Implementation and Analysis of Selected Audio Processing Algorithms

(Implementacja i analiza
wybranych algorytmów przetwarzania dźwięku)

Weronika Tarnawska

Praca inżynierska

Promotor: dr hab. Paweł Woźny

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

5 czerwca 2025

Abstract

This thesis focuses on the implementation of audio processing algorithms that enable the modification of audio signals by applying sound effects or noise reduction. The methods employed are based on the Fast Fourier Transform (FFT), which facilitates the analysis and processing of signals in the frequency domain. Various approaches will be compared in terms of performance and the quality of the processed audio.

Praca dotyczy implementacji algorytmów przetwarzania dźwięku, które umożliwiają modyfikację sygnału audio poprzez nałożenie efektów dźwiękowych lub redukcję szumów. Wykorzystane zostaną metody oparte na szybkiej transformacji Fouriera (FFT), pozwalające na analizę i przetwarzanie sygnałów w dziedzinie częstotliwości. Porównane zostaną różne podejścia pod kątem ich wydajności oraz jakości uzyskanego dźwięku.

Contents

Chapter 1

Introduction

TODO

Chapter 2

DSP basics

This chapter was mainly based on [2] and [8].

2.0.1 Analog Sound and Digital Representation

Sound in the physical world is manifested as pressure variations propagating through a medium (e.g., air). These variations are continuous in both time and amplitude. To process sound using digital systems (i.e., computers), one must convert this continuous signal $x(t)$ into a sequence of finite, discrete values x_n. The conversion occurs in two stages:

- *Sampling*: Capturing the signal at discrete time instants.
- *Quantization*: Approximating the signal amplitudes to a finite set of levels.

2.0.2 Sampling and the Nyquist Theorem

Sampling is the process of measuring the amplitude of the continuous-time signal $x(t)$ at a uniform rate. Formally, the sampled signal is given by

$$x_n \coloneqq x(nT_s), \tag{2.1}$$

where $T_s > 0$ is the *sampling period* and $f_s \coloneqq \frac{1}{T_s}$ is the *sampling frequency*. According to the *Nyquist-Shannon sampling theorem* [8, §3], to perfectly reconstruct a signal with maximum frequency $f_{\max}$, the sampling frequency must satisfy

$$f_s \geq 2f_{\max}. \tag{2.2}$$

The minimum required sampling rate $2f_{\max}$ is known as the *Nyquist rate*. If $f_s < 2f_{\max}$, *aliasing* occurs: higher-frequency components are indistinguishably folded into lower frequencies.

2.0.3 Quantization

While sampling discretizes time, *quantization* discretizes amplitude. Each sample x_n is mapped to the nearest level in a finite set of L uniformly spaced quantization levels. If the full-scale range of the signal is $[-x_{\max}, x_{\max}]$, then the quantization step size is

$$\Delta \coloneqq \frac{2X_{\max}}{L} = \frac{2X_{\max}}{2^B}, \tag{2.3}$$

where B is the number of bits per sample. Quantization introduces an error called *quantization noise* — usually a random additive noise, uniformly distributed between $\pm\frac{1}{2}\Delta$.

2.0.4 Pulse-Code Modulation and WAV Files

Pulse-code modulation (PCM) is the standard method used to digitally represent analog signals. In PCM, each quantized sample is encoded as a binary word of B bits. The resulting bitstream can then be stored or transmitted. One of the most common container formats for PCM audio is the WAV (Waveform Audio File Format), which wraps raw PCM data with a header specifying parameters such as sampling frequency f_s and bit depth B.

2.0.5 Time-Domain and Frequency-Domain Representations

Once audio is digitized, it can be analysed in different domains:

Time Domain: Direct inspection of the sample sequence x_n reveals amplitude variations over time.

Frequency Domain: By applying the discrete Fourier transform (introduced in the next section), one obtains the spectral representation of the signal. This representation reveals how the signal's energy is distributed across different frequencies, allowing us to analyze which frequency components are present and in what proportion.

2.1 DFT & FFT

2.1.1 Complex Numbers

Definition and Basic Properties

A *complex number* z is typically expressed in the form:

$$z = a + bi \qquad (a, b \in \mathbb{R}),$$

where:

- a is the *real part* of z,
- b is the *imaginary part* of z,
- i is the *imaginary unit*, defined as $i^2 = -1$.

The *complex conjugate* of z, denoted $\overline{z}$, is:

$$\overline{z} \coloneqq a - bi.$$

The *modulus* or *magnitude* of z, denoted $|z|$, is given by

$$|z| \coloneqq \sqrt{a^2 + b^2}.$$

Observation 2.1. Note that $z\overline{z} = |z|^2$.

Polar Form and Euler's Formula

A complex number can also be represented in polar form as

$$z = |z|(\cos(\phi) + i\sin(\phi)),$$

where ϕ represents the angle on a *complex plane*, i.e., $\phi = \arctan(b/a)$ for $a \neq 0$. Note that if $a = 0$, the number is purely imaginary, so $z = |z|i$ (i.e., $\phi = \frac{\pi}{2}$).

Using a famous Euler's formula,

$$e^{i\phi} = \cos(\phi) + i\sin(\phi),$$

we can express z in terms of its magnitude and angle, i.e., in the form

$$z = |z|e^{i\phi}.$$

This *polar form* simplifies the multiplication of complex numbers. Given two complex numbers $z_1 \coloneqq |z_1|e^{i\phi_1}$ and $z_2 \coloneqq |z_2|e^{i\phi_2}$, their product is

$$z_1 z_2 = |z_1||z_2|e^{i(\phi_1+\phi_2)}.$$

This property shows that multiplying two complex numbers combines their magnitudes and adds their angles, which geometrically corresponds to scaling and rotating in the complex plane.

2.1.2 The Fourier Transform Family

Before diving into the *Discrete Fourier Transform* (DFT) and its efficient implementation via the *Fast Fourier Transform* (FFT), it is useful to recognize that all Fourier transforms belong to one of four related categories:

- **Aperiodic-Continuous** - transforms signals that extend to both positive and negative infinity without repeating in a periodic pattern into a continuous frequency spectrum. This type of Fourier Transform is called *the Fourier Transform.*

- **Periodic-Continuous** - transforms continuous-time signals that repeat periodically from negative to positive infinity (e.g., sine waves) into a discrete set of frequencies. This type is called *the Fourier Series.*

- **Aperiodic-Discrete** - transforms non-repeating signals that extend to both positive and negative infinity and are sampled at discrete time instants into a continuous periodic frequency function. This type is called *the Discrete-Time Fourier Transform, DTFT.*

- **Periodic-Discrete** - transforms signals that repeat periodically from negative to positive infinity and are sampled at discrete time instants into a discrete set of frequency bins. This type is called *the Discrete Fourier Transform, DFT.*

All four variants share the same basic idea: they convert a time-domain signal into a frequency-domain representation by expressing it as a combination of sinusoids.

In practice, digital systems only store a finite block of samples. To work around this, we imagine extending our finite data by repeating the block indefinitely in both directions, making it a periodic, infinite, discrete signal. With this interpretation, the DFT becomes applicable and computable on real-world data.

In this thesis, we focus exclusively on the DFT, as it is the only Fourier transform that can be directly implemented on a digital computer. The other transform variants are still useful - textbooks and research papers often rely on the continuous or infinite forms when explaining concepts or deriving formulas, before eventually translating the results into the DFT framework for practical use.

2.1.3 Discrete Fourier Transform (DFT)

The Discrete Fourier Transform (DFT) converts a time-domain signal into its frequency-domain components, representing each frequency as a complex number.

The DFT formula for a discrete signal $x = \Big(x_0, x_1, \dots x_{N-1}\Big)$ with N samples is

$$Y_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi \frac{k}{N} n} \qquad \Big(0 \leq k \leq N-1\Big).$$

The result is a set of complex numbers $Y = \Big(Y_0, Y_1, \dots, Y_{N-1}\Big)$, where each Y_k provides the amplitude and phase for a sinusoid at some frequency f_k.

In expanded form using Euler's identity, each Y_k can be expressed as:

$$Y_k = \sum_{n=0}^{N-1} x_n \left(\cos\left(2\pi \frac{k}{N} n\right) - i \sin\left(2\pi \frac{k}{N} n\right) \right).$$

This expression reveals that the DFT is comparing the input signal x to complex sinusoids (sine and cosine components) of increasing frequencies. Each Y_k value captures the "amount" of the corresponding frequency in the signal by summing contributions across time samples with a phase rotation specific to k ($0 \leq k \leq N-1$).

Interpretation Using Complex Plane Geometry

The DFT can be interpreted geometrically in the complex plane, where each term in the summation represents a rotation around the unit circle. Below is a breakdown of how each element of the DFT formula contributes to capturing frequency components in a signal. This perspective draws from a video by 3Blue1Brown [4] and a blog post by Elan Ness-Cohn [5].

- **Rotation Representing Frequency**: The term $e^{-i2\pi \frac{k}{N} n}$ in the DFT formula rotates clockwise around the unit circle, with a frequency determined by the fraction $\frac{k}{N}$. For each unit increment in n, the angle increases by $-2\pi \frac{k}{N}$ radians, creating a full clockwise rotation over N steps. The negative sign in the exponent indicates this clockwise direction.
- **Example with Increasing Frequencies**:
  - $f_1(\phi) \coloneqq e^{i\phi}$ traces a circle of radius 1 counterclockwise as ϕ increases.
  - $f_2(\phi) \coloneqq e^{i2\phi}$ traces a circle of radius 1 but rotates twice as quickly, completing two circles for every one circle traced by $e^{i\phi}$.
  - $f_3(n) \coloneqq e^{i2\pi n}$ completes one full rotation each time n increments by 1, returning to the starting point.
  - $f_4(n) \coloneqq e^{-i2\pi n}$ rotates clockwise with each increment of n.
  - $f_5(n) \coloneqq e^{-i2\pi n \frac{k}{N}}$ completes $\frac{k}{N}$ of a circle with each increment of n, resulting in k full rotations over N samples.
- **Frequency Resolution**: The DFT evaluates the signal's components at discrete frequencies (also called *frequency bins*) $f_k = \frac{k}{N} f_s$, where f_s is the sampling frequency.
- **Summing Contributions**: Each signal sample x_n is multiplied by a rotating complex exponential. The rotation frequency $\frac{k}{N}$ determines how many rotations the term makes as n progresses:
  - For $k = 1$, the term $e^{-i2\pi \frac{1}{N} n}$ rotates once per N samples, aligning once around the unit circle.

  - For $k = 2$, the term $e^{-i2\pi\frac{2}{N}n}$ rotates twice per N samples, completing two circles.
  - In general, $e^{-i2\pi\frac{k}{N}n}$ traces out k circles for each complete set of N samples.

  As n increases, the points $x_n e^{-i2\pi\frac{k}{N}n}$ are distributed around the circle. If a particular frequency f_k is present in the signal, these points will tend to align more on one side of the circle. This asymmetry leads to a non-zero sum, Y_k, indicating a frequency match.

- **Magnitude and Phase of** Y_k: The magnitude $|Y_k|$ is proportional to the amplitude of the frequency component corresponding to f_k, although additional scaling is required to retrieve the exact amplitude. The angle of Y_k, gives the phase of this frequency component in the time domain.

- **Intuitive Interpretation of the DFT Sum**:
  - We can think of each sample x_n as being "spread" around the circle according to $e^{-i2\pi\frac{k}{N}n}$.
  - The sum $\sum_{n=0}^{N-1} x_n e^{-i2\pi\frac{k}{N}n}$ acts similarly to a weighted average, or center of mass, of these points. A non-zero result for Y_k suggests a frequency component matching f_k, as points cluster asymmetrically, generating a meaningful magnitude $|Y_k|$.

DFT Output Amplitudes

Let N be the DFT length and consider exactly an integer number of periods of the input within those N samples.

