . 87
6.4. IIR Filter Design . . . 88
6.4.1. Steps of Filter Design Using Pole-Zero Placement Method . . . 88
6.4.2. Steps of Filter Design Using Bilinear Transformation Technique . . . 88
6.5. Pole-Zero Method . . . 89
6.6. Bilinear Transformation Technique (BLT) . . . 92
6.6.1. Example 1: Design a high pass filter from low pass prototype filter using Bilinear Transformation . . . 92
6.6.2. Example 2: Design a Butterworth Low-Pass Digital Filter using Bilinear Transformation . . . 94
6.6.3. Example 3: Design a Chebyshev Type II Digital Bandstop Filter using Bilinear Transformation 96
Post-laboratory Problems . . . 99
Laboratory 7. Application of Signal Processing (Part - 1) . . . 101
7.1. Objective . . . 101
7.2. Case study 1: Separation of Signals Using FIR Filter . . . 101
7.3. Case Study 2: Denoising of the Signal Using FIR Filter . . . 103

7.4. Case Study 3: Voice/Silence Detection (Simple Feature Extraction) 105
7.4.1. Feature Extraction vs. Digital Filtering in VAD 105
7.4.2. Methodology 105
Post-laboratory Problems 109
Laboratory 8. Application of Signal Processing (Part - 2) 111
8.1. Objective 111
8.2. Case study 4: ECG Baseline Wander Removal (High-Pass Filtering) 111
8.3. Case Study 5: Power Line Hum Removal (Notch Filtering) 114
8.4. Case Study 6: Removing "Salt-and-Pepper" Noise from an Image 118
Post-laboratory Problems 121

Laboratory 1

Introduction to the DSP Lab

1.1. Objective

The main objectives for this lab are to introduce students to the foundational concepts of digital signal processing and Python programming using the Spyder IDE. By the end of the lab, students will be able to simulate and visualize various continuous-time signals, gaining a practical understanding of their characteristics and behavior.

1.2. Digital Signal Processing

A **signal** is a function of a set of independent variables, with time being perhaps the most prevalent single variable. The signal itself carries some kind of information available for observation. Most signals of interest in practice are recorded values of a physical quantities, represented as a 1-D functions, such a time function, **x(t)** like speech or 2-D/3-D functions, such as a spatial function **f(x,y)** like image or function of both space and time **f(x,y,t)** like video. A signal carries **information** and contains **energy**.

By the term **processing**, we mean operating in some fashion on a signal to extract some useful **information**. In many cases, this processing will be a nondestructive "**transformation**" of the given data signal; however, some essential processing methods turn out to be irreversible and thus destructive.

Finally, the word **digital** means that the processing is done with a digital computer or specific-purpose digital hardware. For better clarification, we can briefly define **analog**, **discrete**, and **digital** signals as follows:

— **Analog Signal:** a function x(t), continuous in amplitude, of a continuous independent variable **t** (e.g., time).
— **Discrete Signal:** a function $\mathbf{x[n] = x(nt_0)}$, continuous in amplitude, but defined only at a set of discrete values of the independent variable, $\mathbf{t = 0, \pm t_0, \pm 2t_0, ...}$
— **Digital Signal:** a discrete signal with quantized (finite) amplitude values. For example, there are $2^8 = 256$ gray scale levels in an 8-bit digital image (Figure 1.1).



Figure 1.1: A digital image of a mirrorless camera (Photo courtesy:Rodneil Quiteles)

1.3. Advantages and Applications of Digital Signal Processing

Advantages of using digital signals, including digital signal processing (DSP) and communication systems, include the following:

— Digital signals can convey information being less susceptible to noise, distortion, and interference.
— Digital circuits can be reproduced easily in mass quantities at comparatively low costs.
— Digital signal processing is more flexible because DSP operations can be altered using digitally programmable systems.
— Digital signal processing is more secure because digital information can be easily encrypted and compressed.
— Digital systems are more accurate, and the probability of error occurrence can be reduced by employing error detection and correction codes.
— Digital signals can be easily stored on any magnetic media or optical media using semiconductor chips.

Applications of DSP are increasing in many areas where analog electronics are being replaced by DSP chips, and new applications are depending on DSP techniques. With the cost of DS processors decreasing and their performance increasing, DSP will continue to affect engineering design in our modern daily lives. Some application examples using DSP are listed in Table 1.1.

Table 1.1: Application of Digital Signal Processing

Application Field	Description
Speech Processing	*The use of speech signals for speech recognition, speaker identification, and text-to-speech conversion.*
Audio Signal Processing	*Processing of musical signals for equalization, noise reduction, and special effects.*
Telecommunications	*Modem technologies, mobile phones, satellite communication,n and radar.*
Image and Video Processing	*For image enhancement, compression, pattern recognition, and computer vision.*
Seismology	*For earthquake detection, location, and analysis.*
Biomedical Signal Processing	*For processing signals from ECG, EEG, etc.*
Radar	*The use of DSP for target detection and tracking.*

However, the list in the table by no means covers all DSP applications. Many more areas are increasingly being explored by engineers and scientists. Applications of DSP techniques will continue to have profound impacts and improve our lives.

1.4. DSP Nomenclatures and Topics

Figure 1.2 shows a broad overview of digital signal processing. Analog signals enter an ADC from the left, and samples exit the ADC from the right, and may be 1) processed strictly in the discrete time domain (in which samples represent the original signal at instants in time) or they may be 2) converted to a frequency domain representation (in which samples represent amplitudes of particular frequency components of the original signal) by a time-to-frequency transform, processed in the frequency domain, then converted back to the discrete time domain by a frequency-to-time transform. Discrete-time domain samples are converted back to the continuous time domain by the DAC. Note that a particular signal processing system might use only time domain processing, only frequency domain processing, or both time and frequency domain processing, so either or both of the signal processing paths shown in Figure 1.2 may be taken in any given system.

1.4.1. Time Domain Processing

In time domain processing, the variation of the input and output variables is observed with respect to time. The examples of time domain processing include –

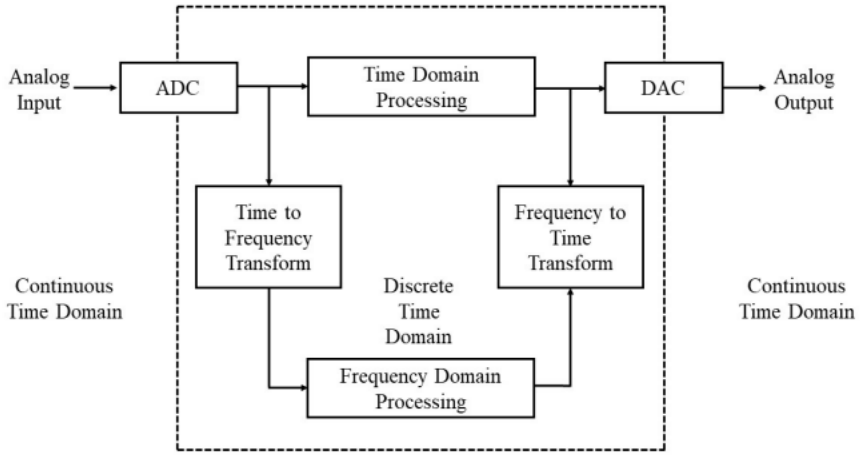


Figure 1.2: A broad, conceptual overview of digital signal processing

— Basic signal generations.
— Signal manipulation in the time domain.
— Signal composition and decompositions.
— Linear Time Invariant (LTI) system response in time domain and convolution.
— Auto-correlation and Cross-correlation, etc.

1.4.2. Frequency Transforms

A time-to-frequency transform operates on a block of time domain samples and evaluates the frequency content thereof. A set of frequency coefficients is derived, which can be used to quantify the amplitudes (and usually phases) of frequency components of the original signal, or the coefficients can be used to reconstruct the original time, domain samples using an inverse transform (a frequency-to-time transform). The most well-known and widely-used of these transforms is the **DFT** (Discrete Fourier Transform), usually implemented by the **FFT** (Fast Fourier Transform), the name of a class of algorithms that allow efficient computation of the **DFT**. In addition, **Z-transforms** also convert a time domain signal to the complex frequency domain.

1.4.3. Frequency Domain Processing

Most signal processing that can be done in the time domain can also be equivalently done in the frequency domain. Each domain has certain advantages for a given type of problem. Time domain filtering, for example, can be performed using frequency transforms such as the DFT, and in certain cases, efficiency can be greatly improved using this technique. A second use is in digital filter design, in which the desired filter frequency response is specified in the frequency domain, i.e., as a set of DFT coefficients, for example. Yet a third and very prevalent use is **Transform Coding**, in which signals are coded using a frequency transform (usually eliminating as much redundant information as possible) and then reconstructed from the transform coefficients. Transform Coding is a powerful tool for compression algorithms, such as those employed with MP3 (MPEG-2 Layer-3) for audio signals, JPEG, a common image compression format, etc. The use of such compression algorithms has revolutionized the audio and video fields, making storage of audio and video data very economical and deliverable via the Internet.

1.5. An Overview of Using Python in DSP Lab

Python is a high-level, interpreted programming language known for its clear syntax and readability, making it an excellent choice for beginners and experienced developers alike. With a vast ecosystem of

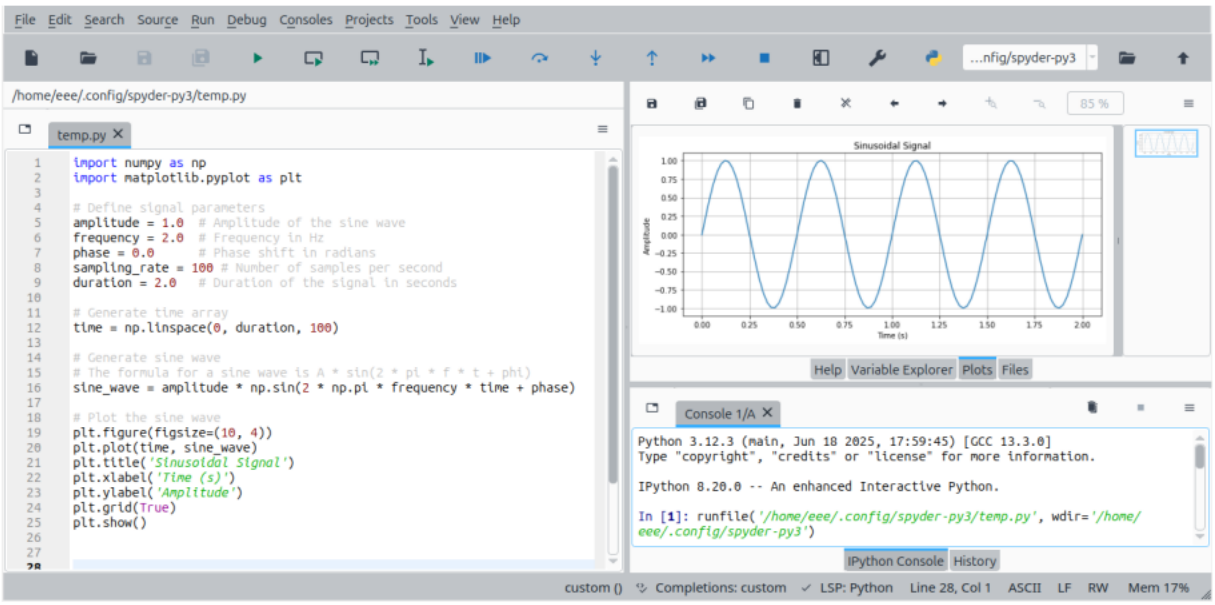


Figure 1.3: Spyder interface

libraries and frameworks, it supports a wide range of applications, from web development and data science to automation and machine learning. Its versatility and strong community support have made it one of the most popular programming languages today.

Python's importance in modern Digital Signal Processing (DSP) cannot be overstated. While traditionally, languages like MATLAB and C++ have dominated the field, Python has emerged as a powerful alternative due to several key factors. Libraries such as **NumPy** provide highly optimized N-dimensional array objects and mathematical functions, enabling efficient manipulation of large datasets and signals. **SciPy** builds on this foundation by offering a comprehensive collection of algorithms for signal processing, including functions for filtering, convolution, and transformations. For visualization, **Matplotlib** allows for the creation of publication-quality plots of signals and their frequency responses.

The open-source nature of Python and its libraries fosters collaboration and innovation, giving researchers and engineers access to cutting-edge tools and algorithms. This combination of powerful libraries, ease of use, and a vibrant community has made Python an indispensable tool for research, prototyping, and even deployment in the modern DSP workflow.

1.5.1. Spyder, the IDE of our choice

Spyder (Scientific Python Development Environment) is a powerful, open-source integrated development environment (IDE) designed specifically for scientific computing and data analysis in Python. It provides a comprehensive set of tools, including a code editor with syntax highlighting, a variable explorer, an interactive console, and a dedicated plotting pane. Its multi-pane layout is highly intuitive, making it a favorite among engineers and scientists.

We use Spyder for our DSP lab experiments because its environment is perfectly suited for a scientific workflow. Getting started with Spyder is a straightforward process, especially for those familiar with MATLAB's layout. Figure 1.3 shows the multi-pane interface of Spyder. On the left, you can find the **Editor**, where you write and save your Python code. On the bottom-right, the **IPython Console** serves as your command prompt, allowing you to execute code interactively and see output immediately. On the top-right, the **Variable Explorer** provides a live view of all the variables in your workspace, which is incredibly useful for debugging and inspecting data. Finally, the **Plots** pane will automatically display any graphs you generate with libraries like *Matplotlib*, which is crucial for visualizing and analyzing signal behavior.

Spyder's organized layout makes it easy to manage your files, write scripts, and see results. This integrated approach streamlines the process of writing and debugging, making it a very effective tool for educational purposes.

By default, Spyder is pre-loaded with essential scientific libraries, including **NumPy**, **SciPy**, **Matplotlib**, and **Pandas**. For this course, you will need three more specialized packages: **Lcapy**, **Control**, and **NeuroKit2**. These packages are not included in a default Spyder installation. To install these three libraries directly from the Spyder environment, navigate to the built-in IPython Console within the IDE and run the following command.

```
pip install lcapy control neurokit2
```

1.6. Basics of Python

1.6.1. The `print()` function

In Python, you use the `print()` function to display output on the screen, printing text, variables, and expressions. You can print strings, numbers, and even results of calculations directly within the parentheses, often using quotation marks for text. The `print()` function automatically adds a space between multiple items and a newline character at the end of each statement.

Printing Text (Strings)

To print text, enclose the text in quotes (either single ' or double ") inside the `print()` function

```
print("Hello World!")   # Output: Hello World!
print('This is a string.')  # Output: This is a string.
```

Printing Variables

You can store data in variables and then print those variables.

```
name = "Alice"
age = 30
print("Name:", name)# Output: Name: Alice
print("Age:", age)  # Output: Age: 30
```

Printing Expressions and Numbers

You can also print the results of mathematical operations directly.

```
print(10 + 5)   # Output: 15
print(3 * 4)# Output: 12
print("The result is:", 2 ** 3) # Output: The result is: 8
```

Using F-Strings for Formatting

For more control, especially when embedding variables into strings without unwanted spaces, use f-strings.

```
score = 95
print(f"Your score is: {score}")# Output: Your score is: 95
```

1.6.2. Variables and Data Types

In Python, you do not need to declare a variable's type. You assign a value to it, and Python figures out the data type automatically. For numerical analysis, the most common types are integers (`int`) and floating-point numbers (`float`).

```
# Integer (for whole numbers)
count = 10
print("The count is:", count)

# Float (for numbers with a decimal point)
pi_value = 3.14159
print("The value of pi is:", pi_value)

# Simple arithmetic
result = 15 / 4
print("15 divided by 4 is:", result)
```

1.6.3. Lists and Arrays

A **list** is a fundamental data structure in Python that can hold a collection of items. For numerical work, it is often more efficient to use a **NumPy** array, which is specifically designed for numerical operations.

```
# A basic Python list
my_list = [1, 2, 3, 4, 5]
print("My list is:", my_list)
```

A list is a flexible, built-in container that can hold elements of different data types, while an array (from modules like array or NumPy) is a more memory-efficient and performant structure for storing a large, homogeneous collection of numerical data. To use arrays, you need to import the NumPy library.

```
import numpy as np
# Create a NumPy array from a list
my_array = np.array([1, 2, 3, 4, 5])
print("My array is:")
print(my_array)

# Numerical operations with arrays
added_array = my_array + 10
print("Array with 10 added to each element:")
print(added_array)

multiplied_array = my_array * 2
print("Array with each element multiplied by 2:")
print(multiplied_array)
```

1.6.4. Loops

Loops are essential for performing repetitive tasks, such as iterating over all the elements in a list or array. The `for` loop is the most common type. Following is an example of a `for` loop to print each element of a list.

```
numbers = [10, 20, 30, 40]
print("Using a for loop to print numbers:")
for num in numbers:
    print(num)
```

To show the output in one single line, you can write the following code.

```
numbers = [10, 20, 30, 40]
print("Using a for loop to print numbers:")
for num in numbers:
    print(num, " ", end="")
```

Following is an example of calculating the sum of an array's elements. This is a common numerical analysis task.

```
import numpy as np

data = np.array([1.2, 3.4, 5.6, 7.8, 9.0])
total_sum = 0
for value in data:
    total_sum = total_sum + value
print("The sum of the data is:", total_sum)
```

However, NumPy has a built-in `sum()` function which is much more efficient.

```
import numpy as np

data = np.array([1.2, 3.4, 5.6, 7.8, 9.0])
print("The sum of the data is:", np.sum(data))
```

1.6.5. Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions are critical for organizing your code and making it more readable and maintainable. Following is an example of a simple function to calculate the average.

```
def calculate_average(data_list):
    total = sum(data_list)
    average = total / len(data_list)
    return average

# Using the function
data_points = [10, 15, 20, 25, 30]
avg = calculate_average(data_points)
print("The average of the data points is:", avg)
```

Following is another function to convert Celsius to Fahrenheit.

```python
def celsius_to_fahrenheit(celsius):
    fahrenheit = (celsius * 9/5) + 32
    return fahrenheit

temp_c = 25
temp_f = celsius_to_fahrenheit(temp_c)
print(f"{temp_c} C is equal to {temp_f} F.")
```

1.6.6. Importance of Indentation in Python

Indentation is not merely a stylistic choice in Python; it is the fundamental mechanism that defines code structure and execution flow. Unlike many other languages that use braces ({}) to denote blocks of code (such as loops or functions), Python uses whitespace (tabs or spaces). This rule forces clear, readable code, but means that even a single missing or misplaced tab/space will prevent the program from running by raising an `IndentationError` or executing logic incorrectly. Every line belonging to a control structure (such as `if`, `else`, or `for`) must be indented at the same level relative to the statement above it. Figure 1.4 shows the correct indentation in Python.

```python
score = int(input("Enter your score: "))
if score >= 80:
    print("Grade A+")
    print("Excellent work!")
elif score >= 40:
    print("Grade A or below")
    print("Good effort.")
else:
    print("You've failed.")
```

(a) Indentation is correct

```python
score = int(input("Enter your score: "))
if score >= 80:
print("Grade A+")
print("Excellent work!")
elif score >= 40:
print("Grade A or below")
print("Good effort.")
else:
print("You've failed.")
```

(b) Indentation is incorrect

```python
  score = int(input("Enter your score: "))
if score >= 80:
print("Grade A+")
        print("Excellent work!")
            elif score >= 40:
print("Grade A or below")
    print("Good effort.")
    else:
            print("You've failed.")
```

(c) Indentation is incorrect

```python
score = int(input("Enter your score: "))
 if score >= 80:
  print("Grade A+")
   print("Excellent work!")
    elif score >= 40:
     print("Grade A or below")
      print("Good effort.")
       else:
        print("You've failed.")
```

(d) Indentation is incorrect

Figure 1.4: Illustration of correct and incorrect Python indentation for defining code flow

1.7. Generation of Continuous-Time Signals

1.7.1. Generation of Sinusoidal Signal

Sinusoidal signals are periodic functions that are based on the sine or cosine function. The expression for the sinusoidal signal is given by

$$x(t) = A\sin(2\pi ft + \phi)$$

In the above equation, A represents the amplitude of the signal, f denotes the frequency of the signal, and ϕ indicates the phase of the signal. Below is the Python code to generate this sinusoidal signal. The output of the following code is shown in Figure 1.5

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

t = np.linspace(0,1,1000) # Defining the time scale
A = 5                     #Amplitude of sine wave
f = 5                     #Frequency of sine wave
ph = 0                    #Phase of sine wave
x = A*np.sin(2*np.pi*f*t+ph)  #Expression of sine wave
plt.plot(t,x)           #Plotting the sine wave
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('A={}V,F={} Hz,$\phi={}^\circ$'.format(A,f,ph))
```

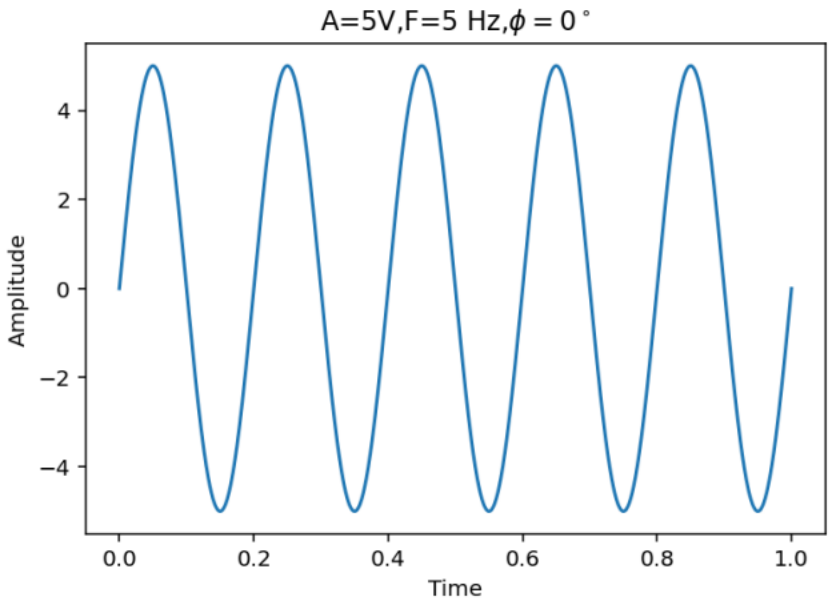


Figure 1.5: Result of 1.7.1

1.7.2. Sinusoidal Signal with Different Phase

Generate a sine wave of amplitude = 1 V, frequency = 5 Hz, and four different phase angles, namely, 0°, 90°, 180°, and 270°. You should present the results in one figure using `subplot()`. The Python code, which does this task, is shown below, and the corresponding output is shown in Figure 1.6.

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

t = np.linspace(0,1,100)
A = 1                   #Amplitude
f = 5                   #Frequency
phi = [0,90,180,270]  #Phase
for i in range(len(phi)):
    x = A*np.sin(2*np.pi*f*t+phi[i]*np.pi/180)
    plt.subplot(2,2,i+1)
    plt.plot(t,x),plt.xlabel('Time (t)'),plt.ylabel('Amplitude (V)')
```

```
    plt.title('$\Phi ={}^\circ $'.format(phi[i]))
    plt.tight_layout()
```

Here, `plt.tight_layout()` automatically adjusts subplot parameters to prevent overlaps between plots, axis labels, and titles within a figure, creating a more compact and visually organized display. It works by adjusting spacing and repositioning axes to ensure all elements fit within the figure area without being cut off or interfering with each other.

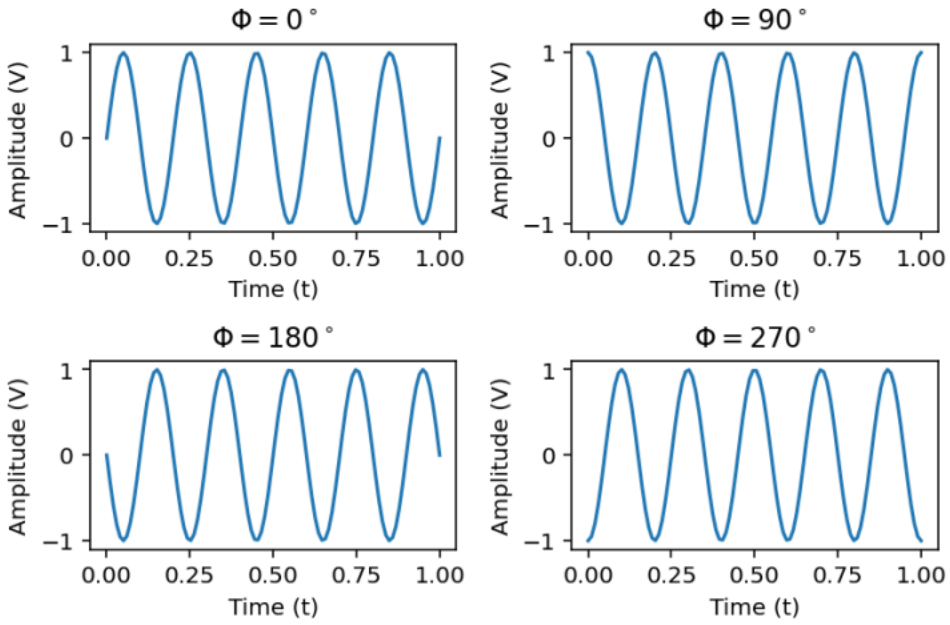


Figure 1.6: Result of 1.7.2

1.7.3. Generation of Three-Phase Sinusoidal Signal

The expressions for three-phase sinusoidal signals are given by

$$x(t) = A\sin(2\pi ft) \tag{1.1}$$

$$x(t) = A\sin(2\pi ft - 120^\circ) \tag{1.2}$$

$$x(t) = A\sin(2\pi ft - 240^\circ) \tag{1.3}$$

The python code, which generates the three-phase sinusoidal waveforms, is shown below, and the corresponding output is shown in Fig. 1.6.

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

t = np.linspace(0,1,100)
A = 1                                   #Amplitude
f = 5                                   #Frequency
phi_1,phi_2,phi_3 = 0, 120, 240   #Three different phases of sine wave
x1 = A*np.sin(2*np.pi*f*t+phi_1*np.pi/180)
x2 = A*np.sin(2*np.pi*f*t+phi_2*np.pi/180)
x3 = A*np.sin(2*np.pi*f*t+phi_3*np.pi/180)
plt.plot(t,x1,'b')    #Plotting the result
plt.plot(t,x2,'r--')
```

```
plt.plot(t,x3,'g-.')
plt.xlabel('Time (t)'),plt.ylabel('Amplitude (V)')
plt.title('Three phase sinusoidal signal')
plt.legend(['Phase-1','Phase-2','Phase-3'],loc=1)
```

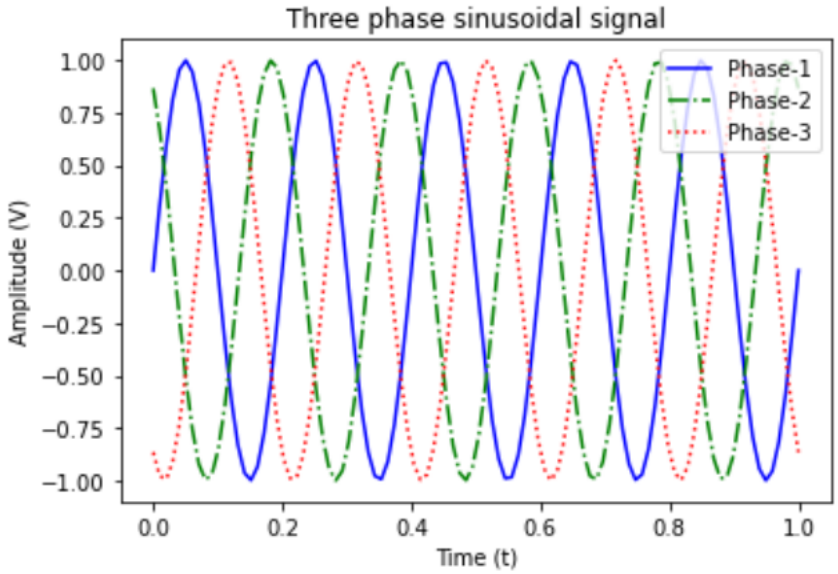


Figure 1.7: Result of 1.7.3

Post-laboratory Problems

1. Compile and run the codes used for **plotting** and **subplotting** different signals and show the output waveforms.
2. Generate the following sinusoidal signal $x(t) = A\sin(2\pi ft + \phi)$ with the amplitude $A = 2V$, frequency $f = 10Hz$ and phase $\phi = 0$. Let the length of the signal be 3 periods. Plot the signal.
3. Generate a sinusoidal signal with 10 kHz and show it for 10ms. Assume amplitude values and necessary data. Use the `sin()` function first. Then use the `cos()` function to repeat the same.
4. Solve all the exercise problems of the textbook from page 31.

Laboratory 2

Sampling and Quantization of Signals

2.1. Objective

The objective of this laboratory is to understand the core processes of converting a continuous-time signal into a discrete-time signal. This involves exploring sampling and quantization, which are critical steps in this conversion. Additionally, the laboratory aims to demonstrate how to reconstruct the original signal from its sampled version. By simulating and visualizing these processes, students will gain a practical understanding of how signals are digitized and manipulated.

2.2. Analog to Digital Conversion

Most signals we encounter in the real world, such as speech and biological signals like ECG (electrocardiogram) or PPG, are analog in nature. To process these signals, we use Digital Signal Processing (DSP), which offers significant advantages over analog processing, including greater accuracy, improved storage capabilities, and cost efficiency. For this reason, analog signals must first be converted into a digital format.

This conversion process involves three main steps: sampling, quantization, and encoding. Figure 2.1 shows these steps. Once processed digitally, the signal must be converted back to an analog form through a reconstruction process to be used by devices like speakers or monitors. In this lab, we will discuss the key concepts and issues related to the initial sampling and the final reconstruction of continuous-time signals.

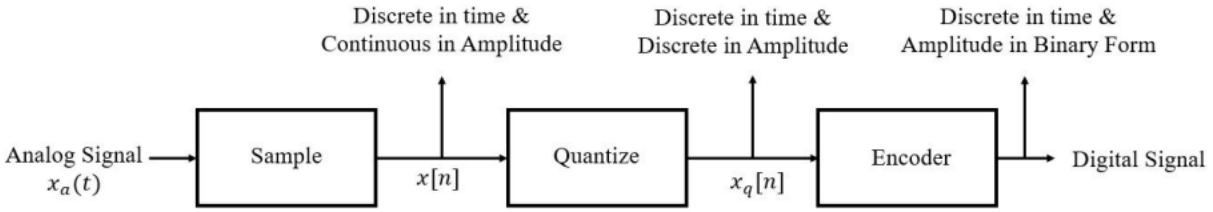


Figure 2.1: Analog to Digital conversion process

2.3. Sampling

Sampling is the fundamental process of converting a continuous-time signal into a discrete-time one, serving as the first step in the analog-to-digital conversion chain. One of the most common methods is periodic sampling, which involves recording the exact value of the continuous signal at uniformly spaced intervals. These intervals are defined by the sampling period (T_s) and its reciprocal, the sampling frequency ($f_s = \frac{1}{T_s}$).

Mathematically, the sampling process can be visualized in two stages. First, the continuous-time signal is multiplied by an impulse train - a series of impulses - to create a sampled signal. This impulse train is then converted into a numerical sequence. This two-stage representation provides a convenient way to model the transition from the continuous to the discrete domain. The periodic impulse train is given by

$$s(t) = \sum_{n=-\infty}^{\infty} \delta(t - nT_s) \tag{2.1}$$

where $\delta(t)$ is the Dirac delta function. The product of *s(t)* and continuous signal, $x_c(t)$ is:

$$x_s(t) = x_c(t)s(t) \tag{2.2}$$

Figure 2.2 shows the process of sampling using impulse train modulation. Note that, $x_s(t)$ is still, in a sense, a continuous signal which is *zero*, except at integer multiples of T_s. On the other hand, $x[n]$ is indexed on the integer variable n.

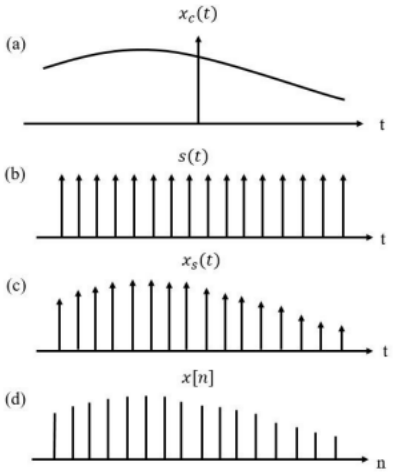


Figure 2.2: (a) Continuous-time Signal, $x_c(t)$ (b) Impulse Train, s(t) (c) Signal, $x_s(t) = x_c(t)s(t)$ (d) Discrete-time signal obtained from sampling.

2.3.1. Violation of Sampling Theorem

We must realize that the samples must be sufficiently close, and the Sampling Rate must bear a certain relation to the highest frequency component of the original signal. The **sampling theorem**, also known as **Nyquist-Shannon Sampling Theorem**, gives us the minimum sampling frequency for a continuous-time signal so that it can be uniquely reconstructed from its samples. If the sampling frequency is less than this minimum value, a phenomenon known as **aliasing** occurs.

For example, in the case of a continuous time sinusoidal signal, $x_c(t) = \cos 4000\pi t$, the signal frequency is $2000Hz$. So in order to reconstruct the signal uniquely from its samples, the sampling frequency should be at least 4000Hz. Note that, $\mathrm{f_N}$ mentioned in the sampling theory is the maximum frequency content of the signal. If a signal is sampled below the Nyquist rate, it should be analyzed in the frequency domain to understand what happens.

Sampling Theorem:

Let $x_c(t)$ be a bandlimited signal with

$$X_c(\mathrm{f}) = 0 \text{ for } |\mathrm{f}| \geq \mathrm{f}_N.$$

Then $x_c(t)$ is uniquely determined by its samples $x[n] = x_c(nT_s), \mathrm{n} = 0, \pm 1, \pm 2....$, if the sampling frequency

$$\mathrm{f}_s = \frac{2\pi}{T_s} \geq 2f_N.$$

The frequency f_N is commonly referred to as the *Nyquist frequency*, and the frequency $2\mathrm{f_N}$ as the *Nyquist rate.*

Figure 2.3 shows the frequency-domain representation of the impulse train sampling. Figure 2.3(c) shows the sampling of $x_c(t)$ with sampling frequency $f_s > 2f_N$. Here the replicas of $X_c(f)$ do not overlap and therefore, $x_c(t)$ can be recovered from $x_s(t)$ with an ideal low pass filter with cut-off frequency

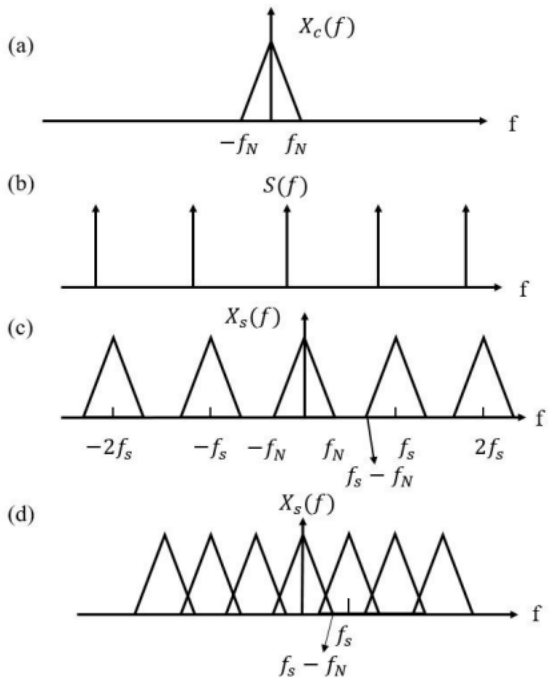


Figure 2.3: Frequency-domain representation of sampling in the time domain. (a) Spectrum of the original signal. (b) Fourier Transform of the sampling function. (c) Fourier Transform of the sampled signal with $f_s > 2f_N$. (d) Fourier Transform of the sampled signal with $f_s < 2f_N$

(f_c) such that $f_N \leq f_c \leq (f_s - f_N)$. However, Figure 2.3(d) shows the case where the sampling frequency $f_s < 2f_N$. Here, the copies of $X_c(f)$ overlap, so that when they are added together, $X_c(f)$ is no longer recoverable by low-pass filtering. In this case, the reconstructed output is related to the original continuous-time input through a distortion which is known as **aliasing**. So aliasing is an unwanted case of sampling, where the minimum condition for accurate sampling is not met. Physically, in aliasing, signals of different frequencies become indistinguishable from each other. The high-frequency components of the signal spectrum take the identity of the lower frequencies in the spectrum of the sampled signal. Aliasing can be avoided by sampling the signal at a higher rate than the Nyquist rate or by using an anti-aliasing filter, which is essentially a low pass filter.

2.3.2. Sampling in Time Domain

Let us consider a continuous-time signal $x_c(t) = 10\cos(200\pi t)$. Frequency of the signal is $F_0 = 100$ Hz and time period is $T_0 = \frac{1}{100}$ = 0.01s. So the corresponding Nyquist rate is 200Hz. At first, we'll sample the signal at a rate of 800Hz, which is much higher than the Nyquist rate. From figure 4.5, it is seen that the original signal can be uniquely identified from its samples. The code for this is given below. And Figure 2.4 shows the outcome of the following code.

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

Fo = 100                    # Signal Frequency in Hz
To = 1 / Fo             # Signal Period in seconds
Fs = 800                    # Sampling Frequency > Nyquist rate
Ts = 1 / Fs             # Sampling Period in seconds
t = np.linspace(0, 3 * To, 1000)
xc = 10 * np.cos(2 * np.pi * Fo * t)
t1 = np.arange(0, 3 * To + Ts, Ts)
xs = 10 * np.cos(2 * np.pi * Fo * t1)
```

```
plt.subplot(211)
plt.plot(t, xc, color='b')
plt.stem(t1, xs, linefmt='r')
plt.title('Continuous and Sampled Signals')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.grid()
plt.subplot(212)
plt.stem(t1, xs, linefmt='r-')
plt.title('Sampled Signal Only')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.grid()
plt.tight_layout()
```

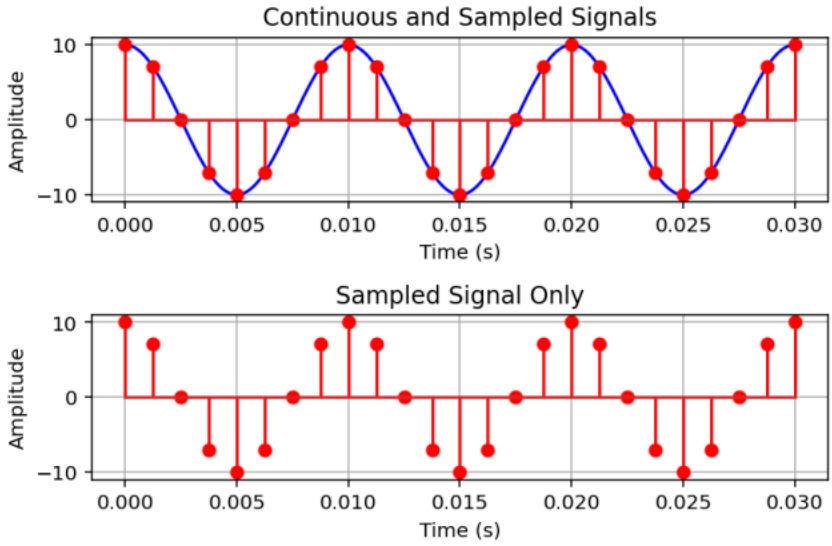


Figure 2.4: (a) Continuous-time signal and its samples obtained by sampling at a rate of 800Hz (b) Samples of the continuous-time signal in (a)

2.3.3. Aliasing in Time Domain

Now let us sample the signal $x_c(t)$ at a sampling frequency equal to the Nyquist rate i.e., 200Hz, and the corresponding output is shown in Figure 2.5. It is seen from the figure that we can recognize the signal from its samples, although this is more difficult than the previous case, where the sampling rate is higher than this. However, the trend in the signal's behaviour can be recognized from its samples. In fact, the samples only capture the extremes of each period of the cosine oscillation. This is the significance of "twice the highest frequency of the signal" value for sampling frequency. We can say that, the samples are just enough to capture the signal's oscillation if it is sampled exactly at the Nyquist rate.

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

Fo = 100                    # Signal Frequency in Hz
To = 1 / Fo             # Signal Period in seconds
```

```
Fs = 200                    # Sampling Frequency = Nyquist rate
Ts = 1 / Fs      # Sampling Period in seconds
t = np.linspace(0, 3 * To, 1000)
xc = 10 * np.cos(2 * np.pi * Fo * t)
t1 = np.arange(0, 3 * To + Ts, Ts)
xs = 10 * np.cos(2 * np.pi * Fo * t1)

plt.subplot(211)
plt.plot(t, xc, color='b')
plt.stem(t1, xs, linefmt='r')
plt.title('Continuous and Sampled Signals')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.grid()
plt.subplot(212)
plt.stem(t1, xs, linefmt='r-')
plt.title('Sampled Signal Only')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.grid()
plt.tight_layout()
```

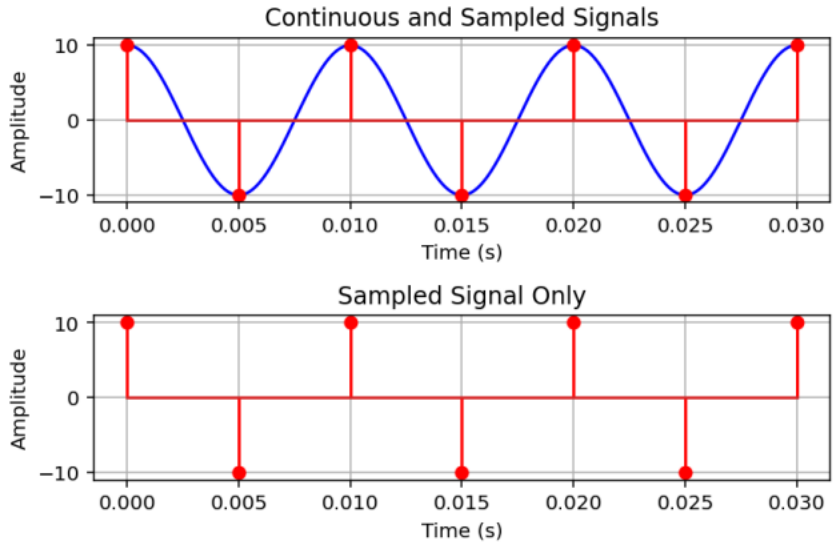


Figure 2.5: (a) Continuous-time signal and its samples obtained by sampling at a rate of 200Hz
(b) Samples of the continuous-time signal in (a)

Finally, we will sample the signal at a frequency lower than the Nyquist rate, and we will observe aliasing. If the sample frequency drops too low, the samples seem to come from a different signal. For example, if the signal $x_c(t) = 10\cos(200\pi t)$ is sampled at a frequency of 150Hz, the resulting discrete time signal is,

$$\begin{aligned} \text{x[n]} &= x(nT_s) = 10\cos(200\pi n T_s) \\ &= 10\cos\left(200\pi n \frac{1}{150}\right) \\ &= 10\cos\left(\frac{4\pi n}{3}\right) = 10\cos\left(\frac{2\pi n}{3}\right) = 10\cos\left(2\pi\frac{n}{3}\right) \end{aligned}$$

Therefore, the sampled signal will be an alias of another signal $x_{ac}(t) = 10\cos(100\pi t)$. Let us verify the result from the following code. The result is shown in the Figure 2.6.