1. If the input is a real sinusoid of amplitude A_0,

$$x_n = A_0 \cos\left(2\pi k_0 n/N + \phi\right), \quad n = 0, \ldots, N-1,$$

   then the nonzero DFT coefficient at bin k_0 has magnitude

$$|X_{k_0}| = \frac{A_0 N}{2}.$$

2. If the input is a complex exponential of amplitude A_0,

$$x_n = A_0 e^{i(2\pi k_0 n/N + \phi)},$$

   then the DFT coefficient at bin k_0 has magnitude

$$|X_{k_0}| = A_0 N.$$

3. If the input is shifted by a constant value D_0 (i.e., the average of x is D_0), then the zero-frequency coefficient is

$$X_0 = D_0 N.$$

To express any DFT magnitude in decibels (dB), use

$$M_{\mathrm{dB}} \;=\; 20\log_{10}\left(|X_{k_0}|\right).$$

Important properties

These properties are described in [10].

Fact 2.2 (Conjugate Symmetry of the DFT)**.** *If x is a real-valued length-L signal, then its L-point DFT satisfies the conjugate-symmetry property*

$$X_k = \overline{X_{(L-k) \bmod L}} \qquad k = 0, 1, \ldots, L-1.$$

As a consequence, if $\check{x}$ denotes the time-reversed signal, defined by $\check{x}_n \coloneqq x_{L-1-n}$, then

$$DFT\Big(\check{x}\Big)(k) = \overline{DFT\Big(x\Big)(k)}.$$

Observation 2.3 (Interpretation of DFT bins beyond the Nyquist frequency)*.* The N-point DFT always computes spectrum samples at frequencies

$$f_k = \frac{k}{N} f_s, \quad k = 0, 1, \ldots, N-1.$$

By the Nyquist theorem, only components with $f \in [0, f_s/2]$ can be unambiguously represented. The DFT bins beyond $k > N/2$, for real input signals, correspond to the mirror image of the positive-frequency spectrum, and do not carry any additional information by themselves.

Fact 2.4 (Linearity of the DFT)**.** *The discrete Fourier transform is a linear operator. In particular, for any two length-L signals a and b, and for constants $\alpha, \beta \in \mathbb{R}$,*

$$\begin{aligned}
DFT\Big(\,\alpha a + \beta b\,\Big)(k) &= \sum_{n=0}^{L-1} (\alpha a_n + \beta b_n) e^{-j2\pi kn/L} \\
&= \sum_{n=0}^{L-1} \alpha a_n e^{-j2\pi kn/L} + \sum_{n=0}^{L-1} \beta b_n e^{-j2\pi kn/L} \\
&= \alpha \sum_{n=0}^{L-1} a_n e^{-j2\pi kn/L} + \beta \sum_{n=0}^{L-1} b_n e^{-j2\pi kn/L} \\
&= \alpha\; DFT\Big(a\Big)(k) + \beta\; DFT\Big(b\Big)(k).
\end{aligned}$$

Computational Complexity

A straightforward DFT calculation has a computational complexity of $O(N^2)$ due to the nested summation over N samples.

2.1.4 Spectral leakage and windowing

Spectral leakage in the DFT arises from two related effects:

1. **Scalloping loss.** The DFT tells us "how much of each frequency" is present in the signal for a set of discrete frequency bins. What if the signal contains a frequency $f_\star$, but we don't measure exactly that one, because $f_\star \neq \frac{k}{N} f_s$? In that case, that frequency "leaks" into several neighbouring bins.

2. **Boundary discontinuities.** When we compute the DFT we always take a finite segment of the signal - a *window* - which by default is rectangular. The DFT treats this segment as periodic. If you select a segment of length N, and repeat it infinitely, there is a discontinuity at the edges (the signal value drops to zero immediately after the segment). This jump itself produces a spectrum that obscures the original.

These problems are mitigated by multiplying the signal by a tapering mask w that attenuates values at the window edges while keeping central values more pronounced. The effect is:

- **Reduction of discontinuities (primary goal).** Window functions smooth the edges (bringing them close to zero), eliminating the abrupt truncation, which significantly reduces leakage caused by the edge jumps.

- **Partial improvement of scalloping loss.** Windowing slightly reduces the amplitudes of leaked energy, but does not increase the fundamental frequency resolution. Complete elimination of scalloping loss would require the window length to match an integer number of signal periods, which is impractical.

Example windows

Most window functions are positive and symmetric bell-shaped curves that weight the center of each frame more heavily than its edges. This reduces spectral leakage at the frame boundaries, but "loses" a bit of information at the edges.

When processing a signal in frames (as will be done for some use-cases in §3), the frames are usually overlapped in time, to compensate the loss at the edges.

Some commonly used windows and their typical frame overlaps are:

- *Hanning window*

$$w_n := 0.5 - 0.5 \cos\Big(\frac{2\pi\, n}{N-1}\Big), \quad n = 0, 1, \ldots, N-1.$$

  Typical overlap is about half of frame size.

- *Hamming window*

$$w_n \coloneqq 0.54 - 0.46\cos\Big(\frac{2\pi\, n}{N-1}\Big), \quad n = 0, 1, \ldots, N-1.$$

  Typical overlap is about half of frame size.

- *Blackman window*

$$w_n \coloneqq 0.42 - 0.50\cos\Big(\frac{2\pi\, n}{N-1}\Big) + 0.08\cos\Big(\frac{4\pi\, n}{N-1}\Big), \quad n = 0, 1, \ldots, N-1.$$

  Typical overlap is about $\frac{2}{3}$ of frame size.

- *Rectangle window*
  This is equivalent to applying no window, so no overlapping is needed.

There are many other window functions. More on window functions can be found in [11, §*Spectrum Analysis Windows*].

2.1.5 Fast Fourier Transform (FFT)

The *Fast Fourier Transform* (FFT) is a method for computing the Discrete Fourier Transform (DFT) efficiently. It reduces the computational complexity from $O(N^2)$ to $O(N \log N)$, where the $\log N$ complexity arises due to the divide-and-conquer approach.

The presentation of the FFT algorithm below closely follows the derivation provided in Lyons' textbook [2], with additional reference to the Wikipedia article [3].

Radix-2 DIT Cooley-Tukey FFT

Several FFT algorithms exist. The Cooley-Tukey algorithm is the simplest and one of most commonly used. In this section, we derive the Radix-2 DIT (Decimation in Time) Cooley-Tukey FFT algorithm, which reduces an N-point DFT into two $N/2$-point DFTs.

Radix-2 means we reduce an N-point problem into two subproblems of size $N/2$. The Cooley-Tukey algorithm achieves that by separating the even and odd terms in the DFT summation. DIT (Decimation in Time) indicates that the decomposition occurs in the time domain. For this algorithm, N must be a power of 2, which can be achieved by zero-padding if necessary.

Dividing into even and odd terms

We start with the DFT formula,

$$Y_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi \frac{kn}{N}}.$$

Splitting the summation into terms with even indices and odd indices, we get

$$Y_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-i2\pi \frac{2mk}{N}} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{-i2\pi \frac{(2m+1)k}{N}}.$$

Now, we can factor out $e^{-i2\pi \frac{k}{N}}$ from the second summation because $e^{-i2\pi \frac{(2m+1)k}{N}} = e^{-i2\pi \frac{2mk}{N}} \cdot e^{-i2\pi \frac{k}{N}}$, i.e.,

$$Y_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-i2\pi \frac{2mk}{N}} + e^{-i2\pi \frac{k}{N}} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-i2\pi \frac{2mk}{N}}.$$

Next, we substitute $\frac{2mk}{N} = \frac{mk}{N/2}$ in the exponents and obtain

$$Y_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-i2\pi \frac{mk}{N/2}} + e^{-i2\pi \frac{k}{N}} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-i2\pi \frac{mk}{N/2}}.$$

Letting $E_k \coloneqq \sum_{m=0}^{N/2-1} x_{2m} e^{-i2\pi \frac{mk}{N/2}}$ and $O_k \coloneqq \sum_{m=0}^{N/2-1} x_{2m+1} e^{-i2\pi \frac{mk}{N/2}}$, we get

$$Y_k = E_k + e^{-i2\pi \frac{k}{N}} O_k.$$

Computing second half of Y-s

Now let us compute $Y_{k+N/2}$ (for $k < N/2$). We substitute k with $k + N/2$,

$$Y_{k+N/2} = \sum_{m=0}^{N/2-1} x_{2m} e^{-i2\pi \frac{m(k+N/2)}{N/2}} + e^{-i2\pi \frac{k+N/2}{N}} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-i2\pi \frac{m(k+N/2)}{N/2}}.$$

We can rewrite $e^{-i2\pi \frac{m(k+N/2)}{N/2}}$ as $e^{-i2\pi \frac{mk}{N/2}} \cdot e^{-i2\pi \frac{m(N/2)}{N/2}}$. Since $e^{-i2\pi m} = 1$, the second factor simplifies to 1 and we get

$$Y_{k+N/2} = \sum_{m=0}^{N/2-1} x_{2m} e^{-i2\pi \frac{mk}{N/2}} + e^{-i2\pi \frac{k+N/2}{N}} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-i2\pi \frac{mk}{N/2}}.$$

We can further simplify $e^{-i2\pi \frac{k+N/2}{N}}$ as $e^{-i2\pi \frac{k}{N}} \cdot e^{-i2\pi \frac{N/2}{N}}$. Since $e^{-i2\pi \frac{N/2}{N}} = e^{-i\pi} = -1$, we obtain

$$Y_{k+N/2} = \sum_{m=0}^{N/2-1} x_{2m} e^{-i2\pi \frac{mk}{N/2}} - e^{-i2\pi \frac{k}{N}} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-i2\pi \frac{mk}{N/2}}$$

which finally gives

$$Y_{k+N/2} = E_k - e^{-i2\pi \frac{k}{N}} O_k.$$

Result of the Recursive Decomposition

We now have

$$Y_k = E_k + e^{-i2\pi \frac{k}{N}} O_k$$

and

$$Y_{k+N/2} = E_k - e^{-i2\pi \frac{k}{N}} O_k.$$

This recursive approach allows us to compute two DFTs of size $N/2$ (namely E_k and O_k), from which we can obtain the DFT of size N in $O(N)$ time.

Computational Complexity

For each stage of the recursion, we have N terms divided into two parts, each requiring $N/2$ operations. We can combine both parts to get full solution in $O(N)$. This gives the following recurrence relation

$$T(N) = 2T(N/2) + O(N),$$

where $T(N)$ denotes time needed to solve instance of size N. We can solve this recurrence equation using, e.g., *master method* [6, §4], and get

$$T(N) = O(N \log N).$$

Pseudocode

Here is a pseudocode implementation of the Radix-2 Cooley-Tukey FFT algorithm:

Algorithm 1 Recursive Cooley-Turkey FFT algorithm

function FFT(x)
    $N \leftarrow \text{length}(x)$
    if $N = 1$ **then**
        return x
    end if
    $X_{\text{even}} \leftarrow \text{FFT}(x[0], x[2], \ldots, x[N-2])$
    $X_{\text{odd}} \leftarrow \text{FFT}(x[1], x[3], \ldots, x[N-1])$
    for $k = 0$ to $N/2 - 1$ **do**
        $t \leftarrow e^{-2\pi ik/N} \cdot X_{\text{odd}}[k]$
        $X[k] \leftarrow X_{\text{even}}[k] + t$
        $X[k + N/2] \leftarrow X_{\text{even}}[k] - t$
    end for
    return X
end function

Accuracy

The accuracy of the FFT algorithm can be discussed under three interrelated aspects: frequency resolution, spectral leakage, and numerical precision.

Frequency resolution: By the sampling theorem (§2.0.2), a real-valued signal sampled at rate f_s contains no information above $f_s/2$. An N-point FFT yields spectral bins at frequencies

$$f_k = \frac{k}{N} f_s, \quad k = 0, 1, \ldots, \frac{N}{2},$$

so that the bin-to-bin spacing (frequency resolution) is

$$\Delta f = \frac{f_s}{N}.$$

Thus, increasing N improves resolution by reducing Δf, but at the cost of larger frame sizes and higher computational work.

Spectral leakage: Because the FFT operates on a finite segment of length N, even a pure tone will "leak" energy into neighbouring bins unless its period fits exactly in the segment. Window functions (see 2.1.4) reduce leakage by tapering the frame edges, but they also attenuate signal energy at the boundaries. Frame overlap compensates for this loss, though it cannot entirely eliminate leakage without discarding some information — especially in the first and last frames, where overlap is impossible.