```python
import numpy as np
import matplotlib.pyplot as plt

Fo = 100         # Original Signal Frequency in Hz
To = 1/Fo        # Original Signal Period in seconds
Fo1 = 50         # Aliased Siganl Frequency in Hz
To1 = 1/Fo1      # Aliased signal period in seconds
Fs = 150         # Sampling Frequency < Nyquist rate
Ts = 1/Fs        # Sampling Period in seconds
t = np.linspace(0, 3 * To + Ts, 1000)
xc = 10 * np.cos(2 * np.pi * Fo * t)        # Continuous signal
t1 = np.arange(0, 3 * To + Ts, Ts)
xs = 10 * np.cos(2 * np.pi * Fo * t1)
t2 = np.linspace(0, 3 * To+ Ts, 1000)
xc2 = 10 * np.cos(2 * np.pi * Fo1 * t2) # Discrete signal
plt.plot(t, xc, 'b')
plt.plot(t2, xc2, 'g--')
plt.stem(t1, xs, 'r')
plt.legend([f'Original Signal ({Fo} Hz)',
            f'Aliased Signal ({Fo1} Hz)',
            f'Sampled Values ({Fs} Hz)'], loc=1)
plt.title('Aliasing: Sampling Below the Nyquist Rate')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.grid()
```

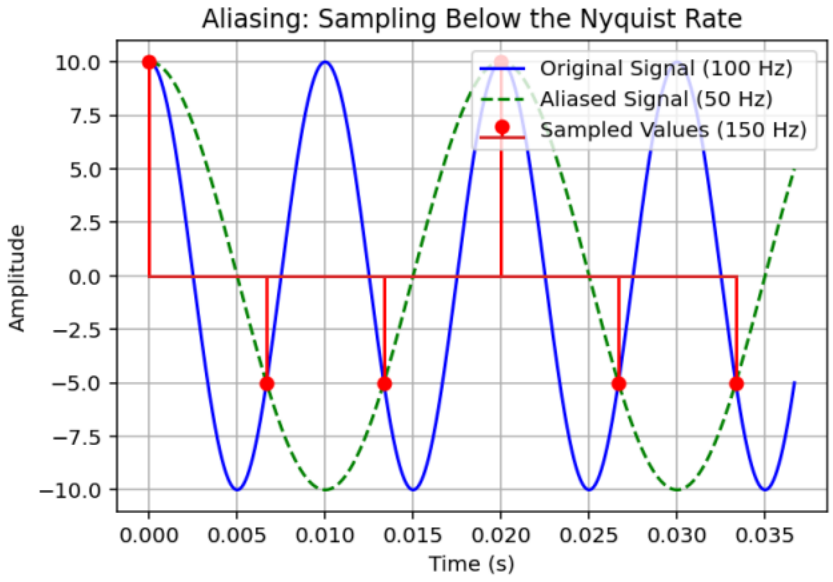


Figure 2.6: Result of Aliasing in the time domain

2.4. Quantization

Quantization is the process of mapping a large set of values to a smaller, discrete set. If the signal values are represented using b bits, there are 2^b distinct values that the signal can assume. These are known as

the Quantization levels. Figure 2.7 visually demonstrates the critical role of quantization levels in signal processing. The number of levels determines the granularity or resolution of the digital signal.

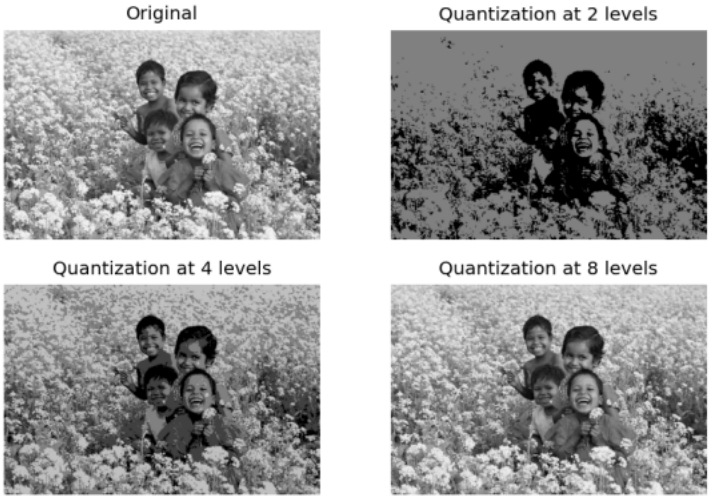


Figure 2.7: Effect of Quantization on an Image

Quantization can be broadly classified into two main types: (a) *uniform* and (b) *non-uniform* quantization. In a *uniform quantizer*, the mapped input signal is divided into steps of equal size. The two most common types of uniform scalar quantizers are the (a) *mid-tread* and (b) *mid-rise quantizers*. Mid-tread quantizers are often considered better than mid-rise quantizers because they suppress low-magnitude noise and effectively handle zero-valued input. Therefore we will investigate the mid-tread quantizers in this experiment.

2.4.1. Mid-Tread Quantizer

The relationship between the input and output of a mid-tread uniform quantizer is given by

$$y[n] = Q \times \left\lfloor \frac{x[n]}{Q} + \frac{1}{2} \right\rfloor \tag{2.3}$$

In the above equation, $x[n]$ represents the input signal to be quantized and $y[n]$ represents the quantized signal, Q denotes the quantization step size. The expression for quantization step size can be computed by

$$Q = \frac{x_{max} - x_{min}}{L} = \frac{Dynamic\ range}{L} \tag{2.4}$$

where, $Dynamic\ range$ represents the difference between the maximum and minimum value of the signal and L denotes the number of reconstruction levels (quantization levels). The expression for L is given by

$$L = 2^b \tag{2.5}$$

where, b is the number of bits used to represent the signal.

We will perform a uniform mid-tread quantization on a 5 Hz sawtooth signal, experimenting with different bit rates of 1, 2, 4, and 8. As the number of bits increases, the quantized signal will more closely resemble the original input signal. The Python code to perform this experiment is shown below, and the simulation results are displayed in the Figure 2.8.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

t=np.linspace(0,1,100)
x=signal.sawtooth(2*np.pi*5*t)  #Generate the input signal
```

```
DR=np.max(x)-np.min(x)           #Dynamic range
b=[1,2,4,8]                      #Number of bits
for i in range(len(b)):
    L=2**b[i]                    #Quantization level
    q=DR/L                       #Quantization step size
    y=np.sign(x)*q*np.floor((abs(x)/q)+(1/2))
    plt.subplot(2,2,i+1)
    plt.plot(t,x,'b')
    plt.plot(t,y,'r')
    plt.xlabel('Time')
    plt.ylabel('Amplitude')
    plt.title(f'Quantization with {b[i]} bits')
    plt.tight_layout()
```

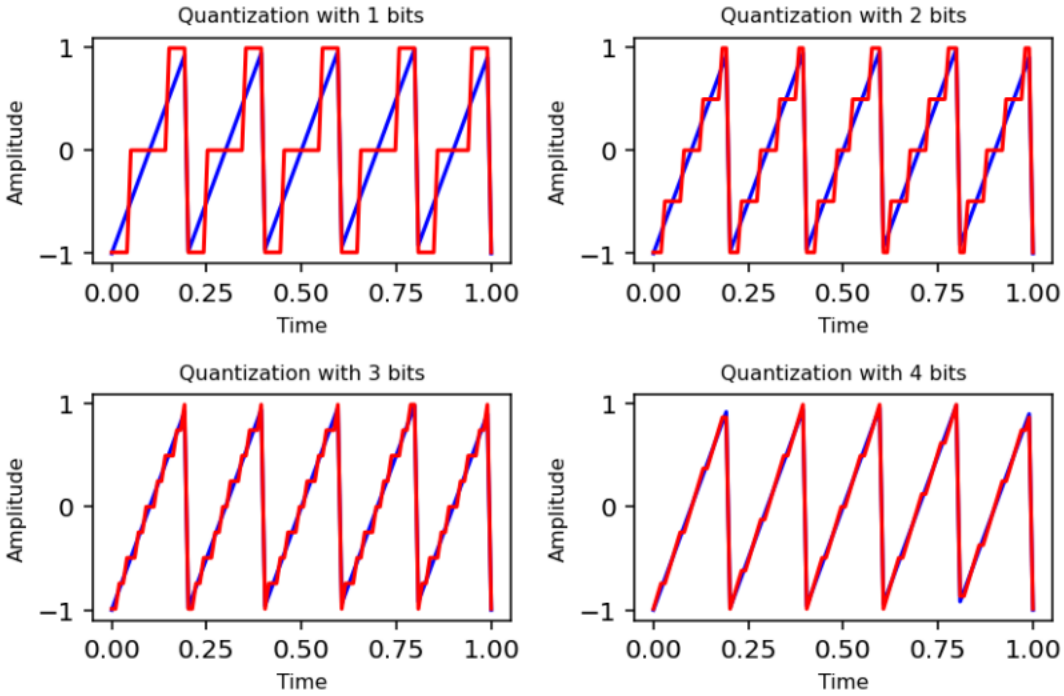


Figure 2.8: Quantization of an input sawtooth signal using mid-tread quantizer

2.4.2. Quantization Levels

Quantization is a fundamental, non-linear, and irreversible process that maps a large set of values to a smaller, finite set. This process inherently leads to a loss of information, which is quantified as quantization error. The error signal is defined as the difference between the reconstructed quantized signal and the original input signal. And heavily depends on the quantization levels. The quantization levels are defined by the Equation 2.5. A higher number of bits, and consequently, more quantization levels, reduces the quantization error. This reduction in error directly translates to a more accurate and higher-fidelity digital signal. This principle applies to all forms of digital media, from audio and video to images. The following code illustrates this phenomenon.

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

```

```
num_bits = [1, 2, 3, 4]
amplitude_max = 2
frequency = 2
sampling_rate = 500
duration = 1
t = np.linspace(0, duration, sampling_rate * duration)
analog_sig = amplitude_max * np.sin(2 * np.pi * frequency * t)
for i in range(0, len(num_bits), 1):
  num_levels = 2**num_bits[i]
  quantization_step = (2*amplitude_max)/num_levels
  quantized_sig = np.round((analog_sig+amplitude_max)/quantization_step)
  quantized_sig_approx = (quantized_sig*quantization_step)-amplitude_max
  quantization_error = analog_sig - quantized_sig_approx
  plt.subplot(2,2,i+1)
  plt.plot(t, analog_sig, 'y', linewidth=4)
  plt.plot(t, quantized_sig_approx, 'b', linewidth=1)
  plt.plot(t, quantization_error, 'r', linewidth=1)
  plt.title(f'Quantization at {num_bits[i]} bits')
  plt.xlabel('Time (s)')
  plt.ylabel('Amplitude')
  plt.tight_layout()
  plt.grid()
```

The result of the above code is shown in the Figure 2.9, which displays the relationship between an original analog signal and its digital representation. The green line represents the original continuous analog signal. The black line is the approximate quantized signal after being quantized to a discrete number of levels. The red line displays the quantization error, which is the difference between these two signals at each point in time.

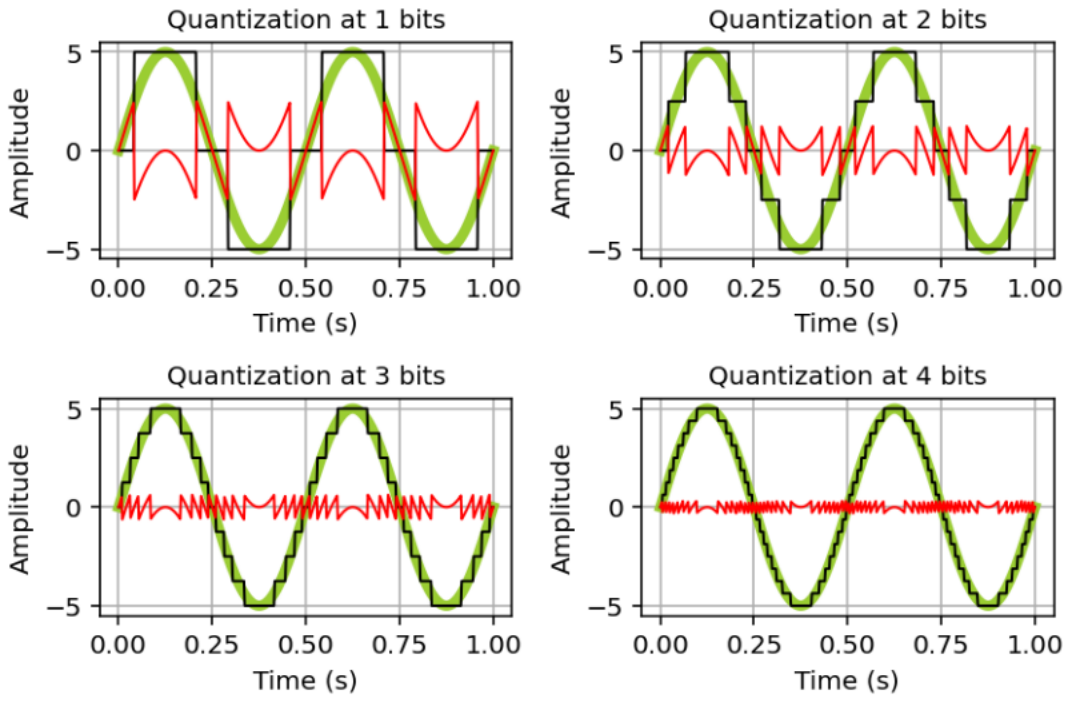


Figure 2.9: Effects of number of bits on signal quantization

2.5. Signal Reconstruction

Signal reconstruction is an attempt to obtain the continuous-time signal from the samples. This is also termed as Interpolation. Different types of interpolation schemes include (1) *zero-order hold interpolation*, (2) *first-order hold or linear interpolation*, and (3) *ideal interpolation*.

Between zero-order hold, first-order hold, and ideal interpolation, the ideal interpolation is considered the best because it perfectly reconstructs the original continuous signal from its discrete samples, assuming the sampling rate meets the Nyquist-Shannon sampling theorem.

— Zero-order hold is the simplest method. It holds the value of a sample until the next sample is available, creating a stair-step approximation of the original signal. This method introduces significant distortion and is not an accurate representation of the signal.
— First-order hold is a slight improvement. It connects consecutive sample points with a straight line, which is a linear approximation of the original signal. While better than a zero-order hold, it still results in a piecewise linear approximation that can smooth out sharp changes and introduce errors.
— Ideal interpolation, also known as $sinc$ interpolation, is mathematically perfect. It uses the $sinc$ function to "fill in the gaps" between samples. Each sample is multiplied by a shifted and scaled $sinc$ function, and all of these functions are summed to reconstruct the original signal. This method does not introduce any distortion or loss of information, making it the theoretical ideal for signal reconstruction.

While zero-order and first-order holds are practical and computationally simple, they are approximations. Ideal interpolation, based on the $sinc$ function, offers a perfect reconstruction of the original signal, which is why it is considered the superior method. Therefore, we consider the ideal interpolation for signal reconstruction in this experiment.

In order to reconstruct a signal from its samples, we need to pass the samples through a low pass filter whose cut-off frequency(f_c) satisfies the condition: $f_N \leq f_c \leq (f_s - f_N)$. In time domain this implies multiplying the samples by an interpolation function or $sinc$ function. In other word, if we consider low pass filter (LPF) a system, the output of the system will be the convolution sum of the samples and the impulse response ($h_r(t)$) of the LPF. $h_r(t)$ for an LPF with cutoff frequency $\frac{\pi}{T_s}$ is given by

$$h_r(t) = \frac{\sin\left(\frac{\pi t}{T_s}\right)}{\frac{\pi t}{T_s}} \tag{2.6}$$

This impulse response is shown in Figure 2.10.

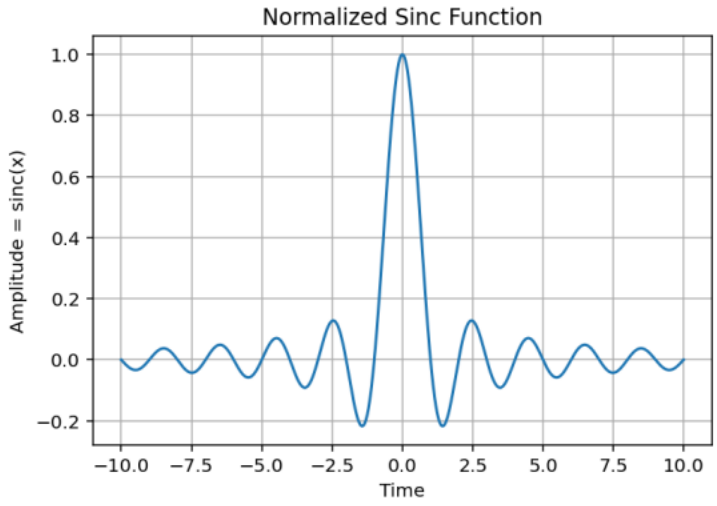


Figure 2.10: Impulse Response of an ideal Low Pass Filter.

Below is the python code to generate the above $sinc$ function.

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-10, 10, 1000)
y = np.sinc(x)
plt.plot(x, y)
plt.title("Normalized Sinc Function")
plt.xlabel("Time")
plt.ylabel("Amplitude = sinc(x)")
plt.grid()
plt.show()
```

The final reconstructed signal $x_r(t)$ which is the convolution sum of $x[n]$ and $h_r(t)$ is given by the formula:

$$x_r(t) = \sum_{n=-\infty}^{\infty} x[n] \frac{\sin\left(\frac{\pi(t-nT_s)}{T_s}\right)}{\frac{\pi(t-nT_s)}{T_s}} \tag{2.7}$$

Use the Equation 2.7 to reconstruct a signal from its samples. Consider sampling frequency higher than the Nyquist rate so that you can properly reconstruct the original signal.

```python
import numpy as np
import matplotlib.pyplot as plt

Fo = 100        # Signal frequency in Hz
To = 1 / Fo     # Signal period
Fs = 800        # Sampling frequency in Hz
Ts = 1 / Fs     # Sampling period
tc = np.arange(0, 3 * To, To / 100)
xc = 5 * np.cos(200 * np.pi * tc)
ts = np.arange(0, 3 * To + Ts, Ts)
xs = 5 * np.cos(200 * np.pi * ts)
N = len(ts)     # number of samples

xr = np.zeros(len(tc))
sinc_fun = np.zeros((N, len(tc)))

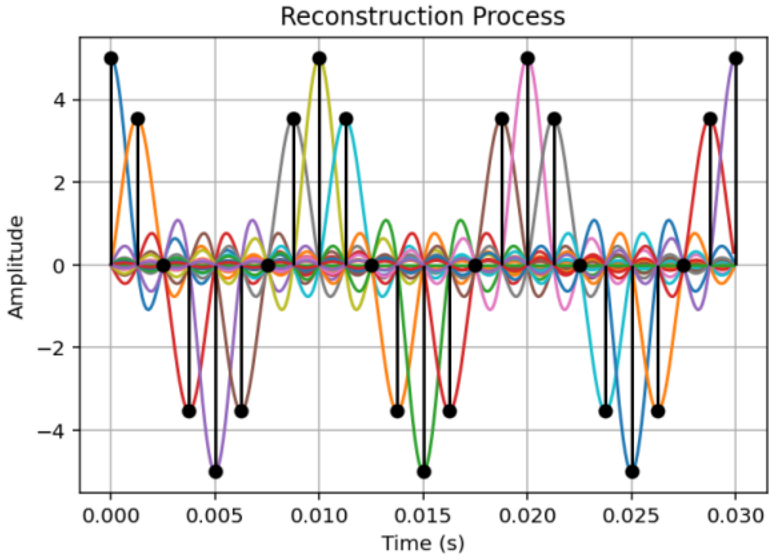
for t_idx in range(len(tc)):
    for n_idx in range(N):
        # Sinc function, calculated for each sample and each time point
        sinc_val = np.sinc((tc[t_idx] - n_idx * Ts) / Ts)
        # Populate the sinc_fun matrix for later visualization
        sinc_fun[n_idx, t_idx] = sinc_val
        # This is the sum of x(nTs)*sinc((t-nT)/T)
        xr[t_idx] += xs[n_idx] * sinc_val

plt.figure()
plt.grid()
for n_idx in range(N):
    plt.plot(tc, xs[n_idx] * sinc_fun[n_idx, :])
plt.stem(ts, xs, 'k', markerfmt='ko', basefmt=" ", label='Sampled Points')
plt.title('Reconstruction Process')
plt.xlabel('Time (s)')
```

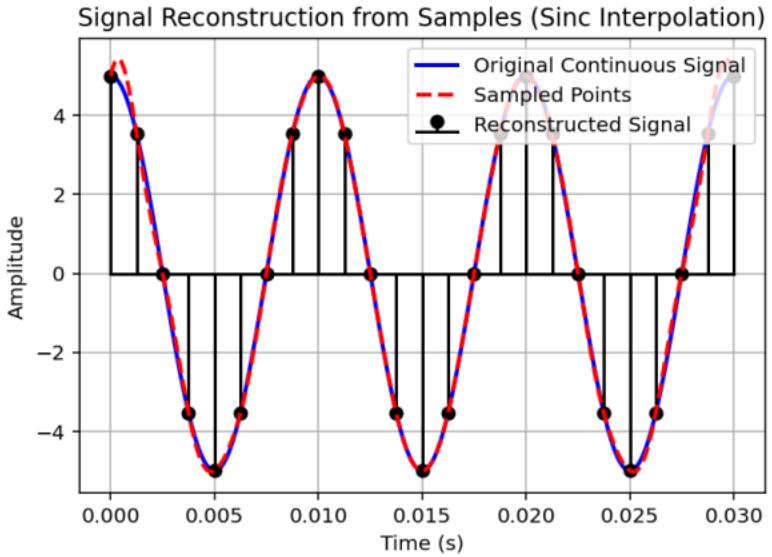
```
plt.ylabel('Amplitude')

plt.figure()
plt.plot(tc, xc, 'b-', linewidth=2, label='Original Continuous Signal')
plt.stem(ts, xs, 'k', markerfmt='ko', label='Sampled Points')
plt.plot(tc, xr, 'r--', linewidth=2, label='Reconstructed Signal')
plt.title('Signal Reconstruction from Samples (Sinc Interpolation)')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.legend()
plt.grid()
```

By running the above code, the original signal, its samples and reconstructed signal - all will be displayed just like the Figure 2.11.



(a) Sinc function, calculated for each sample and each time point



(b) Sinc Interpolation

Figure 2.11: Reconstruction of a signal from its samples

Post-laboratory Problems

1. Consider $\mathbf{x(t) = 10\cos(120\pi t) + 5\sin(100\pi t + 30^o) + 4\sin(150\pi t + 45^o)}$. Sample the signal for i) $F_s = 4F_o$, ii) $F_s = 2F_o$ and iii) $F_s = F_0$ where F_o is the maximum frequency content of $x(t)$. Show the original signal and its samples in one figure for 3 cycles.
2. Consider a signal $\mathbf{x(t) = 10\cos(250\pi t + 60^o) + 5\sin(200\pi t + 75^o)}$. Convert the signal into a sequence $x[n]$ for $-10 \leq n \leq 10$ by choosing an appropriate sampling frequency. Show i) original signal, ii) sampled signal, iii) discrete signal, and iv) final output.
3. Consider $\mathbf{x(t) = \frac{1}{2}\sin(14\pi t) + \frac{1}{3}\sin(18\pi t) + \frac{1}{5}\sin(24\pi t) + \frac{1}{7}\sin(30\pi t)}$. Sample the signal on the interval $0 \leq t \leq 2$ at i) $F_s = 5F_o$ and ii) $F_s = 1.5F_o$. Where F_o is the maximum frequency content of $x(t)$. Reconstruct the signal in each case from its samples. What is the alias signal's time domain expression in case (ii).
4. There is another way to reconstruct the original signal by using a built-in function `resample()` which performs ideal (sinc) interpolation. It takes the original signal, and the desired number of output samples.. Write a Python code using this function to reconstruct the signal of the previous problem.
5. Solve all the exercise problems of the textbook from page 64 to page 67.

Laboratory 3

Generation and Operation on Discrete-Time Sequence

3.1. Objective

The objective of this laboratory is to provide a practical understanding of discrete-time signals and fundamental mathematical operations within digital signal processing. Students will first learn to generate several standard discrete-time signals, including the unit sample, unit step, and unit ramp sequences. Following this, the focus will shift to applying essential signal manipulation techniques. The lab will cover key operations such as folding, shifting, and scaling of discrete-time sequences. Finally, students will perform more complex operations, including the convolution sum to analyze system responses and both autocorrelation and cross-correlation to explore the relationships between signals.

3.2. Generation of Discrete-Time Signals

Signals are categorized as either continuous or discrete, depending on the nature of their independent variables. A continuous-time signal is represented as x(t), where the independent variable, t, can take on any real value. Think of it like a smooth, unbroken line on a graph, such as a sine wave.

In contrast, a discrete-time signal is denoted as x[n], and its independent variable, n, can only take integer values. These signals appear as a series of distinct points on a graph, rather than a continuous line. For instance, a sequence of daily stock prices is a discrete-time signal. These signals are crucial because they can be used to construct more complex signals.

There are several types of discrete-time signals, such as the unit sample, unit step, unit ramp, and real and complex exponential signals, etc.

Let say, $x[n] = \{4, 6, 1, -1, -2, \underset{\uparrow}{1}, 3, 2, 5, 4, -3\}$ is a discrete signal. The following Python code demonstrate the process to generate $x[n]$. Figure 3.1 shows the generated signal.

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.array([4,6,1,-1,-2,1,3,2,5,4,-3])
n = np.arange(-5,6)

plt.stem(n,x)
plt.xticks(n)
plt.xlabel('n')
plt.ylabel('x[n]')
plt.grid(linestyle='--')
```

3.2.1. Generation of the Unit Impulse Sequence

It is also called *unit sample sequence* or *Kronecker Delta function* and is denoted by $\delta[n]$. It is defined by

$$\delta[n] = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases} \tag{3.1}$$

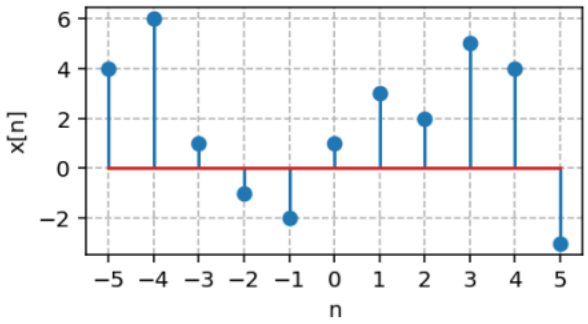


Figure 3.1: Discrete Signal, $x[n] = \{4, 6, 1, -1, -2, \underset{\uparrow}{1}, 3, 2, 5, 4, -3\}$

More generally,

$$\delta[n - n_0] = \begin{cases} 1, & n = n_0 \\ 0, & n \neq n_0 \end{cases} \tag{3.2}$$

The Python code to generate the unit impulse sequence using the logical operation is given below. The corresponding output of the following code is shown in Figure 3.2.

```python
import numpy as np
import matplotlib.pyplot as plt

def unit_impulse(n0, n1, n2):
        """
        Args:
        n0 (int): The location of the impulse.
        n1 (int): The starting index of the sequence.
        n2 (int): The ending index of the sequence.
        """
        n = np.arange(n1, n2 + 1, 1)
        x = (n == n0)
        return x, n


n0 = 0
n1 = -5
n2 = 5
x, n = unit_impulse(n0, n1, n2)
plt.stem(n, x)
plt.title('\delta[n]')
plt.xlabel('n (Index)')
plt.ylabel('x[n] (Amplitude)')
plt.grid(True, linestyle='--')
```

3.2.2. Generation of the Unit Step Sequence

The unit step sequence, denoted by $u[n]$ is defined as:

$$u[n] = \begin{cases} 1, & n \geq 0 \\ 0, & n < 0 \end{cases} \tag{3.3}$$

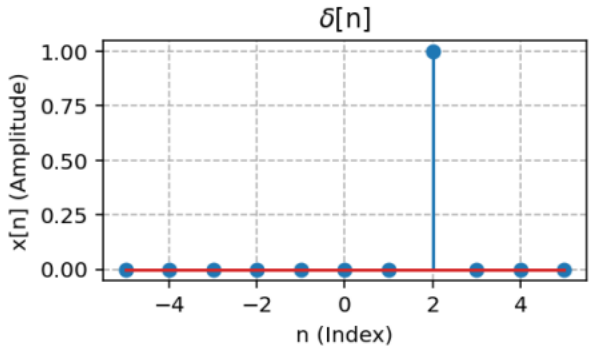


Figure 3.2: Unit impulse sequence at $n = 2$

More generally,

$$u[n - n_0] = \begin{cases} 1, & n \geq n_0 \\ 0, & n < n_0 \end{cases} \tag{3.4}$$

The python code, which generates unit step signal is shown below, and the corresponding output is shown in Figure 3.3.

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

def unit_step(n0, n1, n2):
        n = np.arange(n1, n2 + 1, 1)
        x = (n >= n0)
        return x, n


n0 = 2
n1 = -5
n2 = 8
x, n = unit_step(n0, n1, n2)
plt.stem(n, x)
plt.title('u[n]')
plt.xlabel('n (Index)')
plt.ylabel('x[n] (Amplitude)')
plt.grid(True, linestyle='--')
plt.xticks(n)
```

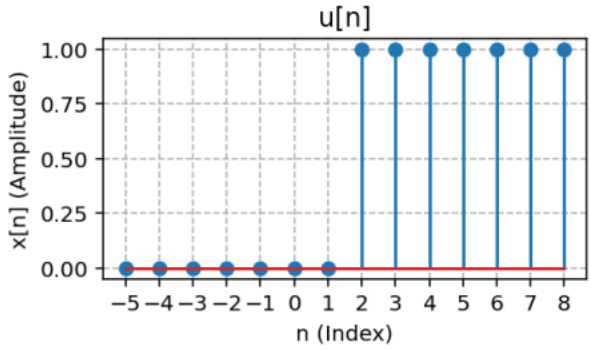


Figure 3.3: Unit step sequence at $n = 0$

3.2.3. Generation of the Unit Ramp Signal

The mathematical expression of the unit ramp sequence $r[n]$ is written as

$$r[n] = \begin{cases} n, & n \geq 0 \\ 0, & n < 0 \end{cases} \tag{3.5}$$

More generally,

$$r[n - n_0] = \begin{cases} n, & n \geq n_0 \\ 0, & n < n_0 \end{cases} \tag{3.6}$$

The python code, which generates unit ramp signal is shown below, and the corresponding output is shown in Figure 3.4.

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

def unit_ramp(n0, n1, n2):
        n = np.arange(n1, n2 + 1, 1)
        x = (n - n0) * (n >= n0)
        return x, n


n0 = 2
n1 = -5
n2 = 8
x, n = unit_ramp(n0, n1, n2)
plt.stem(n, x)
plt.title('r[n]')
plt.xlabel('n (Index)')
plt.ylabel('x[n] (Amplitude)')
plt.grid(True, linestyle='--')
plt.xticks(n)
```

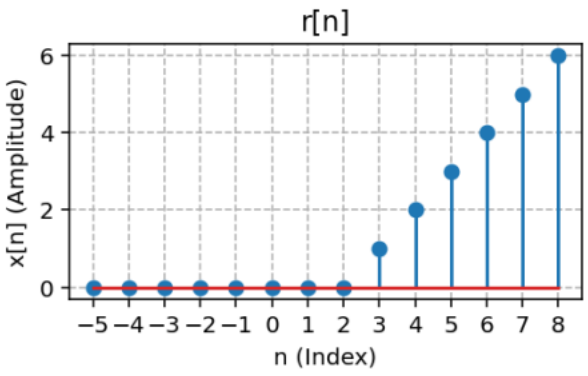


Figure 3.4: Unit step sequence at $n = 2$

3.3. Mathematical Operation on Discrete-Time Signals

3.3.1. Amplitude scaling

If x[n] is the input signal, the scaling of the signal $x[n]$ by a factor of A is represented as

$$y[n] = Ax[n] \tag{3.7}$$

Here, if $A > 1$, the operation is called as amplification, $A < 1$ represents attenuation. If $A = 1$, the output follows the input, it is called as input follower or buffer.

Now, generate unit sample signal and perform the amplitude scaling for three different values of A, namely: $A = 1$, $A = 2$, $A = 0.5$ and $A = -3$. The input and output signal are plotted in Figure 3.5.

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

n = np.arange(-10, 11, 1)
A = [1, 2, 0.5, -3]
x = (n>=0)

for i, A in enumerate(A):
        y = A * x
        plt.subplot(2, 2, i+1)
        plt.stem(n, y)
        plt.xlabel('n')
        plt.ylabel(f'y[n]=A[{i}]x[n]')
        plt.grid(True, linestyle='--')
plt.tight_layout()
```

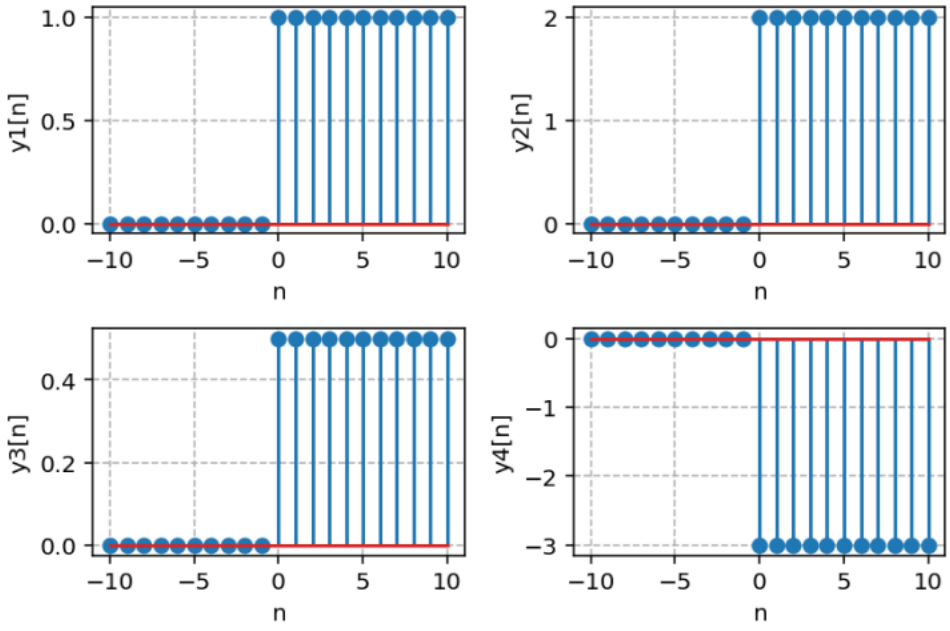


Figure 3.5: Amplitude Scaling

3.3.2. Amplitude Shifting

If $x[n]$ is the input signal, the amplitude shifting of the signal $x[n]$ by a factor of C is represented as

$$y[n] = x[n]C \tag{3.8}$$

Let $x[n]$ represent the discrete-time sinusoidal signal, and perform the DC offset of this signal x[n] to obtain the signals $y1[n] = x[n] + C$ and $y2[n] = x[n] - C$. The value of C for this experiment is to be chosen as 5. The following python code performs this task and the corresponding output is shown in Figure 3.6.

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

Fs = 100
n = np.arange(0, 40, 1)
x = np.sin(2*np.pi*(5/Fs)*n)
plt.subplot(3,1,1)
plt.stem(n,x),
plt.xlabel('n')
plt.ylabel('x[n]')
offset = [5,-5]   # Perform DC offset
y1 = x + offset[0]
y2 = x + offset[1]
plt.subplot(3,1,2)
plt.stem(n,y1)
plt.xlabel('n')
plt.ylabel('y1[n]')
plt.subplot(3,1,3)
plt.stem(n,y2)
plt.xlabel('n')
plt.ylabel('y2[n]')
plt.tight_layout()
```

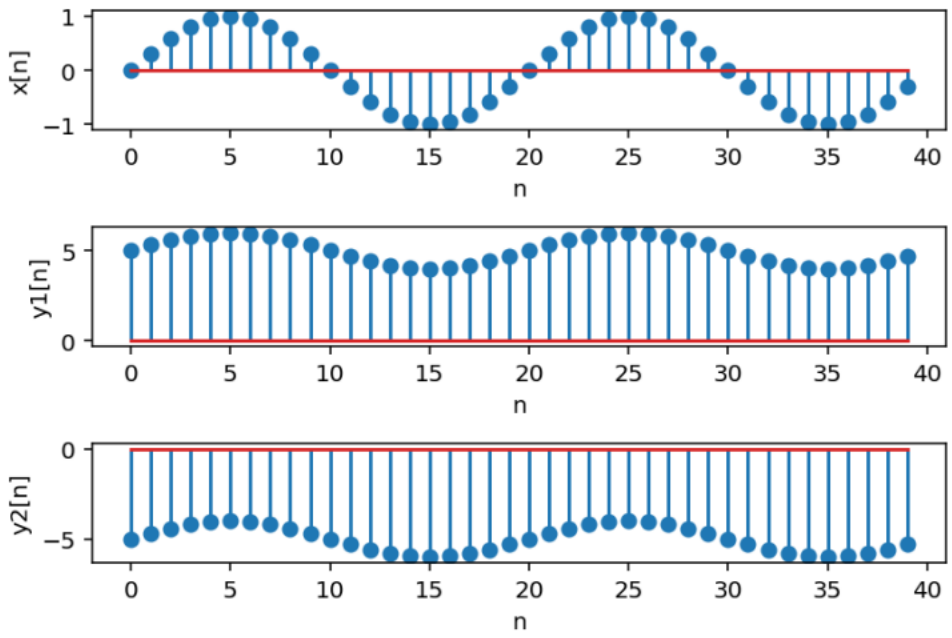


Figure 3.6: Amplitude Shifting (DC Offset)

3.3.3. Product of Two Signals

The product of two signals x1[n] and x2[n] is represented by

$$y[n] = x_1[n] \times x_2[n] \tag{3.9}$$

The following Python code obtains the product of the two signals given by $x_1[n] = sin(2\pi f_1 n)$ and $x_2[n] = cos(2\pi f_2 n)$. Consider f1 = f2 = 5 Hz. The plot is shown in Figure 3.7.

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

def signal_mult(x1, n1, x2, n2):
        min_n = min(np.min(n1), np.min(n2))
        max_n = max(np.max(n1), np.max(n2))
        n = np.arange(min_n, max_n + 1)
        y1 = np.zeros(len(n))
        y2 = np.zeros(len(n))
        y1[np.where(np.isin(n, n1))] = x1
        y2[np.where(np.isin(n, n2))] = x2
        y = y1*y2
        return y, n

n1 = np.arange(-2, 3)
n2 = np.arange(-3, 2)
x1 = np.array([1, -2, 3, -4, 5])
x2 = np.array([5, 4, -3, 2, 3])
y, n = signal_mult(x1, n1, x2, n2)

plt.subplot(3, 1, 1)
plt.stem(n1, x1)
plt.xlabel('n')
plt.ylabel('$x_1[n]$')
plt.xticks(n1)

plt.subplot(3, 1, 2)
plt.stem(n2, x2)
plt.xlabel('n')
plt.ylabel('$x_2[n]$')
plt.xticks(n2)

plt.subplot(3, 1, 3)
plt.stem(n, y)
plt.xlabel('n')
plt.ylabel('$y[n]$')
plt.xticks(n2)

plt.tight_layout()
```

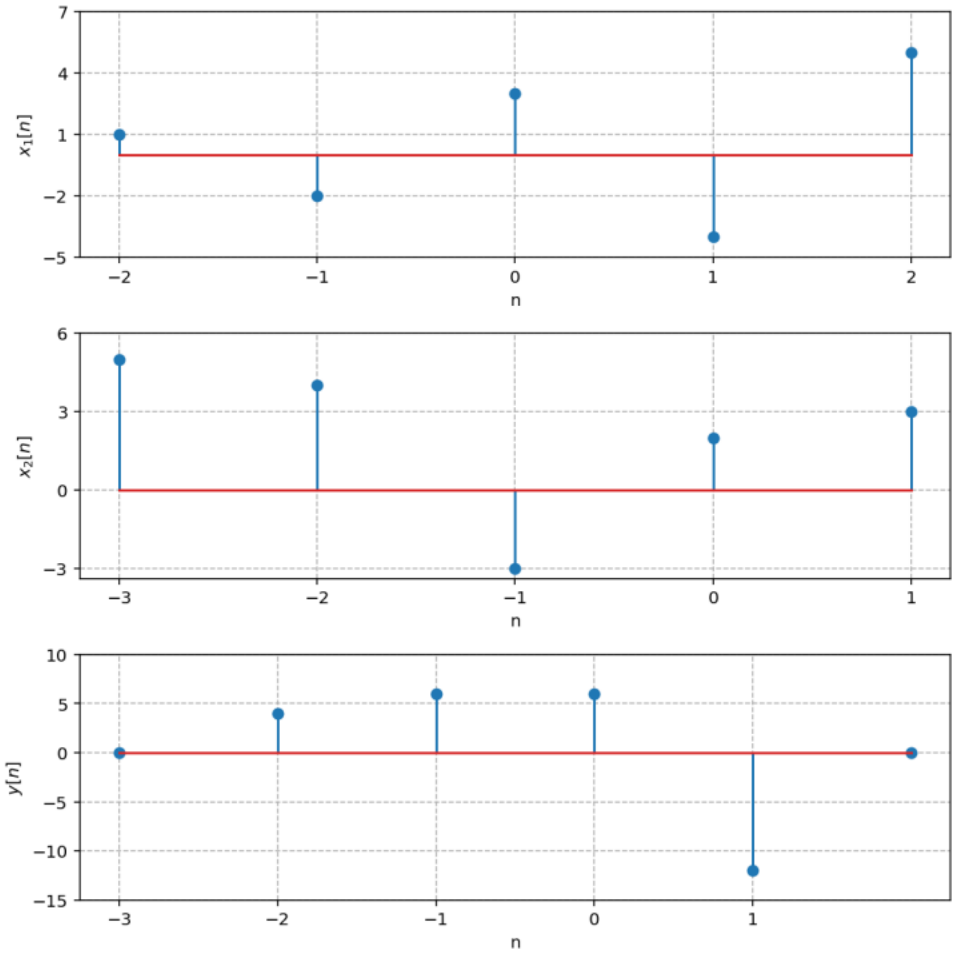


Figure 3.7: Product of two signals

3.3.4. Signal Addition

The signal addition results in a change in the amplitude of the signal. Two signals $x_1[n]$ and $x_2[n]$ are added together to obtain the resultant output signal $y[n]$, which is given by

$$y[n] = x_1[n] + x_2[n] \tag{3.10}$$

Let $x_1[n] = sin(2\pi f_1 n)$ and $x_2[n] = cos(2\pi f_2 n)$. The signal $y[n]$ is the addition of these two discrete sinusoidal signals. The python code which performs this task is shown below, and the corresponding output is shown in Figure 3.8.

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

def signal_add(x1, n1, x2, n2):
        min_n = min(np.min(n1), np.min(n2))
        max_n = max(np.max(n1), np.max(n2))
        n = np.arange(min_n, max_n + 1)
        y1 = np.zeros(len(n))
        y2 = np.zeros(len(n))
        y1[np.where(np.isin(n, n1))] = x1
        y2[np.where(np.isin(n, n2))] = x2
        y = y1+y2
        return y, n
```

```
14
15  n1 = np.arange(-2, 3)
16  n2 = np.arange(-3, 2)
17  x1 = np.array([1, -2, 3, -4, 5])
18  x2 = np.array([5, 4, -3, 2, 3])
19  y, n = signal_add(x1, n1, x2, n2)
20
21  plt.subplot(3, 1, 1)
22  plt.stem(n1, x1)
23  plt.xlabel('n')
24  plt.ylabel('$x_1[n]$')
25  plt.xticks(n1)
26
27  plt.subplot(3, 1, 2)
28  plt.stem(n2, x2)
29  plt.xlabel('n')
30  plt.ylabel('$x_2[n]$')
31  plt.xticks(n2)
32
33  plt.subplot(3, 1, 3)
34  plt.stem(n, y)
35  plt.xlabel('n')
36  plt.ylabel('$y[n]$')
37  plt.xticks(n2)
38
39  plt.tight_layout()
```

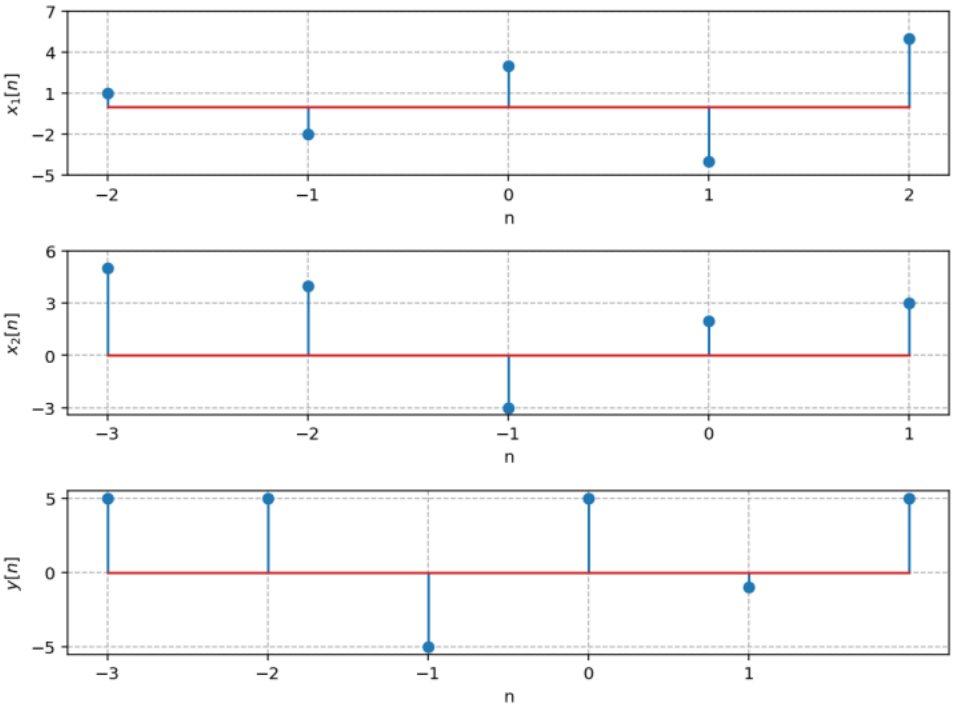


Figure 3.8: Addition of two signals

3.3.5. Time Scaling Operation

Time scaling operations can be classified into two types, namely, (1) *downsampling* and (2) *upsampling*.

Downsampling

The downsampling of the signal $x[n]$ by a factor of M is represented as

$$y[n] = x[Mn] \tag{3.11}$$

where M is an integer. Here $M-1$ samples will be discarded between two consecutive samples. Downsampling by a factor of 2 is represented as $y[n] = x[2n]$.

Upsampling

The upsampling of the signal $x[n]$ by a factor of L is represented by

$$y[2n] = x[\frac{n}{L}] \tag{3.12}$$

The upsampling operation is basically inserting $L-1$ zeros between two consecutive samples. For $L = 2$, the above expression can be written as $\frac{n}{2}$.

The following python code performs both downsampling and upsampling by a factor of 2, and the corresponding output is shown in Figure 3.9.

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

def upsampling(x, L):
        N = len(x)              # Get the length of the input signal
        y = np.zeros(L*N)        # Performing upsampling operation
        y[::L] = x
        return y

def downsampling(x, M):
        return x[::M]        # Performing downsampling operation

n = np.arange(-10, 10, 1)
Fs = 100
x = np.sin(2*np.pi*(5/Fs)*n) # The the original signal

M = 2
m_down = n[::M]
y_down = downsampling(x, M)

L = 2
y_up = upsampling(x, L)
m_up = np.arange(0, len(y_up))

plt.subplot(3, 1, 1)
plt.stem(n, x)  # Plot the original signal
plt.title('Original Signal x[n]')
plt.xlabel('n')
plt.ylabel('x[n]')
plt.xticks(n)
plt.grid(linestyle='--')

```

```
plt.subplot(3, 1, 2)
plt.stem(m_down, y_down)
plt.title(f'Downsampled Signal y[n] (M = {M})')
plt.xlabel('n')
plt.ylabel('y[n]')
plt.xticks(m_down)
plt.grid(linestyle='--')

plt.subplot(3, 1, 3)
plt.stem(m_up, y_up)  # Plot the downsampled signal
plt.title(f'Upsampled Signal y[n] (L = {L})')
plt.xlabel('n')
plt.ylabel('y[n]')
plt.grid(linestyle='--')
plt.tight_layout()
```

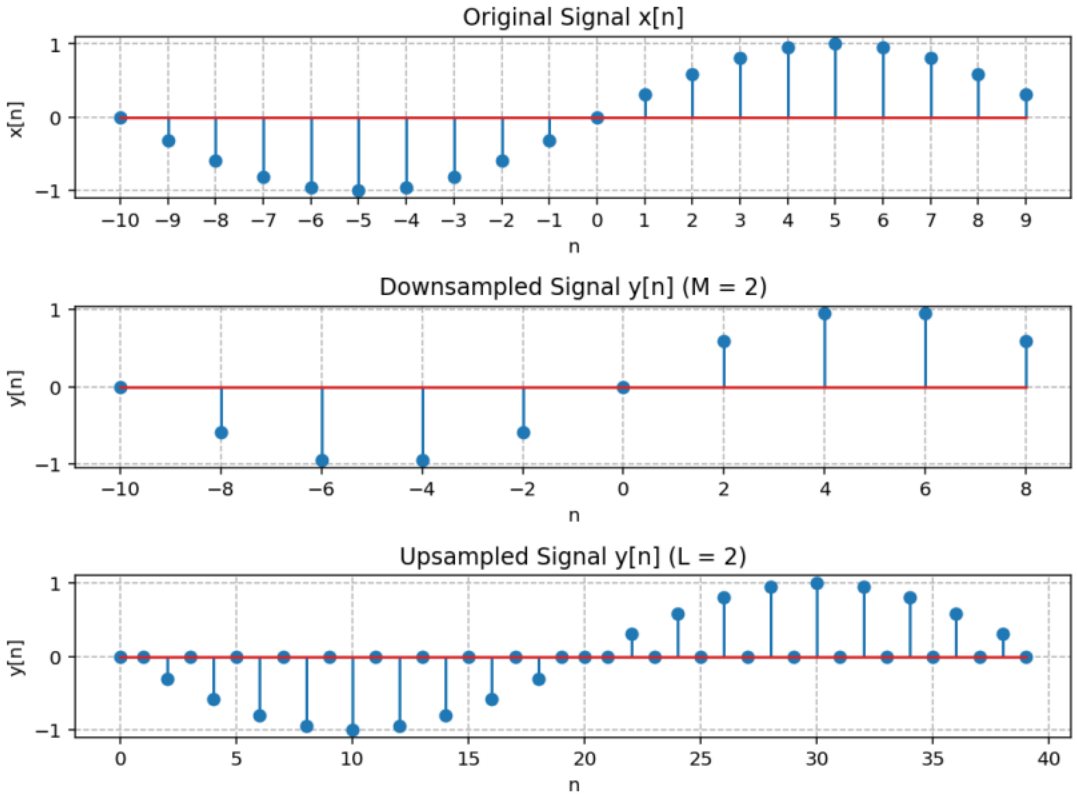


Figure 3.9: Time scaling operation

3.3.6. Time Shifting Operation

The time shifting operation can be broadly classified into two types: (1) *delay operation* and (2) *advance operation*. The *delaying* of the input signal by a factor of k units is expressed as

$$y[n] = x[n - k] \tag{3.13}$$

The *advance* of the input signal $x[n]$ by a factor of k units is expressed as

$$y[n] = x[n + k] \tag{3.14}$$

In the both cases, k must be a positive integer.

The following Python code performs both delay and advance operations by a factor of k units on the unit step signal. First, the unit step signal is generated, and then, it is delayed by a factor of 5 units. Finally, the original unit step signal is advanced by the factor of 5 units. The corresponding output is shown in Figure 3.10.

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

def signal_shift(x, n1, n0):
        n = n1 + n0
        y = x
        return y, n

n=np.arange(-10,11)
x=(n>=0)          # Generation of u[n]
k=5
y1, n1 = signal_shift(x, n, k)
y2, n2 = signal_shift(x, n, -k)

plt.subplot(3,1,1)
plt.stem(n,x)
plt.xticks(n)
plt.xlabel('n')
plt.ylabel('x[n]')
plt.title('u[n]')
plt.xlim(min(n2) - 1, max(n1) + 1)
plt.subplot(3,1,2)
plt.stem(n1,y1)
plt.xticks(n1)
plt.xlabel('n')
plt.ylabel('y1[n]')
plt.title(f'u[n-{k}]')
plt.xlim(min(n2) - 1, max(n1) + 1)
plt.subplot(3,1,3)
plt.stem(n2,y2)
plt.xticks(n2)
plt.xlabel('n')
plt.ylabel('y2[n]')
plt.title(f'u[n+{k}]')
plt.xlim(min(n2) - 1, max(n1) + 1)

plt.tight_layout()
```

3.3.7. Time Reversal Operation

The time reversal of a signal $x[n]$ is a fundamental operation in Digital Signal Processing (DSP). It is denoted as $x[-n]$. This process, also known as *signal folding*, involves flipping the signal horizontally, effectively reflecting it about the vertical axis (the origin). In essence, every sample of the signal is moved to the corresponding negative time index. For example, the value of the signal at $n = 3$ is moved to $n = -3$, and the value at $n = -2$ is moved to $n = 2$.

Following Python code demonstrate the time reversal operation. Figure 3.11 shows the corresponding output.

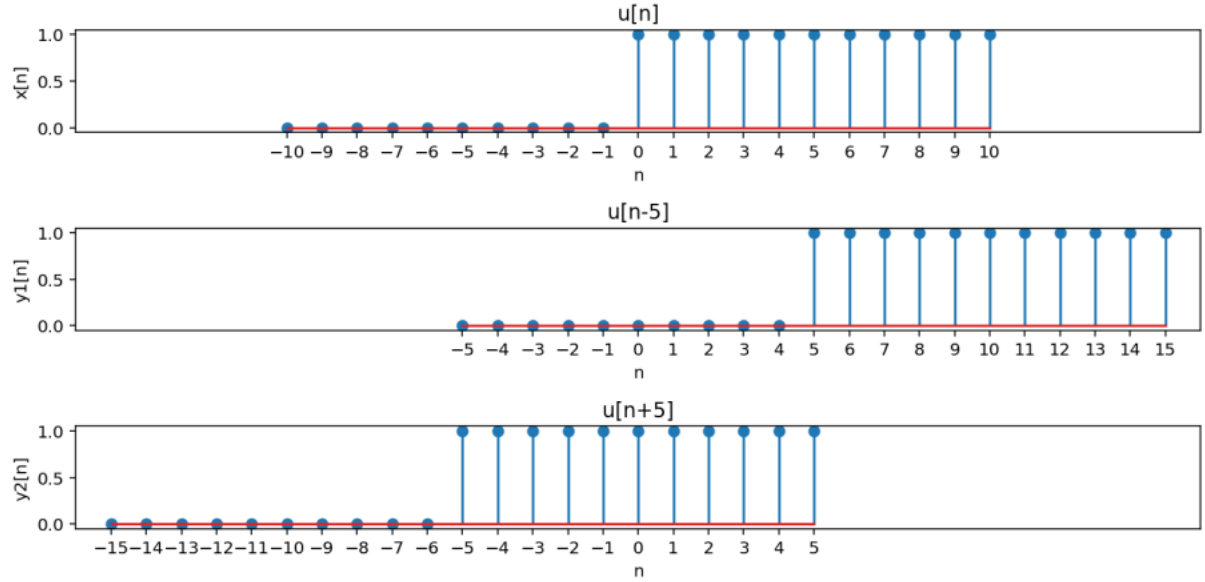


Figure 3.10: Time shifting - delay and advance operation

```python
import numpy as np
import matplotlib.pyplot as plt

def signal_fold(x, n):
        y = np.fliplr([x])[0]   # Flip the signal array left to right
        m = -np.fliplr([n])[0]  # Flip the time index array and negate it
        return y, m

n = np.arange(-10, 11)
x = n
y, m = signal_fold(x, n)

plt.subplot(2, 1, 1)
plt.stem(n, x)
plt.title('Original Signal x[n]')
plt.xlabel('n')
plt.ylabel('x[n]')
plt.xticks(n)
plt.yticks(np.arange(-10,11,5))
plt.grid(True, linestyle='--')

plt.subplot(2, 1, 2)
plt.stem(m, y)
plt.title('Folded Signal y[n] = x[-n]')
plt.xlabel('n')
plt.ylabel('y[n]')
plt.grid(True, linestyle='--')
plt.xticks(m)
plt.yticks(np.arange(-10,11,5))
plt.tight_layout()
```

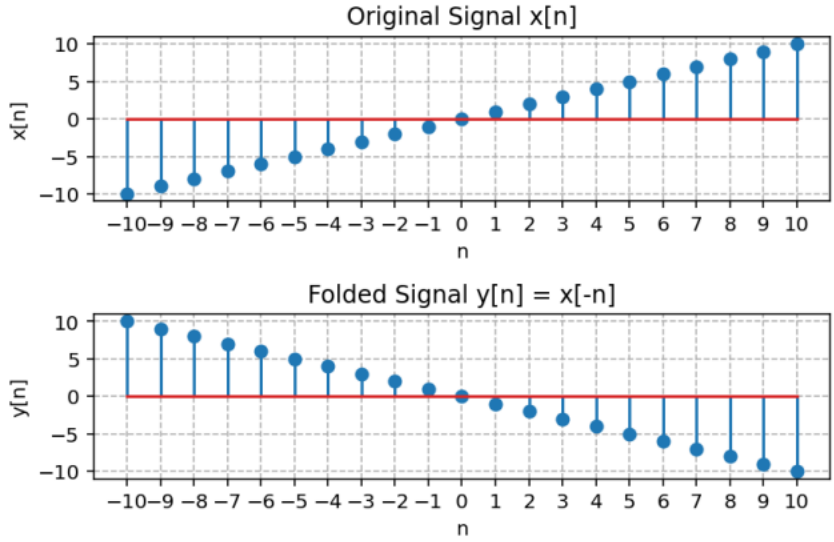


Figure 3.11: Time reversal operation

3.4. Convolution

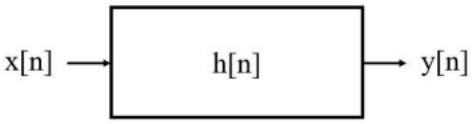


Figure 3.12: Linear Time Invariant System

Convolution is an important operation in digital signal processing, because many DSP algorithms use convolution operations in one form or other. The most common application of convolution operation is filtering. It can be used for signal enhancement. It mathematically describes the relationship between the input, the output, and the impulse response of a linear time-invariant (LTI) system, shown in Figure 3.12. This relationship is defined by the *convolution sum*, which provides a formal method for calculating the output signal, $y[n]$, from an input signal, $x[n]$, and a system's impulse response, $h[n]$. The mathematical expression for discrete-time convolution is given by:

$$y[n] = x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k] \tag{3.15}$$

In this formula, the output at time n is a weighted sum of the input signal, where the weights are determined by the system's impulse response. This operation can be conceptually understood as a *"flipping and shifting"* of the impulse response across the input signal. The result is a profound tool for analyzing how an LTI system modifies an input signal over time. Consequently, convolution is essential for designing digital filters used in a wide array of applications, from audio and image processing to telecommunications.

Following Python code illustrates the fact that the convolution of any signal ($x[n]$) with unit sample signal ($\delta[n]$) will result in the same signal $x[n]$. The output is shown in the Figure 3.13.

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

def convolution_sum(x, nx, h, nh):
        kmin = np.min(nx) + np.min(nh)
        kmax = np.max(nx) + np.max(nh)
```

```
        k = np.arange(kmin, kmax + 1)
        y = np.convolve(x, h)
        return y, k

nx = np.arange(-5,6)
x = 5-np.abs(nx)
h = (nx==0)
nh = nx
y, n = convolution_sum(x, nx, h, nh)

plt.subplot(3,1,1)
plt.stem(nx,x)
plt.xticks(nx)
plt.xlabel(''),
plt.ylabel('x[n]')
plt.subplot(3,1,2)
plt.stem(nh,h)
plt.xticks(nh)
plt.xlabel('n')
plt.ylabel('h[n]')
plt.subplot(3,1,3)
plt.stem(n,y)
plt.xticks(n)
plt.xlabel('n'),
plt.ylabel('y[n]')
plt.tight_layout()
```

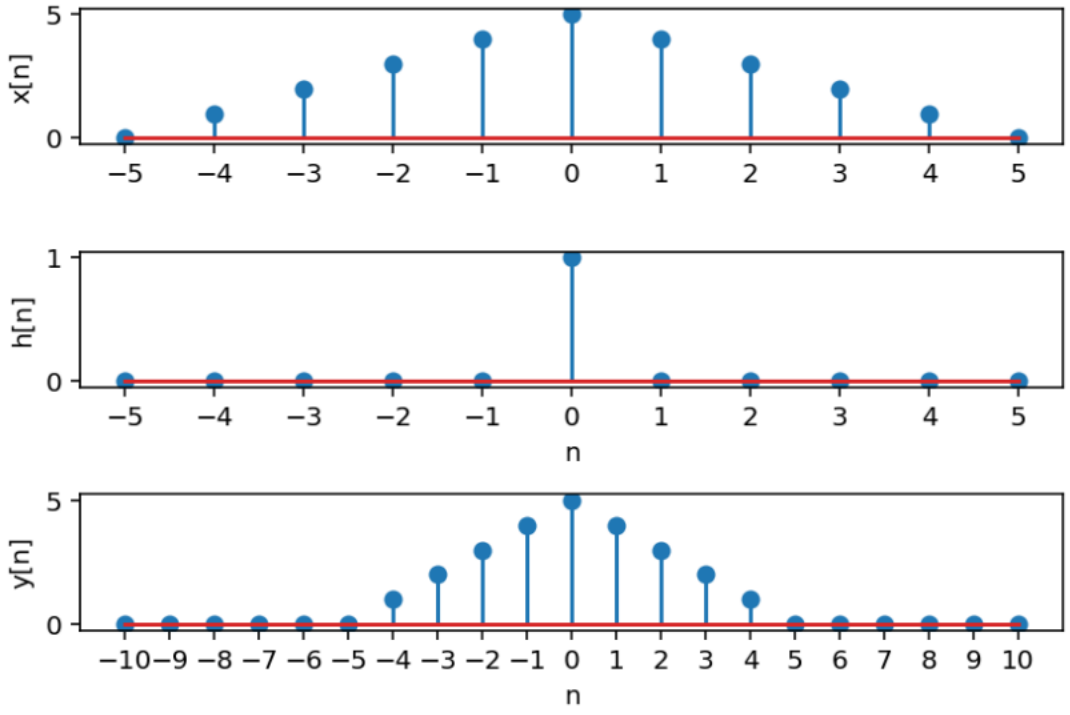


Figure 3.13: Convolution

3.4.1. Properties of Convolution

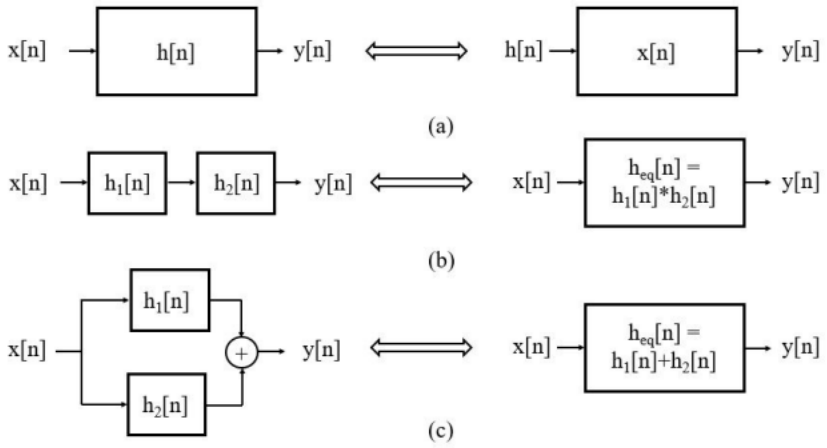


Figure 3.14: Interpretation of convolution properties from a system point of view.

Commutative Property

Commutative property states that the order in which two sequences are convolved is not important. Mathematically, this means that

$$y[n] = x[n] * h[n] = h[n] * x[n] \tag{3.16}$$

From a system point of view, this property states that a system with input x[n] and impulse response h[n] behaves exactly the same way as a system with input h[n] and impulse response x[n]. This is shown in Figure 3.14(a).

Associative Property

Convolution operator satisfies associative property which is

$$y[n] = (x[n] * h_1[n]) * h_2[n] = x[n] * (h_1[n] * h_2[n]) \tag{3.17}$$

From a system point of view this property states that if two systems with impulse response $h_1[n]$ and $h_2[n]$ are cascaded as shown in Figure 3.14(b), the equivalent system will have an impulse response equal to the convolution of $h_1[n]$ and $h_2[n]$.

$$h_{\text{eq}}[n] = h_1[n] * h_2[n] \tag{3.18}$$

Distributive Property

The distributive property states that

$$x[n] * (h_1[n] + h_2[n]) = x[n] * h_1[n] + x[n] * h_2[n] \tag{3.19}$$

This property implies that if two systems with impulse response $h_1[n]$ and $h_2[n]$ are connected in parallel as shown in Figure 3.14(c), the equivalent system will have an impulse response equal to the summation of $h_1[n]$ and $h_2[n]$.

$$h_{\text{eq}}[n] = h_1[n] + h_2[n] \tag{3.20}$$

The python code to illustrate the commutative property of convolution is given below, and the corresponding output is shown in Figure 3.15.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
```

```
def convolution_sum(x, nx, h, nh):
        kmin = np.min(nx) + np.min(nh)
        kmax = np.max(nx) + np.max(nh)
        k = np.arange(kmin, kmax + 1)
        y = np.convolve(x, h)
        return y, k

nx = np.arange(-5,6)
x = np.array([2,-3,4,-2,1,3,1,-1,4,-4,3])
h = np.array([-3,-2,-1,0,1,2,1,0,-1,-2,-3])
nh = np.arange(-3,8)
y1, n1 = convolution_sum(x, nx, h, nh)
y2, n2 = convolution_sum(h, nh, x, nx)

plt.subplot(2,1,1)
plt.stem(n1,y1)
plt.xticks(n1)
plt.xlabel('n')
plt.ylabel([np.arange(-10,10,5)])
plt.ylabel('y1[n]')
plt.grid(linestyle='--')
plt.subplot(2,1,2)
plt.stem(n2,y2)
plt.xticks(n2)
plt.xlabel('n'),
plt.ylabel('y2[n]')
plt.grid(linestyle='--')
plt.tight_layout()
```

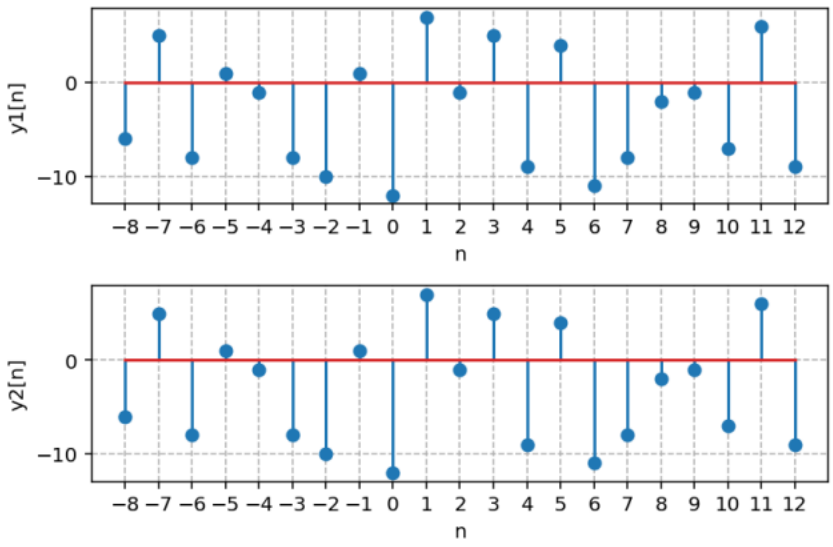


Figure 3.15: Commutative property of convolution

3.5. Correlation

A mathematical operation that closely resembles convolution is correlation. Correlation is computed to measure the degree to which two signals are similar. Correlation is used in radar, active sonar applications

and in digital communication system. Suppose that we have two real signal sequences $x[n]$ and $y[n]$ each with finite energy. The *cross-correlation* of $x[n]$ and $y[n]$ is a sequence $r_xy[l]$ which is defined as:

$$r_{xy}[l] = \sum_{n=-\infty}^{\infty} x[n]y[n-l] \tag{3.21}$$

or equivalently, as

$$r_{xy}[l] = \sum_{n=-\infty}^{\infty} x[n+l]y[n] \tag{3.22}$$

The index l is the (time) shift (or lag) parameters and the subscripts xy on the *cross-correlation* sequence $r_{xy}[l]$ indicate the sequences being correlated. It can easily be checked that

$$r_{xy}[l] = x[l] * y[-l] \tag{3.23}$$

Cross-correlation is simply the convolution of one signal with folded version of another. This makes cross-correlation a non-commutative operation. A distinct peak in the *cross-correlation* function indicates that the two signals are matched for that particular time shift. This has important applications in signal detection and system identification.

In the special case where $y[n] = x[n]$, we have the *autocorrelation* of $x[n]$ which is defined as the sequence:

$$r_{xx}[l] = \sum_{n=-\infty}^{\infty} x[n]x[n-l] \tag{3.24}$$

Autocorrelation function of a periodic signal is periodic. *Autocorrelation* can be used to detect a noisy periodic signal, estimate the impulse response of system and find the period of a periodic signal etc.

In this experiment, sine wave of $5Hz$ is generated. It is stored as the variable x. The sine wave is then corrupted by random noise, which follows normal distribution to obtain the signal y. The autocorrelation of clean sine wave is obtained as $rxx(l)$, and the cross-correlation between the clean and noisy sine wave is obtained as $rxy(l)$. The python code, which performs this task, is given below, and the corresponding output is shown in Figure 3.16.

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

t=np.linspace(0,1,100)
f=5
x=np.sin(2*np.pi*f*t)            # Generation of sine wave
w=2.5*np.random.randn(len(t))    # Generation of noisy signal
y=x+w
N=len(x)
rxx=np.correlate(x,x,mode='full')    # Perform autocorreation
rxy=np.correlate(x,y,mode='full')    # Perform cross-correlation
lag = np.arange(-N+1,N)

plt.subplot(2,2,1)
plt.plot(t,x)
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('Sine wave (x(t))')
plt.subplot(2,2,2)
plt.plot(t,y)
plt.xlabel('Time')
plt.ylabel('Amplitude')
```

```
plt.title('Noisy sine wave(y(t))')
plt.subplot(2,2,3)
plt.plot(lag,rxx)
plt.xlabel('Lag')
plt.ylabel('Autocorrelation')
plt.title('$r_{xx}(l)$')
plt.subplot(2,2,4)
plt.plot(lag,rxy)
plt.xlabel('Lag')
plt.ylabel('Cross correlation')
plt.title('$r_{xy}(l)$')
```

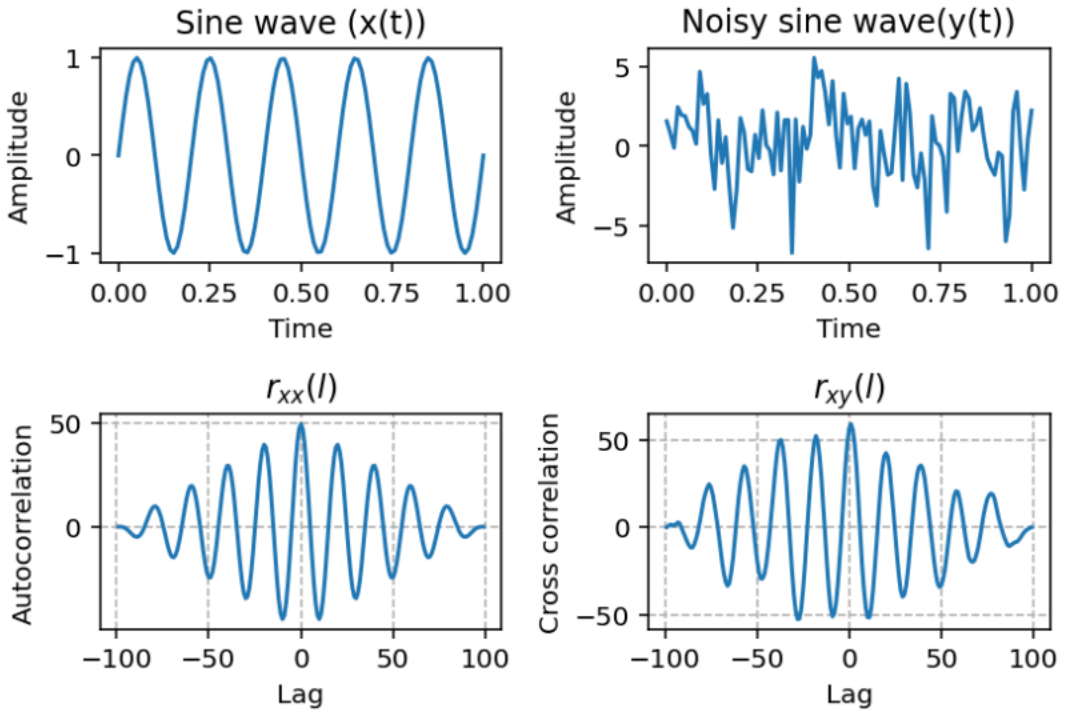


Figure 3.16: Correlation

Post-laboratory Problems

1. Using function `signal_fold()` and `signal_add()`, write a code that will decompose any real signal into its even and odd counterparts. Plot the component.
2. If $x[n] = \{1, 1, 2, 3, \underset{\uparrow}{5}, 8, 13, 21, 34, 55\}$, using the functions `signal_fold()` and `signal_shift()`, show i) $x[-n-3]$ and ii) $x[-n+4]$.
3. Are time shifting and time reversal operations commutative? Verify this by using the Problem 2.
4. Verify the *Associative Property* of Convolution using `convolution_sum` function.
5. Consider an audio signal $x[n] = \cos 0.2\pi n + 0.5 \cos 0.6\pi n$. If the signal is reflected from a barrier and reaches the listeners, they will experience the signal as: $y[n] = x[n] + 0.1x[n-20]$. Generate 200 samples of $y[n]$ and show its autocorrelation. Can you find the delay between $x[n]$ and $y[n]$?
6. Given two signals, $x[n] = \{\underset{\uparrow}{1}, 2, 1, 1\}$ and $y[n] = \{3, \underset{\uparrow}{5}, 8, 13, 21\}$. Using these two signals show that cross-correlation is non-commutative. Show the corresponding cross-correlation output figures.
7. Solve all the exercise problems of the textbook from page 115 to page 117.

Laboratory 4

Transforms and Discrete-Time Systems

4.1. Objective

This experiment is going to connect the three main views of digital signals and systems. The primary goal is to master the Z-transform and the Discrete Fourier Transform (DFT/FFT). Students will learn to calculate both the forward and inverse Z-transforms to analyze discrete systems. Students will also use the Z-transform to determine a system's stability and causality, and they will derive the impulse response. Finally, the students will use all this knowledge to find and plot the system's pole-zero map along with its complete magnitude and phase frequency response.

4.2. Discrete Fourier Transform (DFT)

Discrete Fourier transform (DFT) represents a signal in terms of sinusoids. For a discrete-time signal of length N, the basis functions are sinusoids of length N. Discrete Fourier transform is used when the signal is discrete-time and periodic only. In practice, it calculates the frequency domain representation of aperiodic signals in a given time interval, by assuming their periodic extension. The discrete Fourier transform (DFT) of the signal $x[n]$ of length N is given by

$$\text{DFT: } X[k] = \sum_{n=0}^{N-1} x[n] e^{\frac{-j2\pi kn}{N}}, \text{ k = 0, 1, 2, ...,} N-1 \tag{4.1}$$

$$\text{IDFT: } x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{\frac{j2\pi kn}{N}}, \text{ n = 0,1,2....,} N-1 \tag{4.2}$$

The formulas for DFT and IDFT given by equation (4.1) and 4.2 may be expressed as:

$$\text{DFT: } X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn}, \text{ k = 0, 1, 2, ...,} N-1 \tag{4.3}$$

$$\text{IDFT: } x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] W_N^{-kn}, \text{ n = 0,1,2....,} N-1 \tag{4.4}$$

where, $W_N = e^{-\frac{j2\pi}{N}}$, which is an Nth root of unity. Equation 4.3 can be expressed in the matrix form as:

$$\mathbf{X_N} = \mathbf{W}_N \mathbf{x}_N \tag{4.5}$$

where x_N is the column matrix containing the elements of the sequence $x[n]$ and X_N is the N-point vector containing the frequency samples. Mathematically x_N and X_N is given by:

$$x_N = \begin{bmatrix} x(0) \\ x(1) \\ . \\ . \\ . \\ x(N-1) \end{bmatrix}, X_N = \begin{bmatrix} X(0) \\ X(1) \\ . \\ . \\ . \\ X(N-1) \end{bmatrix}$$

So to match the dimension for matrix multiplication in equation (4.5), W_N must be an $N \times N$ matrix and given by:

$$\mathbf{W}_N = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & W_N & W_N^2 & \cdots & W_N^{N-1} \\ & W_N^2 & W_N^4 & \cdots & W_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \cdots : & \vdots \\ 1 & W_N^{N-1} & W_N^{2(N-1} & \cdots & W_N^{(N-1)(N-1)} \end{bmatrix} \tag{4.6}$$

The IDFT equation in equation(4.4) can also be represented by:

$$\mathbf{x}_N = \mathbf{W}_N^{-1}\mathbf{X}_N = \frac{1}{N}\mathbf{W}_N^*\mathbf{X}_N \tag{4.7}$$

where W_N^* denotes the complex conjugate of the matrix W_N.

4.2.1. Computation of DFT of a sequence using DFT matrix

This experiment deals with the computation of DFT of a input sequence using DFT matrix. Let us consider the input sequence $x[n] = \{1, 1, 1, 1\}$. Hence, the DFT matrix of order 4 is given by

$$\mathbf{W}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix} \tag{4.8}$$

The python code to obtain 4 point DFT of the sequence $x[n]$ using the above DFT matrix is given in below, and the corresponding output obtained is $\{4, 0, 0, 0\}$.

```
from scipy.linalg import dft
import numpy as np

N=4
x=np.ones(N)
W= dft(N)
X=W@x # matrix multiplication
MagX = np.abs(x)
print(MagX)
```

4.2.2. Computation of Inverse DFT through DFT matrix

Let us consider the DFT coefficients $X[k] = \{4, 0, 0, 0\}$. The python code to compute the inverse DFT of $X[k] = \{4, 0, 0, 0\}$ is given below. After executing this code, we get the result of $\{1, 1, 1, 1\}$, which is in agreement with the theoretical result.

```
from scipy.linalg import dft
import numpy as np

N = 4
X = [4,0,0,0]
W = dft(N)
x = (W@X)/N
MagX = np.abs(x)
print(MagX)
```

4.2.3. Computation of N-point DFT of a sequence and reconstruct the sequence from inverse DFT

Now compute the 5 point DFT of the sequence $x[n] = \{1, 1, 0, 0, 1\}$ and reconstruct the original sequence by inverse DFT. The python code for this task is given below. The frequency spectrum and reconstruction of the sequence are shown in Figure 4.1.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import dft

N=5
arr1=np.array([1,1,0,0,1])
arr2=np.zeros(N-len(arr1))
x=np.concatenate ((arr1, arr2), axis=0)
W= dft(N)
X=W@x # matrix multiplication
k = np.arange(N)
MagX = np.abs(X)
PhaseX = np.angle(X)

plt.subplot(2, 1, 1)
plt.stem(k, MagX)
plt.title('Magnitude Spectrum')
plt.xlabel('k')
plt.ylabel('Magnitude')
plt.grid(linestyle='--')

plt.subplot(2, 1, 2)
plt.stem(k, PhaseX)
plt.title('Phase Spectrum')
plt.xlabel('k')
plt.ylabel('Phase (degrees)')
plt.grid(linestyle='--')
plt.tight_layout()

x = (W@X)/N
x = np.real(x)
n = np.arange(N)

plt.figure()
plt.subplot(2,1,1)
plt.stem(n, x)
plt.title('Inverse DFT (Original Signal)')
plt.xlabel('n')
plt.ylabel('x[n]')
plt.grid(linestyle='--')
plt.tight_layout()
```

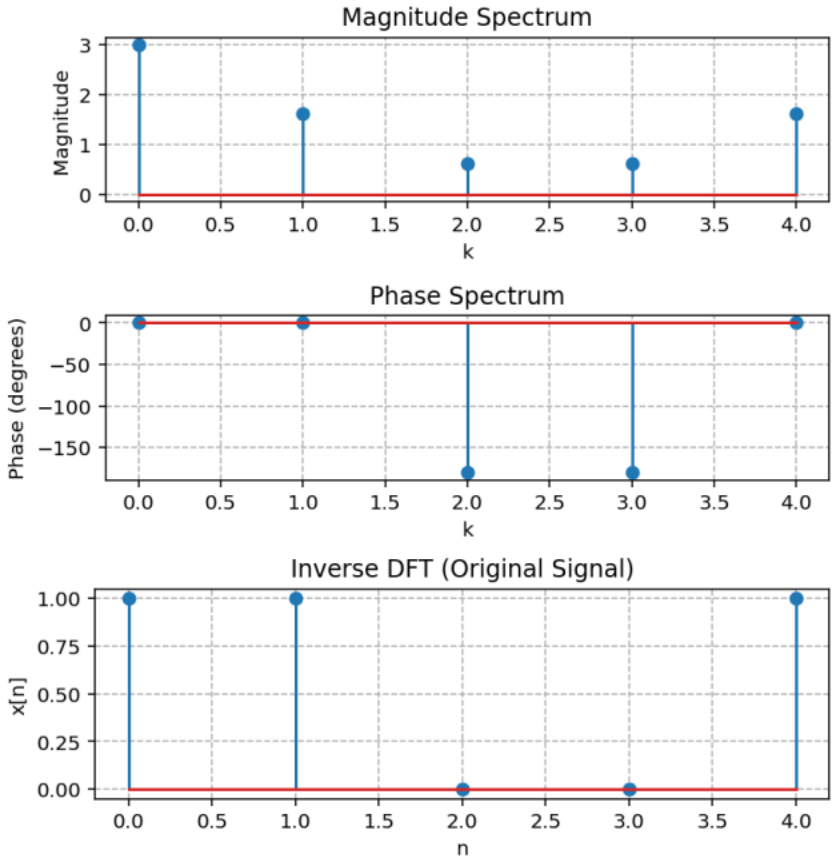


Figure 4.1: Frequency spectrum and reconstruction of a sequence $x[n] = \{1, 1, 0, 0, 1\}$ using DFT and inverse DFT respectively.

4.3. Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) and its inverse, the Inverse Fast Fourier Transform (IFFT), are powerful algorithms used to efficiently convert a signal between its time domain and frequency domain representations. The FFT rapidly computes the Discrete Fourier Transform (DFT), which breaks down a signal into its individual frequency components, revealing their respective amplitudes and phases. This is crucial for numerical computation because it allows engineers and scientists to analyze a signal's hidden properties, such as its frequency content, which might not be obvious in the time domain. The IFFT then provides a way to reconstruct the original signal perfectly from these frequency components. Because of their computational efficiency, the FFT and IFFT are essential for a wide range of applications, including audio processing, image analysis, and telecommunications.

The FFT in Python is primarily implemented using the `numpy.fft` module or the `scipy.fft` module. Both modules provide functions to compute the Discrete Fourier Transform (DFT) efficiently, leveraging the FFT algorithm. in this lab manual we will use the `numpy.fft` module from this point onward.

4.3.1. Computation of N-point FFT of a sequence and reconstruct the sequence from inverse FFT

Now compute the 32 point DFT of the sequence $x[n] = \{1, 1, 1, 1, 1\}$ and reconstruct the original sequence by `fft()` and `ifft()` respectively. The python code for this task is given below. The frequency spectrum and reconstruction of the sequence are shown in Figure 4.2.

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

N = 32
```

```
arr1 = np.array([1, 1, 1, 1, 1])
arr2 = np.zeros(N - len(arr1))
x = np.concatenate((arr1, arr2), axis=0)

X = np.fft.fft(x, N)
k = np.arange(N)
MagX = np.abs(X)
PhaseX = np.angle(X)

plt.subplot(2, 1, 1)
plt.stem(k, MagX)
plt.title('Magnitude Spectrum')
plt.xlabel('k')
plt.ylabel('Magnitude')
plt.grid(linestyle='--')

plt.subplot(2, 1, 2)
plt.stem(k, PhaseX)
plt.title('Phase Spectrum')
plt.xlabel('k')
plt.ylabel('Phase (degrees)')
plt.ylabel('Amplitude')
plt.grid(linestyle='--')

plt.tight_layout()
plt.show()

x_reconstructed = np.fft.ifft(X, N)
x_reconstructed = np.real(x_reconstructed)  # Take only the real part
n = np.arange(N) # Create the time axis for plotting

plt.figure()
plt.subplot(2, 1, 1)
plt.stem(n, x_reconstructed)
plt.title('Inverse DFT (Original Signal)')
plt.xlabel('n')
plt.ylabel('x[n]')
plt.grid(linestyle='--')
plt.tight_layout()
plt.show()
```

4.3.2. Computation of the frequency spectrum of a *sinc* function

Find out the frequency spectrum of a $sinc$ function. Remember that a sinc function is defined by the following equation:

$$\text{sinc}\,(t) = \begin{cases} \frac{\sin \pi t}{\pi t} & t \neq 0 \\ 1 & t = 0 \end{cases} \tag{4.9}$$

In this example, consider $x(t) = \text{sinc}(100t)$. First, sample the signal with sampling period 0.8333 ms and then computes its spectrum using FFT. Following is code for this purpose. And the result is shown in the following Figure 4.3.