Numerical precision and stability: FFT performs a sequence of $\approx \log N$ stages, each involving floating-point operations. Each individual floating-point operation $a \circ b$ (where $\circ \in \{+, -, \times, \div\}$ and a, b are floating-point numbers) is realized as

$$\mathrm{fl}(a \circ b) = (a \circ b)\,(1 + \delta), \quad |\delta| \le \epsilon,$$

where δ is the relative round-off error of that operation and ϵ (machine epsilon) is its upper bound (for IEEE-754 double precision, $\epsilon \approx 2^{-52}$) [12]. Because a Cooley-Tukey FFT comprises $O(\log_2 N)$ such stages, the global relative error accumulates to

$$O(\epsilon \log_2 N),$$

which grows only logarithmically with N.

Despite finite-precision effects, a standard double-precision FFT followed by inverse FFT returns a signal whose difference from the original is too small to hear.

2.1.6 Inverse DFT and FFT

The *Inverse Discrete Fourier Transform (IDFT)* transforms frequency-domain data back into the time domain. The formula for the IDFT is

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} Y_k \, e^{i2\pi \frac{k}{N} n} \qquad \Big(0 \le n \le N-1\Big),$$

where x_n represents the signal in the time domain, Y_k represents the signal in the frequency domain, and N is the number of samples. The only differences from the DFT are the positive sign in the exponent, indicating a reverse rotation in the complex plane, and the factor $\frac{1}{N}$, which normalizes the result to undo the amplitude scaling introduced by the forward transform.

Theorem 2.5 (IDFT(DFT(x)) = x)**.** *Let $x = [x_0, x_1, \ldots, x_{N-1}]$ be a sequence of length N. Define its Discrete Fourier Transform (DFT) by*

$$Y_k = \sum_{n=0}^{N-1} x_n \, e^{-i2\pi \frac{k}{N} n}, \qquad 0 \le k \le N-1,$$

and its Inverse Discrete Fourier Transform (IDFT) by

$$y_n = \frac{1}{N} \sum_{k=0}^{N-1} Y_k \, e^{i2\pi \frac{k}{N} n}, \qquad 0 \le n \le N-1.$$

Then applying IDFT to the DFT of x recovers the original sequence: $y = x$.

Proof. Starting from the definition of the IDFT applied to Y_k:

$$y_n = \frac{1}{N} \sum_{k=0}^{N-1} \Big(\sum_{m=0}^{N-1} x_m \, e^{-i2\pi \frac{k}{N} m} \Big) \, e^{i2\pi \frac{k}{N} n} = \frac{1}{N} \sum_{m=0}^{N-1} x_m \sum_{k=0}^{N-1} e^{i2\pi \frac{k}{N} (n-m)}.$$

Define the inner sum

$$S_{n-m} = \sum_{k=0}^{N-1} e^{i2\pi \frac{k}{N}(n-m)}.$$

Let $r = e^{i2\pi \frac{n-m}{N}}$. Note that

- r is an N-th *root of unity*, meaning that $r^N = 1$.
- If $n - m \equiv 0(\pmod N)$, then $r = e^{i2\pi \cdot \text{integer}} = 1$.
- the sum S_{n-m} is a finite geometric series with ratio r.

Therefore

$$S_{n-m} = \sum_{k=0}^{N-1} r^k = \begin{cases} N, & r = 1 \ (\text{i.e., } n - m \equiv 0 \pmod N), \\ \dfrac{1 - r^N}{1 - r} = 0, & r \neq 1 \ (n - m \not\equiv 0 \pmod N). \end{cases}$$

As $r^N = e^{i2\pi(n-m)} = 1$, when $r \neq 1$ the numerator vanishes exactly. Therefore,

$$S_{n-m} = \begin{cases} N, & n = m, \\ 0, & n \neq m. \end{cases}$$

Plugging back into the expression for y_n,

$$y_n = \frac{1}{N} \sum_{m=0}^{N-1} x_m \, S_{n-m} = \frac{1}{N}(x_n \cdot N) = x_n.$$

Hence IDFT(DFT(x)) reproduces the original sequence x. □

In essence, the DFT "spreads out" the time-domain information across various frequencies, while the IDFT "collects" this spread information back into the time domain.

IDFT Computation

The Cooley-Tukey algorithm used for the FFT can also compute the IFFT. By simply changing the sign of the exponent in the twiddle factors and scaling the result by $\frac{1}{N}$, the algorithm performs the inverse transformation.

Alternatively, one can reuse the standard FFT routine to compute the inverse (as follows from property described in [10, §*Conjugation and Reversal*]). Let $Y = [Y_0, \ldots, Y_{N-1}]$ be the frequency-domain sequence. Define the conjugate sequence $Y'_k = \overline{Y_k}$. Then

$$\mathrm{FFT}(Y')_n = \sum_{k=0}^{N-1} \overline{Y_k}\, e^{-i2\pi \frac{n}{N} k} = \overline{\sum_{k=0}^{N-1} Y_k\, e^{i2\pi \frac{n}{N} k}} = \overline{N\, x_n} = N\, \overline{x_n}.$$

Conjugating again and dividing by N recovers the time-domain samples x_n. Hence the steps are:

1. Conjugate the input: $Y'_k = \overline{Y_k}$.
2. Compute the forward FFT: $x'_n = \mathrm{FFT}(Y')$.
3. Conjugate and normalize: $x_n = \overline{x'_n}/N$.

Algorithm 2 Recursive IFFT via FFT

```
function IFFT(Y[0 ... N − 1])
    Y' ← array of length N
    for k = 0 to N − 1 do
        Y'[k] ← conj(Y[k])
    end for
    x' ← FFT(Y')
    x ← array of length N
    for n = 0 to N − 1 do
        x[n] ← conj(x'[n]) / N
    end for
    return x
end function
```

Iterative FFT implementation

The recursive Cooley-Tukey FFT algorithm can be transformed into an efficient iterative implementation by performing the combining of even and odd terms in-place in a loop over $\log_2 N$ stages. At each stage s, $N/2^s$ DFTs of size 2^s are computed using results from the previous stage, thus eliminating recursive function calls and additional memory allocations. For a detailed description, see [6, §30].

Other FFT variants

- **Cooley-Tukey variations (general radix, DIT/DIF)**
  The algorithm introduced in §2.1.5 involved *radix 2* and *decimation in time (DIT)*. We can also have *decimation in frequency (DIF)* and we can divide into larger number of subproblems – different radices. In general, the algorithm recursively factorizes a DFT of size $N = N_1 N_2$ into N_1 DFTs of size N_2, multiplies by complex roots of unity, then performs N_2 DFTs of size N_1. Smaller of N_1 and N_2 is called the *radix*; if N_1 is the radix it is *decimation in time (DIT)*, if N_2 is the radix it is *decimation in frequency (DIF)*. There are also variation that involve mixing radices, e.g., *mixed-radix* or *split-radix* algorithms. See [13] for details.

- **Prime-factor algorithm**
  Exploits co-prime factorization of N to decompose the DFT similarly to Cooley-Tukey. See [10, §*Fast Fourier Transform (FFT) Algorithms*] for details.

- **Chirp Z-transform (Bluestein's algorithm)**
  Allows an arbitrary-length DFT. Expresses the DFT as a convolution (see §2.2.2) and computes the convolution via convolution theorem (cf. 2.6) using other FFT algorithm. *Rader's algorithm* introduces similar idea but specializes for prime sizes of DFT. This is an alternative to zero-padding the input and computing the FFT of power-of-two length. It provides a way to compute the exact size DFT but at the cost of increased computation time. This approach is useful when processing a signal whose period fits exactly in the segment size, as it helps avoid spectral leakage (cf. §2.1.4). See [10, §*Fast Fourier Transform (FFT) Algorithms*] for details.

2.2 Filtering

2.2.1 LTI systems and filters

Linear time-invariant (LTI) systems are the cornerstone of digital audio filters. An LTI system satisfies:

- **Linearity**: For any inputs x_1 and x_2 with outputs y_1 and y_2, the system response to $a\,x_1 + b\,x_2$ is $a\,y_1 + b\,y_2$.

- **Time invariance**: A shift in the input produces an identical shift in the output: if $x \to y$, then $x_{[n-k]} \to y_{[n-k]}$ for any integer k.

Filters are discrete LTI systems designed to modify an audio signal by emphasizing or attenuating specific frequency components. Their complete behaviour is described by the system's *impulse response*, h, which captures how a single-sample impulse

propagates through the filter. Applying this impulse response to a general input x is achieved via time-domain convolution, yielding the filtered output.

2.2.2 Convolution

Time-domain convolution is the primary method for implementing a linear time-invariant (LTI) filter. Given an input signal $x \in \mathbb{C}^N$ and a filter's impulse response $h \in \mathbb{C}^M$, the output signal $y \in \mathbb{C}^{M+N-1}$ is calculated by the convolution sum,

$$y_n = (x * h)(n) = \sum_{k=-\infty}^{\infty} x_k \, h_{n-k},$$

where we assume $x_k = 0$ for $k < 0$ or $k \geq N$ and $h_k = 0$ for $k < 0$ or $k \geq M$. This infinite-sum definition simplifies for finite-length signals as

$$y_n = \sum_{k=0}^{M-1} x_{n-k} \, h_k, \tag{2.4}$$

where $x_n = 0$ for $n < 0$ or $n \geq N$, yielding $y \in \mathbb{C}^{N+M-1}$.

Convolution is commutative. Indeed, substituting $m = n - k$ in (2.4) gives

$$(x * h)(n) = \sum_{k=0}^{M-1} x_{n-k} \, h_k = \sum_{m=n-M+1}^{n} x_m \, h_{n-m} = (h * x)(n).$$

In practice, for finite signals, we often express y as a sequence of dot products

$$y = \begin{bmatrix} x_{[0]}^T h \\ x_{[1]}^T h \\ \vdots \\ x_{[N+M-2]}^T h \end{bmatrix}, \quad x_{[n]}^T \coloneqq \big[x_n, x_{n-1}, \ldots, x_{n-M+1}\big],$$

where each $x_{[n]}^T h$ is a dot product of the reversed window of x with h. Convolution thus blends each input sample with weighted past samples defined by h, producing the filtered output.

2.2.3 Circular convolution and convolution via frequency domain

An alternative form is *circular convolution*, which assumes periodic extension of the two signals being convolved. The circular convolution of $x \in \mathbb{C}^N$ and $h \in \mathbb{C}^N$ is

$$y_n = (x \circledast h)(n) = \sum_{k=0}^{N-1} x_{(n-k) \bmod N} \, h_k.$$

Theorem 2.6 (Circular convolution corresponds to multiplication in the frequency domain)**.** *Circular convolution in the time domain corresponds to point-wise multiplication in the frequency domain. Specifically, let x and h be real or complex signals of length N, and let $X = DFT\Big(x\Big)$ and $H = DFT\Big(h\Big)$ denote their Discrete Fourier Transforms. Then, the circular convolution $y = x * h$ satisfies*

$$y = IDFT(X \cdot H).$$

Here $\cdot$ denotes point-wise multiplication.

Proof. Let $x, h \in \mathbb{C}^N$ and define $y = x \circledast h$ as their circular convolution. Consider the m-th component of y in the frequency domain,

$$Y_m = \sum_{n=0}^{N-1} y_n \, e^{-j2\pi mn/N}.$$

Substitute the definition of circular convolution:

$$Y_m = \sum_{n=0}^{N-1} \left(\sum_{k=0}^{N-1} x_k \, h_{(n-k) \bmod N} \right) e^{-j2\pi mn/N}.$$

Switching the order of summation:

$$Y_m = \sum_{k=0}^{N-1} x_k \left(\sum_{n=0}^{N-1} h_{(n-k) \bmod N} e^{-j2\pi mn/N} \right).$$

Make the substitution $l = (n-k) \bmod N$, so $n = (l+k) \bmod N$ and:

$$Y_m = \sum_{k=0}^{N-1} x_k \left(\sum_{l=0}^{N-1} h_l e^{-j2\pi m(l+k)/N} \right).$$

Now split the exponential:

$$Y_m = \sum_{k=0}^{N-1} x_k e^{-j2\pi mk/N} \left(\sum_{l=0}^{N-1} h_l e^{-j2\pi ml/N} \right).$$

This gives

$$Y_m = X_m \cdot H_m,$$

which shows that the DFT of the circular convolution is the pointwise product of the DFTs. Applying the inverse DFT yields $y = \mathrm{IDFT}(X \cdot H)$. □

2.2.4 FFT-based convolution

Direct time-domain convolution has complexity $O(NM)$ for signals of lengths N and M. FFT-based convolution reduces this cost by transforming convolution into multiplication via the DFT, achieving $O(L \log L)$ complexity, where L is the chosen FFT length.