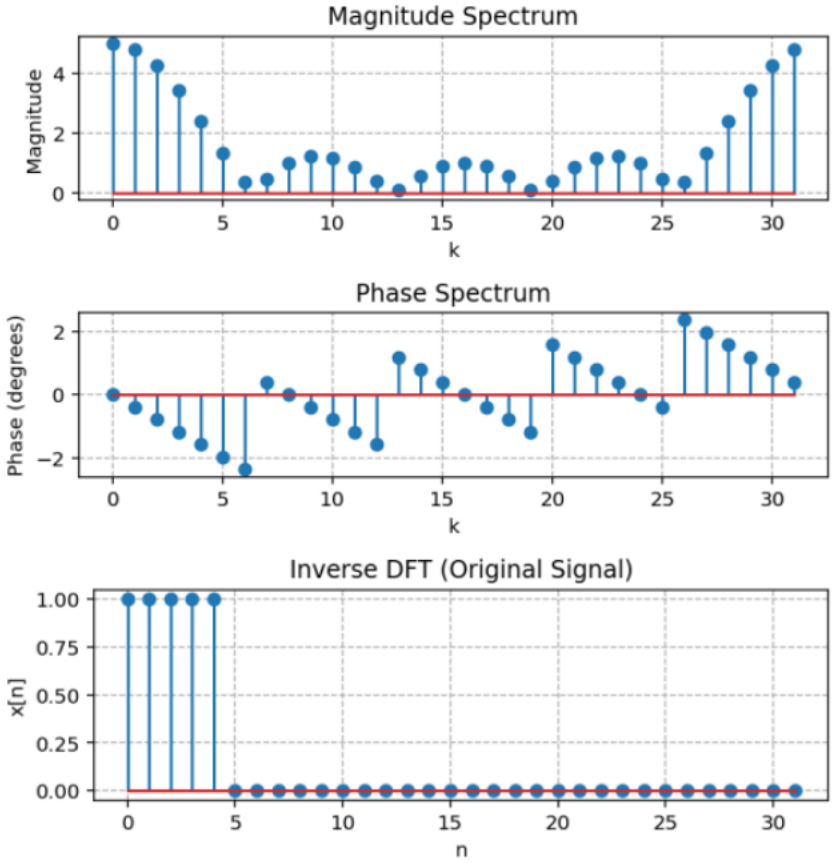


Figure 4.2: Frequency spectrum and reconstruction of a sequence $x[n] = \{1, 1, 1, 1, 1\}$ using FFT and IFFT algorithm respectively.

```python
import numpy as np
import matplotlib.pyplot as plt

t0 = 0.2  # duration of sinc pulse in seconds
ts = 8.3333e-4  # sampling period in seconds
fs = 1 / ts  # sampling frequency in Hz

t = np.arange(-t0 / 2, t0 / 2 + ts, ts)
x = np.sinc(100 * t)
# The argument 100*t is scaled correctly for the desired pulse width.
N = 256  # Number of points for the DFT
X = np.fft.fft(x, N)
freq = np.fft.fftfreq(N, d=ts)
# The fftfreq function is the most robust way to create the frequency axis.
# It handles the zero-centered and positive-only ranges correctly.
plt.figure(figsize=(10, 8))

plt.subplot(3, 1, 1)
plt.plot(t, x, 'b-', linewidth=2)
plt.title('Sinc Pulse')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.grid(True)

```

```
plt.subplot(3, 1, 2)
plt.stem(np.fft.fftshift(freq), np.abs(np.fft.fftshift(X)) / N)
# np.fft.fftshift() centers the zero frequency
# The magnitude is divided by N to scale it correctly
plt.title('Magnitude Spectrum')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.grid(True)

plt.subplot(3, 1, 3)
plt.stem(np.fft.fftshift(freq), np.angle(np.fft.fftshift(X)) * 180 / np.pi)
# np.angle() returns the phase in radians, so we convert it to degrees
plt.title('Phase Spectrum')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Phase (Degrees)')
plt.grid(True)
plt.tight_layout()
```

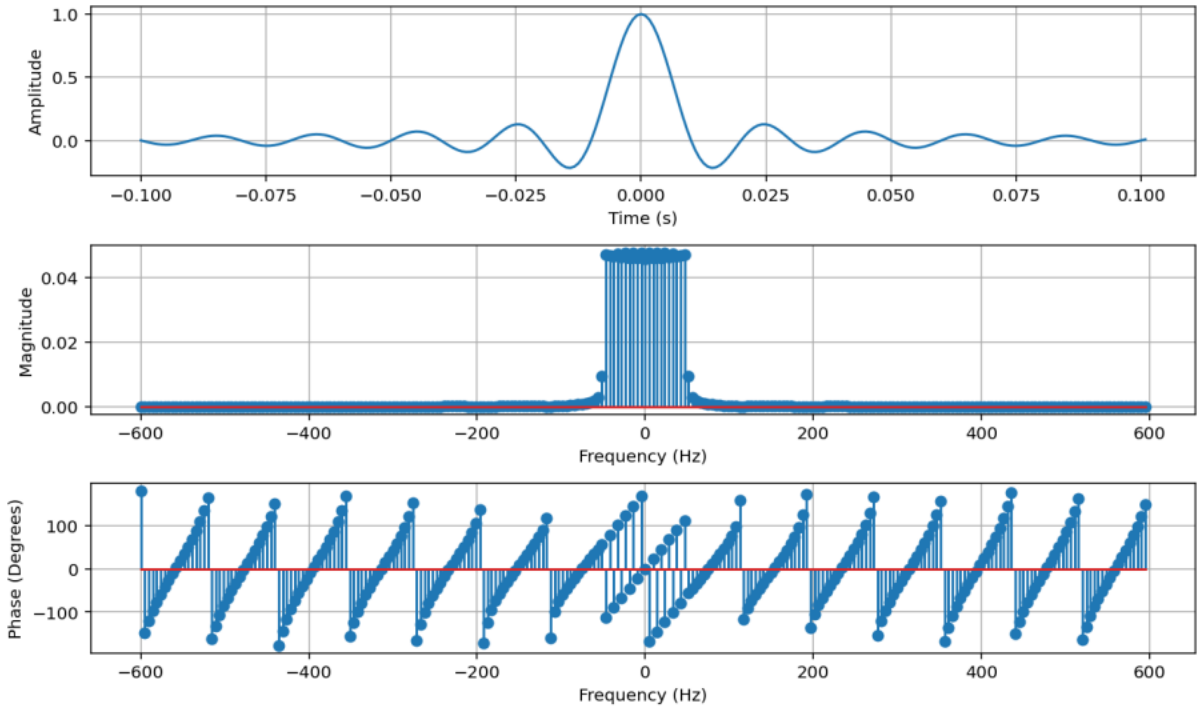


Figure 4.3: Frequency spectrum of a `sinc` pulse.

4.3.3. Convolution using DFT

In continuous time signal, multiplication of two signals in frequency domain implies convolution in time domain. However, in case of DFT, multiplication of two signals in frequency domain corresponds to *circular convolution* in time domain. Circular convolution of two DT signals or sequences $x_1[n]$ and $x_2[n]$ is defined by the following formula:

$$y[m] = \sum_{n=0}^{N-1} x_1[n] x_2([m-n])_N, \text{ m = 0, 1, ..., N-1} \tag{4.10}$$

The basic difference between linear and circular convolution is that, in circular convolution, the folding and shifting (rotating) operations are performed in a circular fashion by computing the index of one of the

sequences modulo N. In linear convolution, there is no modulo N operation. This will be discussed broadly in theory. However, linear convolution can also be computed by multiplying two signals in frequency domain. The steps for computing linear convolution, $y[n] = x[n] * h[n]$ using DFT is given below. We'll only mention the procedure. We won't go into the detailed theory.

1. Let $N = length(x) + length(h) - 1$. Pad each of the original sequences, $x[n]$ and $h[n]$ with zeros until they are both length N.
2. Compute N-point FFT of each padded sequences, $X[k]$ and $H[k]$.
3. Compute Inverse FFT of the product $Y[k] = X[k]H[k]$. This will be equivalent to $y[n] = x[n] * h[n]$.

To compute convolution sum of two sequences in frequency domain, consider two signals, $x[n] = \{\underset{\uparrow}{2}, 1, 2, 1\}$ and $h[n] = \{\underset{\uparrow}{1}, 2, 3, 4\}$. The code is given below.

```python
import numpy as np

x = np.array([2, 1, 2, 1])  # signal x
h = np.array([1, 2, 3, 4, 3])  # signal h
conv_len = len(x) + len(h) - 1
# The length of the linear convolution of two signals with lengths M and N
# is M + N - 1. We must pad the signals to this length for the
x_padded = np.pad(x, (0, conv_len - len(x)))
h_padded = np.pad(h, (0, conv_len - len(h)))
# Pad both signals with zeros so they have a length equal to conv_len.
# This makes the circular convolution result identical to
# the linear convolution.
X = np.fft.fft(x_padded, conv_len)  # Perform DFT
H = np.fft.fft(h_padded, conv_len)  # Perform DFT
Y = X * H
y = np.real(np.fft.ifft(Y, conv_len))
# Perform IDFT to get the result in the Time Domain
print("Signal x:", x)
print("Signal h:", h)
print("Padded signal x:", x_padded)
print("Padded signal h:", h_padded)
print("Length of convolution:", conv_len)
print("Result of linear convolution (y):", y)

y_verified = np.convolve(x, h)
print("Result verified with np.convolve:", y_verified)
# The result can also be verified using NumPy's built-in convolution function
# This is for verification purposes and not part of the DFT method.
```

The result for convolution is given by

```
Signal x: [2 1 2 1]
Signal h: [1 2 3 4 3]
Padded signal x: [2 1 2 1 0 0 0 0]
Padded signal h: [1 2 3 4 3 0 0 0]
Length of convolution: 8
Result of linear convolution (y): [ 2.  5. 10. 16. 18. 14. 10.  3.]
Result verified with np.convolve: [ 2  5 10 16 18 14 10  3]
```

You can verify the result using `convolution_sum` from laboratory 3. In a similar way, correlation can also be calculated using `fft`.

4.3.4. Observe modulation and demodulation of a message signal in frequency domain using FFT

Let $x[n]$ be a discrete signal with with DTFT $X(\Omega)$. If we modulate $x[n]$ by a cosine carrier signal the output signal will simply be the multiplication of $x[n]$ and $\cos(\omega_0 n)$. In frequency domain, the result will be as follows:

$$x(n)\cos\omega_o n \overset{\text{DTFT}}{\longleftrightarrow} \frac{1}{2}\left[X(\omega+\omega_o)+X(\omega-\omega_o)\right] \tag{4.11}$$

So, the original spectrum of x[n] will shift at ω_0 and $-\omega_0$. In our example, we considered a sinc function as a message signal.

$$m(t) = \begin{cases} \sin c(100t), |t| \leq t_o \\ 0, \quad \text{otherwise} \end{cases} \tag{4.12}$$

where, $t_o = 0.1s$. Let, carrier $c(t) = cos(2\pi f_c t)$ where $f_c = 250Hz$. So after modulation, the original signal's spectrum will shift to $250Hz$ and $-250Hz$. Code for modulation is given below. The resulting spectrum is shown in Figure 4.4

```python
import numpy as np
import matplotlib.pyplot as plt
### --- 1. Signal and System Parameters ---
to = 0.2  # duration of the sinc pulse
ts = 0.001  # sampling period
fc = 250  # carrier frequency in Hz
fs = 1 / ts  # sampling frequency in Hz
t = np.arange(-to / 2, to / 2, ts)  # Time axis for the signals
### --- 2. Signal Generation and Modulation ---
m = np.sinc(100 * t)    # The message signal
c = np.cos(2 * np.pi * fc * t)  # The carrier signal,
u = m * c   # DSB-SC modulation: a simple element-wise multiplication
### --- 3. Spectral Analysis using FFT ---
N = 1024  # FFT bin size
freq = np.fft.fftfreq(N, d=1 / fs)
# Generate the correct frequency axis for plotting
M_fft = np.fft.fft(m, N) / fs
# Scale the N-point DFTs by the sampling frequency
U_fft = np.fft.fft(u, N) / fs
### --- 4. Plotting: Message and Modulated Spectra ---
# Plot the spectrum of the message signal
plt.subplot(2, 1, 1)
plt.plot(np.fft.fftshift(freq), np.abs(np.fft.fftshift(M_fft)), linewidth=2)
plt.title('Spectrum of the Message Signal')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.grid(True)
# Plot the spectrum of the modulated signal
plt.subplot(2, 1, 2)
plt.plot(np.fft.fftshift(freq), np.abs(np.fft.fftshift(U_fft)), linewidth=2)
plt.title('Spectrum of the Modulated Signal')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.grid(True)
```

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

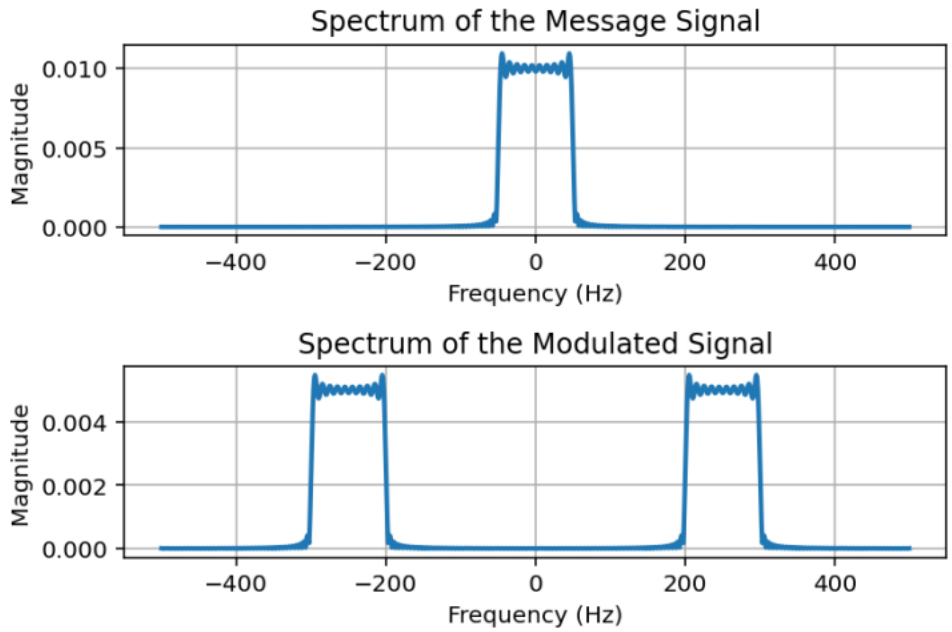


Figure 4.4: After modulation, spectrum of the original signal is shifted to the carrier frequency and magnitude is halved

As a result, the spectrum has three components. The lower frequency component corresponds to the original signal. This component is retrieved by using a low pass filter. To adjust the magnitude of the message signal, a low pass filter with gain 2 is used. The code is given below. The corresponding frequency spectra are shown in Figure 4.5

```
# --- continue the code after 'Modulation' part
# --- 5. Demodulation and Filtering ---
y = u * c   # Demodulation is multiplication by the carrier again
Y_fft = np.fft.fft(y, N) / fs   # N-point DFT of the demodulated signal
# Create a simple low-pass filter in the frequency domain
fcut = 200  # Sets the cutoff frequency for the low-pass filter to 200 Hz.
ncut = int(np.floor(fcut / (fs / N)))
# Calculates the number of frequency bins corresponding to the fcut
H_filter = np.zeros(N, dtype=np.float64)
# Creates an array of zeros to serve as the frequency-domain filter.
H_filter[0:ncut] = 2
# Sets the gain for the positive frequencies up to the cutoff to 2.
H_filter[N - ncut:N] = 2
# Sets the gain for the corresponding negative frequencies to 2,
# which is required for a symmetric filter response.
U_filtered = Y_fft * H_filter
# Apply the filter by multiplying in the frequency domain
# --- 6. Plotting: Demodulated and Filtered Spectra ---
# Plot the spectrum of the demodulated signal
plt.subplot(2, 1, 1)
plt.plot(np.fft.fftshift(freq), np.abs(np.fft.fftshift(Y_fft)))
plt.title('Spectrum of Demodulated Signal')
```

```
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.grid(True)
# Plot the spectrum of the filtered signal
plt.subplot(2, 1, 2)
plt.plot(np.fft.fftshift(freq), np.abs(np.fft.fftshift(U_filtered)))
plt.title('Spectrum of Filtered Demodulated Signal')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.grid(True)
plt.tight_layout()
plt.show()
# --- 7. Plotting: Reconstructed Time-Domain Signal ---
# Use np.real() to remove any imaginary components due to rounding errors
u_reconstructed = np.real(np.fft.ifft(U_filtered) * fs)
# Plot the original message signal
plt.subplot(2, 1, 1)
plt.plot(t, m, linewidth=2)
plt.title('Original Message Signal')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.grid(True)
# Plot the reconstructed signal.
plt.subplot(2, 1, 2)
plt.plot(t, u_reconstructed[:len(t)])
# Slice to match the original signal's length.
plt.title('Reconstructed Signal')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.grid(True)
plt.tight_layout()
plt.show()
```

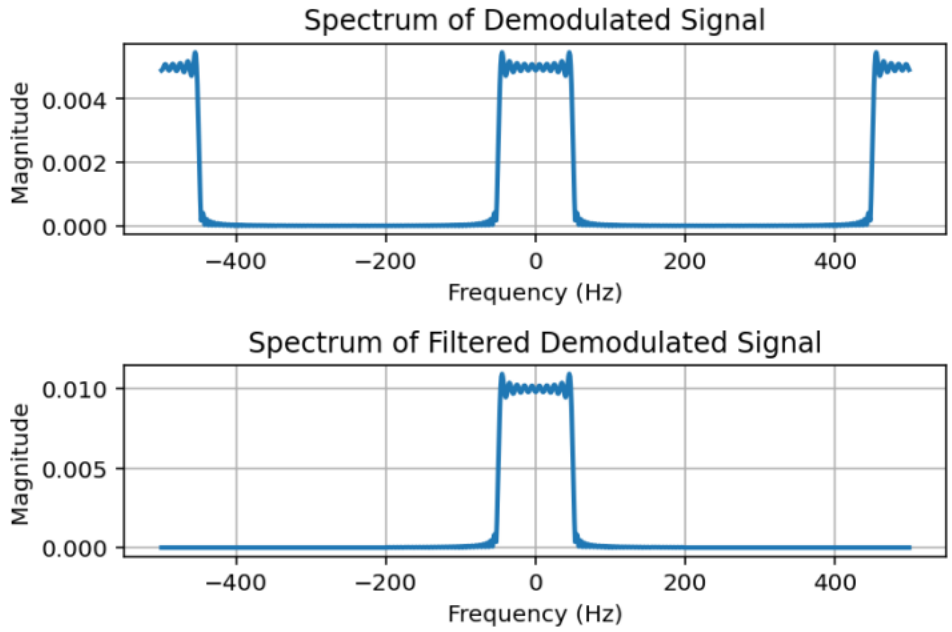


Figure 4.5: Sepctra for demodulation and filtering

The similarities between the original signal and filtered signal in time domain is illustrated in Figure 4.6.

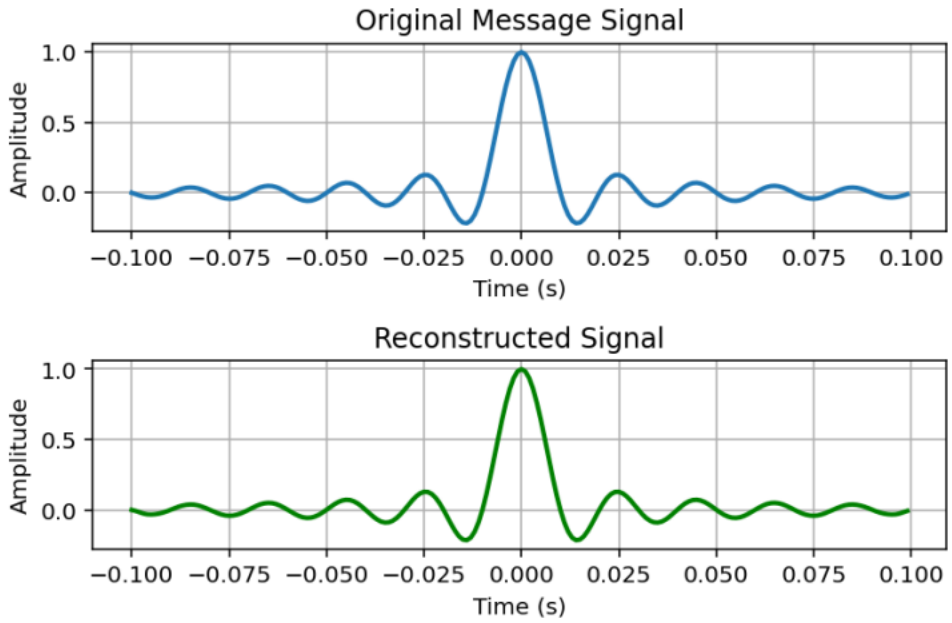


Figure 4.6: Original signal and Reconstructed signal

4.4. Z-Transformation

Z transform is an important tool in analyzing discrete-time (DT) signals and linear time-invariant (LTI) Discrete Time systems. It plays the same role as Laplace transform plays in analyzing continuous time signals and systems. The DFT analyzes the frequency content of finite-length signals at specific frequencies, while the Z-transform is a broader tool for analyzing digital systems and sequences in the complex frequency (z) domain. The DFT can be viewed as a specific case of the Z-transform, evaluated on the *unit circle*. The Z-transform is more generalized because it allows for complex frequency analysis, which helps in analyzing signals that may not be stable or finite, unlike the DFT which is restricted to specific conditions.

The Z-transform of a discrete-time signal $x[n]$ is defined

$$X(z) = \sum_{n=-\infty}^{\infty} x[n]z^{-n} \tag{4.13}$$

where z is a complex variable. Since Z-transform is an infinite power series, it exists only for those values of z for which the series given in 4.13 converges. The *Region of Convergence (ROC)* of $X(z)$ is the set of all values of z for which $X(z)$ attains a finite value. Therefore, while mentioning a Z-transform, its ROC should also be indicated.

Consider a finite-duration signal x[n] = {2,$\underset{\uparrow}{1}$,2,1,3}. Its z-transformed signal,

$$\begin{aligned} X(z) &= \sum_{n=-\infty}^{\infty} x[n]z^{-n} = \sum_{n=-1}^{3} x[n]z^{-n} \\ &= 2z^1 + 1 + 2z^{-1} + z^{-2} + 3z^{-3} \end{aligned} \tag{4.14}$$

ROC of $X(z)$ is entire z-plane except $z = 0$ and $z = \infty$.

4.4.1. Z-Transform of Unit Sample and Unit Step Sequences using `lcapy` package

`lcapy` is a Python package for linear circuit analysis. This pcakage can be used to compute the Z-transform of unit sample and unit step sequences. The python code is shown below.

```
from lcapy import n,delta,us

x =delta(n)
Xz=x.ZT()
print(Xz)
x1 = us(n)
Yz=x1.ZT()
print(Yz)
```

Upon executing the above code, the result obtained in the IPython Console is 1 and $1/(1-1/z)$, which is in agreement with the theoretical result. The python command $delta$ defines the unit impulse sequence, us gives the unit step sequence and ZT obtains the Z-transform. The result is shown below.

```
1
1/(1 - 1/z)
```

4.4.2. Z-Transform of $x[n] = e^{jn}u[n]$, and $x[n] = cos(n)$, and $x[n] = \frac{1}{2}^n u[n]$

The python code to obtain the required Z-transform is given below.

```
from lcapy import n, exp, cos, us

x1 = exp(1j*n)
x2 = cos(n)
x3 = (1/2)**n*us(n)
X1 = x1.ZT()
print(X1)
X2=x2.ZT()
print(X2)
```

```
X3=x3.ZT()
print(X3)
```

Upon executing the above code, the following result is obtained in the IPython Console.

```
z/(z - exp(j))
z*(z - cos(1))/(z**2 - 2*z*cos(1) + 1)
2*z/(2*z - 1)
```

4.4.3. Inverse Z-Transform of $X(z) = z^{-1}$, and $X(z) = \frac{1}{1-z^{-1}}$ using `lcapy` package

The python code computes the inverse Z-transform of $X(z) = z^{-1}$, and $X(z) = \frac{1}{1-z^{-1}}$ is given below.

```
from lcapy import z

X=z**(-1)
x=X.IZT()
print(x)
Y=1/(1-z**(-1))
y=Y.IZT()
print(y)
```

Upon executing the above code, the following result is obtained in the IPython Console.

```
UnitImpulse(n - 1)
Piecewise((1, n >= 0))
```

4.4.4. Inverse Z-Transform using partial fraction expansion

Consider a rational Z-transform X(z) given by:

$$X(z) = \frac{1 + 2z^{-1} - z^{-2}}{1 - z^{-1} + 0.3561z^{-2}} \tag{4.15}$$

Determine its partial fraction expansion. For this type of $X(z)$, first, provide the numerator and denominator polynomials in MATLAB. In this example, numerator polynomial is $num = [1, 2, 1]$ and denominator polynomial is $den = [1, -1, 0.3561]$. And, then use the `scipy.signal.residuez` function to compute the partial fraction expansion. It returns the residues (r), poles (p), and direct terms (k). After that, to reconstruct the original polynomials from the results, use the inverse function is `scipy.signal.invresz`.

```
import numpy as np
from scipy import signal

num = [1, 2, -1]  # Numerator coefficients
den = [1, -1, 0.3561]  # Denominator coefficients
r, p, k = signal.residuez(num, den)
print("--- Partial Fraction Expansion Results ---")
print("Residues (r):")
print(r)
```

```
print("\nPoles (p):")
print(p)
print("\nDirect terms (k):")
print(k)
print("-" * 40)

num_rc, den_rc = signal.invresz(r, p, k)
print("\nReconstructed Numerator:")
print(num_rc)
print("\nReconstructed Denominator:")
print(den_rc)

```

Upon executing the above code, the following result is obtained in the IPython Console.

```
--- Partial Fraction Expansion Results ---
Residues (r):
[1.90409997-1.68222178j 1.90409997+1.68222178j]

Poles (p):
[0.5+0.32572995j 0.5-0.32572995j]

Direct terms (k):
[-2.80819994]
----------------------------------------

Reconstructed Numerator:
[ 1.+0.j  2.+0.j -1.+0.j]

Reconstructed Denominator:
[ 1.    +0.j -1.    +0.j  0.3561+0.j]
```

$X(z)$ can be written as following by using partial fraction expansion.

$$X(z) = -2.8082 + \frac{1.9041 - 1.6822i}{1 - (0.5 + 0.3257i)z^{-1}} + \frac{1.9041 + 1.6822i}{1 - (0.5 - 0.3257i)z^{-1}} \tag{4.16}$$

4.5. Discrete-Time System

Discrete-time (DT) system accepts a discrete-time signal as input and generates a discrete-time signal as the output. The input to the discrete-time system is termed as *excitation* and the output of the system is termed as *response*. The block diagram of DT system is shown in Figure 4.7.

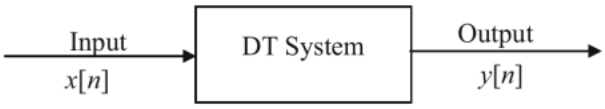


Figure 4.7: Block diagram of a discrete-time system

4.5.1. Difference Equation Representation of Discrete-Time Linear Time-Invariant System

The relationship between the input and output of a discrete-time linear time-invariant (LTI) system is expressed in terms of linear constant coefficient difference equation (LCCDE) as

$$y[n] = \sum_{k=0}^{N-1} a_k y[n-k] + \sum_{k=0}^{M-1} b_k x[n-k] \tag{4.17}$$

where a_k and b_k are the output and input coefficients respectively. The above equation represents the fact that weighted sum of input is equal to the weighted sum of output. Equation 4.17 can be expanded as

$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots + b_M x[n-M] - \{a_1 y[n-1] + \cdots + a_N y[n-N]\} \tag{4.18}$$

Let the relationship between input and output of a discrete-time system is given by

$$y[n] = \frac{1}{2} y[n-1] + x[n] \tag{4.19}$$

The python code, which obtains the output of the system, is given below. The function `signal.lfilter` is used in this code to simulate a LTI digital filter defined by a difference equation. It applies a digital filter to an input signal. It solves the discrete-time difference equation that defines the filter, relating the output signal $y[n]$ to the input signal $x[n]$. The corresponding system output is shown in Figure 4.8.

```
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
n=np.arange(0,4)
x=np.ones(len(n)) #Input
coeff_x=[1]
coeff_y=[1,-1/2]
y=signal.lfilter(coeff_x,coeff_y,x)

plt.stem(y)
plt.xlabel('n')
plt.ylabel('y[n]')
plt.tight_layout()
```

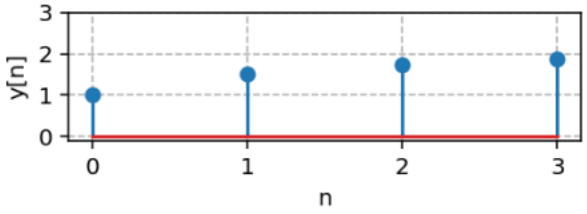


Figure 4.8: Output of the system having difference equation of $y[n] = \frac{1}{2} y[n-1] + x[n]$

4.5.2. Impulse Response of Discrete-Time System

Impulse response is the reaction of the discrete-time system to unit sample input signal, whereas step response is the reaction of the system to unit step input signal. The discrete-time system is completely characterised by its impulse response. The meaning is, if one knows the impulse response of the system, it is possible to infer whether the system is causal and stable from it.

If $x[n]$ represents the input signal, $h[n]$ represents the impulse response and $y[n]$ represents the output of the system. If the system is a LTI system, then the output of the system is expressed as

$$y[n] = x[n] * h[n] \tag{4.20}$$

In the above expression, $*$ indicates the convolution operation. If the input to the system is unit sample signal, then the output of the system is given by

$$y[n] = \delta[n] * h[n] \tag{4.21}$$

Convolution of any signal with unit sample signal results in the same signal; hence, the above equation can be expressed as

$$y[n] = h[n] \tag{4.22}$$

Thus, the impulse response of the system is the reaction of an LTI system to unit sample input signal.

An LTI discrete-time system is defined by the difference equation $y[n] = x[n] + y[n-1]$. Plot the impulse response of the system. W already know, impulse response is the response of the system to unit sample input signal. The python code, which computes the impulse response of the system is shown below, and the corresponding output is shown in Figure 4.9.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

def unit_impulse(n0, n1, n2):
        n = np.arange(n1, n2 + 1, 1)
        x = (n == n0)
        return x, n
n0 = 0
n1 = 0
n2 = 50
x, n = unit_impulse(n0, n1, n2) # Generation of unit sample signal
x = x.astype(float) # Convert from boolean to floating-point number
coeff_x=[1]     # Define the system
coeff_y=[1,-1]
h=signal.lfilter(coeff_x,coeff_y,x)    # Obtaining the impulse response
plt.stem(h)
plt.xlabel('n')
plt.ylabel('h[n]')
plt.title('Impulse response (h[n])')
```

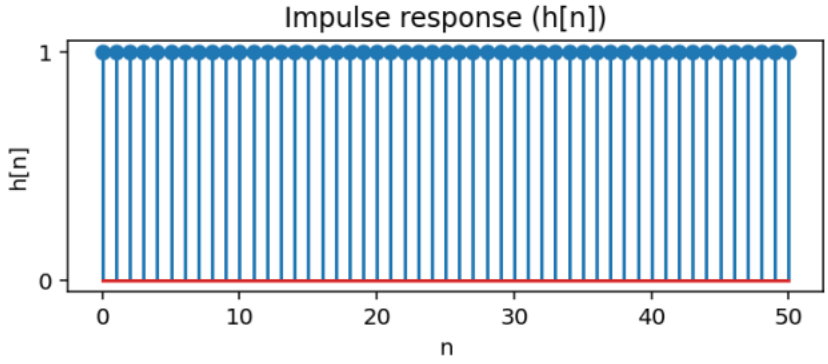


Figure 4.9: Impulse response of the discrete-time system defined as $y[n] = x[n] + y[n-1]$

4.5.3. Pole-Zero Plot of Discrete-Time System

The pole-zero plot of a discrete-time system is plotted in the Z-plane. The position on the complex plane is represented by $re^{j\omega}$. consider a rational $X(z)$ in the following form

$$X(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots\cdots + b_M z^{-M}}{a_0 + a_1 z^{-1} + a_2 z^{-2} + \cdots\cdots + a_N z^{-N}} \tag{4.23}$$

Here, $a_N \neq 0$. Zeros of a Z-transform $X(z)$ are the values of z for which $X(z) = 0$ and poles are the values of z for which $X(z) = \infty$. In other words, the roots of numerator in Equaition (4.23) are called zeros and the roots of denominator in Equation (4.23) are called poles of $X(z)$. Since the numerator and denominator in Equation (4.23) are polynomials, they can be expressed in the following factored form

$$X(z) = \frac{B(z)}{A(z)} = \frac{b_0}{a_0} z^{-M+N} \frac{(z - z_1)(z - z_2)\cdots(z - z_M)}{(z - p_1)(z - p_2)\cdots(z - p_N)} \tag{4.24}$$

Thus $X(z)$ has M finite zeros at $z = z_1, z_2, \cdots z_M$ (roots of the numerator polynomial), N finite poles at $z = p_1, p_2 \cdots p_N$ (the roots of the denominator polynomial) and $|N - M|$ zeros (if $N > M$) or poles (if $N < M$) at $z = 0$. For example consider a Z-transform $X(z) = \dfrac{z+1}{(z - \frac{1}{2})(z + \frac{3}{5})}$. The numerator has a root at $z = -1$ and two poles at $z = \dfrac{1}{2}$ and $z = -\dfrac{3}{5}$. Thus the zero of $X(z)$ is 1 and poles are $\{\dfrac{1}{2}, -\dfrac{3}{5}\}$.

X(z) can be graphically represented by a *pole-zero plot* in the complex plane, which shows the location of poles by crosses ($\times$) and the location of zeros by circles ($\circ$). The multiplicity of multiple-order poles or zeros is indicated by a number close to the corresponding cross or circle. By definition, the ROC of a Z-transform should not contain any poles.

Pole Zero Location and System's Behavior

If $x[n]$ is the input sequence to an LTI system with impulse response $h[n]$, the output sequence $y[n]$ is given by the following equation for convolution sum.

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k] \tag{4.25}$$

Using the convolution property (*the convolution of $x[n]$ and $h[n]$ in time domain corresponds to multiplication in z-domain)*, we can write equation (6.11) in the z-domain as:

$$Y(z) = X(z)H(z) \tag{4.26}$$

where $Y(z)$, $X(z)$ and $H(z)$ are the Z-transform of $y[n]$, $x[n]$ and $h[n]$ respectively. The transfer function of the discrete-time system is given by

$$H(z) = \frac{Y(z)}{X(z)} \tag{4.27}$$

We are going to look at how a system's pole location determines its behavior over time. For signals that start at $n = 0$ (causal signals), the unit circle ($\mid z \mid = 1$) acts as the essential reference line. This circle is important because it is where the Z-transform equals the DFT. The time behavior of the signal is based on the pole's location relative to the unit circle (see Figure 4.10):

— Inside the Unit Circle ($\mid z \mid < 1$): The signal decays over time (stable).
— On the Unit Circle ($\mid z \mid = 1$): The signal is constant (marginally stable).
— Outside the Unit Circle ($\mid z \mid > 1$): The signal grows over time (unstable).

A pair of complex conjugate poles results in an exponentially weighted sinusoidal signal. The amplitude of the signal is growing if $r > 1$, constant if $r = 1$ (sinusoidal signals) and decaying if $r < 1$ (see Figure 4.11)

The Region of Convergence (ROC) of the system function $H(z)$ dictates the system's properties. An LTI system is causal if its ROC is the exterior of a circle ($r < \infty$), and it is BIBO stable if the ROC includes the unit circle. Combining these, a causal LTI system is stable if and only if all the poles of $H(z)$ lie strictly inside the unit circle. The ROC can be visually identified directly from the pole-zero plot.

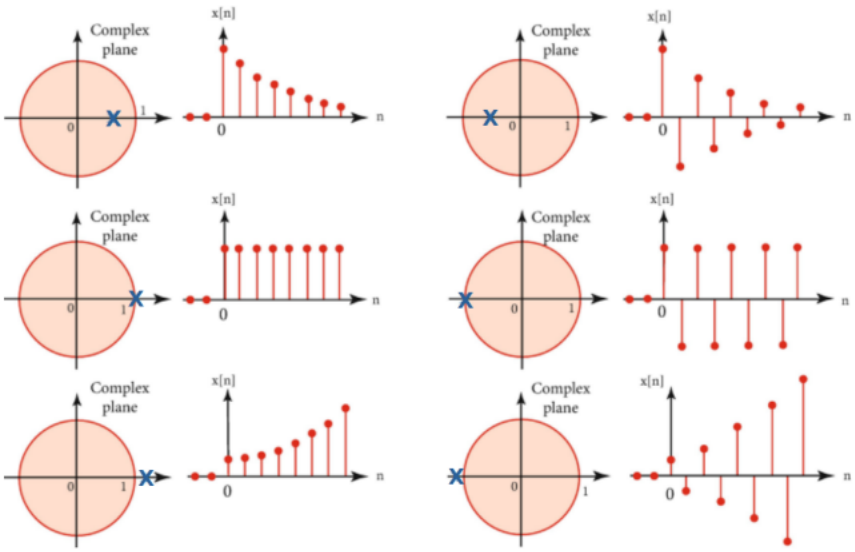


Figure 4.10: Time-domain behavior of a single real pole causal signal as a function of the pole location with respect to the unit circle.

Pole Zero Location and System's Behavior

Consider a discrete-time system with the following transfer function

$$H(z) = \frac{1 + 2z^{-1} - z^{-2}}{1 - z^{-1} + 0.3561z^{-2}} \tag{4.28}$$

The built-in function `pzmap` in `control` library is utilized to plot the Z-plane of the above system. The python code does this task is shown below. The pole-zero plot is shown in Figure 4.12.

```
import control as ct

num = [1, 2, -1]
den = [1, -1, 0.3561]
sys = ct.tf(num, den, dt=True)
ct.pzmap(sys, title=False)
```

In Figure 4.12, the symbol ∘ represents a zero and the symbol × represents a pole. The plot includes the unit circle which is the dotted line. This is the set of points z to which the Z-transform equals the DFT.

4.5.4. Frequency Response of a Discrete-Time System

Frequency reponse of a system is equivalent to evaluating the transfer function $H(z)$ of that system on the unit circle, i.e. $z = e^{j\omega}$.

$$H\left(e^{j\omega}\right) = H(z)|_{z=e^{j\omega}} \tag{4.29}$$

Using this methodology, it is possible to sketch the frequency response of the system that is related to its poles and zeros. The transfer function of a discrete-time system can be considered in the following form:

$$H(z) = \frac{b_0 + b_1 z^{-1} + \cdots\cdots + b_n z^{-n}}{a_0 + a_1 z^{-1} + \cdots\cdots + a_m z^{-m}} = \frac{B(z)}{A(z)} \tag{4.30}$$

To estimate the frequency response of the above system, `freqz` from the `scipy.signal` package is necessary. `freqz` function uses an FFT-based approach to compute the frequency response. The function has a variety of formats. A useful format is `w, h = signal.freqz(num, den, worN=N, fs=Fs)` where the variables num and den are the vectors of the numerator and denominator polynomials. In case of transfer function, vector num is related to the input of a system and vector den is related to the output

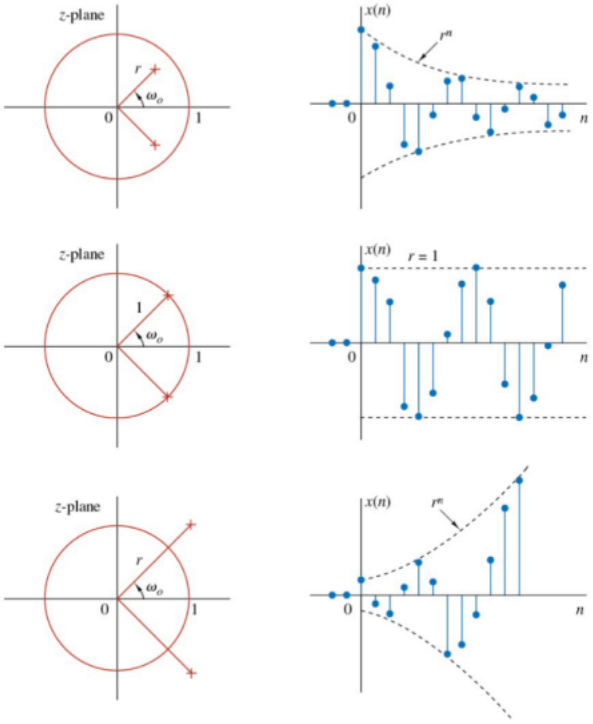


Figure 4.11: A pair of complex conjugate poles corresponding to causal signals with oscillatory behavior.

of a system. These coefficients control the response of a system. fs is the sampling frequency and $worN$ is the number of frequency between 0 and $\frac{Fs}{2}$. This function returns w and h, where, h is the the frequency response, and w is the frequencies at which h was computed, in the same units as fs.

Consider the transfer function of the following discrete-time system.

$$H(z) = \frac{1 - 1.6180z^{-1} + z^{-2}}{1 - 1.5371z^{-1} + 0.9025z^{-2}} \tag{4.31}$$

Following code is used to compute the frequency response of the above system. The output is shown in Figure 4.13.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

num = [1, -1.6180, 1]
den = [1, -1.5371, 0.9025]
Fs = 512
N = 256
w, h = signal.freqz(num, den, worN=N, fs=Fs)

plt.subplot(2, 1, 1)
magnitude_db = 20 * np.log10(np.abs(h))
plt.plot(w, magnitude_db)
plt.title('Magnitude Response')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude (dB)')
plt.grid(True, linestyle='--')

plt.subplot(2, 1, 2)
```

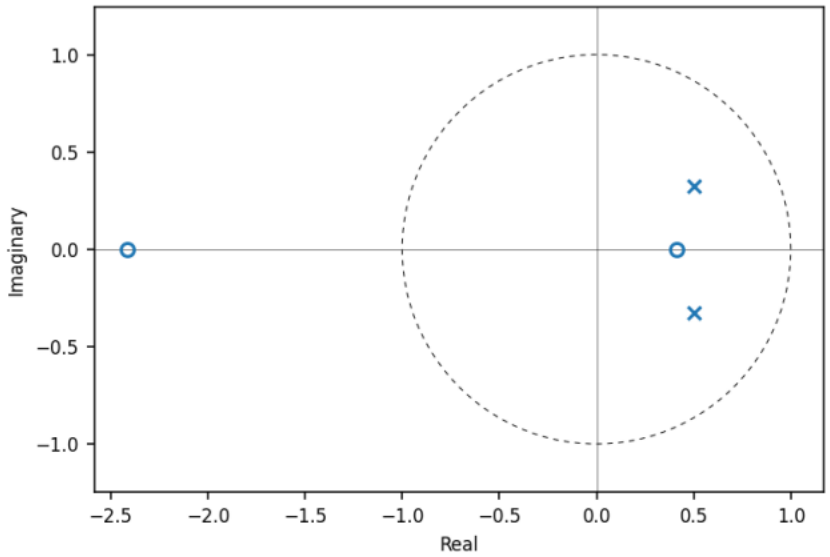


Figure 4.12: Pole-zero plot of H(z) given by equation 4.28

```
phase_degrees = np.angle(h, deg=True)
plt.plot(w, phase_degrees)
plt.title('Phase Response')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Phase (Degrees)')
plt.grid(True, linestyle='--')

plt.tight_layout()
```

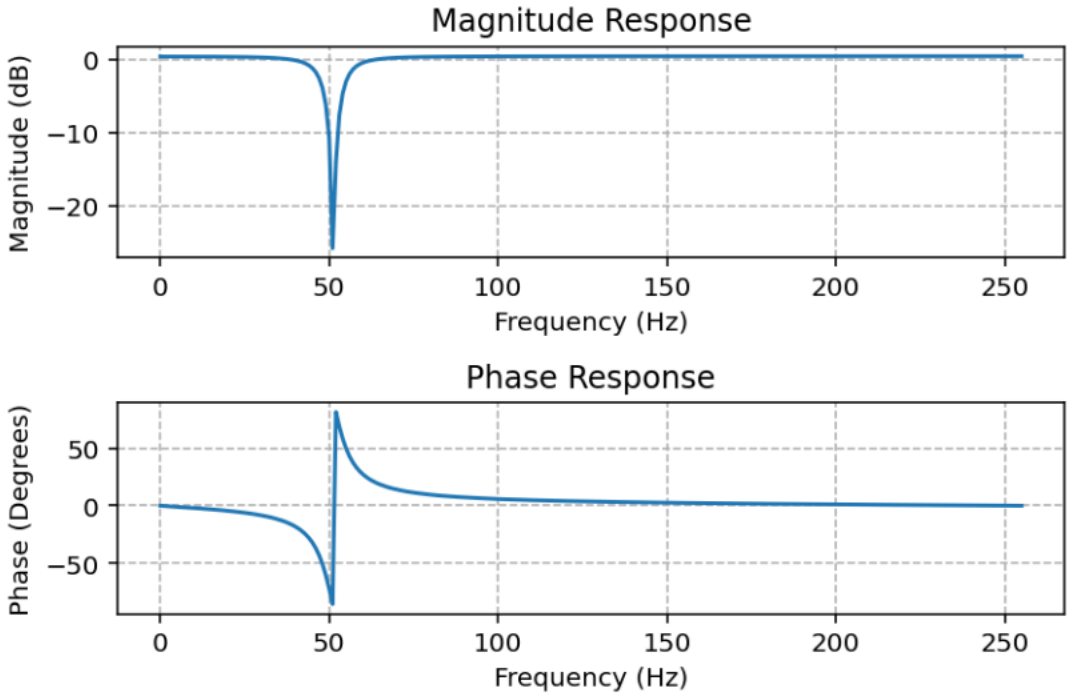


Figure 4.13: Frequency Response of H(z) given by equation 4.31

This is the response of a notch filter. A notch filter is a highly specialized band-stop filter designed to

create a very sharp, narrow rejection band. The filter's magnitude response confirms this function, showing a deep, sudden dip, or "notch," precisely at the $50Hz$ center frequency. This allows the filter to efficiently reject signals at $50Hz$ while leaving surrounding frequencies virtually untouched. In practical equipment, the $50Hz$ notch is crucial for eliminating the persistent power line hum (mains noise) introduced by electrical circuits. Technically, the width and depth of this notch are controlled by adjusting the filter's denominator coefficients (den), which govern the location of the filter's poles in the z-plane, allowing for precise tuning of the notch's sharpness.

Post-laboratory Problems

1. A causal, LTI system is described by the second-order difference equation: $y[n]-4y[n-1]+4y[n-2] = x[n] - x[n-1]$. Determine impulse response of the system (i) by solving the difference equation and (ii) by using `lfilter` function. Show the output for $-20 \leq n \leq 25$. Show the output of the system if the input is $x[n] = (5 + 3\cos 0.2\pi n + 4\sin 0.6\pi n)u[n]$.
2. Consider the system $H(z) = \dfrac{z^{-1} + \frac{1}{2}z^{-2}}{1 - \frac{3}{5}z^{-1} + \frac{2}{25}z^{-2}}$. Determine the frequency response of the system.
3. Consider the system: $H(z) = \dfrac{z^3 - 2z^2 + 2z - 1}{(z-1)(z-0.5)(z-0.2)}$. ROC of the z transform is given by: $\mid z \mid > 2$. Show the pole-zero diagram of the system. Is the system stable?
4. Determine the partial fraction expansion of $X(z) = \dfrac{2z^4 + 16z^3 + 44z^2 + 56z + 32}{3z^4 + 3z^3 - 15z^2 + 18z - 12}$. Find its pole-zero plot and determine the ROC of the Z-transform. What are the ROCs for which the system is causal and stable? Also determine the system's frequency response.
5. Consider a signal $x(t) = sinc^2(100t)$. Sample the signal at a frequency $5KHz$ and find the DFT spectrum.
6. Consider a signal $x[n] = \{\underset{\uparrow}{1}, 1, 2, 3, 5, 8, 13\}$. Find autocorrelation of $x[n]$ using FFT. Show the spectrum and time domain plot for the result.
7. Consider the following system shown in Figure 4.14. If

$$m_1(t) = \begin{cases} \sin c(100t), |t| \leq t_o \\ 0, \quad \text{otherwise} \end{cases}, \quad m_2(t) = \begin{cases} \sin c^2(100t), |t| \leq t_o \\ 0, \quad \text{otherwise} \end{cases}$$

   where $t_0 = 0.1s$. Let carrier $c_1(t) = \cos(2\pi f_{c1}t)$ where $f_{c1} = 250Hz$ and $c_2(t) = \cos(2\pi f_{c2}t)$ where $f_{c2} = 750Hz$. Obtain magnitude spectrum of m_1, m_2 and final output y(t) of the system.

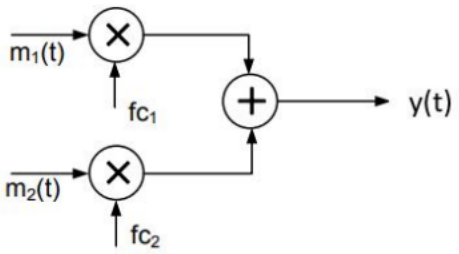


Figure 4.14: Block diagram of FDM transmitter

8. Solve all the exercise problem of the textbook from page 161 to page 163.
9. Solve all the exercise problem of the textbook from page 227.

Laboratory 5

Digital Filter Design (Part - 1)

5.1. Objective

The primary goal of this laboratory exercise is to understand the fundamental concepts and specifications of digital filters, focusing particularly on the Finite Impulse Response (FIR) filter architecture. This involves learning to design FIR filters from specifications using the windowing method, as well as designing, implementing, and analyzing other key filter types, including the first-order Infinite Impulse Response (IIR) filter, the moving average filter, and the digital resonator.

5.2. Digital Filters

In digital signal processing, filter design is the process of converting required frequency response specifications into a working numerical algorithm. This involves finding the correct coefficients for a difference equation and choosing a suitable filter architecture. Since filters are based on rational transfer functions, the design is often a polynomial optimization problem. Additional decisions involve minimizing the filter's computational cost (operations and storage) and selecting the best algorithmic structure to implement the difference equation.

5.3. Design Specification of a Practical Filter

In designing any filter, the ideal frequency response characteristics are desirable but they are not realizable and necessary in most practical applications. In the Figure 5.1, the magnitude characteristics of a physically realizable low pass filter is shown. The bold black line represents the ideal response. In practical filter, magnitude of $H(e^{j\omega}$ in the passband (the allowable frequency range) is not constant like the ideal response but has a small amount of ripple. Similarly in the stop band (the forbidden frequency), the filter response $|H(e^j\omega)|$ is not zero. The transition of the frequency response from passband to stopband defines the *transition band* or *transition region* of the filter. The bandedge frequency ω_p defines the edge of the passband while the frequency ω_s denotes the beginning of the stopband. The width of the transition band is $\omega_s - \omega_p$. The width of the passband is usually called the *bandwidth* of the filter. If δ_p is the amount of ripple in the passband, the magnitude $|H(e^{j\omega}|$ varies between the limits $1 \pm \delta_p$.

The ripple in the stopband of the filter is denoted by δ_s. Usually the response of any filter is given in log scale. In that case, passband ripple and stopband attenuation in decibels are given by

$$\begin{aligned} &\text{Passband ripple, } R = -20\log_{10}(1-\delta_p) \\ &\text{Stopband Attenuation, } A = 20\log_{10}\delta_s \end{aligned} \tag{5.1}$$

In any filter design problem, the following requirements are specified:
— The maximum tolerable passband ripple
— The maximum tolerable stopband ripple
— The passband edge frequency, ω_p
— The stopband edge frequency, ω_s

Based on these specifications, we can select the filter coefficients $\{a_k\}$ and $\{b_k\}$ in the frequency response characteristics.

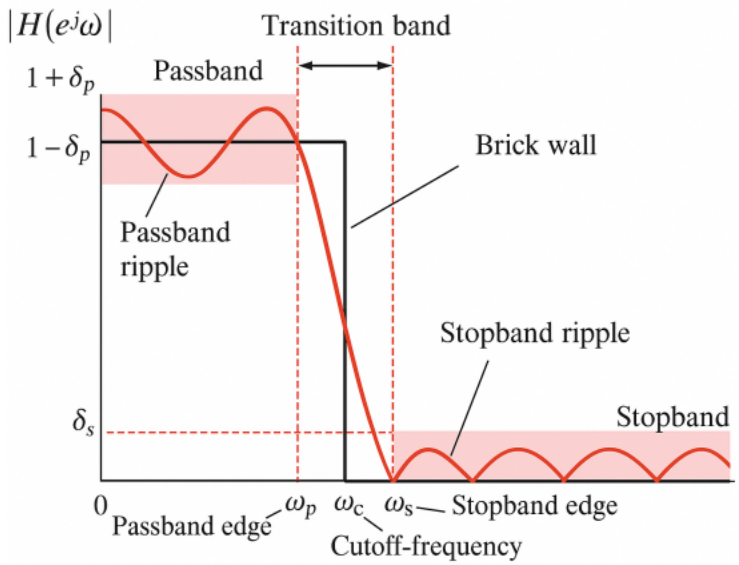


Figure 5.1: Magnitude characteristics of physically realizable filters

5.4. Classification of Digital Filters

Digital filters allow us to selectively modify a signal's frequency content. To effectively design and analyze these systems, we must understand their fundamental classifications. Filters are typically grouped based on their structure (how they compute the output, either Finite or Infinite Impulse Response) and their design objective (what specific frequency behavior they achieve). A comprehensive overview of common filter types and their corresponding categories are shown below.

5.4.1. Classification by Structure (Impulse Response)

Filters are fundamentally defined by whether their output depends only on current and past inputs (**FIR** - *Finite Impulse Response*) or if it also depends on past outputs (**IIR** - *Infinite Impulse Response*). This classification is shown in Table 5.1.

Table 5.1: Filter classification by structure (impulse response)

Category	Filters Belonging to this Category	Key Characteristic
IIR Filters	IIR Filter, First-Order IIR Filter, M-Point Exponentially Weighted Moving Average Filter (EWMA), Digital Resonator, Notch Filter, All-Pass Filter, Comb Filter, Butterworth Filter, Chebyshev Filter	The impulse response continues indefinitely. They use feedback (past outputs) and are efficient but can be unstable.
FIR Filters	FIR Filters, Moving Average Filter	The impulse response settles to zero in a finite amount of time. They use no feedback and are inherently stable.

5.4.2. Classification by Design/Response Type

Beyond structure, filters are often categorized by their explicit function in the frequency domain. Response type tells us what the filter is trying to accomplish - for instance, achieving a perfectly flat passband or completely eliminating a single noise frequency. Understanding these design goals is critical because it dictates the complexity and trade-offs required during the implementation phase. This section classifies filters based on their primary intended application. This classification is shown in Table **??**.

Table 5.2: Filter classification by design/response type

Response Type	Filters Belonging to this Category	Function
Classic Analog Prototypes	Butterworth Filter, Chebyshev Filter, Elliptic Filter	These names define the mathematical shape of the filter's magnitude response (e.g., flat passband, ripple in the passband, etc.). They are usually implemented digitally as IIR filters.
Specific Frequency Shaping	Notch Filter, Digital Resonator, Comb Filter, All-Pass Filter	**Notch:** Blocks a narrow band of frequencies (e.g., 50 Hz noise). **Resonator:** Highly selective band-pass filter. **Comb:** Creates multiple evenly spaced notches/peaks. **All-Pass:** Changes phase but not magnitude.
Smoothing / Averaging	Moving Average filter, M-Point Exponentially Weighted Moving Average Filter (EWMA)	Used for smoothing time-series data. MA is FIR; EWMA is IIR (as it uses weighted feedback).
Fundamental Structure	First-Order IIR Filter	The simplest IIR structure, involving only one previous output sample (one pole).

5.4.3. FIR and IIR Filters

Digital filters require both the time and frequency domain techniques. This is because the desired filter characteristics are specified in the frequency domain but filters are usually described by difference equation in time domain. In the filter design process, we determine the coefficients of the filter's difference equation. A general linear and time invariant causal digital filter with input $x[n]$ and output $y[n]$ can be described by linear constant coefficient difference equations of the form:

$$y[n] = -\sum_{k=1}^{N} b_k y[n-k] + \sum_{k=0}^{M} a_k x[n-k] \tag{5.2}$$

where a_k and b_k are the coefficients which characterizes the filter. a_k's are the feedforward coefficients (affect the input signal) and b_k's are the feedback coefficients (affect the output signal). In case of impulse response, the input is a delta function. If we denote the impulse response $h[n]$, Equation (**??**) becomes

$$h[n] = -\sum_{k=1}^{N} b_k h[n-k] + \sum_{k=0}^{M} a_k \delta[n-k] \tag{5.3}$$

If we take Z-transform of Equation (5.2), we get the system function as:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{a_k \sum_{k=0}^{M} z^{-k}}{1 + \sum_{k=1}^{N} b_k z^{-k}} \tag{5.4}$$

The system function of the filter has M zeros and N poles. Depending on the nature of the impulse response, digital filters are of two types - infinite impulse response (IIR) and finite impulse response (FIR) filters. If $b_k = 0$ for all k in Equation (5.3), then the filter is FIR filter. In case of FIR filter, $h[n]$ and $H(z)$ are given by:

$$\begin{aligned} h[n] &= \sum_{k=0}^{M} a_k \delta[n-k] \\ H(z) &= \sum_{k=0}^{M} a_k z^{-k} \end{aligned} \tag{5.5}$$

Since in case of FIR filter, (5.5) is not recursive, the impulse response has finite duration M and hence the name finite impulse response filter. On the other hand, $b_k \neq 0$ represents an IIR filter. In this case, the Equation (5.3) is recursive and generates an impulse response which is non-zero for $n \rightarrow \infty$. The block diagram of FIR and IIR filter is shown in the Figure 5.2.

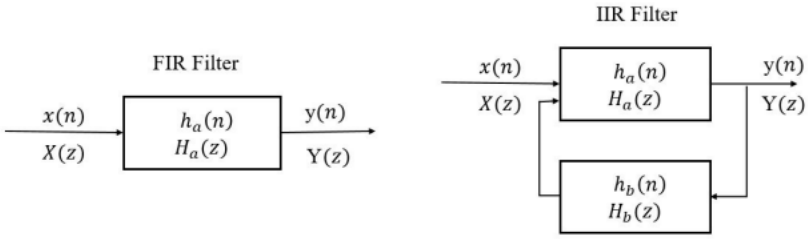


Figure 5.2: Block Diagram of FIR filter and IIR Filter

5.5. Characteristics of The Traditional Digital Filter

5.5.1. First-Order IIR Filter

The transfer function of a first-order IIR filter is given by

$$H(z) = \frac{1}{1 - p_1 z^{-1}} \tag{5.6}$$

If the pole lies on the unit circle $(p_1 = 1)$, the transfer function of the filter is given by

$$H(z) = \frac{1}{1 - z^{-1}} \tag{5.7}$$

This experiment is about obtaining the characteristics of the filter like *(a) impulse response, (b) pole-zero plot* and *(c) magnitude and phase responses* of the filter. The python code which obtains the characteristics of the first-order IIR filter is given below, and the corresponding output is shown in Figure 5.3.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
import control as ct

x = np.zeros(30)
x[0] = 1 # Unit impulse delta[n]
num = [1]
den =  [1, -1] # H(z) = 1 / (1 - z^-1)
fs = 100 # Sample frequency (for plotting frequency response)
sys = ct.tf(num, den, dt=True)
h = signal.lfilter(num, den, x) # Obtain the impulse response
w, H = signal.freqz(num, den)
# w: normalized angular frequency (rad/sample), H: complex response

plt.figure(figsize=(10, 8))

plt.subplot(2, 2, 1)    # Impulse Response (Time Domain)
plt.stem(h)
plt.xlabel('n')
plt.ylabel('h[n]')
```

```
plt.title('Impulse Response (Unit Step)')
plt.grid(linestyle=':')

plt.subplot(2, 2, 2)    # Magnitude Response (Frequency Domain)
plt.plot((w / np.pi) * fs / 2, 20 * np.log10(np.abs(H)))
# Convert normalized frequency 'w' to Hz (w * fs / (2*pi))
plt.xlabel('Normalized Frequency (rad/sample)')
plt.ylabel('Magnitude (dB)')
plt.title('Magnitude Response (Low-Pass)')
plt.grid (linestyle=':')

plt.subplot(2, 2, 3)    # Pole-zero plot
ct.pzmap(sys, title=False)

plt.subplot(2, 2, 4)    # Phase Response (Frequency Domain)
plt.plot((w / np.pi) * fs / 2, np.angle(H))
plt.xlabel('Normalized Frequency (rad/sample)')
plt.ylabel('Phase (Radians)')
plt.title('Phase Response')
plt.grid(linestyle=':')
plt.tight_layout()
```

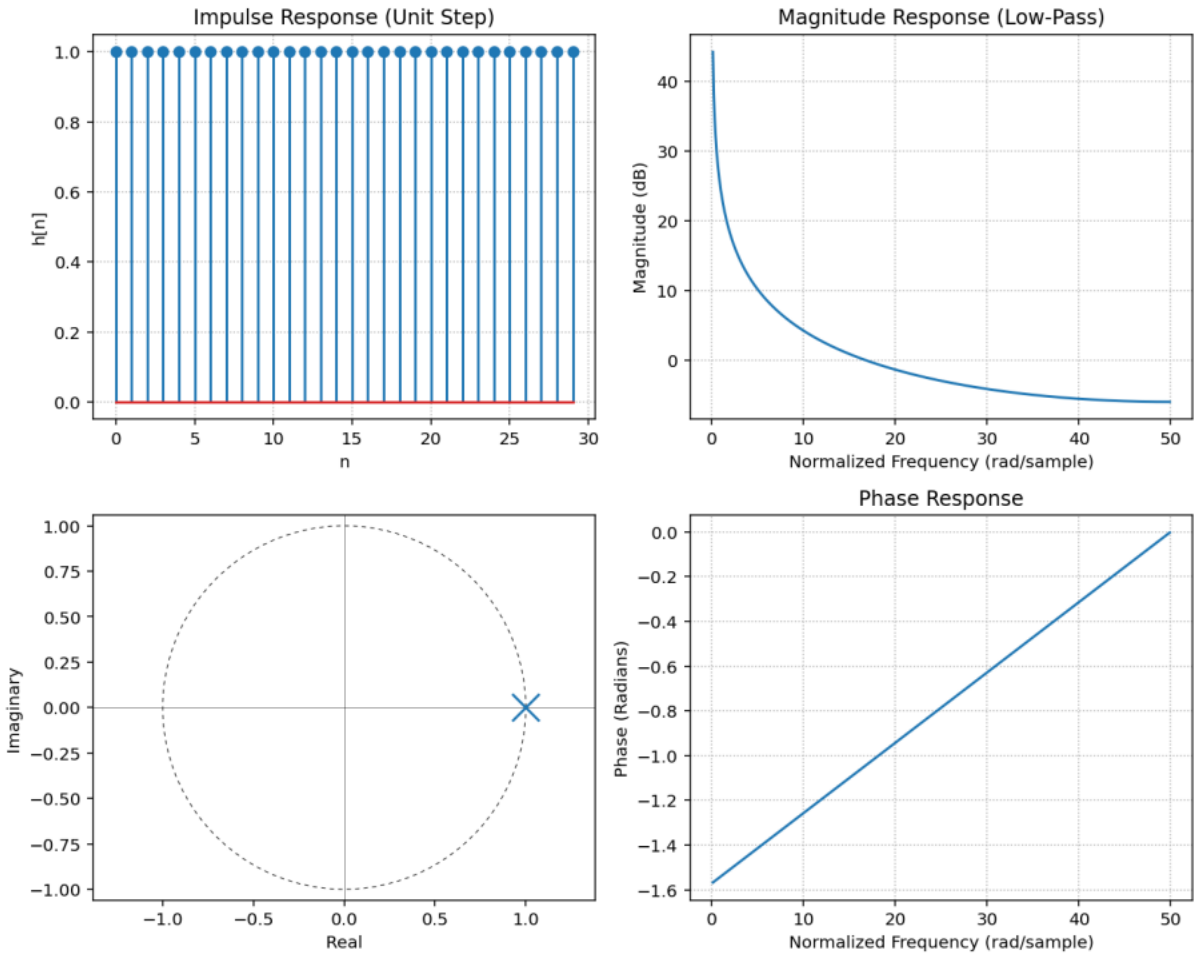


Figure 5.3: Characteristics of first-order IIR filter

From the Figure 5.3, the analysis of the First-Order IIR Filter confirms its expected behavior, as the impulse response is obtained as a unit step function. Correspondingly, the frequency response indicates that the filter operates as a lowpass filter. However, a closer look at stability reveals a critical issue - while a discrete-time system requires poles to be within the unit circle for strict BIBO stability, the pole-zero plot shows the single pole lying directly on the unit circle, implying only marginal stability. This assessment is supported by the time domain analysis, where the unit step impulse response is not absolutely summable (a requirement for BIBO stability), definitively classifying the filter as unstable.

5.5.2. Moving Average filter

The relationship between the input and output of M-point moving average filter is given by

$$y[n] = \frac{1}{M} \sum_{k=0}^{M-1} x[n-k] \tag{5.8}$$

For a three-point moving average filter, M = 3, substituting M = 3 in Equation (5.8), we get the input-output relationship as

$$y[n] = \frac{1}{3} \{x[n] + x[n-1] + x[n-2]\} \tag{5.9}$$

This experiment tries to obtain the characteristics of a moving average filter using python. The python code, which obtains the characteristics of three-point moving average filter, is shown below, and the corresponding output is shown in Figure 5.4.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
import control as ct

x = np.zeros(30)
x[0] = 1
num = [1/3,1/3,1/3]
den =  [1]
fs = 100
h = signal.lfilter(num, den, x)
w, H = signal.freqz(num, den)
sys = ct.tf(num, den, dt=True)
plt.figure(figsize=(10, 8))

plt.subplot(2, 2, 1)
plt.stem(h)
plt.xlabel('n')
plt.ylabel('h[n]')
plt.title('Impulse Response (Unit Step)')
plt.grid(linestyle=':')

plt.subplot(2, 2, 2)
plt.plot((w / np.pi) * fs / 2, 20 * np.log10(np.abs(H)))
plt.xlabel('Normalized Frequency (rad/sample)')
plt.ylabel('Magnitude (dB)')
plt.title('Magnitude Response (Low-Pass)')
plt.grid (linestyle=':')

plt.subplot(2, 2, 3)
```

```
ct.pzmap(sys, title=False, marker_size=16)

plt.subplot(2, 2, 4)
plt.plot((w / np.pi) * fs / 2, np.angle(H))
plt.xlabel('Normalized Frequency (rad/sample)')
plt.ylabel('Phase (Radians)')
plt.title('Phase Response')
plt.grid(linestyle=':')
plt.tight_layout()
```

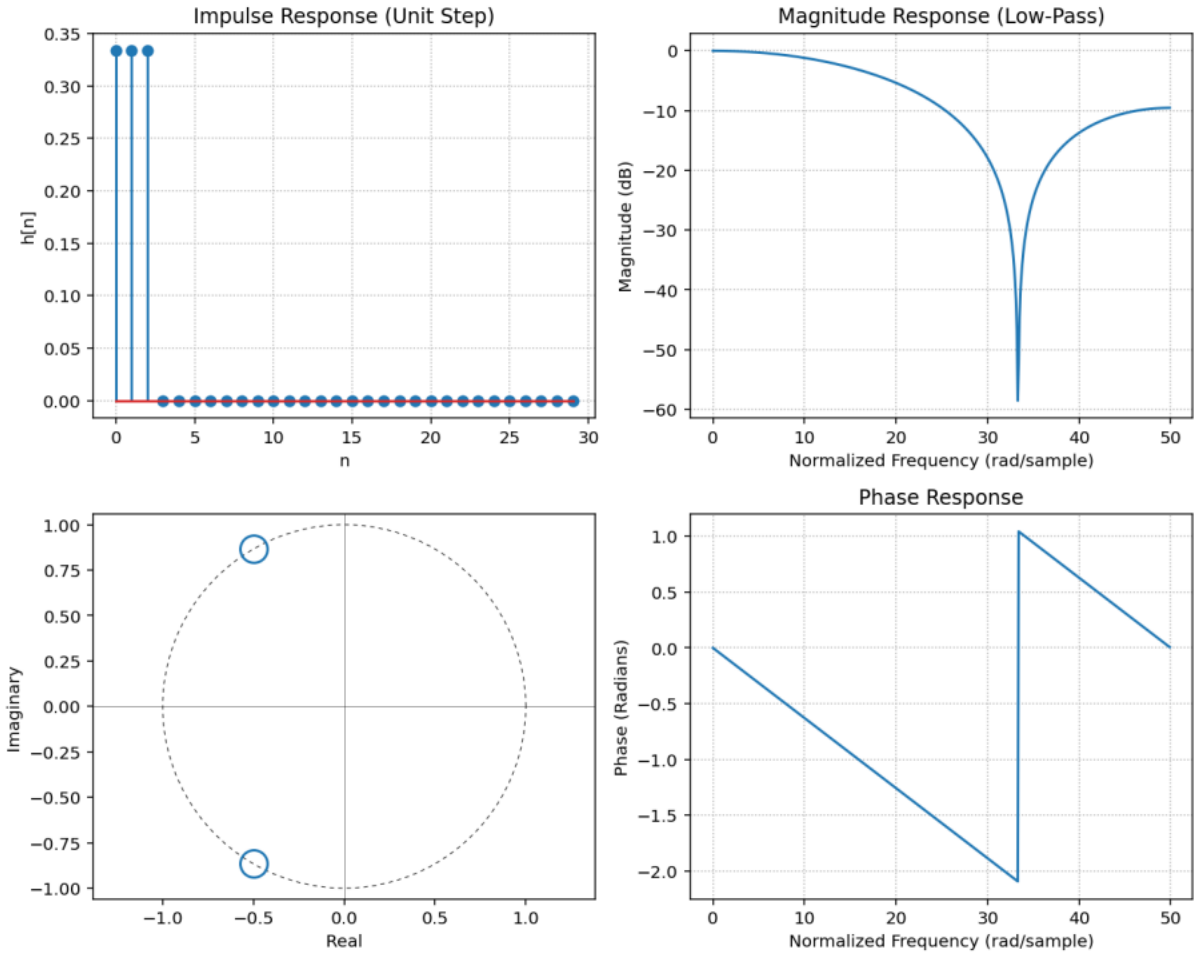


Figure 5.4: Characteristics of three-point moving average filter

Based on the Figure 5.4, the Moving Average (MA) filter demonstrates three key characteristics: inherent stability, lowpass functionality, and linear phase. The filter is fundamentally stable because its impulse response is finite and quickly returns to zero. This absolute stability is visually confirmed by the pole-zero plot. The filter has two finite zeros and zero finite poles - this confirms it is an all-zero filter (no poles outside the unit circle). This design causes it to act as a lowpass filter, effectively smoothing the signal by passing low frequencies while rejecting high-frequency noise. This frequency rejection is enforced by the zeros located directly on the unit circle, which create the deep dips (nulls) seen in the magnitude response. Furthermore, the filter maintains linear phase in the passband, ensuring that the filtered signal is not distorted in time, as all frequency components are delayed by the same constant amount.

5.5.3. Digital Resonator

Digital Resonator is a specialized filter designed to exhibit its strongest response at a single, target input frequency. This makes resonators essential components in various applications, such as communication receivers and AM/FM demodulators. Structurally, a digital resonator is a two-pole bandpass filter that achieves its sharp, high-magnitude peak through a pair of complex-conjugate poles placed very close to the unit circle. If one pole is positioned at $p_1 = re^{j\omega}$, the second must be at $p_2 = re^{-j\omega}$, where r (the distance from the origin) must be less than one $(0 < r < 1)$ to maintain stability. The system's behavior is then mathematically described by its transfer function.

$$H(z) = \frac{1}{(1 - p_1 z^{-1})(1 - p_2 z^{-1})} \tag{5.10}$$

Substituting $p_1 = re^{j\omega}$ and $p_2 = re^{-j\omega}$ and using the Euler's formula, we get,

$$H(z) = \frac{1}{1 - 2rz^{-1}\cos(\omega) + r^2 z^{-2}} \tag{5.11}$$

This experiment analyses the concept of digital resonator using python. The python illustration of digital resonator with two complex conjugate poles occurring at $r = 0.90$ is shown below, and the corresponding output is shown in Figure 5.5

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
import control as ct

x = np.zeros(30)
x[0] = 1
r = 0.9
fs = 100 #Sampling frequency
fc = 10 #Cutoff frequency
w = 2*np.pi*(fc/fs)
num = [1]
den = [1,-2*r*np.cos(w),r**2]
h = signal.lfilter(num, den, x)
w, H = signal.freqz(num, den)
sys = ct.tf(num, den, dt=True)
plt.figure(figsize=(10, 8))

plt.subplot(2, 2, 1)
plt.stem(h)
plt.xlabel('n')
plt.ylabel('h[n]')
plt.title('Impulse Response (Unit Step)')
plt.grid(linestyle=':')

plt.subplot(2, 2, 2)
plt.plot((w / np.pi) * fs / 2, 20 * np.log10(np.abs(H)))
plt.xlabel('Normalized Frequency (rad/sample)')
plt.ylabel('Magnitude (dB)')
plt.title('Magnitude Response (Low-Pass)')
plt.grid (linestyle=':')

plt.subplot(2, 2, 3)
```

```
ct.pzmap(sys, title=False, marker_size=16)

plt.subplot(2, 2, 4)
plt.plot((w / np.pi) * fs / 2, np.angle(H))
plt.xlabel('Normalized Frequency (rad/sample)')
plt.ylabel('Phase (Radians)')
plt.title('Phase Response')
plt.grid(linestyle=':')
plt.tight_layout()
```

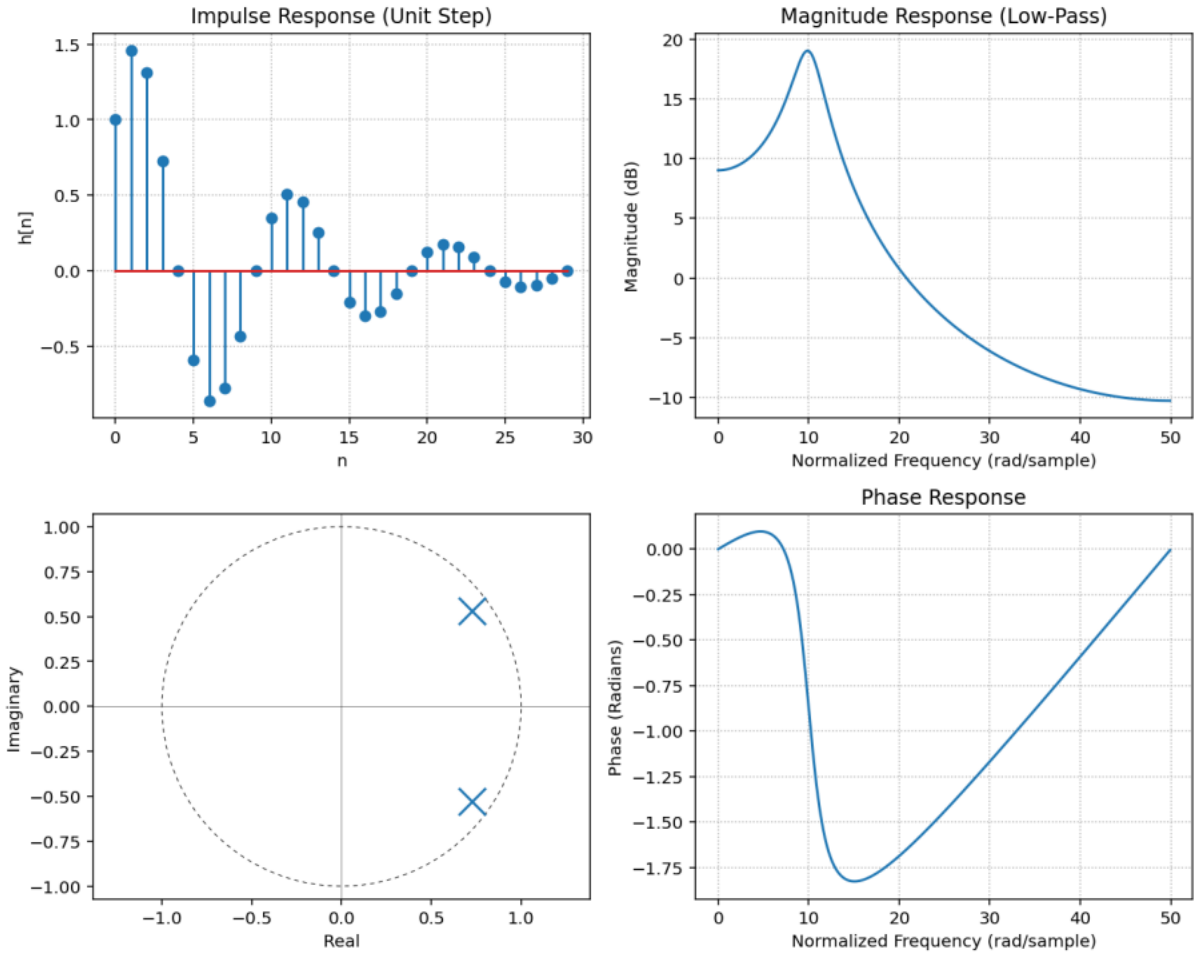


Figure 5.5: Characteristics of a digital resonator with two complex conjugate poles occurring at $r = 0.90$

From the Figure 5.5 it is obvious that, when the resonator is excited by a brief unit sample signal (an impulse), it produces an oscillating output in the time domain, which slowly decays over time. This damped, oscillating response immediately suggests the filter acts as a frequency-selective element. Indeed, the magnitude response confirms this by showing a single, very high and narrow peak, proving the filter functions as a narrow bandpass filter (a resonator). This resonant behavior is structurally rooted in the pole-zero plot, which clearly shows two complex conjugate poles situated extremely close to the unit circle. Since the poles are slightly inside the circle, the oscillation decays, ensuring the system is stable. Finally, the phase response exhibits non-linear characteristics, which is typical for IIR filters like this resonator, meaning that different frequencies passing through the peak are delayed by varying amounts.

5.6. FIR Filter Design *(using Window Method)*

Since in case of FIR filter, Equation 5.5 is not recursive, the impulse response has finite duration M and hence the name finite impulse response filter. The FIR filter in Equation 5.5 has filter length of $M+1$. FIR Filters can have exactly linear phase response and very simple to implement.

Frequency response of a filter $H_D(\omega)$ is related to its impulse response $h_D[n]$ by the Inverse Discrete time Fourier Transform:

$$h_D[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} H_D(\omega)e^{j\omega n}d\omega \tag{5.12}$$

The subscript 'D' is used to denote the ideal behavior. So if $H_D(\omega)$ is known, $h_D[n]$ can be determined from Equation 5.12. Let us illustrate this with an example of Low Pass Filter (LPF). Consider an ideal LPF with cut-off frequency ω_c (normalized) and frequency response $H_D(\omega)$ given by:

$$H_D\left(e^{jw}\right) = \begin{cases} 1, |\omega| \leq \omega_c \\ 0, \omega_c < |\omega| \leq \pi \end{cases} \tag{5.13}$$

Impulse response, $h_D[n]$ for this filter can be found by:

$$\begin{aligned} h_D[n] &= \frac{1}{2\pi}\int_{-\pi}^{\pi} 1 \times e^{j\omega n}d\omega \\ &= \frac{1}{2\pi}\int_{-\pi}^{\pi} e^{j\omega n}d\omega \\ &= \frac{2f_c\sin(n\omega_x)}{n\omega_c}, n \neq 0, -\infty \leq n \leq \infty \\ &= 2f_c, n = 0 \end{aligned} \tag{5.14}$$

Now as $h_D[n]$ is infinite duration signal, its z transform $H_D(z)$ will also be of infinite duration:

$$\begin{aligned} H(z) &= \sum_{n=-\infty}^{\infty} h(n)z^{-n} \\ &= \cdots + h(-2)z^2 + h(-1)z^1 + h(0) + h(1)z^{-1} + h(2)z^{-2} + \cdots \end{aligned} \tag{5.15}$$

It is apparent from Equation (5.14) and (5.15) that $h_D[n]$ is both non-causal and infinite in duration. Hence FIR filter given in Equation (5.14) is not practically realizable.

An obvious solution is to truncate the filter coefficients to $i = -\frac{M}{2}, \cdots 0, \cdots + \frac{M}{2}$, then shifting to the right by $\frac{M}{2}$ gives the filter coefficient for an M^{th} order physically realizable Low-Pass Filter. However this introduces undesirable ripples and overshoot (Gibbs phenomenon). Figure 5.6 illustrates the effects of truncating the filter coefficients. Increasing the filter length does not reduce the problems.

Window method is developed to get rid of this undesirable Gibbs oscillation in the passband and stop-band of the designed FIR filter. As Gibbs oscillations are the result of abrupt truncation of the infinite length filter coefficient sequence. Hence a window function is required which is symmetrical and can gradually weight the desired FIR coefficients down to zeros at the both ends for the range of $-M \leq n \leq M$. Applying the window sequence to the filter coefficients gives

$$h_w[n] = h[n]\dot{w}[n] \tag{5.16}$$

where w[n] is the window function. Desirable frequency characteristics can be obtained by making a better selection for the window, w[n]. There are several window functions are used for this purpose. Some of them are as follows:

1. Rectangular window:

$$w_{rec}[n] = 1, \quad 0 \leq n \leq M-1$$

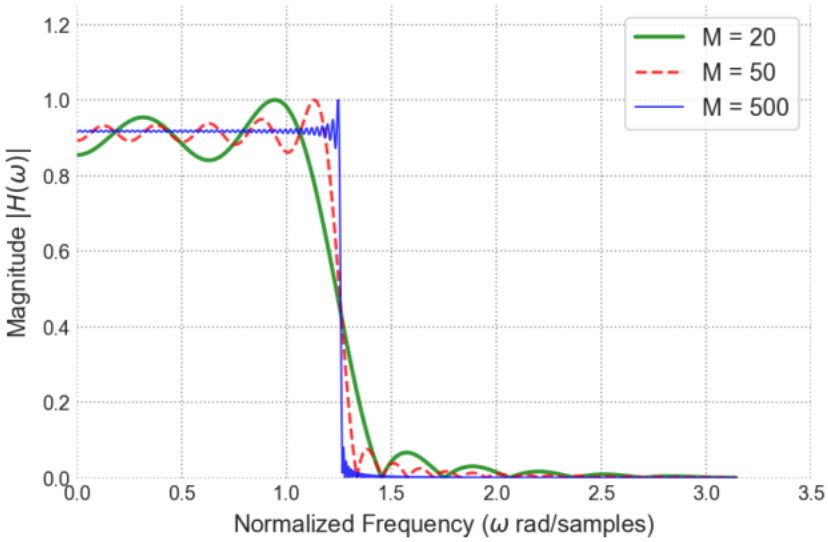


Figure 5.6: Effects on the frequency response of truncating the ideal impulse response to $M = 20, 50 and 500$ coefficients.

2. Triangular (Bartlett) window:

$$w_{tri}[n] = \begin{cases} \dfrac{2n}{M-1}, 0 \leq n \leq \frac{M-1}{2} \\ 2 - \dfrac{2n}{M-1}, \frac{M-1}{2} \leq n \leq M-1 \end{cases}$$

3. Hanning window:

$$w_{han}[n] = 0.5 - 0.5\cos\left(\frac{n\pi}{M}\right), 0 \leq n \leq M \text{(total M+1 points)}$$

4. Hamming window:

$$w_{ham}[n] = 0.54 - 0.46\cos\left(\frac{n\pi}{M}\right), 0 \leq n \leq M$$

5. Blackman window:

$$w_{\text{black}}[n] = 0.42 - 0.5\cos\left(\frac{n\pi}{M}\right) + 0.08\cos\left(\frac{2n\pi}{M}\right), 0 \leq n \leq M$$

Shapes of different window functions in time domain is given in Figure 5.7 for $M = 64$.

In window-based filter design, there are two key frequency domain effects important to the design: the transition band roll-off and the passband and stopband ripple. Two parameters in the spectrum influence these filter properties.

— The filter's roll-off is related to the width of the center lobe of the window's magnitude spectrum
— The ripple is influenced by the ratio of the mainlobe amplitude to the first sidelobe amplitude (or difference in dB scale).

These two window spectrum parameters are not independent. Theoretical values of these two parameters for the aforementioned windows are given in Table 5.3.

In Python, `firwin()` function from `scipy.signal` is used to design a FIR filter by windowing method. Table 5.4 shows the list of the available windows from `scipy.signal` along with their characteristics. The syntax for this function is:

```
signal.firwin(numtaps, cutoff, window='blackman', pass_zero='lowpass', fs=Fs)
```

Here, **`numtaps`** is the length of the filter (number of coefficients, i.e. the filter order + 1). `numtaps` must be odd if a passband includes the Nyquist frequency. **`cutoff`** is the cutoff frequency of filter. **`window`** is the desired window to be used for the filter. **`pass_zero`** is a string argument for the desired filter type, such as `'bandpass'`, `'lowpass'`, `'highpass'`, `'bandstop'`.

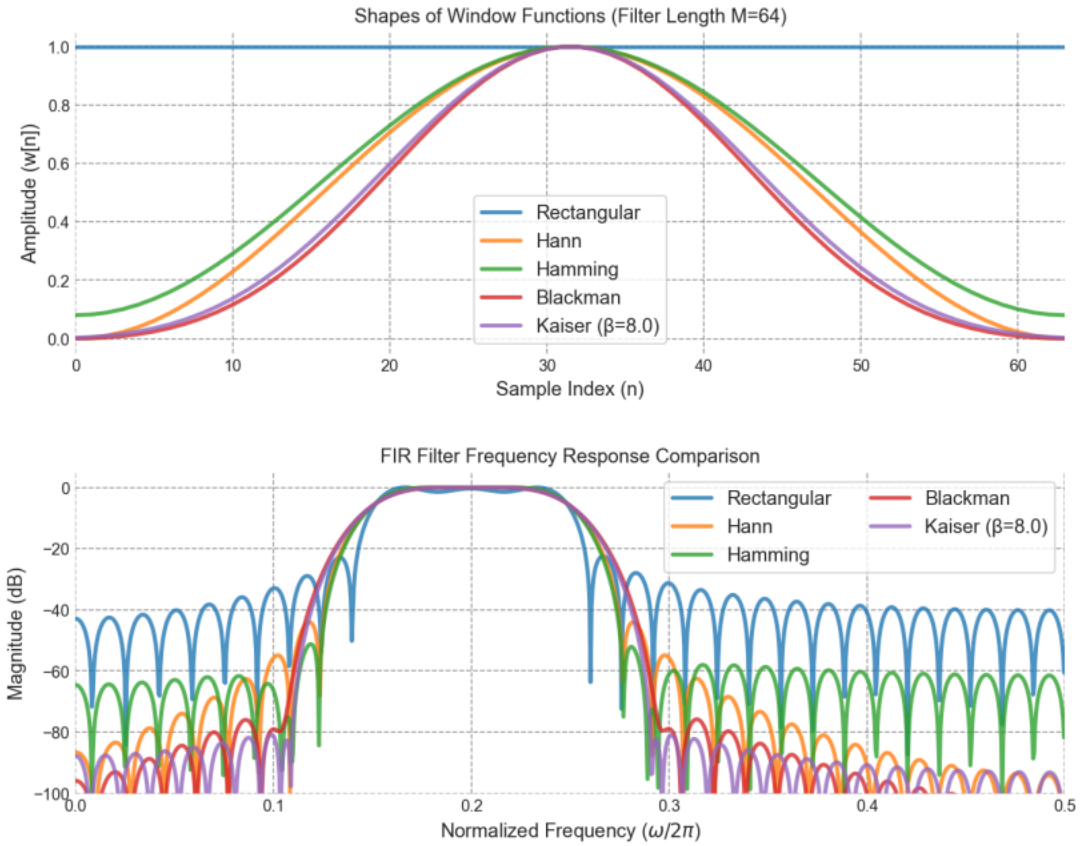


Figure 5.7: Shapes of Window functions for the case of M = 64 (Upper Figure). Frequency response of different window functions (Lower Figure)

Example 1: Design a FIR low pass filter

Design a low pass FIR filter with the following specifications:

— Passband edge frequency 1.5 KHz
— Transition Width 0.5 KHz
— Stopband Attenuation >50 dB
— Sampling Frequency 8 KHz

From Table 5.3, we find that Hamming or Blackman window provides stopband attenuation $> 50dB$ and hence can be used to design this filter. In this example we will use the *Blackman window*. According to the formula of transition width the value of M can easily be computed. Following is the code for designing this specific FIR filter. The result of the following code is shown in Figure 5.8.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

Fs = 8000.0   # Sampling frequency (Hz)
TW = 500.0    # Transition width (Hz)
PBE = 1500.0  # Passband Edge frequency (Hz)

M = int(np.round(6 * Fs / TW))
filter_length = M + 1

Fc = PBE + TW / 2.0                # Cutoff frequency

a = signal.firwin(
```

Table 5.3: Comparison of Common Window Functions for FIR Filter Design.