Overlap-add

The idea is to break the full linear convolution into a sum of smaller convolutions. That is, if we write x as a sum of blocks of length K

$$x = \sum_i x_{[i]}, \qquad x_{[i]} = [0, \ldots, x_{iK}, \ldots, x_{(i+1)K-1}, 0, \ldots]$$

then the full convolution becomes

$$x * h = \sum_i x_{[i]} * h.$$

Time-domain aliasing

When we compute convolution via DFT, we always get circular convolution. If we do not zero-pad sufficiently, the periodic extension of the signals will overlap with themselves, corrupting the result. This is known as *time-domain aliasing*.

To avoid this, we zero-pad the input blocks $x_{[i]}$ and the filter h so that the circular convolution performed via FFT produces exactly the same output as linear convolution over the block.

The time-domain convolution result has length $L = N + M - 1$, N and M being the lengths of the inputs. If we want to convolve the same signals via frequency domain we need to make sure the frequency-domain result will be at least this length, otherwise it will get wrapped around. Thus we need to extend both input vectors to length L before performing the DTF of each.

FFT-convolution

The FFT-based convolution proceeds as follows:

1. **Partitioning**: Split the input signal x into non-overlapping blocks of length $L - M + 1$, where M is the length of the impulse response h. Denote the i-th block as $x_{[i]} \in \mathbb{C}^{L-M+1}$.

2. **Zero-padding**: Each $x_{[i]}$ is extended to length L by appending $M - 1$ zeros. The filter h is also zero-padded to length L.

3. **FFT and multiply**:

$$y_{[i]} = \mathrm{IDFT}\Big(\mathrm{DFT}\Big(x_{[i]}\Big) \cdot \mathrm{DFT}\Big(h\Big)\Big).$$

4. **Overlap-Add**: Each block $y_{[i]}$ has length L. The i-th block is placed at offset $i(L - M + 1)$ in the output, so that the last $M - 1$ samples are added (overlapped) with the next block.

Block size, FFT length and compexity

Let us assume that lengths of the inputs are N and M and $N \geq M$.

Let K be the chosen block length ($K \geq M$). To avoid aliasing, we must choose $L \geq M + K - 1$. In practice we usually round L up to the power of two for efficient FFT computation, so let $L \coloneqq \text{round\_up\_pow2}(K + M - 1)$. There are $\lceil N/K \rceil$ blocks to process. Each block requires two FFT and one IFFT of length L, costing $O(L \log L)$ each, plus $O(L)$ point-wise multiplication. Combining the result blocks at the end can be done in $O(N + M - 1) = O(N)$. Thus, the total complexity is

$$T(N, K) = O\big(N + \lceil N/K \rceil \, (L \log L)\big) = O\Big(\tfrac{N}{K} L \log L\Big).$$

Compare this to:

- **Direct convolution:** $O(NM)$.
- **Single-block FFT ($K = N$):** one transform of length $L_0 = \text{round\_up\_pow2}(N + M - 1)$ without overlap-add at the end, gives $O(L_0 \log L_0) = O(N \log N)$.
- **Block processing:** Asymptotically $O\Big(\tfrac{N}{K} L \log L\Big)$, which assuming $K \geq M$ gives $O\Big(N \log K\Big)$. This optimizes memory and latency for real-time applications by using smaller K, at the cost of increased constant factors.

In practice, K is chosen to balance computation (smaller L) and overhead (more blocks). For non real-time processing, and if N is not much larger that M, choosing $K = N$ (single FFT) is a good idea. For low-latency or streaming, K is set by latency constraints. However, this FFT-based method only outperforms direct convolution when the signals are sufficiently long; for short inputs, direct time-domain convolution remains faster.

Windowing

Optionally, a window function may be applied to each $x_{[i]}$ block before FFT to reduce spectral leakage (see §2.1.4).

2.2.5 Frame-by-frame processing in frequency domain

When analysing a long signal whose frequency content changes over time, we usually don't take the DFT of the whole signal at once. It would tell us *what* frequencies are present but not *when* they occur. Instead we process the input signal in short, possibly overlapping, segments (frames). We compute the DFT of each segment and this way we obtain a time-frequency representation that shows how the spectrum evolves over time. This is called the *Short-time fourier transorm (STFT)* (for mathematical details see [11, §*Short-time fourier transorm*]).

The STFT-based frame-by-frame processing of the input signal x is performed as follows:

1. Choose frame length L, analysis window w and hop size $R = L - [\textit{window overlap}]$.
2. Split the input signal x into frames of length L, each frame starting R samples after the previous one.
3. Multiply each frame by window:
$$x_{[m],n} = x_{n+mR}\, w_n \qquad n = 0, \ldots, L-1.$$
4. Compute the L-point DFT of each windowed frame:
$$X_{[m],k} = \sum_{n=0}^{L-1} x_{[m],n}\, e^{-i\,2\pi\,k\,n/L} \qquad k = 0, \ldots, L-1.$$
5. Store the resulting complex spectra $X_{[m]}$ in a 2D array.
6. Perform your frequency-domain processing on each frame's spectrum.
7. For each processed spectrum $\widetilde{X}_{[m]}$, compute the inverse DFT:
$$\tilde{x}_{[m],n} = \frac{1}{L} \sum_{k=0}^{L-1} \widetilde{X}_{[m],k}\, e^{i\,2\pi\,k\,n/L} \qquad n = 0, \ldots, L-1.$$
8. Construct the output signal y by overlap-adding the reconstructed frames:
$$y_n = \sum_{m=0}^{M-1} \tilde{x}_{[m],\,n-mR}, \quad n = 0, \ldots, N-1,$$
where we define $\tilde{x}_{[m],k} = 0$ for $k < 0$ or $k \geq L$ and N is the length of output.

Chapter 3

Noise reduction methods

In this chapter we present three practical methods for reducing noise in digital audio: the Wiener filter, the LMS adaptive filter, and spectral subtraction. Although most theoretical derivations in the literature operate on continuous, infinite-length signals and rely on stochastic signal models, our focus here is on concrete implementations that operate on finite frames of discrete, real-valued signals.

We assume that our input signals are discrete-time and real-valued, i.e., already sampled and quantized. This matches real-world scenarios, where audio processing is performed PCM data read from an audio file.

In the rest of this chapter the following notation will be used:

- Upper-case letters (e.g. X) denote signals in the frequency domain, and lower-case letters (e.g. x) denote the same signals in the time domain.
- x is the observed (noisy) input signal.
- d is the desired (clean) signal.
- e is the error signal (difference between desired and output).
- n is the noise signal.
- w represents the filter coefficients.
- $\hat{\square}$ denotes an estimate of $\square$.
- $\check{\square}$ denotes an inverted version of $\square$.
- $\square_{[m]}$ denotes the m-th frame (vector) of signal $\square$.
- $\square_n$ denotes the n-th sample (or frequency bin) of signal $\square$.
- $\square_{[m],n}$ denotes the n-th sample of the m-th frame.

This chapter is primarily based on [7].

3.0.1 Remarks on stochastic modeling

The theory of noise reduction filters like Wiener and adaptive filters is usually based on stochastic process modeling. This means that signals are treated as *random variables* that evolve over time. Such formalism allows the use of expected values and correlation functions in derivations.

However, fully developing the theory of stochastic processes is beyond the scope of this thesis. Instead, we adopt a practical engineering perspective:

- Treat each signal simply as a finite sequence of samples.
- Assume *local stationarity*: in implementation we slice the data into short frames within which the signal's statistical properties (mean, correlation) vary negligibly. This lets us apply the *mean*, *autocorrelation* and *cross-correlation* formulas introduced in the next section.

The stochastic model is well introduced in [7, §3] and in [14].

3.0.2 Statistical properties of signals

Mean value

An estimate of a *mean value* of N samples segment of a discrete-time signal $x \in \mathbb{R}^N$ is obtained as

$$E\Big[x\Big] \coloneqq \frac{1}{N} \sum_{m=0}^{N-1} x_m.$$

Signal energy

The *energy* of a discrete-time signal x of length N is defined as the sum of squared amplitudes,

$$E_x \coloneqq \sum_{n=0}^{N-1} x_n^2.$$

Signal power

The *power* of the signal is its energy averaged over the number of samples,

$$P_x \coloneqq \frac{1}{N} \sum_{n=0}^{N-1} x_n^2 = \frac{E_x}{N}.$$

Signal-to-Noise Ratio (SNR)

The *signal-to-noise ratio* (SNR) is a measure of the relative strength of a desired signal d to additive noise n for signal $x = d + n$.

It is defined as the ratio of signal power to noise power,

$$\text{SNR} \coloneqq \frac{P_d}{P_n}.$$

It is common to express SNR in logarithmic scale - in decibels - i.e.,

$$\text{SNR}_d B \coloneqq 10 \log_1 0(\text{SNR}).$$

In the frequency domain one often uses a *spectral SNR* defined per frequency bin k,

$$\xi(k) \;\coloneqq\; \frac{S_{dd}(k)}{S_{nn}(k)},$$

where $S_{dd}(k)$ and $S_{nn}(k)$ are the *power spectral densities* (see §3.0.2) of the clean signal and noise, respectively.

The time-domain SNR in (3.0.2) is a global measure over the entire signal duration, while the spectral SNR $\xi(k)$ describes the local ratio of signal to noise power in each frequency bin k.

Autocorrelation

Autocorrelation describes how similar a signal is to itself after a certain time shift.

For an N-sample segment of a discrete-time signal $x \in \mathbb{R}^N$, the linear estimate of autocorrelation is defined as

$$r_{xx}(k) \coloneqq \frac{1}{N} \sum_{m=0}^{N-1} x_m x_{m+k}.$$

where values outside the signal range are treated as zero, i.e., $x_{m+k} = 0$ for $m + k \notin [0, N-1]$.

Alternatively, one can define the circular autocorrelation

$$\overset{(\text{circ})}{r}_{xx}(k) \coloneqq \frac{1}{N} \sum_{m=0}^{N-1} x_m x_{(m+k) \bmod N}.$$

In this work, unless otherwise stated, we refer to the linear definition.

Cross-correlation

Cross-correlation measures the similarity between two different signals at different points in time.

The cross-correlation between two signals $x, y \in \mathbb{R}^N$ is estimated as

$$r_{xy}(k) \coloneqq \frac{1}{N} \sum_{m=0}^{N-1} x_m y_{m+k},$$

with $y_{m+k} = 0$ for out-of-range indices.

The circular cross-correlation is

$$\overset{(\text{circ})}{r}_{xy}(k) \coloneqq \frac{1}{N} \sum_{m=0}^{N-1} x_m y_{(m+k) \bmod N}.$$

Again, default is the linear variant.

Power Spectral Density (PSD)

The *Power Spectral Density* is Fourier transform of autocorrelation, and describes how the power of a signal is distributed over frequency.

Power spectral density of discrete-time signal $x \in \mathbb{R}^N$ can be estimates as

$$S_{xx}(k) = \frac{1}{N} |X_k|^2 \qquad k = 0, \ldots, N-1, \tag{3.1}$$

where X is frequency domain representation of x. The PSD is actually defined for infinite signals (N approaches infinity and X is DTFT (for discrete), or Fourier transform (for continuos) x, cf. §2.1.2). Taking $X = \text{DFT}\Big(x\Big)$ and using (3.1) is a common method of estimating PSD called *periodogram*. There are also methods (*Welch's method* and *Bartlett's method*) that involve averaging periodograms across frames. All three methods are described in detail in [14, §8].