Window Name	Stopband Attenuation (A_s, dB)	Peak Sidelobe (Relative, dB)	Transition Width (rad/sample)	Key Design Consideration
Rectangular	-21	-13	$\frac{4\pi}{M}$	Best frequency resolution (narrowest main lobe), but worst stopband rejection (highest ripple).
Bartlett (Triangular)	-25	-25	$\frac{8\pi}{M}$	Simple, provides moderate rejection; generally used for quick, non-critical designs.
Hann	-44	-31	$\frac{8\pi}{M}$	Excellent balance between transition width and sidelobe suppression. A common default choice.
Hamming	-53	-41	$\frac{8\pi}{M}$	Optimized to minimize the first sidelobe, leading to better attenuation than Hann for the same M.
Blackman	-74	-57	$\frac{12\pi}{M}$	Highest fixed stopband attenuation (lowest ripple), but requires the largest M for a given transition sharpness.
Kaiser	Tunable (Set by β)	Tunable (Set by β)	$\frac{A_s - 7.9}{2.285 \cdot M \cdot TW}$	Highly flexible. Allows independent specification of the desired stopband attenuation and transition width by adjusting the β parameter.

```
        numtaps = filter_length,
        cutoff = Fc,
        window = 'blackman',
        pass_zero = 'lowpass',
        fs = Fs
        )

N = 512 # number of points for calculating frequency response
f, h = signal.freqz(a, 1, worN=N, fs=Fs)

plt.figure(figsize=(8, 6))

plt.subplot(2, 1, 1)
plt.plot(f, 20 * np.log10(np.abs(h)), linewidth=2)
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude (dB)")
plt.axvline(PBE, color='red', linestyle='--',
        label=f'Passband Edge ({PBE} Hz)')
plt.axvline(PBE + TW, color='purple', linestyle='--',
        label='Stopband Edge (2000 Hz)')
plt.grid()
plt.legend()
```

Table 5.4: FIR Windows Functions from `scipy.signal`

Window Name	Required Parameter	Description / Characteristics
`boxcar`	None	The simplest window. Narrowest main lobe, but the worst sidelobe leakage (spectral leakage).
`triang`	None	A triangular window. Better sidelobe suppression than boxcar, but a wider main lobe.
`bartlett`	None	Mathematically identical to the triangular window.
`hann`	None	A raised cosine window. Good balance of main lobe width and sidelobe attenuation.
`hamming`	None	Optimized to minimize the amplitude of the first sidelobe (no zeros in the Fourier Transform).
`blackman`	None	A three-term cosine window. Offers very high sidelobe attenuation.
`kaiser`	β	A tunable window. β controls the trade-off between main lobe width and sidelobe level.
`gaussian`	σ	The standard deviation σ (in samples) controls the spreading.
`blackmanharris`	None	A four-term cosine window offering excellent sidelobe suppression.
flattop	None	Designed for highly accurate amplitude measurements due to its very flat main lobe.
`tukey`	α	A rectangular window with raised-cosine edges. α is the fraction of the window tapered.
`chebwin`	Attenuation (A)	Optimized to provide the narrowest main lobe for a specified constant sidelobe attenuation A.
`exponential`	Decay (τ)	A two-sided exponential decay window where τ is the time constant of decay.
`parzen`	None	A power of a cosine function, offering smooth roll-off.
`welch`	None	A parabolic-shaped window, often used in power spectrum estimation (*Welch's method*).

```
37
38 plt.subplot(2, 1, 2)
39 plt.plot(f, np.degrees(np.unwrap(np.angle(h))), linewidth=2, color='green')
40 plt.xlabel("Frequency (Hz)")
41 plt.ylabel("Angle (Degree)")
42 plt.grid()
43 plt.tight_layout()
44
45 print(f"Filter Order (M): {M}")
46 print(f"Filter Length (Taps): {filter_length}")
47 print(f"Compensated Ideal Cutoff Frequency: {corner} Hz")
```

From the Figure 5.8, the magnitude response falls gradually after transition band edge and there are ripples with attenuation $> 50dB$ in the stopband. The phase reponse of the FIR filter is observed to be linear as expected. The same filter for the above specifications can also be designed by Hamming window.

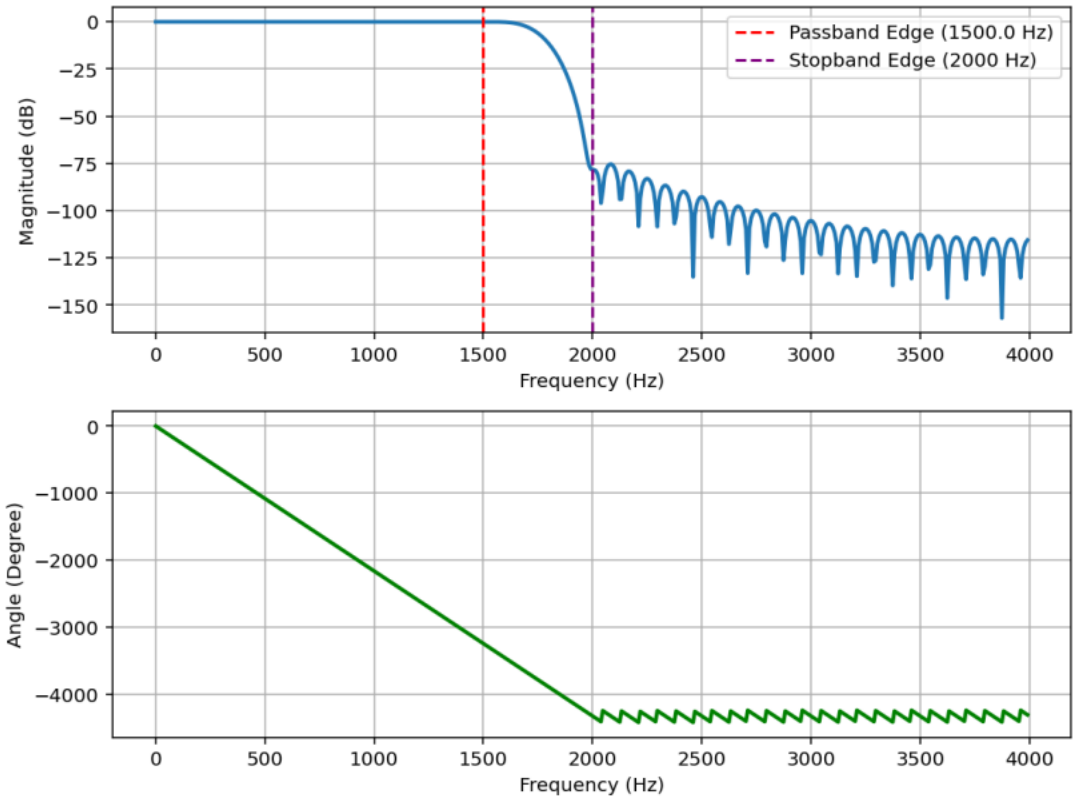


Figure 5.8: Magnitude and Phase response of the designed FIR filter

Example 2: Design a FIR band pass filter

Design a bandpass filter, which will pass signal in the frequency range 10 - 20 Hz. Choose 100 Hz as the sampling frequency. The order of the filter is 50, and use the *Hamming window*.

The python code to design the filter corresponding to the desired bandpass filter is given below. The result of the following code is shown in Figure 5.9.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

Fs = 100   # Sampling frequency (Hz)
M = 50    # Filter order
Fl = 10  # Lower cutoff frequency (Hz)
Fu = 20   # Upper cutoff frequency (Hz)
filter_length = M + 1
Fc = [Fl, Fu] # Normalized cutoffs

a = signal.firwin(
        numtaps=filter_length,
        cutoff=Fc,
        window='hamming',
        pass_zero='bandpass',
        fs=Fs
        )

N = 512 # number of points for calculating frequency response
f, h = signal.freqz(a, 1, worN=N, fs=Fs)
```

```
plt.figure(figsize=(8, 6))

plt.subplot(2, 1, 1)
plt.plot(f, 20 * np.log10(np.abs(h)))
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude (dB)")
plt.axvline(Fl, color='red', linestyle='--',
        label=f'Lower cutoff ({Fl} Hz)')
plt.axvline(Fu, color='purple', linestyle='--',
        label=f'Upper cutoff ({Fu} Hz)')
plt.axhline(-3, color='orange', linestyle='-.', label='-3 dB Mark')
plt.grid()
plt.legend()

plt.subplot(2, 1, 2)
plt.plot(f, np.degrees(np.unwrap(np.angle(h))), color='green')
plt.xlabel("Frequency (Hz)")
plt.ylabel("Angle (Degree)")
plt.grid()
plt.tight_layout()
```

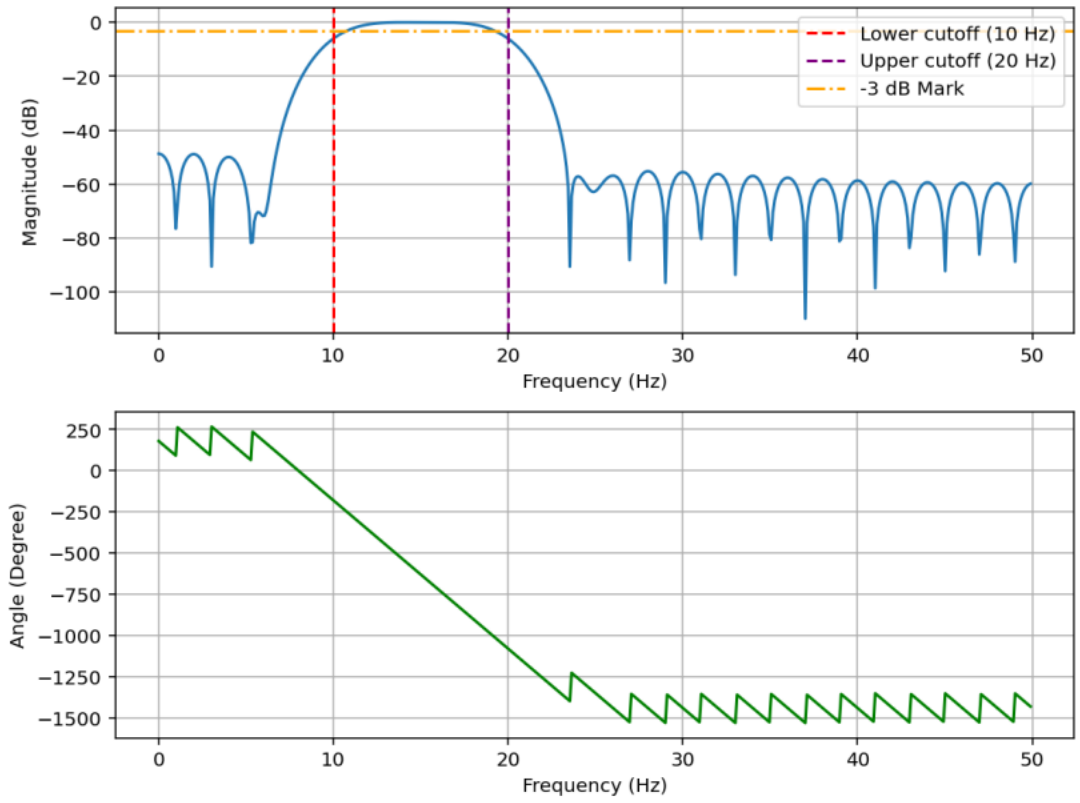


Figure 5.9: Magnitude and Phase response of the designed FIR filter

In the Figure 5.9, the magnitude response of the FIR Bandpass Filter effectively isolates the 10-20 Hz target frequency band, showing near 0 dB gain (minimal attenuation) in the passband. In the stopbands (below 10 Hz and above 20 Hz), the filter achieves significant attenuation (over -60 dB), with the stopband ripples being characteristic of the Hamming window design. Additionally, the phase response plot confirms

the filter's linear phase characteristic within the passband, ensuring all frequencies experience the same time delay and thus preserving the signal's waveform without phase distortion.

Post-laboratory Problems

1. Plot the magnitude and phase responses for a 50th order Highpass filter assuming a sampling frequency of 5KHz and a cut-off frequency of 1KHz for each of the windows mentioned in the lab manual.
2. Design an FIR bandpass filter to meet the following specifications:
   — passband 150-250 Hz
   — transition width 50 Hz
   — passband ripple 0.05 dB
   — stopband attenuation 50 dB
   — sampling frequency 1 KHz
   Choose a suitable window method and explain why your reasoning of choosing that particular window. Plot the magnitude and phase response of the FIR filter.
3. Obtain the characteristics of the filters that have the following transfer functions
   a) $H(z) = \frac{1}{1+2z^{-1}}$
   b) $H(z) = \frac{1}{1-2z^{-1}}$
   c) $H(z) = 1 - 2z^{-1}cos(\omega) + z^{-2}$
   d) $H(z) = \frac{z^{-1}-a}{1-az^{-1}}$, where $|a| < 1$
4. Design a High-Pass FIR Filter that will block signals below 150 Hz. The specifications are -
   — Filter Type: High-Pass
   — Passband Edge Frequency: 150 Hz (signals at or above 150 Hz should pass)
   — Sampling Frequency: 1000 Hz
   — Filter Order: 60
   — Window Type: Blackman window (known for very high stopband attenuation)
5. Design a Band-Stop FIR Filter that is specifically intended to reject (or notch out) the frequencies between 45 Hz and 55 Hz (often used to eliminate 50 Hz or 60 Hz power line noise).
   — Filter Type: Band-Stop
   — Stopband Frequencies: 45 Hz to 55 Hz
   — Sampling Frequency: 300 Hz
   — Filter Order: 120
   — Window Type: Kaiser window (requires specifying the beta parameter, β. Use $\beta = 8.0$ for high stopband rejection).
6. Design a Low-Pass FIR Filter that will pass signals up to 20 kHz. This design will highlight the Gibbs phenomenon due to the chosen window.
   — Filter Type: Low-Pass
   — Cutoff Frequency: 20 kHz
   — Sampling Frequency: 100 kHz
   — Filter Order: 40
   — Window Type: Rectangular window (known for its narrow transition band but high passband ripple)
7. Solve all the exercise problem of the textbook from page 258.
8. Solve all the exercise problem of the textbook from page 298 to page 300.

Laboratory 6

Digital Filter Design (Part - 2)

6.1. Objective

This lab is designed to provide a comprehensive understanding of Infinite Impulse Response (IIR) Filters, focusing on the core theoretical and practical steps required for their implementation. The primary goal is to learn how to design digital IIR filters from specifications by mastering key techniques, including the pole-zero placement method and the conversion method known as the bilinear transformation. Through practical design and analysis exercises, the lab will specifically focus on two foundational IIR filter types: the Butterworth filter and the Chebyshev filter, enabling the calculation of necessary coefficients and the analysis of their distinct frequency responses.

6.2. IIR Filters

In IIR filter, the current output is a function of the current and previous inputs and past outputs. The relationship between the input and output of an IIR filter is given by

$$y[n] = -\sum_{k=1}^{N} b_k y[n-k] + \sum_{k=0}^{M} a_k x[n-k] \tag{6.1}$$

where a_k and b_k are the coefficients which characterizes the filter. a_k's are the feedforward coefficients (affect the input signal) and b_k's are the feedback coefficients (affect the output signal).

From Equation 6.1, it is possible to observe that the current output is a function of current and previous inputs and past outputs. Thus, IIR filters are *recursive filters*. In a recursive filter, the current output depends on both the input and previously calculated outputs. The word *recursive* literally means *running back* and refers to previously calculated output values that go back into calculating the current output along with input values.

Upon taking Z-transform of the input-output relation given in Equation 6.1, the transfer function expression for IIR filter is given by

$$H(z) = \frac{Y(z)}{X(z)} = \frac{a_k \sum_{k=0}^{M} z^{-k}}{1 + \sum_{k=1}^{N} b_k z^{-k}} \tag{6.2}$$

The IIR filter is defined by its infinite impulse response, meaning the output signal never completely dies out. A consequence of this infinite and non-symmetrical response is that the filter cannot achieve linear phase and will delay different frequencies by different amounts (non-constant group delay). Furthermore, unlike other filters, stability is not guaranteed; if a pole lies outside the unit circle, the filter becomes unstable, causing the output to grow infinitely. Despite these complexities, the primary advantage of the IIR filter is its extreme efficiency, as it can meet design specifications using the smallest possible number of coefficients.

6.3. FIR vs IIR - Which one to Choose?

The decision between an FIR and an IIR filter comes down to a critical trade-off between efficiency and signal quality. IIR filters are significantly more efficient because they meet specifications using far fewer computational steps, and they are quick to design using simple, pre-existing formulas. However, they cannot achieve linear phase, which means they distort the timing of the signal. Conversely, FIR filters can

achieve perfect linear phase, preventing this timing distortion, and they offer great flexibility in shaping the frequency response. The cost of this precision is that FIR filters are generally less efficient (requiring more steps) and must be designed using complex computer algorithms, rather than simple equations. Therefore, the choice depends entirely on the application: if signal integrity (linear phase) is a must-have, choose FIR; if minimizing cost and processing power is the priority, choose IIR.

6.4. IIR Filter Design

Designing a digital IIR filter essentially means finding the correct numbers, or coefficients, for the filter's equation (Equation 6.1) so that it meets all the performance requirements. There are two main ways to do this:

1. The simple way is the *Pole-Zero Placement* method. Here, you manually place points (called poles and zeros) on a chart called the z-plane until the filter achieves the desired frequency response. This method is quick and useful for designing basic filters, like a notch filter that blocks a specific frequency.
2. The standard way (which is more effective for complex designs) is to first design a classic analog filter (a continuous one) that meets the goals. Then, you use a mathematical process to convert that analog design into an equivalent digital IIR filter. Most professional IIR filters are created this way. The three most common methods for this conversion are the *Impulse Invariant*, the *Matched Z-Transform*, and the *Bilinear Transformation Technique*.

This laboratory explores both the *Pole-Zero Placement* and the *Bilinear Transformation Technique* methods to design the IIR filters.

6.4.1. Steps of Filter Design Using Pole-Zero Placement Method

By strategically placing poles and zeros on the z-plane, simple low pass or other frequency selective filters can be designed. While designing simple digital filters using pole-zero placement method the following steps are to be followed:

— The location of poles and zeros are to be determined from the desired filter specifications.

— The transfer function $H(z)$ is written in terms of poles and zeros. If $z_1, z_2 \cdot z_M$ are the zeros of the transfer function and $p_1, p_2 \cdots p_N$ are the poles, then

$$H(z) = \frac{(z - z_1)(z - z_2) \cdots (z - z_M)}{(z - p_1)(z - p_2) \cdots (z - p_N)} \tag{6.3}$$

— From H(z), the corresponding difference equation is obtained and finally the filter coefficients $\{a_k\}$ and $\{b_k\}$ are determined.

6.4.2. Steps of Filter Design Using Bilinear Transformation Technique

Most of the digital filters are designed from their analog counterparts using a transformation. In bilinear transformation method, an analog filter $H(s)$ (where $H(s)$ is the S-domain representation of the transfer function of the system) is converted into a digital filter $H(z)$ by replacing s as follows:

$$s = \frac{2}{T} \frac{1 - z^{-1}}{1 + z^{-1}} \tag{6.4}$$

The above transformation maps H(s) from the s-plane to H(z) in z-plane. However, the direct replacement of s in H(s) as shown in equation (**??**) may lead to a digital filter with undesirable response. This is because the behavior of the desired filter response $H(e^{j\omega})$ at some analog frequency $\omega = \omega_a$ does not appear at the digital frequency ω_a but at frequency ω_d, where

$$\omega_d = \frac{2}{T} \Big(\tan^{-1} \frac{\omega_a T}{2} \Big) \tag{6.5}$$

This is termed as *frequency warping*. ω_a is an analog frequency (rad/s), which exists only in the theoretical design step of the analog filter $H(s)$. The overall goal of IIR filter design is to create a digital filter $H(z)$ that

has a specific cutoff frequency, say f_c (or ω_d). When you use the Bilinear Transformation to map the analog filter $H(s)$ to the digital filter $H(z)$, the digital frequency ω_d you get is not the same as the original analog frequency you put in. This is called *frequency warping*. To counteract this warping, you must calculate a special *warped* analog frequency, ω_a, using the digital frequency ω_d you actually want. This ω_a is the frequency that the analog filter must be designed to have as its cutoff. When the analog filter is designed with this special ω_a cutoff and then transformed, its final digital cutoff ω_d will correctly land exactly where you want it. For this reason, before replacing s by the formula given in Equation 6.4, the cut-off frequency of the filter must be pre-warped by the following equation:

$$\omega_a = k \tan\left(\frac{\omega_d T}{2}\right) \tag{6.6}$$

where,
$\omega_d =$ specified cutoff frequency
ω_a prewarped cutoff frequency
$T =$ sampling period
Now if we summarize the steps for this process:

— Given the digital filter frequency specifications, we need to prewarp the digital frequency specifications to the analog frequency specifications. For the low pass filter and high pass filter:

$$\omega_a = \frac{2}{T} \tan\left(\frac{\omega_d T}{2}\right) \tag{6.7}$$

For the bandpass and bandstop filter:

$$\omega_{al} = \frac{2}{T} \tan\left(\frac{\omega_{dl} T}{2}\right), \quad \omega_{ah} = \frac{2}{T} \tan\left(\frac{\omega_{dh} T}{2}\right) \tag{6.8}$$

— Then we need to perform prototype transforming by replacing s in the transfer function, $H(s)$ using one of the following transformations, depending on the type of filter required:

$$\begin{aligned} s &= \frac{s}{\omega_a} && \text{from lowpass to lowpass} \\ s &= \frac{\omega_a}{s} && \text{from lowpass to highpass} \\ s &= \frac{s^2 + \omega_0^2}{sW} && \text{from lowpass to bandpass} \\ s &= \frac{sW}{s^2 + \omega_0^2} && \text{from lowpass to bandstop} \end{aligned} \tag{6.9}$$

where, $\omega_0 = \sqrt{\omega_{al}\omega_{ah}}$ and $W = \omega_{ah} - \omega_{al}$

— Finally we need to substitute $s = \frac{2}{T}\frac{1 - z^{-1}}{1 + z^{-1}}$ in $H(s)$ to obtain the digital filter.

6.5. Pole-Zero Method

Design a bandpass digital filter using pole-zero placement method to meet the following specifications:
— complete signal rejection at dc and 250 Hz
— a narrow passband centered at 125 Hz
— a 3dB bandwidth of 10 Hz
The sampling frequency for the above filter is 500 Hz.

To design this filter, first determine the pole-zero location of the desired filter. Since complete rejection of signals is required at DC i.e. $0Hz$ and $250Hz$, we need to place the zeros at an angle $0°$ and $2\pi \times \frac{250}{500} = \pi$

on the unit circle. To have a narrow passband centered at $125Hz$, we need to place poles at $2\pi \times \frac{125}{500} = \frac{\pi}{2}$. Now in order to ensure that the filter coefficients are real, we need to have a complex conjugate pole pair and so the angle corresponding to the pole will be $\pm\frac{\pi}{2}$.

Now determine the magnitude of the poles which is governed by the bandwidth specifications. An approximate relationship between r (for $r > 0.9$) and bandwidth (bw) is given by:

$$r \approx 1 - \left(\frac{\text{bw}}{F_s}\right)\pi \tag{6.10}$$

For the given specifications, $bw = 10Hz$ and $F_s = 500Hz$. Plugging this number in equation(6.10), we get $r = 0.937$. The pole-zero diagram for this filter is shown in Figure 6.1.

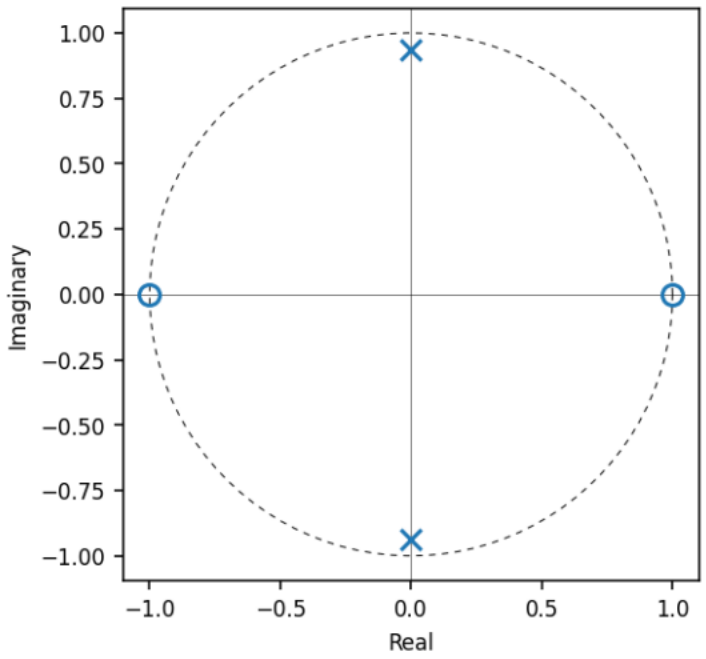


Figure 6.1: Pole-zero plot of the filter of the designed IIR filter

The transfer function of the filter can be written in Z-domain:

$$\begin{aligned} H(z) &= \frac{(z-1)(z+1)}{(z - re^{j\pi/2})(z - re^{-j\pi/2})} \\ &= \frac{z^2 - 1}{z^2 + 0.877969} \\ &= \frac{1 - z^{-2}}{1 + 0.877969z^{-2}} \end{aligned} \tag{6.11}$$

From the Equation 6.11 we find that the filter is a second order one with the following coefficients:

$$\begin{aligned} a_0 &= 1 \quad & b_1 &= 0 \\ a_1 &= 0 \quad & b_2 &= 0.877969 \\ a_2 &= -1 \end{aligned}$$

The code for the filter's magnitude response is given below. The frequency response of the filter is shown the Figure 6.2.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy import signal
```

```
import control as ct

Fs = 500  # Sampling frequency (Hz)
bw = 10   # Bandwidth (Hz)
Fr1 = 0   # DC frequency
Fr2 = 250 # Nyquist frequency (Fs/2)
# Angle corresponding to the zeros (theta = 2*pi*F/Fs)
theta1 = 2 * np.pi * (Fr1/Fs)
theta2 = 2 * np.pi * (Fr2/Fs)
# Zero locations on the Z-plane
z1 = np.exp(1j * theta1)  # z = 1 (DC)
z2 = np.exp(1j * theta2)  # z = -1 (Nyquist)

Fp = 125  # Passband center frequency (Hz)
# Angle corresponding to the pole
thetap = 2 * np.pi * (Fp / Fs)
# Pole magnitude (r) based on 3dB bandwidth approximation
r = 1 - (bw / Fs) * np.pi
# Pole locations (complex conjugate pair)
p1 = r * np.exp(1j * thetap)
p2 = r * np.exp(-1j * thetap)
# Collect all poles and zeros
all_zeros = np.array([z1, z2])
all_poles = np.array([p1, p2])
b = np.poly(all_zeros).real  # Numerator coefficients
a = np.poly(all_poles).real  # Denominator coefficients
print(f"Filter Numerator (b): {b}")
print(f"Filter Denominator (a): {a}")
# Calculate the response. worN=256 is the number of points.
w, h = signal.freqz(b, a, worN=256, fs=Fs)

plt.figure(1, figsize=(7, 6))
# Subplot 1: Magnitude Response
plt.subplot(2, 1, 1)
# Normalize magnitude by the peak value
plt.plot(w, np.abs(h) / np.max(np.abs(h)), linewidth=2)
plt.title('Magnitude Response (Normalized)')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Normalized Magnitude')
plt.grid(True)
# Subplot 2: Phase Response
plt.subplot(2, 1, 2)
# Convert phase from radians to degrees
plt.plot(w, np.angle(h) * 180 / np.pi, color='green', linewidth=2)
plt.title('Phase Response')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Phase (Degrees)')
plt.grid(True)
plt.tight_layout()
# Create the discrete-time transfer function object
sys = ct.tf(b, a, dt=1/Fs)
plt.figure(2, figsize=(4, 4))
```

```
# Use pzmap to automatically generate the pole-zero plot
ct.pzmap(sys, title=False, marker_size=8)
```

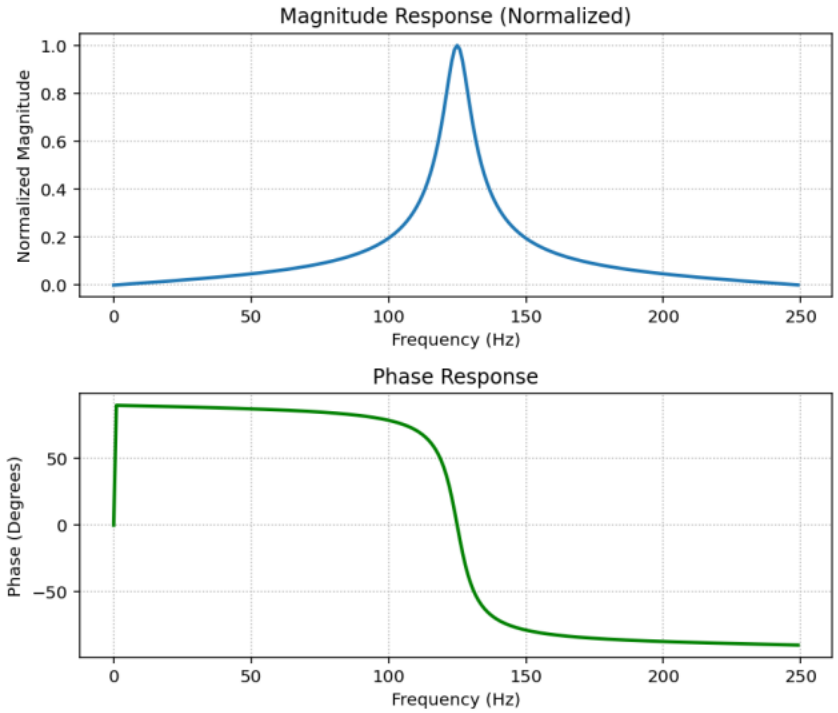


Figure 6.2: Magnitude (normalized) and phase response of the designed IIR bandpass filter

From the filter response in the Figure 6.2, it is seen that the filter has a narrow passband with a peak at frequency 125 Hz and at 0 and 250 Hz, the filter's magnitude is zero which were the design criteria. *(Verify that the filter has a 3 dB bandwidth of 10Hz.).* Thus by pole-zero placement method, we can design simple filters like the one shown in this example.

6.6. Bilinear Transformation Technique (BLT)

6.6.1. Example 1: Design a high pass filter from low pass prototype filter using Bilinear Transformation

Design a high pass filter assuming sampling frequency 150 Hz and cut-off frequency 30 Hz.

For the above filter, first calculate the pre-warped analog frequency. For 30 Hz cut-off frequency the corresponding digital frequency, $\omega_d = 2\pi \times 30 = 60\pi$ and the prewarped analog frequency,

$$\omega_a = \frac{2}{T_s} tan\left(\frac{\omega_d T_s}{2}\right) = 217.9628 rad/s$$

. Now to design a high-pass digital filter from a low pass filter, use the required transformation shown in the Table 6.1.

After this transformation of s, substitute $s = \frac{2}{T_s}\frac{1-z^{-1}}{(1+z^{-1}}$ to complete the bilinear transformation. This is done by the `bilinear()` function from `scipy.signal` package. Syntax for the function is

```
a, b = bilinear(A, B, Fs)
```

Here, A and B are the numerator and denominator vectors of the analog filter respectively. a and b are the numerator and denominator vectors of the digital filter respectively. In other words, a is the feedforward coefficient vector and b is the feedback coefficient vector of the digital filter. Fs is the sampling frequency.

Table 6.1: Analog Low-Pass Filter Transformations

Filter Transformation	Python function	Parameters
Low-pass to low-pass	`lp2lp(a, b, w0)`	b: Numerator polynomial coefficients. a: Denominator polynomial coefficients. wo: Desired cutoff (rad/s)
Low-pass to high-pass	`lp2hp(a, b, w0)`	b: Numerator polynomial coefficients. a: Denominator polynomial coefficients. wo: Desired cutoff (rad/s)
Low-pass to band-pass	`lp2bp(a, b, w0, bw)`	b: Numerator polynomial coefficients. a: Denominator polynomial coefficients. wo: Desired passband center (rad/s) bw: Desired passband width (rad/s)
Low-pass to band-stop	`lp2bs(a, b, w0, bw)`	b: Numerator polynomial coefficients. a: Denominator polynomial coefficients. wo: Desired stopband center (rad/s) bw: Desired stopband width (rad/s)

After obtaining $H(z)$ use the `freqz()` function to determine the frequency response of the filter. The code of the above procedure is given below. Figure 6.3 shows the frequency response of the desired filter.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

Fc = 30  # Cut-off frequency (Hz)
Fs = 150 # Sampling frequency (Hz)
Ts = 1 / Fs # Sampling period (s)

wd = 2 * np.pi * Fc     # Pre-warping Calculation
wa = 2 / Ts * np.tan(wd * Ts / 2)

print(f"Sampling Frequency (Fs): {Fs} Hz")
print(f"Digital Cut-off Frequency (Fc): {Fc} Hz")
print(f"Pre-warped Analog Cut-off (wa): {wa:.2f} rad/s")

num_lp = [1]    # Analog Low-Pass Prototype
den_lp = [1, 1]
# Low-Pass to High-Pass Transformation (Analog Domain)
num_hp_a, den_hp_a = signal.lp2hp(num_lp, den_lp, wa)
# Bilinear Transformation (Analog to Digital)
a, b = signal.bilinear(num_hp_a, den_hp_a, Fs)
N = 512 # Number of FFT points
f, hz = signal.freqz(a, b, worN=N, fs=Fs)
# --- Phase Response Processing ---
phi_rad = np.unwrap(np.angle(hz))
phi_deg = np.degrees(phi_rad)

plt.subplot(2, 1, 1)    # Subplot 1: Magnitude Response
plt.plot(f, np.abs(hz), linewidth=2)
plt.ylabel("Magnitude Response")
plt.xlim(0, Fs/2)
```

```
plt.grid(True)

plt.subplot(2, 1, 2)    # Subplot 2: Phase Response
plt.plot(f, phi_deg, color='orange', linewidth=2)
plt.ylabel("Phase Response (Degrees)")
plt.xlabel("Frequency (Hz)")
plt.xlim(0, Fs/2)
plt.grid(True)

plt.tight_layout()
plt.show()

print(f"\nDigital Filter Numerator: {a}")
print(f"Digital Filter Denominator: {b}")
```

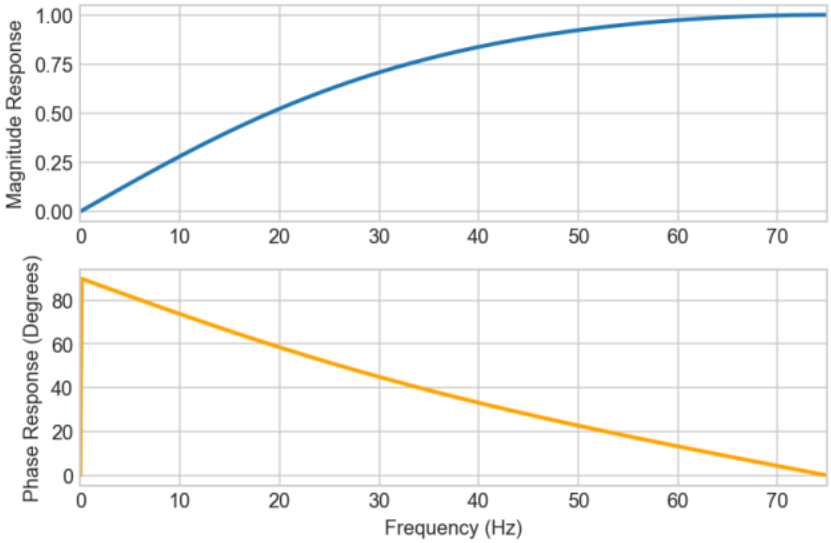


Figure 6.3: Magnitude and Phase response of the designed IIR filter

6.6.2. Example 2: Design a Butterworth Low-Pass Digital Filter using Bilinear Transformation

Design a Butterworth filter using bilinear transformation technique (BLT) that has a passband gain of 0 to $-3dB$ cut-off frequency of $2kHz$, and attenuation of at least $20dB$ for frequencies greater than $5kHz$. Assume the sampling frequency to be $20kHz$ and sampling period $T = 1s$.

To design the desired filter we use the following `butter()` function from the `scipy.signal` package.