As the PSD and autocorrelation form a Fourier-transform pair,an estimate of PSD can also be obtained as the DFT of circular autocorrelation

$$S_{xx}(k) = \sum_{n=0}^{N-1} \overset{(\text{circ})}{r}_{xx}(n) \cdot e^{-i2\pi kn/N} \qquad k = 0, \ldots, N-1. \tag{3.2}$$

Explanation: equivalence of both estimation methods. The equivalence of (3.1) and (3.2) follows from the property that circular convolution in time corresponds to multiplication in frequency (see theorem 2.6). Let us define the autocorrelation vector of a length-N sequence x as

$$r_{xx} \coloneqq \Big[\overset{(\text{circ})}{r}_{xx}(0), \overset{(\text{circ})}{r}_{xx}(1), \ldots, \overset{(\text{circ})}{r}_{xx}(N-1)\Big]^T,$$

where

$$\overset{(\text{circ})}{r}_{xx}(k) = \frac{1}{N} \sum_{m=0}^{N-1} x_m x_{(m+k) \bmod N}.$$

By definition this is exactly the circular convolution (see §2.2.3) of x with its time-reversed version $\check{x}_n = x_{(-n) \bmod N}$, scaled by $1/N$,

$$r_{xx} = \frac{1}{N}(x \circledast \check{x}).$$

Therefore the DFT of r_{xx} can be obtained using the method described in §2.2.4, as

$$\begin{aligned} S_{xx}(k) &= \mathrm{DFT}\Big(r_{xx}\Big)(k) \\ &= \mathrm{DFT}\Big(\frac{1}{N}(x \circledast \check{x})\Big)(k) \\ &= \frac{1}{N} \cdot \mathrm{DFT}\Big(x\Big)(k) \cdot \mathrm{DFT}\Big(\check{x}\Big)(k) \\ &= \frac{1}{N} \cdot X_k \cdot \overline{X}_k && (\textit{form 2.2: } \mathrm{DFT}\Big(\check{x}\Big)(k) = \overline{X_k}) \\ &= \frac{1}{N}|X_k|^2. && (\textit{form 2.1}) \end{aligned}$$

□

If we wish to estimate the PSD using the *linear* autocorrelation, avoiding edge overlap, we should zero-pad the sequence x to a length of at least $2N-1$, take the DFT of this zero-padded sequence, and use (3.1), but for range $k = -N+1, \ldots, N-1$.

Alternatively we can compute the linear autocorrelation $r_{xx}(n)$ for $n = -(N-1), \ldots, N-1$, and take the DFT

$$S_{xx}(k) = \sum_{n=-N+1}^{N-1} \overset{(\mathrm{circ})}{r_{xx}}(n) \cdot e^{-i2\pi kn/(2N-1)} \qquad k = -N+1, \ldots, N-1.$$

Cross Power Spectral Density (CPSD)

The *Cross Power Spectral Density* is the Fourier transform of cross-correlation.

For two discrete-time signals $x, y \in \mathbb{R}^N$, the CPSD can be estimated as the DFT of their cross-correlation,

$$S_{xy}(k) = \sum_{n=0}^{N-1} \overset{(\mathrm{circ})}{r_{xy}}(n) \cdot e^{-i2\pi kn/N}, \tag{3.3}$$

or, like PSD (3.1),

$$S_{xy}(k) = \frac{1}{N} X_k \cdot \overline{Y_k}, \tag{3.4}$$

where X and Y and DFTs of x and y respectively and $\overline{Y_k}$ denotes the complex conjugate of Y_k.

The equivalence of both definitions can be shown similarly to the explanation for PSD 3.0.2.

3.0.3 How to find an extremum of multivariable function?

For a single-variable function f, the derivative f' tells us how quickly the function changes at a given point. If we want to understand how a function $f(x_1, x_2, \ldots, x_n)$

changes in a multidimensional space, we need *partial derivatives*, which describe how the function changes with respect to each variable separately. The *gradient* is a vector containing these partial derivatives.

Definition of the Gradient

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a multivariable function that is differentiable (i.e., all partial derivatives exist). The gradient of this function is given by

$$\nabla f(x_1, x_2, \ldots, x_n) \coloneqq \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}. \tag{3.5}$$

Thus, the gradient consists of the partial derivatives of the function with respect to each variable.

Direction of the Steepest Ascent

The gradient points in the direction of the steepest increase of the function. This means that if you are at a point $(x_1, x_2, \ldots, x_n)$ and want to move in a way that maximally increases the function f, you should follow the direction of the gradient vector ∇f.

Zero Gradient - Critical Points

Just as for a single-variable function, where $f'(x) = 0$ indicates a potential minimum or maximum, in the case of multivariable functions,

$$\nabla f(x_1, x_2, \ldots, x_n) = 0$$

means that the function does not change in any direction. Such a point is a candidate for an extremum (a minimum, maximum, or saddle point).

Finding Extrema Using the Gradient

To find the minimum or maximum of a function $f(x_1, x_2, \ldots, x_n)$, we follow these steps:

1. **Compute the gradient** $\nabla f(x_1, x_2, \ldots, x_n)$.
2. **Find points where the gradient is zero**, solving the system of equations:

$$\frac{\partial f}{\partial x_1} = 0, \quad \frac{\partial f}{\partial x_2} = 0, \quad \ldots, \quad \frac{\partial f}{\partial x_n} = 0$$

3. Determine the nature of the point (minimum, maximum or saddle point), for example, by analysing second derivatives.

3.0.4 Mean square error

In signal processing, the *mean square error (MSE)* is a fundamental metric used to evaluate the performance of filters. It quantifies the average squared difference between the desired signal and the estimated signal produced by a system or filter.

Let $x \in \mathbb{R}^N$ be the *input signal* and $d \in \mathbb{R}^N$ the *desired signal*. Passing x through a filter produces an estimate $\hat{d}$ of d, and we define the *error signal* as

$$e \coloneqq d - \hat{d}. \tag{3.6}$$

To quantify performance, we use the *mean square error* (MSE)

$$E\Big[e^2\Big] \coloneqq \frac{1}{N} \sum_{m=0}^{N-1} e_m^2. \tag{3.7}$$

Minimizing the MSE yields filter parameters that, on average, make the estimate $\hat{d}$ follow the desired signal d as closely as possible, with larger deviations penalized more heavily by the squaring operation.

3.1 Wiener Filter

3.1.1 Wiener filter derivation

Objective

Let $x \in \mathbb{R}^N$ be the *input signal* and $d \in \mathbb{R}^N$ the *desired signal*.

The goal is to find a filter coefficient vector

$$w \coloneqq \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{p-1} \end{bmatrix},$$

that, when convolved with the input signal, produces an *estimate of the desired signal*

$$\hat{d} \coloneqq \begin{bmatrix} \hat{d}_0 \\ \hat{d}_1 \\ \vdots \\ \hat{d}_{N-1} \end{bmatrix}.$$

We'll call the length of the coefficient vector w — denoted by p — the *filter order*. To apply the filter, we extract overlapping slices of the input signal, each of length p. Let $x_{[n]}$ denote the slice starting at index n, ordered from x_n down to $x_{n-(p-1)}$,

$$x_{[n]} \coloneqq \begin{bmatrix} x_n \\ x_{n-1} \\ x_{n-2} \\ \vdots \\ x_{n-(p-1)} \end{bmatrix}.$$

To simplify notation, we define the input matrix X, whose rows are transposed slices of the input signal.

$$X \coloneqq \begin{bmatrix} x_{[0]}^T \\ x_{[1]}^T \\ \vdots \\ x_{[N-1]}^T \end{bmatrix} = \begin{bmatrix} x_0 & x_{-1} & x_{-2} & \dots & x_{1-p} \\ x_1 & x_0 & x_{-1} & \dots & x_{1-p} \\ \vdots & & & & \\ x_{N-1} & x_{N-2} & x_{N-3} & \dots & x_{N-p} \end{bmatrix}$$

The n-th sample of the output is given by the dot product of the filter coefficients and the corresponding slice of the input signal

$$\hat{d}_n \coloneqq \sum_{k=0}^{p-1} w_k x_{n-k} = w^T x_{[n]} = x_{[n]}^T w. \tag{3.8}$$

The full estimated signal is then computed as a matrix-vector product

$$\hat{d} = Xw.$$

The goal is to determine w such that the mean squared error (MSE) of the error signal $e = d - \hat{d}$ is the smallest.

Calculating MSE

Let us use (3.8) to expand the term for a single sample of the squared error

$$\begin{aligned} e_n^2 &= (d_n - w^T x_{[n]})^2 && (3.9)\\ &= d_n^2 - 2w^T x_{[n]} d_n + (w^T x_{[n]})^2 && (3.10)\\ &= d_n^2 - 2w^T x_{[n]} d_n + w^T x_{[n]} \cdot x_{[n]}^T w. && (3.11) \end{aligned}$$

Notice that the result of multiplying horizontal vector (w^T) by vertical vector ($x_{[n]}$) is a scalar, therefore $w^T x_{[n]} = x_{[n]}^T w$.

Now we can plug (3.11) into the expression for mean squared error (3.7)

$$\begin{aligned} E\Big[e^2\Big] &= \frac{1}{N}\sum_{n=0}^{N-1} e_n \\ &= \frac{1}{N}\sum_{n=0}^{N-1}\Big(d_n^2 - 2w^T x_{[n]} d_n + w^T x_{[n]} x_{[n]}^T w\Big) \\ &= \Big(\frac{1}{N}\sum_{n=0}^{N-1} d_n^2\Big) - 2w^T\Big(\frac{1}{N}\sum_{n=0}^{N-1} x_{[n]} d_n\Big) + w^T\Big(\frac{1}{N}\sum_{n=0}^{N-1} x_{[n]} x_{[n]}^T\Big)w. \end{aligned}$$

Let us introduce the following notations:

- $\frac{1}{N}\sum_{n=0}^{N-1} d_n^2$ is the autocorrelation of the desired signal at lag 0: $r_{dd}(0)$,
- $\frac{1}{N}\sum_{n=0}^{N-1} x_{[n]} d_n$ is the cross-correlation vector between the input signal and the desired signal: r_{xd}

$$\frac{1}{N}\sum_{n=0}^{N-1} x_{[n]} d_n = \frac{1}{N}\sum_{n=0}^{N-1}\begin{bmatrix} x_n \\ x_{n-1} \\ x_{n-2} \\ \vdots \\ x_{n-(p-1)} \end{bmatrix} d_n = \begin{bmatrix} \frac{1}{N}\sum_{n=0}^{N-1} x_n d_n \\ \frac{1}{N}\sum_{n=0}^{N-1} x_{n-1} d_n \\ \vdots \\ \frac{1}{N}\sum_{n=0}^{N-1} x_{n-p+1} d_n \end{bmatrix} = \begin{bmatrix} r_{xd}(0) \\ r_{xd}(1) \\ \vdots \\ r_{xd}(p-1) \end{bmatrix} =: r_{xd} \tag{3.12}$$

- $\frac{1}{N}\sum_{n=0}^{N-1} x_{[n]} x_{[n]}^T$ is autocorrelation matrix of the input signal: R_{xx}

$$\frac{1}{N}\sum_{n=0}^{N-1} x_{[n]} x_{[n]}^T = \begin{bmatrix} r_{xx}(0) & r_{xx}(1) & \cdots & r_{xx}(p-1) \\ r_{xx}(1) & r_{xx}(0) & \cdots & r_{xx}(p-2) \\ \vdots & \vdots & \ddots & \vdots \\ r_{xx}((p-1)) & r_{xx}((p-2)) & \cdots & r_{xx}(0) \end{bmatrix} =: R_{xx}, \tag{3.13}$$

  R_{xx} is a $p \times p$ matrix and each entry $r_{xx}(i-j)$ corresponds to the autocorrelation at lag $i-j$.

Substituting these notations into the expression for the MSE we obtain

$$E\Big[e_n^2\Big] = r_{dd}(0) - 2\, w^T\, r_{xd} + w^T R_{xx} w. \tag{3.14}$$

Minimizing MSE Using the Gradient Method

Let us have a look at the MSE equation in expanded form:

$$E\Big[e_n^2\Big] = r_{dd}(0) - 2\sum_{k=0}^{p-1} w_k\, r_{xd}(k) + \sum_{k=0}^{p-1}\sum_{i=0}^{p-1} w_k\, w_i\, r_{xx}(k-i).$$

Here we can notice that it is a quadratic function of w, therefore has single minimum.

To minimize the MSE with respect to the filter weight vector w, we differentiate the MSE expression with respect to w and set the resulting gradient equal to zero.

We define the gradient as

$$\frac{\partial}{\partial w} E\left[e_n^2\right] \coloneqq \begin{bmatrix} \frac{\partial\left(E\left[e_n^2\right]\right)}{\partial w_0} \\ \frac{\partial\left(E\left[e_n^2\right]\right)}{\partial w_1} \\ \vdots \\ \frac{\partial\left(E\left[e_n^2\right]\right)}{\partial w_{p-1}} \end{bmatrix}. \tag{3.15}$$

Using the fact that derivative of a sum is sum of derivatives we can rewrite (3.15) as

$$\frac{\partial}{\partial w} E\left[e_n^2\right] = \frac{\partial\left(r_{dd}(0)\right)}{\partial w} - \frac{\partial\left(2\, w^T r_{xd}\right)}{\partial w} + \frac{\partial\left(w^T R_{xx} w\right)}{\partial w}.$$

Since $r_{dd}(0)$ is a constant with respect to w, its derivative is zero. Differentiating the remaining terms and applying standard rules of multivariable calculus [9, §2], we note that:

- The derivative of the linear term $2\, w^T r_{xd}$ with respect to w is $2\, r_{xd}$.
- The derivative of the quadratic form $w^T R_{xx} w$ (with symmetric R_{xx}) with respect to w is $(R_{xx} + R_{xx}^T)w$, which equals to $2R_{xx}w$, as R_{xx} is symmetric.

Thus, we obtain

$$\frac{\partial}{\partial w} E\left[e_n^2\right] = -2\, r_{xd} + 2\, R_{xx} w. \tag{3.16}$$

Setting the gradient to zero, we have

$$-2\, r_{xd} + 2\, R_{xx} w = 0,$$

which after rearrangement gives

$$R_{xx} w = r_{xd}. \tag{3.17}$$

This equation is known as the **Wiener-Hopf equation**.

As R_{xx} is a *Toeplitz matrix* the equation (3.17) can be solved using *Levinson-Durbin Recursion* [14, §5]. If R_{xx} is invertible, a matrix inversion method, like *QR decomposition* can be used as well. However, for audio processing, more commonly used method is just using the frequency domain representation of wiener filter described in §3.1.2.

Note: Why Use the MSE?

In the derivation of the Wiener filter, the error signal is defined in vector form as

$$e = d - Xw,$$

where d is the desired signal vector, X is the input signal matrix, and w is the filter coefficient vector. In expanded form we have

$$\begin{bmatrix} e_0 \\ e_1 \\ \vdots \\ e_{N-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{N-1} \end{bmatrix} - \begin{bmatrix} x_0 & x_{-1} & x_{-2} & \dots & x_{1-p} \\ x_1 & x_0 & x_{-2} & \dots & x_{1-p} \\ \vdots & \vdots & \vdots & \ddots & \\ x_{N-1} & x_{N-2} & x_{N-3} & \dots & x_{N-p} \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{p-1} \end{bmatrix}.$$

The matrix equation behaves differently depending on the relation between the number of samples N and the filter length p:

- $N = p$: The system is square. If X is invertible, there is exactly one solution $w = X^{-1}d$, yielding $e = 0$.
- $N > p$: The system is *over-determined*. There are more equations than unknowns, so a solution that gives $e = 0$ usually doesn't exist. This is the common practical scenario, and the optimal solution is obtained by minimizing an average error cost function, such as the mean square error.
- $N < p$: The system is *under-determined* - may have infinitely many solutions that result in $e = 0$.

3.1.2 Wiener filter in frequency domain

Let us denote

- D desired signal d in frequency domain,
- X input signal x in frequency domain,
- W frequency-domain filter coefficients (i.e., *frequency response*),
- S_{xx} – power spectrum density of x,
- S_{xd} – cross-power spectrum density of x and d.

We can get the frequency-domain wiener filer equation by taking the fourier transform of the time domain equation (3.17).

Starting from

$$r_{xd} = R_{xx}w,$$

in expanded form

$$\begin{bmatrix} r_{xd}(0) \\ r_{xd}(1) \\ r_{xd}(2) \\ \vdots \\ r_{xd}(p-1) \end{bmatrix} = \begin{bmatrix} r_{xx}(0)w_0 + r_{xx}(1)w_1 + r_{xx}(2)w_2 + \ldots + r_{xx}(p-1)w_{p-1} \\ r_{xx}(1)w_0 + r_{xx}(0)w_1 + r_{xx}(1)w_2 + \ldots + r_{xx}(p-2)w_{p-1} \\ r_{xx}(2)w_0 + r_{xx}(1)w_1 + r_{xx}(0)w_2 + \ldots + r_{xx}(p-3)w_{p-1} \\ \vdots \\ r_{xx}(p-1)w_0 + r_{xx}(p-2)w_1 + r_{xx}(p-3)w_2 + \ldots + r_{xx}(0)w_{p-1} \end{bmatrix},$$

we note that the expression for a single row is circular convolution (cf. §2.2.3) of r_{xx} and w

$$\begin{aligned} r_{xd}(n) &= \sum_{m=0}^{p-1} r_{xx}((n-m) \bmod p) w_m \\ &= (r_{xx} \circledast w)(n), \end{aligned}$$

with $r_{xx} \coloneqq [r_{xx}(0), r_{xx}(1), \ldots, r_{xx}(p-1)]^T$.

We now take the Fourier transform of boh sides. By definition $\mathrm{DFT}\Big(r_{xd}\Big)$ is the cross-power spectrum S_{xd}. The $\mathrm{DFT}\Big(r_{xx} \circledast w\Big)$ is

$$\mathrm{DFT}\Big(r_{xx} * w\Big)(k) = \mathrm{DFT}\Big(r_{xx}\Big)(k) \cdot \mathrm{DFT}\Big(w\Big)(k) = S_{xx}(k)W_k,$$

because convolution in time domain corresponds to multiplication in frequency domain 2.6 and $\mathrm{DFT}\Big(r_{xx}\Big)(k)$ is by definition (cf. §3.0.2) $S_{xx}(k)$.

Finally we get

$$S_{xx}(k)W_k = S_{xd}(k) \qquad k = 0, \ldots, p-1. \tag{3.18}$$

3.1.3 Practical Wiener Filter for Additive Noise Reduction

We will assume the desired signal d is observed in additive noise n, thet is the input signal x is given by

$$x_m = d_m + n_m \qquad m = 0, \ldots, M-1$$

In the frequency domain we have

$$X_k = D_k + N_k,$$

where X, D, and N denote the DFTs of x, d, and n, respectively.

We will also assume that signals d and n are uncorrelated, i.e., $r_{dn}(k) = 0$ for all integer lags k.

We observe only the noisy signal x and a noise-only reference n, but the clean signal d is unknown. The objective is to recover d using frequency domain wiener filter, i.e., to find frequency response W, that produces an estimate $\hat{d}$ of d, that minimizes the mean-square error.

Lemma 3.1 (Wiener-Hopf Equations for Uncorrelated Signals)**.** *If the Wiener filter input x satisfies the additive model above, and d and n are uncorrelated, i.e., then the Wiener filter equation* (3.17) *simplify to*

$$w\,(R_{dd} + R_{nn}) = r_{dd},$$

and in the frequency domain (3.18) *to*

$$(S_{dd}(f) + S_{nn}(f))\,W_f = S_{dd}(f).$$

Proof. We split the proof into three parts:

1. Decomposition of r_{xx} and S_{xx}. By the definition of autocorrelation (cf. §3.0.2),

$$\begin{aligned} r_{xx}(k) &= \frac{1}{N}\sum_{m=0}^{N-1} x_m\, x_{m+k} && (3.19)\\ &= \frac{1}{N}\sum_{m=0}^{N-1} (d_m + n_m)(d_{m+k} + n_{m+k}) && (3.20)\\ &= \frac{1}{N}\sum_{m=0}^{N-1} (d_m d_{m+k} + d_m n_{m+k} + n_m d_{m+k} + n_m n_{m+k}) && (3.21)\\ &= r_{dd}(k) + \underbrace{r_{dn}(k)}_{0} + \underbrace{r_{nd}(k)}_{0} + r_{nn}(k) && (3.22)\\ &= r_{dd}(k) + r_{nn}(k), && (3.23) \end{aligned}$$

and taking the DFT gives

$$S_{xx}(f) = S_{dd}(f) + S_{nn}(f),$$

where S_{xx}, S_{dd}, S_{nn} are power spectral densities (see §3.0.2) of x,d,n respectively.

2. Cross-correlation between x and d. Using the cross-correlation definition (see §3.0.2),

$$\begin{aligned} r_{xd}(k) &= \frac{1}{N}\sum_{m=0}^{N-1} x_m\, d_{m+k} = \frac{1}{N}\sum_{m=0}^{N-1} (d_m + n_m) d_{m+k} && (3.24)\\ &= r_{dd}(k) + \underbrace{r_{nd}(k)}_{0} = r_{dd}(k), && (3.25) \end{aligned}$$

and similarly in the frequency domain, using the definition of CPSD (cf. §3.0.2),

$$S_{xd}(f) = S_{dd}(f).$$

3. Substitution into the Wiener-Hopf equations. The time-domain Wiener-Hopf system is given by (3.17)

$$w\, R_{xx} = r_{xd},$$

and its frequency-domain counterpart (3.18) by

$$S_{xx}(f)\, W_f = S_{xd}(f).$$

Substituting (3.23) and (3.25) yields

$$w\, (R_{dd} + R_{nn}) = r_{dd},$$

and substituting (3.1.3) and (3.1.3) in frequency gives

$$\left(S_{dd}(f) + S_{nn}(f)\right) W_f = S_{dd}(f).$$

□

Looking at the Wiener-Hopf equations, we face the practical challenge that the true spectrum $S_{dd}(f)$ of the clean signal d is unavailable. However, since we can estimate $S_{xx}(f)$ and we have access to a noise-only reference allowing us to estimate $S_{nn}(f)$, we can compute an estimate of the desired spectrum from (3.1.3)

$$\hat{S}_{dd}(f) = S_{xx}(f) - S_{nn}(f). \tag{3.26}$$

Substituting $\hat{S}_{dd}(f)$ into the Wiener filter formula from the lemma 3.1 yields the practical frequency response:

$$W_f = \frac{\hat{S}_{dd}(f)}{\hat{S}_{dd}(f) + S_{nn}(f)} = \frac{S_{xx}(f) - S_{nn}(f)}{S_{xx}(f)}.$$

Practical Steps for Wiener Denoising

Given a noisy observation x and noise-only fragment n, we carry out additive noise reduction using the Wiener filter as follows:

1. **STFT:** Split x and n into frames, apply a suitable window function to each frame, and compute the FFT of each windowed frame.
2. **Estimate Noise PSD S_{nn}:** Get S_{nn} by one of two approaches:
   - *Noise only is same length as the signal:* For each frame of noise, compute its periodogram and use this as S_{nn} while processing the respective frame of x.
   - *Small noise sample, but with assumption that noise is stationary:* Compute periodograms for all noise frames, then average them to obtain a single estimate of S_{nn}.

3. **Frame-wise Wiener Filtering:** We now consider a single noisy frame and denote its spectrum simply by X, omitting the frame index for clarity. The following steps are performed for each frame independently:

   - **Estimate Signal PSD S_{dd}:** Compute the periodogram S_{xx} (see §3.0.2). Then estimate the clean-signal PSD using (3.26). Note that PSD is by definition non-negative; to preserve this property
   $$S_{dd}(f) = \max\big(S_{xx}(f) - S_{nn}(f),\, 0\big).$$
   - **Compute Wiener Response $W(k)$:**
   $$W(k) = \frac{S_{dd}(k)}{S_{dd}(k) + S_{nn}(k) + \epsilon},$$
   where ϵ is a small constant to prevent division by zero.
   - **Apply Filter:**
   $$\hat{D}_k = W_k X_k.$$

4. **Inverse STFT:** For each filtered frame of the result, compute the inverse FFT and then use the overlap-add method to reconstruct the time-domain estimate $\hat{d}$.