```
butter(N, Wn, btype='lowpass', analog=False, output='ba', fs=None)
```

Here, N is the order of the filter. Wn is the cutoff frequency or frequencies. $btype$ is the type of filter. The options are *'lowpass', 'highpass', 'bandpass', 'bandstop'*. $analog$ - when *True*, returns an analog filter, otherwise a digital filter is returned. $output$ is type of output. The types are numerator/denominator (*'ba'*), pole-zero (*'zpk'*), or second-order sections (*'sos'*). Fs is the sampling frequency of the digital system.

The python code of the above desired Butterworth low-pass digital filter is given below. Figure 6.4 shows the frequency response of the desired filter.

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

```
from scipy import signal

F_sample = 20000  # Sampling frequency (Hz)
F_pass = 2000     # Passband edge frequency (2 kHz)
F_stop = 5000     # Stopband edge frequency (5 kHz)
Ap = 3            # Maximum passband attenuation (dB)
As = 20           # Minimum stopband attenuation (dB)
T = 1.0 / F_sample

omega_pa = (2.0/T)*np.tan(np.pi*F_pass/F_sample) # Pre-warped passband freq.
omega_sa = (2.0/T)*np.tan(np.pi*F_stop/F_sample) # Pre-warped stopband freq.

print(f"Sampling Frequency (Fs): {F_sample / 1000:.0f} kHz")
print(f"Pre-warped Analog Passband Omega_pa: {omega_pa:.2f} rad/s")
print(f"Pre-warped Analog Stopband Omega_sa: {omega_sa:.2f} rad/s")
print("-" * 30)

# --- Determine Analog Butterworth Filter Order (N) ---
N, omega_c=signal.buttord(omega_pa,omega_sa,Ap,As,analog=True)

print(f"Selected Filter Order (N): {N}")
print(f"Analog Cutoff Frequency (omega_c): {omega_c:.2f} rad/s")
print("-" * 30)
# --- Design the Digital Filter using Scipy (BLT is applied internally) ---
b, a = signal.butter(
        N,                  # Filter Order
        F_pass,             # Cutoff Frequency in Hz
        btype='lowpass',        # Low-pass filter
        analog=False,       # Design a digital filter
        output='ba',        # Return num (b) and den (a) coefficients
        fs=F_sample         # Provide sampling frequency for BLT pre-warping
        )

print("Digital Filter Coefficients:")
print(f"Numerator (b): {b}")
print(f"Denominator (a): {a}")
print("-" * 30)
# Calculate the frequency response of the digital filter
w, h = signal.freqz(b, a, fs=F_sample)

magnitude_db = 20 * np.log10(abs(h))    # Convert magnitude to dB
f_hz = w  # w is in Hz because we passed fs=F_sample to freqz

plt.plot(f_hz / 1000, magnitude_db)
plt.xlabel('Frequency (kHz)')
plt.ylabel('Magnitude (dB)')
plt.grid(linestyle='--')
plt.xlim(0, F_sample / 2000) # Plot up to Nyquist frequency (10 kHz)
plt.ylim(-40, 5)
# Passband edge (2 kHz, -3 dB)
plt.plot(F_pass/1000, -Ap, 'go',
        label=f'Passband Edge ({F_pass/1000:.1f} kHz, -{Ap} dB)')
plt.axvline(F_pass / 1000, color='g', linestyle='--')
```

```
plt.axhline(-Ap, color='g', linestyle='--')
# Stopband edge (5 kHz, -20 dB)
plt.plot(F_stop / 1000, -As, 'ro',
         label=f'Stopband Edge ({F_stop/1000:.1f} kHz, -{As} dB)')
plt.axvline(F_stop / 1000, color='r', linestyle='--')
plt.axhline(-As, color='r', linestyle='--')

plt.legend()
plt.show()
```

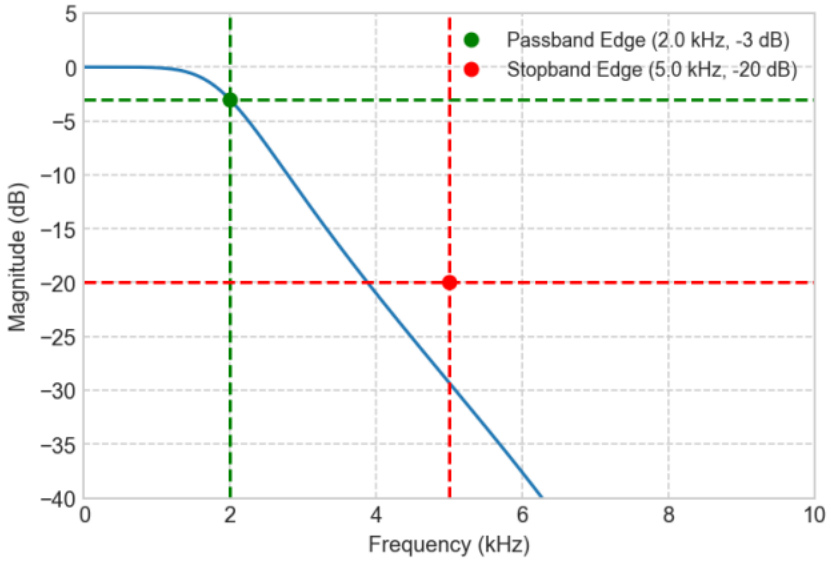


Figure 6.4: Magnitude and Phase response of the designed butterworth low-pass digital filter

The digital filter design process leverages multiple functions within the `scipy.signal` package. Initially, the filter order and analog cutoff frequency are determined using the `signal.buttord()` function, based on the required passband and stopband characteristics. Subsequently, the `signal.butter()` function is used to compute the numerator and denominator coefficients of the resulting analog filter. These analog coefficients are then transformed into the final digital filter coefficients by applying the bilinear transformation (BLT), using the `signal.bilinear()` function. Once the Python code is executed, the simulated frequency response, shown in Figure 6.4, confirms that the results align closely with the established theoretical design specifications.

6.6.3. Example 3: Design a Chebyshev Type II Digital Bandstop Filter using Bilinear Transformation

Design of a Chebyshev Type II digital bandstop filter for the following specifications:
(1) Stopband frequencies are from $1500Hz$ to $2500Hz$ with an attenuation of $40dB$.
(2) Passband frequencies are below $1000Hz$ and above $3000Hz$ with the passband ripple of $3dB$.
Consider the sampling frequency $Fs = 8kHz$. Use BLT method for transformation.
To design the desired filter we use the following `cheby2()` function from the `scipy.signal` package.

```
cheby2(N, rs, Wn, btype='low', analog=False, output='ba', fs=None)
```

Here, N is the order of the filter. rs is the minimum attenuation required in the stop band. Specified in decibels, as a positive number. Wn is the cutoff frequency/frequencies. $btype$ is the type of filter. The options are *'lowpass', 'highpass', 'bandpass', 'bandstop'*. $analog$ - when *True*, returns an analog filter, otherwise a digital filter is returned. $output$ is type of output. The types are numerator/denominator (*'ba'*), pole-zero (*'zpk'*), or second-order sections (*'sos'*). Fs is the sampling frequency of the digital system.

The python code of the above desired Chebyshev Type II Bandstop digital filter is given below. Figure 6.5 shows the frequency response of the desired filter.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

Fs = 8000       # Sampling frequency (Hz)
Ap = 3          # Maximum passband loss/ripple (dB)
As = 40         # Minimum stopband attenuation/ripple (dB)
# Passband edges (frequencies to KEEP)
Fp1 = 1000      # 1.0 kHz
Fp2 = 3000      # 3.0 kHz
# Stopband edges (frequencies to REJECT)
Fs1 = 1500      # 1.5 kHz
Fs2 = 2500      # 2.5 kHz

Nyquist = Fs / 2.0  # Normalize Frequencies for SciPy
Wp = [Fp1 / Nyquist, Fp2 / Nyquist] # Normalized Passband Edges (Wp)
Ws = [Fs1 / Nyquist, Fs2 / Nyquist] # Normalized Stopband Edges (Ws)

print(f"Sampling Frequency (Fs): {Fs/1000:.1f} kHz")
print(f"Passband Loss (Ap): {Ap} dB (Monotonic Passband)")
print(f"Stopband Attenuation (As): {As} dB (Equiripple Stopband)")
print(f"Normalized Passband Edges (Wp): [{Wp[0]:.3f}, {Wp[1]:.3f}]*Nyquist")
print(f"Normalized Stopband Edges (Ws): [{Ws[0]:.3f}, {Ws[1]:.3f}]*Nyquist")
print("-" * 40)

# --- Determine Filter Order (N) and Natural Frequencies (Wn)---
# It uses the passband loss (gpass) and the stopband attenuation (gstop).
N, Wn = signal.cheb2ord(
        wp=Wp,      # Passband normalized frequencies
        ws=Ws,      # Stopband normalized frequencies
        gpass=Ap,   # Maximum loss in the passband (Ap)
        gstop=As,   # Minimum attenuation in the stopband (As)
        analog=False # Design a digital filter
        )

print(f"Required Filter Order (N): {N}")
print(f"Chebyshev Natural Frequencies (Wn): \
        [{Wn[0]:.3f}, {Wn[1]:.3f}]*Nyquist")
print("-" * 40)

# --- Design the Digital Filter Coefficients using cheby2 ---
# cheby2 computes the filter coefficients (b, a).
# It uses the Stopband Attenuation (rs) to define the equiripple floor.
b, a = signal.cheby2(
        N=N,        # Filter Order
        rs=As,      # Stopband ripple/rejection (As)
        Wn=Wn,      # Critical frequencies
        btype='bandstop',
        analog=False, # Design a digital filter
        output='ba'   # Return numerator (b) and denominator (a) coefficients
```

```
        )

print("Digital Filter Coefficients:")
print(f"Numerator (b): {b}")
print(f"Denominator (a): {a}")
print("-" * 40)
# Calculate the frequency response
w, h = signal.freqz(b, a, fs=Fs)

# Convert magnitude to dB
magnitude_db = 20 * np.log10(np.abs(h))
frequencies_hz = w

plt.figure(figsize=(10, 6))
plt.plot(frequencies_hz / 1000, magnitude_db,
        label='Chebyshev Type II Response (Equiripple Stopband)')
plt.xlabel('Frequency (kHz)')
plt.ylabel('Magnitude (dB)')
plt.grid(which='both', axis='both', linestyle='--')

plt.xlim(0, Nyquist / 1000) # Set the x-axis limit up to the Nyquist frequency
plt.ylim(-60, 5) # Set a lower limit to clearly show 40 dB attenuation

# Highlight Passband Edge (Ap)
# The response must be at least -Ap at the passband edge.
plt.axhline(-Ap, color='g', linestyle='--',
        label=f'Max Passband Loss (-{Ap} dB)')

# Highlight Stopband Attenuation (As) - this is the equiripple floor
plt.axhline(-As, color='r', linestyle='--',
        label=f'Equiripple Stopband Floor (-{As} dB)')
plt.fill_between([Fs1 / 1000, Fs2 / 1000], -60, -As, color='pink',
        label='Equiripple Stopband Region')

# Highlight Passband Edges
plt.axvline(Fp1 / 1000, color='b', linestyle=':')
plt.axvline(Fp2 / 1000, color='b', linestyle=':', label='Passband Edges')

plt.legend()
plt.show()
```

Figure 6.5 confirms that the code successfully designed a Chebyshev Type II Bandstop Filter exactly according to the specifications. As expected for a Type II design, the frequency response exhibits a monotonic and flat passband outside the rejection zone (below $1.0kHz$ and above $3.0kHz$), ensuring that the desired frequencies pass through with minimal variation and meeting the $3dB$ passband loss criteria. Crucially, the rejection zone between $1.5kHz$ and $2.5kHz$ clearly demonstrates the characteristic equiripple stopband, where the magnitude response repeatedly touches the dashed red line at $-40dB$, thereby strictly enforcing the minimum stopband attenuation requirement. The filter successfully isolates the band of frequencies intended for removal, confirming that the code was executed and visualized correctly, achieving a smoother passband at the expense of ripple in the stopband.

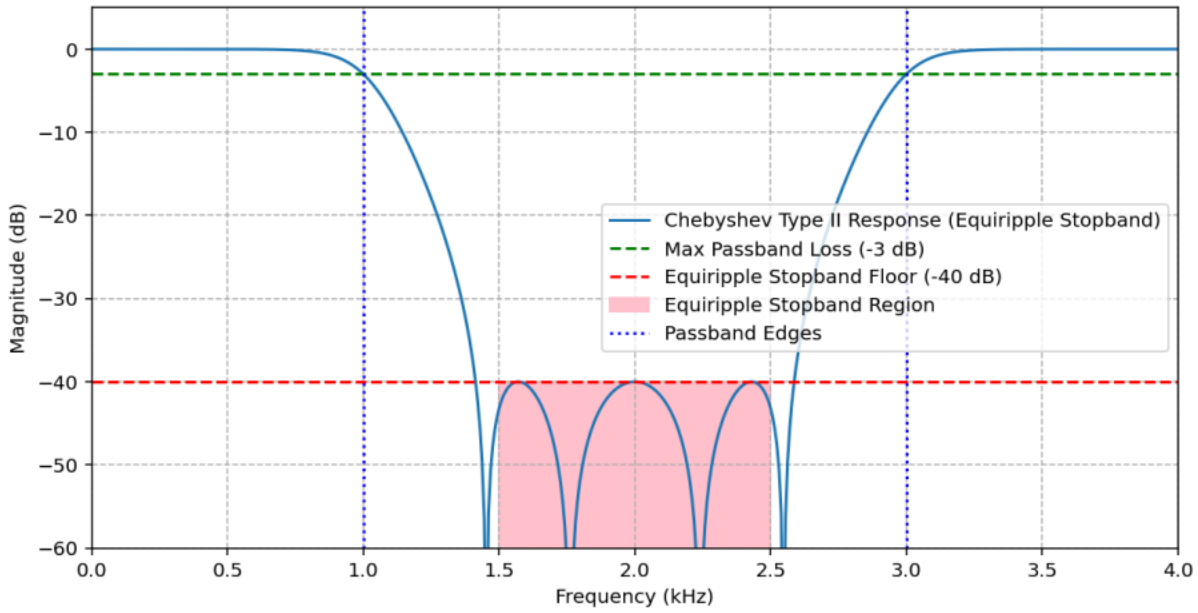


Figure 6.5: Chebyshev Type II Bandstop Digital Filter

Post-laboratory Problems

1. Design a notch filter using pole-zero placement method with following specifications:
   — notch frequency $50Hz$
   — $3dB$ width of the notch ± 5 Hz
   — sampling frequency $500Hz$
   Show the magnitude and phase response of the filter. Consider a signal, $x(t) = 2\cos(60\pi t) + \cos(100\pi t)$. If $x[n]$ is the signal obtained from $x(t)$ by sampling it at a frequency $500Hz$, show the output signal after x[n] is passed through your designed notch filter.
2. Design a bandstop filter with the following property using pole-zero placement method.
   — Center frequency, $\Omega_0 = \dfrac{\pi}{10}$ (complete attenuation)
   — Bandstop width, $\Omega_w = 2\Omega_{\text{cf}} = \dfrac{\pi}{20}$
   Show the frequency response of the filter and mention the filter coefficients.
3. A discrete bandpass filter with following specifications is to be designed using bilinear transformation from a low pass filter prototype.
   — passband, $200 - 300Hz$
   — sampling frequency, $2KHz$
   — filter order, $N = 2$
   The transfer function of prototype low pass filter is $H(s) = \dfrac{1}{s+1}$. Determine the filter coefficients and show the frequency response of the filter.
4. Write a Python code to design a Butterworth digital bandpass filter for the following specifications:
   — Passband frequencies are $1500Hz$ and $2500Hz$.
   — Stopband frequencies are $1000Hz$ and $3000Hz$.
   — Sampling frequency $Fs = 8kHz$.
   — Passband ripple is $1dB$ and stopband attenuation is 30 dB.
   — Use BLT method for transformation.
5. Design of a Chebyshev Type I digital highpass filter for the following specifications:
   — Passband frequency is $2500Hz$.
   — Stopband frequency is $1500Hz$.
   — Sampling frequency $Fs = 8kHz$.
   — Passband ripple is $3dB$ and stop attenuation is $40dB$.
   — Use BLT method for transformation.

6. Design a Chebyshev Type II digital bandpass filter for the following specifications:
   — Passband frequencies are from $1500 - 2500Hz$ with an attenuation of $3dB$.
   — Stopband frequencies are below $1000Hz$ and above $3000Hz$ with the passband ripple of $40dB$.
   — Sampling frequency $Fs = 8kHz$.
   — Use BLT method for transformation.
7. Write a Python code to obtain the magnitude and phase response of Chebyshev Type I bandpass filter with the passband frequencies of $1500 - 3000Hz$. Choose the order of the filter that is 3.
8. Write a Python code to obtain the magnitude and phase response of Chebyshev Type I band reject filter with the stopband frequencies of $1500 - 3000Hz$. Choose the order of the filter that is 3.
9. Solve all the exercise problem of the textbook from page 373 to page 374.

Laboratory 7

Application of Signal Processing (Part - 1)

7.1. Objective

The objective of this laboratory is to provide students with essential practical exposure to real-world Digital Signal Processing (DSP) applications by implementing three fundamental case studies: Separation of Signals Using FIR Filter, which demonstrates isolating frequency components; Denoising of the Signal Using FIR Filter, which focuses on restoring signal quality by removing unwanted noise; and Voice/Silence Detection (Simple Feature Extraction), which teaches signal classification using simple metrics like energy, thereby bridging the gap between theoretical DSP concepts and core industry applications.

7.2. Case study 1: Separation of Signals Using FIR Filter

The signal $x(t)$ is an addition of three signals $x_1(t)$, $x_2(t)$, and $x_3(t)$. The frequencies of the three signals $x_1(t)$, $x_2(t)$, and $x_3(t)$ are $5Hz$, $10Hz$ and $15Hz$, respectively. The signal $x(t)$ is now passed through a lowpass filter whose cut-off frequency is $7Hz$, the order of the filter is 50 and the window chosen is *Hanning*. The problem statement is depicted in the form of block diagram in the Figure 7.1.

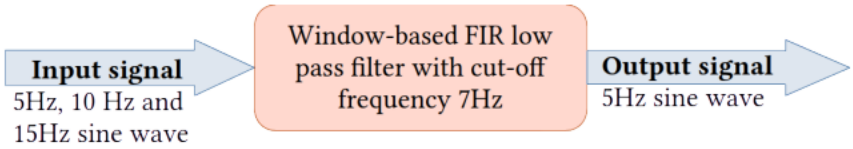


Figure 7.1: Block diagram for case study 1

The whole process of frequency separation can be divided into six following steps.

1. Defining the parameters of sine wave
2. Defining the sampling frequency and number of points in FFT
3. Generation of sine wave
4. Design of FIR LPF
5. Obtaining the spectrum of input and output signal
6. Plotting the results

From the above steps we need to generate an input signal which is the sum of three sine wave frequencies, $5Hz$, $10Hz$ and $15Hz$. A window-based FIR filter has to be designed with a cut-off frequency of $7Hz$, and the number of taps is 51. The window chosen is *Hanning*. The sampling frequency chosen is $100Hz$. The built-in function `firwin`, which is available in `scipy.signal` package, is used to design the filter. The python code, which performs lowpass filtering of the input sine wave with $5Hz$, $10Hz$ and $15Hz$ frequency components, is given below, and the corresponding output is in Figure 7.2.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
from scipy.fftpack import fft,fftfreq
#Step 1: Defining the parameters of sine wave
f1,f2,f3,ph = 5,10,15,0 #Frequency of signal 1 and 2 and phase
```

```
#Step 2: Defining the sampling frequency and number of points in FFT
Fs, N = 100, 256 #Sampling frequency
T = 1/Fs #Sampling period
#Step 3: Generation of sine wave
t = np.linspace(0,N*T,N)
x1 = np.sin(2*np.pi*f1*t+ph)
x2 = np.sin(2*np.pi*f2*t+ph)
x3 = np.sin(2*np.pi*f3*t+ph)
x = x1+x2+x3
#Step 4: Design of LPF
f_cut = 7 # Cut-off frequency
Nyquist_freq = Fs/2
numtaps = 51
cutoff = f_cut/Nyquist_freq
h = signal.firwin(numtaps, cutoff, window='hann', pass_zero='lowpass')
y = signal.lfilter(h,1,x) # Filtering of the signal
#Step 5: Obtaining the spectrum of input and output signal
f_axis = fftfreq(N,T)[0:N//2]
X = fft(x)
Y = fft(y)
#Step 6: Plotting the results
plt.subplot(2,2,1)
plt.plot(t,x)
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('Input Signal')
plt.subplot(2,2,2)
plt.plot(t,y)
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('Filtered Signal')
plt.subplot(2,2,3)
plt.plot(f_axis,2/N*np.abs(X[0:N//2]))
plt.xlabel('Frequency')
plt.ylabel('Magnitude')
plt.title('Spectrum of input Signal')
plt.axvline(f_cut, color='red', linestyle='--', label=f"{f_cut}Hz")
plt.legend()
plt.subplot(2,2,4)
plt.plot(f_axis,2/N*np.abs(Y[0:N//2]))
plt.xlabel('Frequency')
plt.ylabel('Magnitude')
plt.title('Spectrum of filtered Signal')
plt.axvline(f_cut, color='red', linestyle='--', label=f"{f_cut}Hz")
plt.legend()
plt.tight_layout()
```

Based on Figure 7.2, the input signal is clearly a combination of three sine waves at $5Hz$, $10Hz$, and $15Hz$, as evidenced by the corresponding peaks in its spectrum. The resulting output signal is a low-pass filtered version of this input, which successfully retains only the $5Hz$ sine wave component. This is confirmed by the output signal's spectrum, which shows a single, prominent peak at $5Hz$, demonstrating that the filter effectively allowed the $5Hz$ component to pass while blocking the higher frequency components at $10Hz$ and $15Hz$.

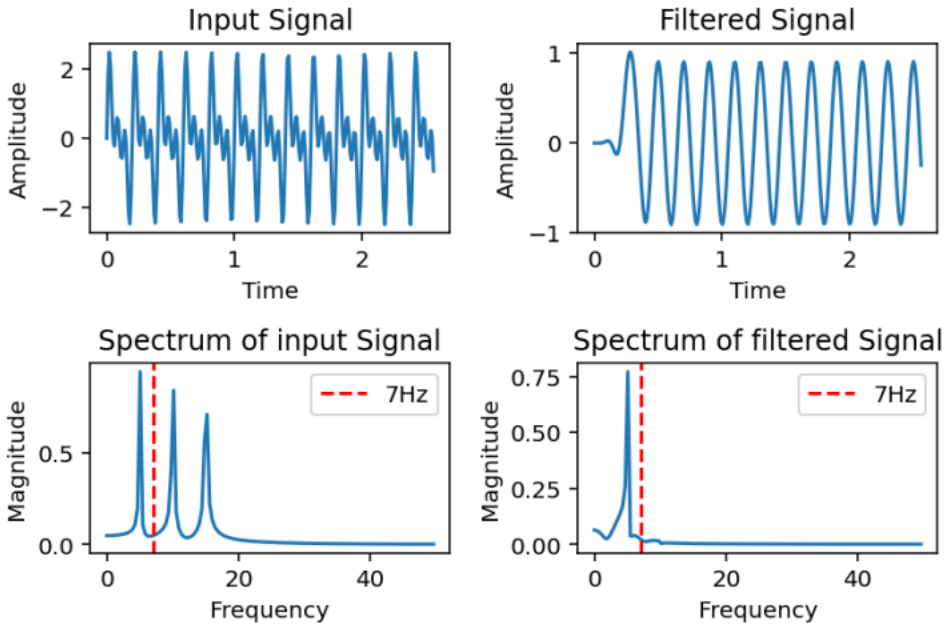


Figure 7.2: Signal separation using FIR filter

7.3. Case Study 2: Denoising of the Signal Using FIR Filter

The signal $x(t)$ is a $5Hz$ sine wave. This signal $x(t)$ is corrupted by white noise, which follows uniform distribution in the range $[0, 1]$. The noisy signal is then passed through FIR lowpass filter. The FIR filter coefficients are generated using the windowing technique. Plot results of the clean, noisy and filtered signals.

The above mentioned process can be divided into following five steps.

1. Generation of clean signal
2. Generation of uniform random noise
3. Generation of noisy sine wave
4. Design of FIR LPF
5. Plotting the results

The python code which performs the above-mentioned task is shown below. The corresponding output is shown in Figure 7.3.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
#Step 1: Generation of clean signal
t=np.linspace(0,1,100)
x=np.sin(2*np.pi*5*t)
#Step 2: Uniform random noise
n=np.random.uniform(0,1,100)
#Step 3: Noisy sine wave
x1=x+n
#Step 4: Design of FIR LPF
numtaps = 21
cutoff = 0.2 # Normalized cutoff frequency
h=signal.firwin(numtaps, cutoff, window='bartlett', pass_zero='lowpass')
y=signal.filtfilt(h,1,x1)
#Step 5: Plotting the result
plt.subplot(3,1,1)
```

```
plt.plot(t,x)
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('Clean signal')
plt.subplot(3,1,2)
plt.plot(t,x1)
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('Noisy signal')
plt.subplot(3,1,3)
plt.plot(t,y)
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('Filtered signal')
plt.tight_layout()
```

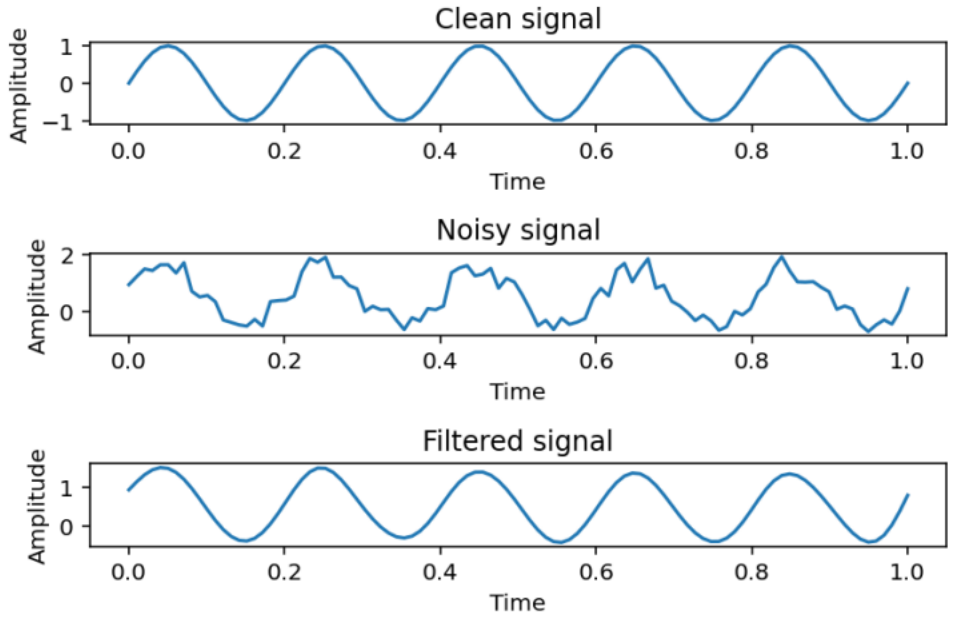


Figure 7.3: Signal denoising using FIR filter

From the above code, the built-in function `random.uniform()` available in the `numpy` library is used here to generate uniformly distributed random noise in the interval 0 to 1. The random noise is added to pure the sine wave to create noisy sine wave. The noisy sine wave is then filtered using FIR filter, obtained using the built-in function `firwin()` available in the `scipy.signal` package.

The normalized cutoff frequency is used here. The value is normalized to the Nyquist frequency ($\frac{F_s}{2}$). Since the Nyquist frequency is the highest frequency that can be represented, a cutoff of 0.2 means the actual cutoff frequency in Hertz is $f_{cutoff} = 0.2 \times \frac{F_s}{2}$. Since the default value of fs in `firwin()` is 2, the cutoff frequency is therefore considered 0.2.

`filtfilt()` function applies the designed filter coefficients (h) to the noisy signal $(x1)$ to produce the final filtered output, y. this `filtfilt()` function performs zero-phase forward and backward filtering. It applies the filter once in the forward direction and then again on the reversed output in the backward direction. By filtering both forward and backward, it ensures the final result has zero phase delay. This is crucial for signal analysis, as it prevents the filtered sine wave from being shifted in time relative to the original clean signal.

From Figure 7.3, we can clearly observe the three stages of signal processing. First, the clean input signal is identified as a smooth sine wave oscillating at $5Hz$. Second, this signal is intentionally corrupted by random noise, resulting in a noisy sine wave where the pure 5 Hz pattern is obscured by high-frequency disturbances. Finally, by passing this distorted data through the low-pass filter, the resulting filtered signal successfully minimizes the impact of the random noise, clearly restoring the original $5Hz$ sine wave pattern.

7.4. Case Study 3: Voice/Silence Detection (Simple Feature Extraction)

Distinguish between periods of spoken speech and periods of silence in an audio recording. This is the first step in many real-world applications like voice activity detection (*VAD*) and audio compression. Write a Python script to load a short audio segment (alternatively, you can generate a simulated signal mixing speech and silence for simplicity), calculate two key features over short time segments (frames), and use a simple threshold to classify each frame as *Speech* or *Silence*.

You need to used the short term energy (STE). It is the time-domain equivalent of Magnitude in the frequency domain, to determine the presence of a signal component.

7.4.1. Feature Extraction vs. Digital Filtering in VAD

We rely on Feature Extraction, specifically the Short-Term Energy (STE), for Voice Activity Detection (VAD) because VAD is fundamentally a classification problem, not a modification problem. A digital filter's primary job is to clean or modify the frequency content of a signal (like denoising or isolating a specific band), resulting only in a cleaner version of the original audio. It does not provide the binary information needed for detection. In contrast, STE calculation directly converts a segment of audio into a single, scalar energy value ($E[m]$). By comparing this concise feature against a simple threshold (E_{th}), we can immediately and efficiently make the required binary decision: whether the current frame contains speech or silence. Therefore, Feature Extraction is the most direct and computationally efficient method to achieve the detection objective.

7.4.2. Methodology

The solution can be divided into four main steps: parameter definition, feature calculation (framing), decision-making (thresholding), and visualization.

Step 1: Defining Parameters and Signal Generation

This initial step establishes the discrete signal environment and creates the test data required for analysis.

i **Define Sampling Parameters:**
   — Sampling Frequency (Fs): typically $8000Hz$ for narrowband speech.
   — Total Duration (Td): e.g., $3s$.
   — Total Number of Samples (N): $N = Fs.Td$
   — Sampling Period (T): $T = \frac{1}{Fs}$

ii **Generate Time Axis (t):** The discrete time vector t runs from 0 to T_d with N points.

iii **Generate the Test Signal ($x[n]$):** The input signal $x[n]$ is constructed by concatenating segments of silence and simulated speech.

$$x[n] = \text{Silence}(0 \text{ s to } 1 \text{ s}) + \text{Speech}(1 \text{ s to } 2 \text{ s}) + \text{Silence}(2 \text{ s to } 3 \text{ s})$$

The speech segment can be approximated by a sine wave of frequency F_{speech} (e.g., 400 Hz) plus a small amount of background noise $noise[n]$:

$$x_{\text{speech}}[n] = A \cdot \sin\left(2\pi F_{\text{speech}} \cdot nT\right) + noise[n]$$

Step 2: Signal Framing and Feature Calculation

To analyze non-stationary signals like speech, the signal must be divided into short, overlapping or non-overlapping frames.

i **Define Frame Size (L):** The frame length L in samples (e.g., 160 samples for $20ms$ at $8kHz$) determines the resolution of the VAD. The total number of frames is $M = \lfloor \frac{N}{L} \rfloor$.

ii **Short-Term Energy (STE) Calculation:** For each frame m (where $m = 0, 1, \ldots, M-1$), the energy $E[m]$ is calculated as the sum of the squares of the samples within that frame $x_{\text{frame}}[n]$. The index n runs across the L samples of the current frame.

$$E[m] = \sum_{n=0}^{L-1} \left(x_{\text{frame}}[n]\right)^2$$

iii **Storage:** The calculated energy values are stored in the $\boldsymbol{E}$ array.

Step 3: Thresholding for Classification

This step converts the continuous feature magnitude (E) into a binary decision (VAD).

i **Define Energy Threshold (E_{th}):** A fixed threshold is selected to discriminate between the high energy of speech and the low energy of silence or noise. This is typically set empirically or as a fraction of the maximum observed energy.

ii **Classification (VAD Decision):** The Voice Activity Detection result, $VAD[m]$, is a binary sequence where 1 indicates speech and 0 indicates silence.

$$VAD[m] = \begin{cases} 1 & \text{if } E[m] > E_{th} \quad \text{(Speech)} \\ 0 & \text{if } E[m] \leq E_{th} \quad \text{(Silence)} \end{cases}$$

Step 4: Plotting the Results

Visualization is essential for validating the algorithm's performance.

i **Plot 1: Original Signal (x):** Display $x[n]$ vs. nT to show the input signal in the time domain.

ii **Plot 2: Energy Feature (E):** Display the calculated Short Term Energy $E[m]$ vs. $time$. This plot includes the decision threshold E_{th} as a horizontal reference line.

iii **Plot 3: Final VAD Decision (VAD):** Display the binary $VAD[m]$ result vs. $time$, often plotted as a step function, to clearly show the segments classified as active speech.

The python code which performs the above-mentioned task is shown below. The corresponding output is shown in Figure 7.4.

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

# --- Step 1: Defining Parameters and Signal Generation ---
# 1. Define Sampling Parameters
FS = 8000           # Sampling Frequency (samples/sec), common for speech
DURATION = 3        # Total signal duration in seconds
N = FS * DURATION   # Total number of samples
# 2. Generate Time Axis (t)
t = np.linspace(0, DURATION, N)
# 3. Generate the Test Signal (x)
# Create a 3-second signal: 1s silence, 1s speech, 1s silence
T_SILENCE = 1
T_SPEECH = 2
# Silence Segment 1 (0s to 1s)
```

```
silence1 = np.zeros(FS * T_SILENCE)
# Speech Segment (1s to 2s) - Simulated as a 400Hz sine wave
f_speech = 400
speech_t = np.linspace(0, T_SPEECH - T_SILENCE, FS * (T_SPEECH - T_SILENCE))
speech = 0.8 * np.sin(2 * np.pi * f_speech * speech_t)
# Silence Segment 2 (2s to 3s)
silence2 = np.zeros(FS * (DURATION - T_SPEECH))
# Combine the segments into the final input signal (x)
x = np.concatenate((silence1, speech, silence2))
x += np.random.normal(0, 0.03, N) # Add slight background noise
# --- Step 2: Signal Framing and Feature Calculation ---
# 1. Define Frame Size (20 ms frame)
FRAME_MS = 20
FRAME_LEN = int(FS * FRAME_MS / 1000) # e.g., 8000 * 0.02 = 160 samples
# 2. Initialize STE Array and iterate
frames = np.array([x[i:i + FRAME_LEN] \
        for i in range(0, N - FRAME_LEN + 1, FRAME_LEN)])
M = frames.shape[0] # Number of frames
# Calculate Short-Term Energy (STE) for each frame
# E[m] = sum(x_frame[n]^2)
E = np.sum(frames**2, axis=1)
# Generate time vector for the STE plot (aligned to the center of each frame)
time_E = np.arange(M) * FRAME_LEN / FS + (FRAME_LEN / 2) / FS
# --- Step 3: Thresholding for Classification (VAD) ---
# 1. Define Energy Threshold
E_TH = 0.5 * np.max(E) # Threshold: 50% of the maximum observed energy
# 2. Classification
VAD = np.where(E > E_TH, 1, 0) # 1 for Speech, 0 for Silence
# --- Step 4: Plotting the Results ---
plt.figure(figsize=(10, 8))
# Plot 1: Original Input Signal
plt.subplot(3, 1, 1)
plt.plot(t, x, label='Input Signal (Simulated Speech)')
plt.title('1. Input Signal (Time Domain)')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.grid(True, linestyle='--')
# Plot 2: Short-Term Energy (STE) Feature
plt.subplot(3, 1, 2)
plt.plot(time_E, E, label='Short-Term Energy (STE)', color='orange')
plt.axhline(E_TH, color='r', linestyle='--', label=f'Threshold: {E_TH:.4f}')
plt.title('2. Short-Term Energy Feature')
plt.xlabel('Time (s)')
plt.ylabel('Energy')
plt.legend()
plt.grid(True, linestyle='--')
# Plot 3: Final VAD Decision
plt.subplot(3, 1, 3)
# The VAD decision is plotted as a step function over the signal's time axis
# Expand VAD vector to match the original sample length for cleaner plotting
VAD_expanded = np.repeat(VAD, FRAME_LEN)[:N]
# Plot the VAD decision line
plt.plot(t, VAD_expanded, label='VAD Decision (1=Speech, 0=Silence)')
```