3.2 Adaptive Filtering: LMS

Adaptive filtering refers to a class of algorithms in which the filter's coefficients are not fixed in advance but instead are updated continuously based on the incoming signal and a performance criterion. This allows the filter to track changes in the signal or noise environment in real time.

In this section we present the Least Mean Squares (LMS) algorithm. LMS is closely related to the Wiener filter – both aim to minimize the mean squared error between the filter output and a desired signal. However, unlike the Wiener solution, which requires knowledge of signal and noise statistics, LMS attempts to learn the optimal filter weights on the fly, based on instantaneous values.

3.2.1 Gradient descent

We aim to minimize a cost function (typically, an error function). Since we do not have access to the full signal at once, we cannot compute the minimum analytically. Instead, we use an iterative method, gradually moving toward lower function values.

Let $J : \mathbb{R}^p \to \mathbb{R}$ denote the cost function we wish to minimize. At each step, we update the weight vector w,

$$w_{\text{new}} = w - \mu \cdot \frac{dJ(w)}{dw},$$

where

- μ is the *step size* (also called learning rate),
- $\frac{dJ(w)}{dw}$ is the *step direction* - gradient (see (3.5)); we move in the opposite direction to it to descend the cost surface (hence the minus sign).

For a scalar function (e.g., single-tap filter), this is intuitive.

- If w is to the left of the minimum, the slope is negative, so subtracting the negative gradient increases w - we move right toward the minimum.
- If w is to the right of the minimum, the slope is positive, and subtracting it decreases w - again, we move toward the minimum.

This intuition extends to multivariable functions, where the gradient points in the direction of steepest ascent, and we step in the opposite direction.

3.2.2 Finding minimal MSE using gradient descent

We now apply gradient descent to minimize the mean square error.

We will use MSE as the cost function

$$J(w) \coloneqq E\Big[e^2\Big] = E\Big[(d - Xw)^2\Big].$$

The update rule becomes

$$w_{[n+1]} = w_{[n]} - \mu \cdot \frac{\partial E\Big[e^2\Big]}{\partial w_{[n]}}.$$

In the section on the Wiener filter §3.1.1, we derived the gradient of the MSE (3.16) as

$$\frac{\partial}{\partial w_{[n]}} E\left[e^2\right] = -2\, r_{xd} + 2\, R_{xx} w_{[n]}.$$

Substituting this into the update rule, we get

$$\begin{aligned} w_{[n+1]} &= w_{[n]} - \mu \cdot \Big(-2\, r_{xd} + 2\, R_{xx} w_{[n]}\Big) \\ &= w_{[n]} + 2\mu \Big(r_{xd} - R_{xx} w_{[n]}\Big). \end{aligned}$$

This update moves the filter coefficients in the direction that reduces the mean squared error, but it still requires knowledge of R_{xx} (3.13) and r_{xd} (3.12).

3.2.3 LMS algorithm derivation

The gradient descent update derived above still depends on the autocorrelation matrix R_{xx} and the cross-correlation vector r_{xd}. This is impractical, as computing them requires knowledge of the full signal statistics, which is generally not available in real-time systems.

The **Least Mean Squares (LMS)** algorithm solves this by replacing the true gradient of the mean squared error with the gradient of the **instantaneous squared error**. This allows for online adaptation using only current signal values.

The new cost function becomes

$$J(w_{[n]}) \coloneqq (e_n)^2 = \left(d_n - w_{[n]}^T x_{[n]}\right)^2.$$

We do not take the expectation, just the squared error at time step n.

The update rule is

$$\begin{aligned} w_{[n+1]} &= w_{[n]} - \mu \cdot \frac{\partial\left(e_n^2\right)}{\partial w_{[n]}} \\ &= w_{[n]} - \mu \cdot \frac{\partial\left(d_n - w_{[n]}^T x_{[n]}\right)^2}{\partial w_{[n]}}. \end{aligned}$$

To compute the gradient, we use the *chain rule for derivatives* [9, §2.5]. Let

$$\begin{aligned} e_n &= d_n - w_{[n]}^T x_{[n]}, \\ J(w_{[n]}) &= e_n^2. \end{aligned}$$

Then

$$\begin{aligned} \frac{\partial J}{\partial e_n} &= 2e_n, \\ \frac{\partial e_n}{\partial w_{[n]}} &= -x_{[n]}, \end{aligned}$$

$$\begin{aligned} \frac{\partial J(w_{[n]})}{\partial w_{[n]}} &= \frac{\partial J}{\partial e_n} \cdot \frac{\partial e_n}{\partial w_{[n]}} \\ &= 2e_n \cdot (-x_{[n]}) = -2e_n x_{[n]}. \end{aligned}$$

Substituting this into the adaptation rule we get

$$w_{[n+1]} = w_{[n]} - \mu \cdot (-2e_n x_{[n]}) = w_{[n]} + 2\mu\, e_n x_{[n]}.$$

In practice, the factor of 2 is absorbed into the step size, introducing $\mu' = 2\mu$, and leading to the final form of the LMS update equation

$$w_{[n+1]} = w_{[n]} + \mu' e_n x_{[n]}. \tag{3.27}$$

This forms the basis of the LMS algorithm, in which the filter coefficients are updated iteratively using only the current input vector $x_{[n]}$ and error e_n.

3.2.4 LMS implementation

Using (3.27) we can write LMS algorithm as follows:

Algorithm 3 LMS adaptive filter

function LMS(x, d, μ, p) ▷ x: input, d: desired, μ: step, p: filter size
  $N \leftarrow length(x)$
  Allocate output arrays y, e of length N and w of size p
  for $n = 0, \ldots, N-1$ **do**
    $x_{[n]} \leftarrow [x_n, x_{n-1}, x_{n-2}, \ldots, x_{n-(p-1)}]^T$
    $y_n \leftarrow w^T x_{[n]}$ ▷ one-sample "convolution" of x with w
    $e_n \leftarrow d_n - y_n$
    $w \leftarrow w + \mu\, e_n\, x_{[n]}$
  end for
  return y, e, w ▷ y: filter output, e: error signal, w: filter weights
end function

In practise w is either initialized with zeros or random, and input vectors x and d are padded with zeros at the front, so that we don't access unexistent samples.

Notice that each iteration computes exactly one sample of the "convolution" between x and w, so the full LMS algorithm is essentially a modified convolution algorithm, that changes filter coefficients each iteration.

3.2.5 LMS adaptive filter for additive noise reduction

Normally we do not have access to the clean signal d – the goal is to recover it. Instead, we have:

$$x = d + n, \quad \text{and} \quad n \quad \text{(noise-only reference)}.$$

We can recover an estimate of d by feeding:

- the noise reference n as the filter input x, and
- the noisy mixture x as the "desired" signal d.

Then the algorithm adapts its weights to produce

$$y \approx \widetilde{n},$$

which is the filter's best estimate of how n appears in x. The LMS error then becomes

$$e = x - y = (d + n) - \widetilde{n} \approx d.$$

Thus, by subtracting the estimated noise from the noisy mixture, the error signal e converges to the clean signal d.

3.3 Spectral Subtraction

The *spectral subtraction* method aims to restore the magnitude or power spectrum of a signal corrupted by additive noise by subtracting an estimate of the noise spectrum from the noisy signal spectrum.

We model the noisy input signal x as the sum of a clean (desired) signal d and additive noise n:

$$x = d + n. \tag{3.28}$$

We split x into overlapping frames of length L. The m-th frame is

$$x_{[m]} = d_{[m]} + n_{[m]}, \tag{3.29}$$

where

$$x_{[m]} = \left[x_{[m],0}, x_{[m],1}, \ldots, x_{[m],L-1}\right]^T$$

is the time-domain vector of frame m. After windowing each frame (see §2.1.4), we take its discrete Fourier transform:

$$X_{[m]} = D_{[m]} + N_{[m]}, \tag{3.30}$$

where $X_{[m]}, D_{[m]}, N_{[m]}$ are the L-point DFTs of the noisy frame, clean frame, and noise frame, respectively. The validity of $\mathrm{DFT}\left(d_{[m]}+n_{[m]}\right) = \mathrm{DFT}\left(d_{[m]}\right)+\mathrm{DFT}\left(n_{[m]}\right)$ follows from the linearity properties recalled in 2.4.

Let $|X_{[m],k}|$ denote the magnitude of the k-th DFT coefficient of the noisy frame, and let $\hat{N}_k$ be our time-averaged noise magnitude at frequency bin k (estimated as in §3.3.1). The general spectral subtraction rule (for exponent b) is

$$A^b_{[m],k} = |X_{[m],k}|^b - \alpha_k \, \hat{N}^b_k, \tag{3.31}$$

where:

- $b = 1$ yields magnitude spectral subtraction, and $b = 2$ yields power spectral subtraction (see §3.3.4);
- α_k is a parameter that controls the subtraction strength (with $\alpha_k = 1$ for full subtraction (the usual choice) and $\alpha_k > 1$ for over-subtraction).

Once $A_{[m],k}$ is computed, we restore a complex spectrum by combining it with the noisy phase $\angle X_{[m],k}$:

$$\hat{D}_{[m],k} = A_{[m],k} \, e^{i\angle X_{[m],k}}, \tag{3.32}$$

and recover the time-domain frame via the inverse DFT:

$$\hat{d}_{[m],l} = \sum_{k=0}^{L-1} \hat{D}_{[m],k} \, e^{i\frac{2\pi k l}{L}}, \quad l = 0, \ldots, L-1. \tag{3.33}$$

Overlap-add (see §2.2.4) of all reconstructed frames yields the final enhanced signal.

3.3.1 Noise estimation

We assume access to M frames of "pure" noise $n_{[m]}$, and we assume that the noise is approximately stationary (it's characteristics do not chande over time). For each noise frame we compute its DFT magnitude $|N_{[m],k}|$, then form a time-averaged noise spectrum:

$$\hat{N}_k^b = \frac{1}{M} \sum_{m=0}^{M-1} |N_{[m],k}|^b, \tag{3.34}$$

where $b = 1$ or 2 matches the exponent used in (3.31).

The noise frames can be acquired either by identifying segments when the desired signal is not present or using a reference microphone thet records noise only (scenario as in LMS setup §3.2.5).

3.3.2 Non-negative spectrum mapping

Subtracting $\alpha_k |\tilde{N}_k|^b$ from $|X_{[m],k}|^b$ can yield negative values, which are not physically meaningful for a magnitude or power spectrum. We therefore apply a mapping function

$$f(A_{[m],k}) = \begin{cases} A_{[m],k}, & A_{[m],k} > \beta |X_{[m],k}|, \\ \beta |X_{[m],k}|, & \text{otherwise}, \end{cases} \tag{3.35}$$

where $\beta \ll 1$ is a small constant (e.g. $\beta = 0.01$). In practice, we replace $A_{[m],k}$ by $f(A_{[m],k})$ in (3.32) before phase restoration.

3.3.3 Distortions

Spectral subtraction is subject to several *processing distortions* that degrade the restored signal. The sources of these include

- *Noisy estimate of noise*: We estimete the noise spectrum $|N_{[k],m}|$ using an average. In reality $|N_{[k],m}|$ jumps up and down, so when we subtract it we sometimes leave bits of noise or remove too much signal.
- *Clipping negative values*: If subtraction gives a negative magnitude, we force it up to a small floor level. That creates random little tones called *musical noise*.

3.3.4 Power and magnitude spectrum subtraction

Spectral subtraction is usually applied either to the magnitude spectrum ($b = 1$) or to the power spectrum ($b = 2$) in (3.31). The two variants are:

Magnitude subtraction ($b = 1$):

$$A_{[m],k} = |X_{[m],k}| - \alpha_k \tilde{N}_k. \tag{3.36}$$

Power subtraction ($b = 2$):

$$A^2_{[m],k} = |X_{[m],k}|^2 - \alpha_k \tilde{N}^2_k. \tag{3.37}$$

Power subtraction ($b = 2$) tends to remove low-energy noise more aggressively, which can achieve lower residual noise levels at the cost of more pronounced "musical noise" artifacts if the noise estimate is inaccurate. Magnitude subtraction ($b = 1$) usually produces smoother, more natural residuals but may leave more noise behind.