```
plt.title('3. Voice Activity Detection (VAD) Result')
plt.xlabel('Time (s)')
plt.ylabel('VAD State')
plt.yticks([0, 1], ['Silence', 'Speech'])
plt.axvline(1.0, color='r', linestyle=':', label='Actual Speech Start/End')
plt.axvline(2.0, color='r', linestyle=':')
plt.legend()
plt.ylim(-0.3, 1.3)
plt.grid(axis='y', linestyle='--')
plt.tight_layout()
plt.show()
```

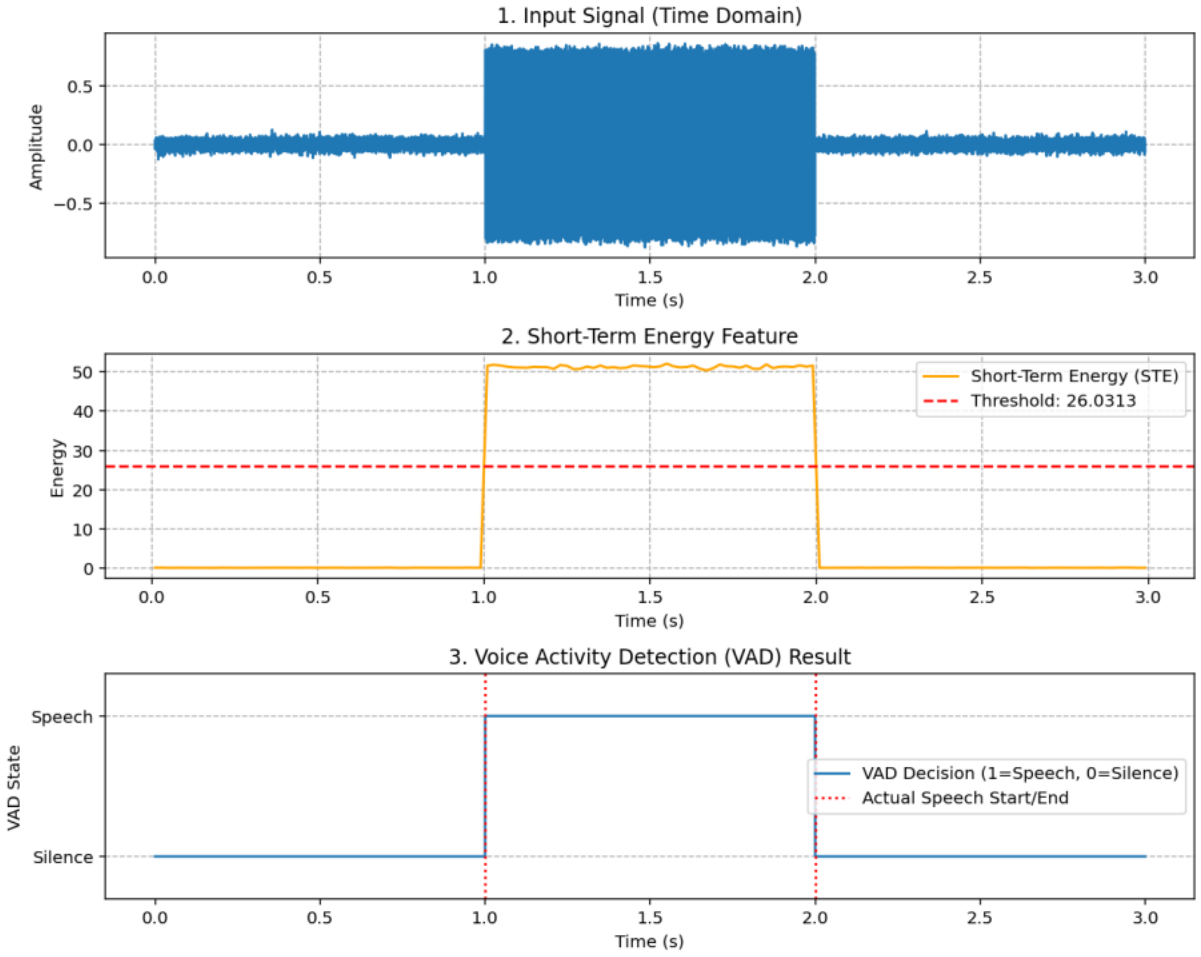


Figure 7.4: Voice/silence detection using feature extraction

Figure 7.4 confirms that the Voice Activity Detection (VAD) method successfully achieved its goal by utilizing the Short-Term Energy (STE) feature. The feature extraction process accurately mapped the high-amplitude speech segment (between $1.0s$ and $2.0s$) to a high STE value and the surrounding silent segments to near-zero STE values. By applying a single threshold (at 26.0313), the VAD algorithm was able to perfectly segment the audio, producing a binary VAD Decision (1 =Speech, 0 =Silence) that precisely matches the actual start and end times of the speech input.

Post-laboratory Problems

1. Repeat *Case Study 3* with an external real life audio signal, instead of simulated signal. In addition, take a longer duratiion for the input signal instead of $3sec$.
2. **Pitch Estimation and Voiced/Unvoiced Classification**
   Accurately determine the fundamental frequency (*Pitch*) of a speech signal and classify each frame as *voiced* (periodic excitation from vocal cords, e.g., vowels) or *unvoiced* (random excitation, e.g., s, f). For this problem, focus on the following aspects.
   — Utilize *Autocorrelation* or *Average Magnitude Difference Function (AMDF)* to estimate the pitch period.
   — Use the *Zero-Crossing Rate (ZCR)* to distinguish between the segments: voiced segments typically exhibit low ZCR, while unvoiced segments show high ZCR.
3. **Formant Tracking for Vowel Recognition using LPC**
   Identify the resonant frequencies of the vocal tract, known as *Formants* ($F1$, $F2$, $F3$) which are critical features for distinguishing different vowel sounds (e.g., i vs. a). For this problem, focus on the following aspects.
   — Apply *Linear Predictive Coding (LPC)*. LPC is used to model the human vocal tract as a time-varying filter, and the roots of the resulting prediction polynomial can be solved to locate the resonant peaks (formants) in the short-time magnitude spectrum.
   — The coefficients a_k of the LPC filter $A(z)$ are derived by minimizing the prediction error:

$$A(z) = 1 - \sum_{k=1}^{p} a_k z^{-k}$$

   where p is the order of the filter.
4. **Speech Enhancement Using Adaptive Noise Reduction Filters**
   Design and implement a system to remove or substantially reduce constant or slowly varying background noise (e.g., fan hum) from a speech recording to improve clarity and the *Signal-to-Noise Ratio (SNR)*. For this problem, focus on the following aspects.
   — Implement an *Adaptive Filter*, such as the *Least Mean Squares (LMS)* algorithm, where the filter coefficients are continuously adjusted to estimate the noise component and subtract it from the noisy input.
   — Utilize *Spectral Subtraction*, which involves estimating the noise power spectrum during non-speech periods and subtracting this estimate from the power spectrum of the noisy speech frame-by-frame.
5. **Speech Feature Extraction for Speaker Identification (MFCCs)**
   Extract robust, speaker-specific features that characterize the unique shape and movement of an individual's vocal tract, which can be used for biometric speaker identification or verification. For this problem, focus on the following aspects.
   — Compute *Mel-Frequency Cepstral Coefficients (MFCCs)*. The process involves transforming the power spectrum to the non-linear *Mel scale* (which aligns better with human perception of pitch), taking the logarithm, and finally applying a Discrete Cosine Transform (*DCT*) to decorrelate the filterbank outputs.
   — The critical step is the conversion from Hertz (f) to the Mel scale (M):

$$M = 2595 \log_{10}\left(1 + \frac{f}{700}\right)$$

6. **Time-Scale Modification without Pitch Distortion**
   Change the playback speed or duration of a speech segment (making it faster or slower) without altering the speaker's *pitch* or *formant structure*, preventing the chipmunk or giant effect. For this problem, focus on the following aspects.
   — Implement the *Pitch-Synchronous Overlap-Add (PSOLA)* algorithm. This method requires accurate *pitch marking* of the signal, where short segments are cut at the beginning of each pitch period and then resynthesized by overlapping and adding these segments at new, modified time intervals.

— Maintaining the phase coherence of the overlapping windows is essential to avoid undesirable artifacts and preserve the speech quality.

7. **Industrial Monitoring: Machine Fault Detection using RMS Energy**
In predictive maintenance for industrial machinery, a sudden increase in vibration or noise often signals an impending mechanical failure (e.g., a damaged bearing or imbalance). This change can be reliably tracked using the *Root Mean Square (RMS) Energy* feature, which measures the signal's power.
— Define the formula for the RMS of a discrete signal $x[n]$ over a short frame of length L. Relate the RMS to the Short-Term Energy (STE) feature used in Voice Activity Detection.
— Suppose a machine's vibration signal is processed using a sliding window. Explain how you would select a dynamic or adaptive threshold (rather than a fixed one) on the RMS feature to reliably detect the moment a machine begins to fail, regardless of its normal operational load.
— Contrast the effect of an *RMS threshold* (detecting an increase in overall vibration intensity) versus a *High-Pass Filter* (designed to remove constant background noise) in a practical industrial setting. Which method is better for detecting a transient, high-amplitude fault, and why?

8. **Biomedical Signal Analysis: Arrhythmia Detection using Zero-Crossing Rate (ZCR)**
The *Zero-Crossing Rate (ZCR)* is a measure of the rate at which the sign of a signal changes. It is an effective feature for differentiating between low-frequency and high-frequency signals. In ECG analysis, certain chaotic arrhythmias (e.g., Ventricular Fibrillation) cause the signal to become highly noisy and high-frequency.
— Write the mathematical definition of the Zero-Crossing Rate for a discrete signal $x[n]$ over a frame L.
— Assume a patient's ECG signal (which is typically low-frequency) transitions suddenly into Ventricular Fibrillation (a chaotic, high-frequency signal). Describe the expected change (increase or decrease) in the ZCR feature during this transition.
— Design a simple, fixed-length thresholding algorithm based on the ZCR feature that could be used in a defibrillator device to automatically flag a severe, high-frequency arrhythmia event.

Laboratory 8

Application of Signal Processing (Part - 2)

8.1. Objective

This laboratory experiment is the continuation of the previous experiment (Laboratory 7). The objective of this laboratory is to bridge the gap between theoretical DSP concepts and practical engineering applications. Students will achieve this by engaging with essential real-life signal processing problems, including filtering artifacts from Electrocardiogram (ECG) data, eliminating common $50Hz$ power line interference, and enhancing image quality through digital filtering. The expected outcome is for students to possess a robust understanding of the practical utility and deployment of various DSP algorithms.

8.2. Case study 4: ECG Baseline Wander Removal (High-Pass Filtering)

The *Electrocardiogram (ECG)* is a vital, non-invasive medical signal that measures the electrical activity of the heart over time, serving as a critical diagnostic tool for cardiac health. *Baseline Wander* is a low-frequency noise artifact that corrupts the ECG signal, typically appearing as a slow, large, undulating drift in the baseline (the zero-voltage reference) caused by factors like patient movement or poor electrode contact. This drift obscures the essential, higher-frequency components of the heart complex (P, QRS, T waves), making clinical interpretation or automated analysis difficult. *ECG Baseline Wander Removal* is thus a necessary signal processing step, which is efficiently accomplished by applying a Digital High-Pass Filter (HPF), typically with a very low cutoff frequency (e.g., $0.5Hz$), to block the slow wander while allowing the diagnostically important ECG components to pass through unaltered, thereby restoring a stable, accurate baseline. Thus, ECG Baseline Wander Removal problem is a suitable example for applying digital filtering to a real-world biomedical signal, directly using the denoising principles seen in the previous case studies.

In this process you will remove low-frequency baseline wander ($\approx 0.1Hz$) while preserving high-frequency heart complexes. To do this you will use IIR Digital High-Pass Filter to block low frequencies but passes high frequencies, directly targeting the low-frequency wander. You will also use the Zero-Phase Filtering (`filtfilt`) to ensure that, the filtered ECG is not shifted in time relative to the original clean signal.

The above ECG Baseline Wander Removal process can be divided into following four steps.

1. **Step 1: Defining Parameters and Signal Generation**
   This step prepares the clean signal and adds the specific noise component.
   i **Define Sampling Parameters:**
      — Choose a Sampling Frequency (F_s) suitable for ECG (e.g., 1000 Hz).
      — Define the Total Duration (T_d).
   ii **Generate Clean ECG Signal** (x): Simulate a $1Hz$ rhythmic signal (like the heart rate) with sharp peaks (to represent the QRS complex, the most important feature) with the help of `ecg_simulate()` function from the `neurokit2` library. This is the desired, high-frequency component.
   iii **Generate Baseline Wander Noise** (n): Generate a very slow, low-frequency sine wave (e.g., $0.2Hz$) with a high amplitude. This simulates the corruption caused by patient movement.
   iv **Generate Noisy Signal** (x_{noisy}): Combine the clean ECG and the wander noise:

$$x_{\text{noisy}}[n] = x_{clean}[n] + n_{wander}[n]$$

2. **Step 2: High-Pass Filter Design (IIR Butterworth)**
   This step converts the frequency specifications into the filter coefficients.
   i **Define Filter Specifications:**

— **Cutoff Frequency** (F_{cutoff}): Choose a value slightly above the noise but below the signal (e.g., 0.5 Hz).
— **Order** (P): Choose a low order (e.g., $P = 4$) for sharp cutoff and minimal ringing.
— **Filter Type**: High-Pass Filter (HPF).

ii **Normalize Cutoff Frequency:** Normalize the analog cutoff frequency to the Nyquist frequency ($F_s/2$).

$$\omega_{\text{norm}} = \frac{F_{\text{cutoff}}}{F_s/2}$$

iii **Design IIR Filter:** Use a digital filter design function (such as `scipy.signal.butter` in Python) to calculate the numerator (B) and denominator (A) coefficients of the IIR High-Pass filter transfer function $H(z)$:

$$H(z) = \frac{\sum_{k=0}^{P} B_k z^{-k}}{1 + \sum_{k=1}^{P} A_k z^{-k}}$$

3. **Step 3: Zero-Phase Filtering**
This step applies the designed filter to remove the noise without introducing time shift.
i Use the `filtfilt()` function from `scipy.signal` package to apply the filter coefficients (B, A) to the noisy signal (x_{noisy}), yielding the final filtered signal, y.

```
y = filtfilt(B, A, x_noisy)
```

ii `filtfilt()` performs the filtering forward and then backward on the reversed signal, ensuring the final output (y) has zero phase delay.

4. **Step 4: Plotting the Results**
Visualization is key to confirming that the baseline wander has been successfully removed.
i Plot the original, ideal ECG signal.
ii Plot the corrupted signal, showing the ECG riding on top of the slow baseline wander.
iii Plot the final output. This plot should clearly show the restored ECG signal with the baseline flattened back to zero.

The python code which performs the above-mentioned task is shown below. The corresponding output is shown in Figure 8.1.

```python
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt
import neurokit2 as nk

# --- Step 1: Defining Parameters and Signal Generation ---
Fs = 1000  # Samples per second
Td = 10       # Seconds
x_clean = nk.ecg_simulate(duration=Td, sampling_rate=Fs, method="ecgsyn")
t = np.arange(0, Td, 1/Fs)
# 3. Generate Baseline Wander Noise (Very Low-Frequency Component)
# Baseline wander is typically 0.05 Hz to 0.5 Hz.
# We'll use 0.2 Hz with high amplitude.
f_wander = 0.2
n_wander = 1 * np.sin(2 * np.pi * f_wander * t)
# 4. Generate Noisy Signal
x_noisy = x_clean + n_wander
# --- Step 2: High-Pass Filter Design (IIR Butterworth) ---
# Define Filter Specifications
f_cutoff = 0.5      # (must be between 0.1 Hz noise and 1 Hz signal)
P = 4               # Filter order (4th order provides a good balance)
# Normalized Cutoff Frequency (normalized to Nyquist frequency Fs/2)
```

```
Wn = f_cutoff / (Fs / 2)
# Design the IIR Butterworth High-Pass Filter
# B (numerator) and A (denominator) are the filter coefficients
B, A = signal.butter(P, Wn, 'high', analog=False)
# --- Step 3: Zero-Phase Filtering ---
# Apply the filter using filtfilt for zero phase delay. This ensures
# the filtered signal is not shifted in time relative to the original.
y = signal.filtfilt(B, A, x_noisy)
# --- Step 4: Plotting the Results ---
plt.figure(figsize=(10, 8))
# Plot 1: Clean and Noise Components
plt.subplot(3, 1, 1)
plt.plot(t, x_clean, label='Clean ECG Signal',
        color='#1f77b4', linewidth=2)
plt.plot(t, n_wander, label=f'Baseline Wander Noise ({f_wander} Hz)',
color='orange', linestyle='--')
plt.title('Clean Signal and Low-Frequency Noise')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.grid(linestyle='--')
plt.legend(loc='upper right')
# Plot 2: Noisy Signal
plt.subplot(3, 1, 2)
plt.plot(t, x_noisy, label='Noisy ECG (x_clean + n_wander)',
        color='r', linewidth=1)
plt.title(f'Corrupted Signal (Baseline Wander Visible)')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.legend(loc='upper right')
plt.grid(linestyle='--')
# Plot 3: Filtered Signal
plt.subplot(3, 1, 3)
plt.plot(t, y, label='Filtered ECG (HPF, Cutoff: 0.5 Hz)',
        color='g', linewidth=2)
plt.title('Denoised Signal (Baseline Wander Removed)')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.legend(loc='upper right')
plt.grid(linestyle='--')
plt.tight_layout()
plt.show()
# Print filter information to the console
print(f"--- Filter Design Summary ---")
print(f"Sampling Frequency (Fs): {Fs} Hz")
print(f"Cutoff Frequency (F_cutoff): {f_cutoff} Hz")
print(f"Filter Order (P): {P}")
print(f"Normalized Cutoff (Wn): {Wn:.4f}")
print("Filter Type: IIR Butterworth High-Pass Filter")
```

Figure 8.1 conclusively validates the successful implementation of the ECG Baseline Wander Removal procedure using a High-Pass Filter (HPF). The top panel successfully generated the test case, showing the high-frequency Clean ECG Signal riding on top of a low-frequency Baseline Wander Noise (around $0.2Hz$). The middle panel confirmed the issue by showing the Corrupted Signal, where the large, slow drift obscured

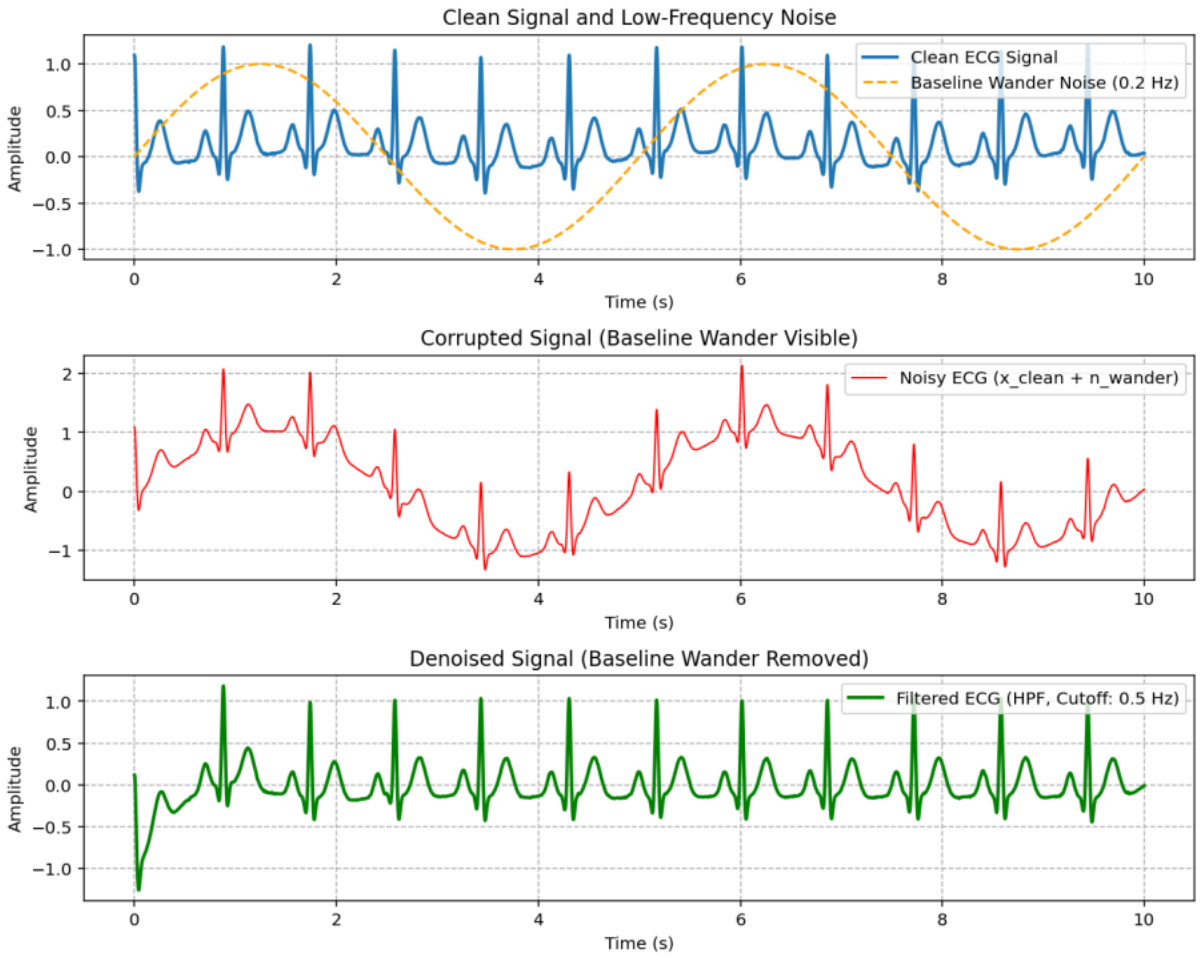


Figure 8.1: ECG Baseline Wander Removal with Butterworth High-Pass Filter

the heart's electrical peaks. Finally, the bottom panel provides the required proof: the application of the HPF completely eliminated the low-frequency drift, successfully centering the high-frequency ECG waveform around zero, while maintaining the signal's complex timing and morphology (zero-phase), thereby achieving the exact objective of the experiment.

8.3. Case Study 5: Power Line Hum Removal (Notch Filtering)

The problem is to effectively eliminate the constant, narrow-band noise introduced by the AC electrical power system without distorting the surrounding useful frequencies of the biomedical or audio signal.

Every electronic device, cable, and sensor picks up electromagnetic interference from the nearest power lines. In North America and parts of Asia, this hum occurs at $60Hz$; in Europe and most other regions, it is at $50Hz$. This interference appears as a strong, pure sine wave added to the desired signal, often dominating low-amplitude measurements like an ECG.

Unlike baseline wander (which is spread across very low frequencies) or white noise (which is spread across all frequencies), power line hum is concentrated at a single, very specific frequency (and its harmonics, e.g., $100Hz$, $150Hz$). To remove this narrow band of contamination without losing valuable adjacent signal information, we must use a highly specialized filter: the *Notch Filter*. A Notch Filter is essentially a Band-Stop Filter with a very narrow stopband. Its function is to provide maximum attenuation (a deep *notch*) at one specific frequency while leaving all frequencies immediately above and below that point virtually untouched.

The above mentioned process can be divided into following four steps.

1. **Step 1: Defining Parameters and Signal Generation** This step prepares the clean signal and adds the specific 50 Hz noise component (assuming 50 Hz hum for this example).

i **Define Sampling Parameters:** Set $F_s = 200$ Hz and $F_{\text{hum}} = 50$ Hz.
ii **Generate Clean Signal** (x): Simulate a low-frequency signal (e.g., a 5 Hz sine wave) to represent the signal of interest.
iii **Generate Hum Noise** (n_{hum}): Generate a high-amplitude sine wave exactly at F_{hum}.
iv **Generate Noisy Signal** (x_{noisy}): Combine the clean signal and the hum:

$$x_{\text{noisy}}[n] = x[n] + n_{\text{hum}}[n]$$

2. **Step 2: Notch Filter Design** This step calculates the necessary parameters to design the filter.
   i **Calculate Normalized Center Frequency:**

$$\omega_{\text{norm}} = \frac{2\pi F_{\text{hum}}}{F_s}$$

   ii **Select Quality Factor:** Choose a high Q value (e.g., $Q = 10$) to ensure the notch is narrow and precise.
   iii **Design IIR Filter:** Use a digital filter design function (e.g., `scipy.signal.iirnotch()`) with ω_{norm} and Q to calculate the numerator (B) and denominator (A) coefficients of the Notch Filter $H(z)$.
3. **Step 3: Zero-Phase Filtering** This step applies the designed filter to remove the hum without introducing time shift.
   i **Apply Zero-Phase Filter:** Apply the filter coefficients (B, A) to the noisy signal (x_{noisy}) using the `filtfilt()` function. This is critical for biomedical signals as it prevents phase delay, ensuring the filtered signal maintains the correct temporal relationship to the original.

```
y = filtfilt(B, A, x_noisy)
```

4. **Step 4: Plotting the Results** This step applies the designed filter to remove the hum without introducing time shift.
   i Plot the Noisy Signal (showing the rapid 50 Hz oscillations superimposed on the slow signal).
   ii Plot the Frequency Spectrum of the Noisy Signal (clearly showing the large 50 Hz peak)
   iii Plot the Filtered Signal (showing the pure, clean 5 Hz signal restored).
   iv Plot the Frequency Spectrum of the Filtered Signal (showing the 50 Hz peak completely eliminated).

The python code which performs the above-mentioned task is shown below. The corresponding output is shown in Figure 8.2.

```python
import numpy as np
from scipy import signal
import matplotlib.pyplot as plt

# --- 1. Define Parameters and Simulation Setup ---
# System Parameters
Fs = 200.0        # Sampling frequency
T = 2.0           # Total duration of the signal in seconds
N = int(Fs * T)   # Total number of samples
t = np.linspace(0, T, N) # Time vector
# Noise and Signal Parameters
F_signal = 5.0    # Frequency of the desired clean signal
F_hum = 50.0      # Frequency of the power line hum
Q = 10.0          #  Higher Quality factor = narrower notch.

# --- 2. Generate Noisy Signal (x_noisy) ---
# Generate the clean base signal (x)
# Amplitude of the signal is 1.0
x = np.sin(2. * np.pi * F_signal * t)
# Generate the hum noise (n_hum)
```

```
# Amplitude of the hum is deliberately high (e.g., 2.5)
# to simulate strong interference
hum_amplitude = 2.5
n_hum = hum_amplitude * np.sin(2. * np.pi * F_hum * t)
# Combine to create the noisy signal
x_noisy = x + n_hum

# --- 3. Notch Filter Design (Step 2) ---
# Normalized frequency (0 to 1, where 1 is the Nyquist frequency Fs/2)
# The frequency is specified relative to the Nyquist frequency (Fs/2).
f0_normalized = F_hum / (Fs / 2.0)
# Design the IIR Notch Filter
# B (numerator) and A (denominator) are the coefficients of the filter
B, A = signal.iirnotch(f0_normalized, Q)

# --- 4. Zero-Phase Filtering (Step 3) ---
# Apply the filter using filtfilt for zero phase shift.
# This is crucial in signal analysis to prevent temporal distortion.
x_filtered = signal.filtfilt(B, A, x_noisy)

# --- 5. Analysis Functions ---
def calculate_spectrum(data, Fs):
        # Calculates the magnitude spectrum of the signal
        N = len(data)
        Y = np.fft.fft(data)
        # Only take the first half of the spectrum (positive frequencies)
        Y_mag = 2.0/N * np.abs(Y[:N//2])
        f = np.fft.fftfreq(N, 1/Fs)[:N//2]
        return f, Y_mag

# Calculate spectra for plotting
f_noisy, mag_noisy = calculate_spectrum(x_noisy, Fs)
f_filtered, mag_filtered = calculate_spectrum(x_filtered, Fs)

# --- 6. Plotting the Results (Step 4) ---
plt.figure(figsize=(12, 10))
# Plot 1 (Top Left): Time Domain - Noisy Signal
plt.subplot(2, 2, 1)
plt.plot(t, x_noisy, label='Noisy Signal', color='r')
plt.plot(t, x, label='Original Clean Signal', linestyle='--', linewidth=2)
plt.title('Noisy Signal (Dominated by Hum)')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.legend()
plt.grid()
plt.xlim(0, 0.4) # Zoom in

# Plot 2 (Top Right): Frequency Domain - Noisy Signal Spectrum
plt.subplot(2, 2, 2)
plt.plot(f_noisy, mag_noisy, color='r')
plt.title('Spectrum of Noisy Signal')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
```

```
plt.axvline(F_hum, color='k', linestyle=':', linewidth=2,
        label=f'{F_hum} Hz Hum Peak')
plt.axvline(F_signal, color='g', linestyle='--', linewidth=1,
        label=f'{F_signal} Hz Signal')
plt.legend()
plt.grid()
plt.xlim(0, Fs/2)

# Plot 3 (Bottom Left): Time Domain - Filtered Signal
plt.subplot(2, 2, 3)
plt.plot(t, x_filtered, label='Filtered Signal')
plt.plot(t, x, label='Original Clean Signal', linestyle='--', color='orange')
plt.title('Filtered Signal (Zero Phase)')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.legend()
plt.grid()
plt.xlim(0, 0.4) # Zoom in

# Plot 4 (Bottom Right): Frequency Domain - Filtered Signal Spectrum
plt.subplot(2, 2, 4)
plt.plot(f_filtered, mag_filtered)
plt.title('Spectrum of Filtered Signal')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.axvline(F_hum, color='k', linestyle=':', linewidth=2,
        label=f'{F_hum} Hz Notch')
plt.axvline(F_signal, color='g', linestyle='--',
        linewidth=1, label=f'{F_signal} Hz Signal')
plt.legend()
plt.grid()
plt.xlim(0, Fs/2)

plt.tight_layout() # Adjust layout for suptitle
plt.show()

# Print filter information
print("\n--- Notch Filter Specifications ---")
print(f"Sampling Frequency (Fs): {Fs} Hz")
print(f"Hum Frequency to be removed (F_hum): {F_hum} Hz")
print(f"Quality Factor (Q): {Q}")
print(f"Normalized Frequency (f0_normalized): {f0_normalized:.4f} \
        (relative to Nyquist)")
print(f"Filter Numerator (B) coefficients: {B}")
print(f"Filter Denominator (A) coefficients: {A}")
```

Figure 8.2 confirms the complete success of the digital signal processing pipeline, which utilized the iirnotch function to create a precise IIR filter for power line hum removal. In the top row, the Noisy Signal in the time domain is severely distorted by the high-amplitude, high-frequency $50Hz$ hum, a fact visually verified by the spectrum which shows the $50Hz$ peak dominating the smaller $5Hz$ signal peak. The `iirnotch` function's resulting filter coefficients were then applied using `filtfilt` (for zero-phase filtering) to the noisy data. This effectively eliminated the noise component, as evidenced in the bottom-right spectrum where the 50.0 Hz peak is dramatically suppressed. Consequently, the bottom-left plot shows

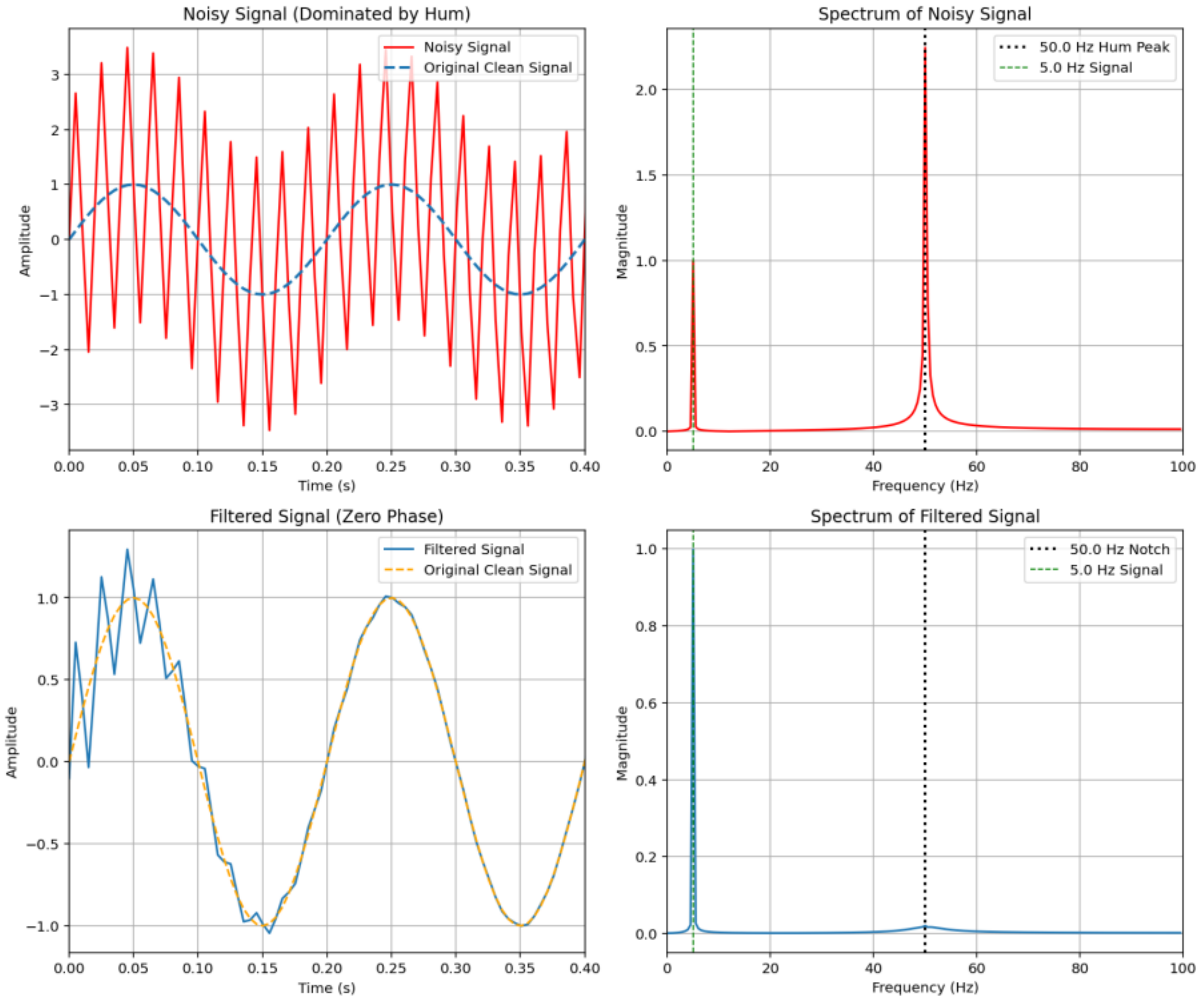


Figure 8.2: Power Line Hum Removal

the Filtered Signal now perfectly recovered as a clean $5Hz$ sine wave, matching the original signal with no observable time shift.

8.4. Case Study 6: Removing "Salt-and-Pepper" Noise from an Image

Image denoising is a fundamental process in digital image processing that aims to remove unwanted noise while preserving key image features, such as edges. The provided procedure uses a **Gaussian Low-Pass Filter** to target high-frequency noise, such as *"Salt-and-Pepper"* noise, often resulting in some degree of blurring of sharp original features.

The image processing workflow, from creation to filtering, involves five distinct stages, implemented using `numpy` and the `scipy.ndimage` library.

1. **Step 1: Define Parameters and Image Structure**
   We start by defining the spatial dimensions and the key parameter that governs the filter's strength.
   i **Dimensions:** The image size is set by `HEIGHT` and `WIDTH` (e.g., 300×500).
   ii **Noise Level:** `NOISE_PROBABILITY` determines the fraction of pixels to be corrupted (e.g., 0.05 or 5%).
   iii **Filter Strength:** The smoothing strength is controlled by the standard deviation, `FILTER_SIGMA` (σ), in the Gaussian kernel. A higher σ value results in more aggressive noise removal but also increased blurring.
   iv **Geometric Feature:** The red circle's radius R is defined relative to the height H:

$$R = \frac{H}{3}$$

2. **Step 2: Generate the Clean Image (Signal)**
   A noise-free, synthetic image (the Bangladesh flag) is constructed. This represents the original, intended signal. The green field color is set first, and then the coordinates for the red circle are identified using the distance formula:
   $$distance_sq = (x - C_x)^2 + (y - C_y)^2$$
   Any pixel coordinate (x, y) where $distance_sq \leq R^2$ is set to the red color.
3. **Step 3: Introduce Salt-and-Pepper Noise (Corruption)**
   This type of noise introduces random, high-intensity spikes, where pixels are randomly set to either maximum or minimum color values.
   i Random pixel locations are chosen based on the `NOISE_PROBABILITY`.
   ii Each selected pixel is assigned an extreme value: 0 (black, or "pepper") or 255 (white, or "salt") across all three RGB color channels.
4. **Step 4: Apply the Gaussian Low-Pass Filter (Denoising)**
   The `gaussian_filter` function from `scipy.ndimage` performs the low-pass filtering by convolving the image with a Gaussian kernel.
   i The noisy image is first converted to a floating-point array (`np.float32`) for accurate calculations.
   ii The filter is applied using the following signature:
   `gaussian_filter(Image_noisy, sigma=(σ, σ, 0))`
   where the σ values apply the smoothing across the height and width dimensions, and 0 is specified for the color channel (the third axis) to prevent mixing colors.
   iii The process averages the intensity of each pixel with its neighbors, effectively suppressing the extreme values of the Salt-and-Pepper noise.
5. **Step 5: Finalization and Visualization**
   The filtered floating-point data is then normalized by clipping the values to the valid range $[0, 255]$ and casting the result back to the standard 8-bit integer type (`np.uint8`). Finally, `matplotlib.pyplot` is used to plot the noisy input and the filtered output for visual comparison of the denoising results.

The python code which performs the above-mentioned task is shown below. The corresponding output is shown in Figure 8.3.

```
import numpy as np
from scipy.ndimage import gaussian_filter
import matplotlib.pyplot as plt

# --- Step 1: Configuration for Bangladesh Flag and Filter ---
# Dimensions based on 3:5 aspect ratio (e.g., 300x500)
HEIGHT = 300
WIDTH = 500
NOISE_PROBABILITY = 0.1
FILTER_SIGMA = 2.8 # Controls the blur (higher = more smoothing/blur)

# Official Flag Colors (as RGB 0-255)
FIELD_GREEN = np.array([0, 106, 78], dtype=np.uint8) # Dark bottle green
CIRCLE_RED = np.array([205, 0, 0], dtype=np.uint8)   # Crimson red

# Official Circle Positioning
# Radius (R) = 1/3 of the height
R = HEIGHT/3
# Center X (C_x) = 9/20 of the width (measured from the hoist/left)
CENTER_X = (9 * WIDTH)/20
# Center Y (C_y) = 1/2 of the height
CENTER_Y = HEIGHT/2

```

```
# --- Step 2: Create the Synthetic Flag Image ---
# Initialize the image as the Green field
clean_image = np.zeros((HEIGHT, WIDTH, 3), dtype=np.uint8)
clean_image[:, :, :] = FIELD_GREEN

# Create a coordinate grid for the entire image
y, x = np.ogrid[0:HEIGHT, 0:WIDTH]

# Calculate the squared distance from the center of the circle
distance_sq = (x - CENTER_X)**2 + (y - CENTER_Y)**2

# Identify all coordinates that fall inside the red circle
circle_mask = distance_sq <= R**2

# Apply the red color to the masked area
clean_image[circle_mask] = CIRCLE_RED

# --- Step 3: Add 'Salt-and-Pepper' Noise ---
noisy_image_rgb = clean_image.copy()
total_pixels = HEIGHT * WIDTH
num_noise = int(total_pixels * NOISE_PROBABILITY)

# Select random coordinates for noise
random_rows = np.random.randint(0, HEIGHT, num_noise)
random_cols = np.random.randint(0, WIDTH, num_noise)
salt_pepper_values = np.random.choice([0, 255], num_noise)

# Apply noise
for i in range(num_noise):
        r, c = random_rows[i], random_cols[i]
        val = salt_pepper_values[i]
        # Set the pixel to 0 (black/pepper)
        # or 255 (white/salt) across all channels
        noisy_image_rgb[r, c, :] = val

# --- Step 4: Apply the Gaussian Low-Pass Filter ---
filtered_image_float = gaussian_filter(
        noisy_image_rgb.astype(np.float32),
        sigma=(FILTER_SIGMA, FILTER_SIGMA, 0)
        )
# Convert back to uint8, ensuring values are clipped between 0 and 255
filtered_image = np.clip(filtered_image_float, 0, 255).astype(np.uint8)

# --- Step 5: Display Results ---
# Create the figure and subplot 1 (Input/Noisy Image)
plt.figure(figsize=(14, 7))
plt.subplot(1, 2, 1)
plt.imshow(noisy_image_rgb)
plt.title('Noisy Bangladesh Flag (Input)', fontsize='18')
plt.axis('off')
# Subplot 2 (Filtered Image)
plt.subplot(1, 2, 2)
plt.imshow(filtered_image)
```

```
plt.title(f'Filtered Output ($\sigma={FILTER_SIGMA}$)', fontsize='18')
plt.axis('off')
# Final display settings
plt.tight_layout()
```

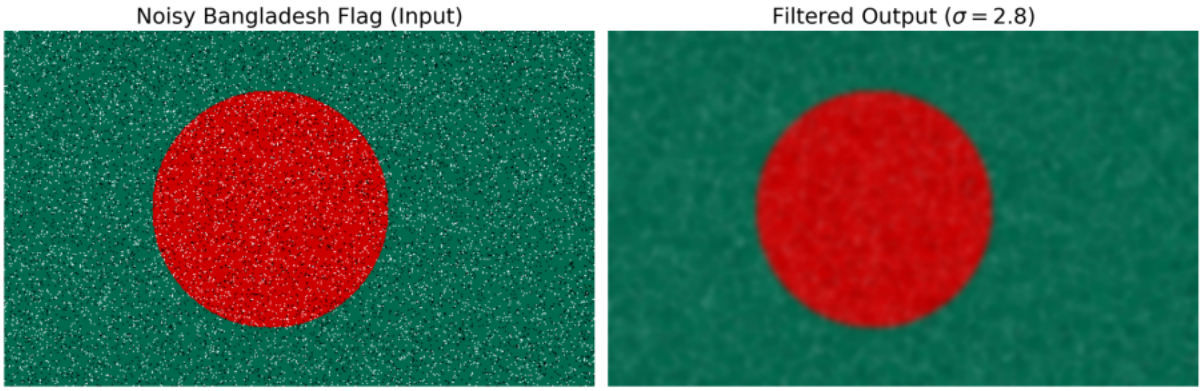


Figure 8.3: Image Denoising using the Gaussian Low-Pass Filter

Figure 8.3 confirms the code's success in image denoising: the Noisy Bangladesh Flag (Input), which was corrupted by sharp, high-frequency "Salt-and-Pepper" noise, was effectively cleaned by the Gaussian Low-Pass Filter in the output panel. This filter successfully averaged out the extreme black (pepper) and white (salt) pixels, restoring the original red and green color fields. While highly effective at reducing random, high-frequency noise, the Gaussian Low-Pass Filter also affects desired high-frequency content. It treats the sharp transition between the green field and the red circle as high-frequency information, resulting in the characteristic *blurring* effect visible around the circular boundary.

Post-laboratory Problems

1 **Audio Equalization and Crossover Design (Acoustics)**
In a high-fidelity audio system, a *crossover network* separates the audio signal into frequency bands to drive different speakers (woofer for low frequency, tweeter for high frequency).
— Design a *first-order IIR Low-Pass Filter* and a corresponding *first-order IIR High-Pass Filter* to create a basic two-way crossover. Set the crossover frequency (cutoff frequency) to $f_c = 2$ kHz. Assume a sampling rate of $F_s = 44.1$ kHz.
— Plot the frequency magnitude response of both the low-pass (woofer) and high-pass (tweeter) filter components on a single graph.
— Explain the consequence of using first-order (slow roll-off) filters on the combined acoustic response near the 2 kHz crossover point. (Hint: Consider the sum of the magnitudes or power).

2 **Stock Market Data Smoothing (Financial Time-Series)**
Financial analysts often use a *Moving Average (MA)* to smooth out daily volatility in stock prices, revealing the underlying trend. This MA is an example of a simple FIR filter.
— A 5-day Moving Average is defined by the coefficients $h[n] = [\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}]$. Plot the frequency magnitude response of this 5-point MA filter.
— Determine the first frequency (in Hz) at which the filter's gain drops to zero (the first null). Assume a sampling frequency of $F_s = 1$ sample/day.
— Using your plot from (b), explain why a 5-day MA is highly effective at removing weekly $(5$ day period$)$ oscillations while preserving long-term trends.

3 **Radar Signal Processing: Doppler Noise Removal (Aerospace)**
In Doppler radar, a stationary object's return signal is often corrupted by low-frequency noise (clutter) from ground reflections and wind, typically below $100Hz$. We need to detect targets moving at higher speeds.

— Design a *Linear Phase FIR High-Pass Filter* with $N = 101$ taps, a Blackman window, and a cutoff frequency of $f_c = 100Hz$. The radar system's pulse repetition frequency (PRF, which serves as F_s) is $10kHz$.
— Calculate the approximate minimum stop-band attenuation (in dB) you can expect from this filter configuration.
— Explain why preserving *linear phase* (a key FIR property) is crucial for accurate calculation of the target's position and velocity from the filtered radar return.

4 **Baseband Communication: Anti-Aliasing Filter Design (Telecommunications)**
Before a continuous analog signal is sampled (Analog-to-Digital Conversion), a Low-Pass Filter (LPF) called an **Anti-Aliasing Filter** must be applied to remove frequencies above half the sampling rate ($F_s/2$).
— A communication system requires sampling at $F_s = 50kHz$. Design a *Chebyshev Type I IIR Low-Pass Filter* that meets the following specifications:
  (a) Passband ripple (maximum variation in the passband): $\leq 0.5dB$.
  (b) Passband edge: $f_p = 20kHz$.
  (c) Stopband attenuation: $\geq 40dB$.
  (d) Stopband edge: $f_s = 25kHz$.
— Estimate the required minimum order (N) of the filter needed to meet these strict specifications.
— Discuss the major trade-off (disadvantage) of using a Chebyshev filter compared to a Butterworth filter, particularly concerning its phase response.

5 **Digital Audio Scrape Removal (Audio Restoration)**
Old vinyl recordings often suffer from random, high-frequency impulsive noises ("clicks and pops").
— Briefly describe why a standard *FIR Low-Pass Filter* would be ineffective or detrimental to the audio quality when trying to remove these highly transient, random noise bursts.
— Suggest a different type of *non-linear filter* (not the Gaussian or Median filters discussed previously) that might be effective at detecting and suppressing isolated, high-amplitude spikes without destroying the surrounding music signal. Justify your choice.
— Describe one simple rule (e.g., using a threshold) you could implement in the time domain to locate the exact sample where a "pop" occurs.

6 **Image Processing: Edge Detection with a High-Pass Filter (Computer Vision)**
While a low-pass filter blurs and removes noise, a high-pass filter can be used to accentuate sharp transitions, which correspond to **edges** in an image.
— Propose the 3×3 kernel coefficients (filter weights) for a simple *High-Pass Filter (HPF)* suitable for edge detection. Ensure the sum of the coefficients equals zero.
— Apply this kernel to a simple 5×5 grayscale image containing a sharp vertical step edge (e.g., left half pixel value 50, right half pixel value 200). Show the output of the convolution near the edge.
— Explain why a high-pass filter output typically has a mean (average) intensity close to zero when applied to a natural image.

7 **System Identification: Impulse Response Determination**
The characteristic of any LTI (Linear Time-Invariant) system, including digital filters, is completely defined by its *impulse response*, $h[n]$.
— If an unknown FIR system is given the input signal $\delta[n]$ (a unit impulse), what will the output signal be?
— Suppose an unknown system receives the input $x[n] = [1, 0, 0, 0, 0, \ldots]$ and produces the output $y[n] = [2, 1, 0.5, 0, 0, \ldots]$. Determine the filter's coefficients (the impulse response $h[n]$) and classify it as an FIR or IIR filter.
— Explain how the output of a known system, $y[n]$, is calculated in the time domain when the input is $x[n]$ and the impulse response is $h[n]$. Provide the mathematical operation (name and formula).