3.3.5 Summary of the Spectral Subtraction Algorithm

The following steps summarize the complete spectral subtraction procedure:

1. Split the noisy time-domain signal x into frames and apply a suitable window w to each frame, then compute STFT (see §2.2.5)
2. Estimate the noise spectrum as in §3.3.1
3. Subtract the noise estimate using (3.36) or (3.37) applying non-negative mapping (3.35).
4. Combine with phase (3.32) to reconstruct the complex spectrum.
5. Inverse STFT and overlap-add (see §2.2.5).

3.3.6 Spectral Subtraction vs. Wiener Filtering

From a theoretical standpoint, both spectral subtraction and Wiener filtering aim to suppress additive noise in the short-time Fourier domain by attenuating frequency bins where noise dominates. The Wiener gain is

$$H_W(k) \;=\; \frac{S_{dd}(k)}{S_{dd}(k) + S_{nn}(k)}.$$

In practice we cannot access the true clean-signal spectrum $S_{dd}(k)$, so we estimate it using (3.1.3) as $\hat{S}_{dd}(k) = \max\left(S_{xx}(k) - S_{nn}(k), 0\right)$, where $S_{xx}(k)$ and $S_{nn}(k)$ are the periodogram-based PSD estimates of the noisy observation $x = d + n$ and a noise-only fragment n. The resulting practical Wiener gain becomes

$$W_k = \max\Big(1 - \frac{S_{nn}(k)}{S_{xx}(k)}, 0\Big).$$

Spectral subtraction in the power-domain, with subtraction factor $\alpha = 1$ and no flooring ($\beta = 0$), computes

$$|D_k|^2 = \max\left(|X_k|^2 - \hat{N}_k^2, 0\right) = |X_k|^2 \max\Big(1 - \frac{\hat{N}_k^2}{|X_k|^2}, 0\Big).$$

Noting that $\hat{N}_k^2$ and $|X_k|^2$ are actually the estimates of $S_{nn}(k)$ and $S_{xx}(k)$, we can say that its power-domain gain is

$$H_k = \max\Big(1 - \frac{\hat{N}_k^2}{|X_k|^2}, 0\Big) \approx \max\Big(1 - \frac{S_{nn}(k)}{S_{xx}(k)}, 0\Big) = W_k.$$

Thus in the special case of power subtraction with $\alpha = 1$, $\beta = 0$, and identical PSD estimates, both methods yield exactly the same gain

$$H_k = W_k = \max\Big(1 - \frac{S_{nn}(k)}{S_{xx}(k)}, 0\Big).$$

The only implementation difference is that spectral subtraction rescales the magnitude $\sqrt{|D_k|^2}$ before re-combining with the original phase, whereas Wiener filtering multiplies the full complex spectrum by W_k.

Intuitive interpretation. The ideal Wiener gain is derived from the signal-to-noise ratio (SNR) in each bin:

$$W_k \;=\; \frac{S_{dd}(k)}{S_{dd}(k) + S_{nn}(k)} \;=\; \frac{\xi(k)}{1 + \xi(k)}, \quad \xi(k) = \frac{S_{dd}(k)}{S_{nn}(k)}.$$

Under high SNR ($\xi(k) \gg 1$), $W_k \approx 1$, under low SNR ($\xi(k) \ll 1$), $W_k \approx 0$.

At frequencies where the estimated SNR is high, both filters apply a gain close to unity, preserving the signal. At frequencies where noise dominates, the gain approaches zero, attenuating noise.

Spectral subtraction can be viewed as a flexible, heuristic approximation of the ideal Wiener solution for infinitely long, stationary stochastic processes. Likewise, the "Wiener filter" implementation presented in §3.1.3 is itself an estimate of the theoretical filter, since it replaces exact power spectral densities by frame-averaged PSD estimates In practice, both approaches approximate the same optimal noise-suppression strategy under different implementation trade-offs.

Chapter 4

Experiments & results

4.1 Implementation and experiments description

4.1.1 Test signals

Generated signals

All generated signals were sampled at $f_s = 44100$ Hz, which is the standard sample rate for cd-format.

- *Sine*: A pure sinusoidal signal of a single frequency f was generated by sampling the function
$$x_m \leftarrow \sin\left(2\pi f \frac{m}{f_s}\right).$$
For a desired signal length N, the time increment is $\frac{m}{f_s}$ for $m = 0, 1, \ldots, N-1$.
- *Chord*: A major chord is created by summing three sinusoidal components: the root frequency f_{root}, its major third, and its perfect fifth. The major-third frequency is
$$f_{\text{third}} = f_{\text{root}} \times 2^{4/12},$$
and the perfect-fifth frequency is
$$f_{\text{fifth}} = f_{\text{root}} \times 2^{7/12}.$$
Each component is generated as a sinusoid (as described above), all of length N samples, and the three waveforms are added sample-by-sample:
$$x_m \leftarrow \sin\left(2\pi f_{\text{root}} \frac{m}{f_s}\right) + \sin\left(2\pi f_{\text{third}} \frac{m}{f_s}\right) + \sin\left(2\pi f_{\text{fifth}} \frac{m}{f_s}\right).$$
- *White noise*: White noise is a random signal with a flat frequency spectrum. In practice, it was generated by drawing each sample randomly from a uniform distribution from $[-1, 1]$. For N samples
$$x_m \leftarrow \text{RandUniform}(-1, 1), \quad m = 0, 1, \ldots, N-1.$$

- *Random-tone*: A signal that consists of a sinusoid whose instantaneous frequency changes at fixed time intervals. Let us define a change period T_{change} (in seconds), which would correspond to length of a segment $L_{\text{change}} = T_{\text{change}} \times f_s$, in which we use one frequency. We initialize a random starting frequency f_{curr} drawn uniformly from a given range (200 Hz to 600 Hz). Then every L_{change} samples we choose new frequency, so if current sample index is m,

  $$\text{if } m \bmod L_{\text{change}} = 0, \quad f_{\text{curr}} \leftarrow \text{RandUniform}(200, 600).$$

  The m-th sample is

  $$x_m \leftarrow \sin\left(2\pi f_{\text{curr}} \frac{m}{f_s}\right).$$

  As a result, the tone "jumps" randomly to a new frequency every T_{change} seconds.

Recorded signals

All recordings were captured using `arecord`, a command-line utility for audio capture under Linux, with the following command:

```
arecord -f S16_LE -c 1 -r 44100 <output_filename>.wav
```

This specifies 16-bit little-endian format (`S16_LE`), a sample rate of 44100 Hz, and a single channel.

- `speech`: A short fragment of spoken language.
- `guitar`: An acoustic guitar playing.
- `microwave`: Noise emitted by a microwave oven.
- `hairdrier`: A hairdryer was recorded while its power setting was varied between low and high airflow modes. This produced a non-stationary noise resembling white noise, with amplitude changes corresponding to the motor speed.

Mixing signals

Mixing with noise: To simulate a noisy environment, clean signal samples d are mixed with noise samples n at a specified noise level ℓ. The noisy signal samples x are formed as

$$x_m \leftarrow d_m \; + \; \ell \, n_m,$$

where ℓ controls the noise amplitude relative to the clean signal. A higher ℓ yields a lower signal-to-noise ratio, with $\ell < 1$ meaning that desired signal dominates and $\ell > 1$ meaning that noise dominates.

Varying noise level: A mechanism to modify the noise, so that it's level will change with time, when it is mixed with the clean signal. First, the noise vector n is passed through a function that divides the noise samples into non-overlapping blocks of L samples ($L = 44100$, corresponding to 1 second). For each block b, a random scale factor $\ell_b \in [0, 1]$ is drawn and applied to all noise samples in that block, i.e.,

$$\tilde{n}_m \leftarrow \ell_b \, n_m, \quad m \in \big[bL,\, (b+1)L - 1\big].$$

Then, in test where variable noise was used, the scaled noise $\tilde{n}_m$ is mixed with the clean signal d_m. This approach was applied in one experiment to produce a non-stationary noise.

4.1.2 Test cases

The names of the test cases (e.g., `sine_white`) appear on the corresponding plots in results section. In these labels, the prefix before the underscore denotes the desired (clean) signal type, and the suffix denotes the noise (unwanted) signal type.

The input for the algorithm was *mixed signal* and *noise signal* as it was before any scaling.

Sine wave with white noise (`sine_white`)

- *Desired signal*: Sinusoid at 440.0 Hz.
- *Noise signal*: White noise.
- *Mixed signal*: Noise samples were scaled by $\ell = 1.1$ before being added to clean samples.
- *Duration*: 6 seconds

This ratio $\ell = 1.1$, which means "a little bit more noise than signal", was chosen to challenge the algorithms under a noise-dominated condition.

Separating two chords (`chord_chord`)

- *Desired signal*: Chord with root frequency 500 Hz (sinusoids at 500 Hz, ≈ 630 Hz, ≈ 749 Hz).
- *Noise signal*: Chord with root frequency 600 Hz (sinusoids at 600 Hz, ≈ 756 Hz, ≈ 898 Hz).
- *Duration*: 6 seconds
- *Mixed signal*: Summed at 1:1 ratio, i.e. $\ell = 1$.

The intention was, that the chords are hard, but possible, to separate. The highest harmonic of the desired chord (749 Hz) is very close to the second harmonic of the noise chord (756 Hz). With an FFT frame size of 2^{14} and a sampling rate of 44100 Hz, the bin separation is about 2.7 Hz, allowing - at least in theory - possible separation. With small frame (e.g. 1024 samples) these frequencies will leak to the same bins, so removing the unwanted chord using FFT-based methods may fail, or artifacts may appear in the output.

Speech with varying-tone noise (`speech_vartones`)

- *Desired signal*: Recorded speech.
- *Noise signal*: Random-tone signal with frequency changes every 1.5 seconds.
- *Duration*: 20 seconds
- *Mixed signal*: The noise amplitude was further scaled by a random factor changing every 1 second. Noise samples n_m (after random time-varying scaling) were multiplied by $\ell = 0.5$ before mixing.

This case tests algorithm performance under time-varying noise when the reference noise is similar - but not identical - to the mixed noise. This is a kind of scenario for which adaptive filters are designed. Note that in this case the desired signal is spectrally complex while the noise is simple (single frequency at a time).

Varying-tone signal with hairdryer noise (`tones_hairdrier`)

- *Desired signal*: Random-tone with frequency changes every 0.5 seconds.
- *Noise signal*: Recorded hairdryer noise.
- *Duration*: 20 seconds
- *Mixed signal*: Noise and desired summed at 1:1 ratio.

Another test where both the desired signal and the noise are time-varying, but in this case, the desired tone is relatively simple (single sinusoid), while the hairdryer noise is spectrally complex.

Speech with guitar noise (`speech_guitar`)

- *Desired signal*: Recorded speech.
- *Noise signal*: Guitar recording.
- *Duration*: Approximately 45 seconds.

- *Mixed signal*: Guitar signal was scaled by $\ell = 0.4$ before being added to speech.

Simulates extracting vocals from a song accompanied by guitar.

Guitar with speech noise (`guitar_speech`)

- *Desired signal*: Guitar recording.
- *Noise signal*: Recorded speech.
- *Duration*: Approximately 45 seconds.
- *Mixed signal*: Speech signal was scaled by $\ell = 0.4$ before being added to guitar.

Inverse of the previous case: extracting the instrument sound in the presence of speech. Expected to be difficult for the tested algorithms.

Speech with real microwave noise (`speech_microvave`)

- *Noise signal*: Recorded microwave oven noise alone.
- *Mixed signal*: Speech with microwave oven hum in the background.

No computer-based mixing. First, a segment of microwave-only noise was recorded. then speech was recorded in the same environment with the microwave running, producing the mixed signal.

This is a fully realistic scenario where only the noisy mixture and a separate noise-only reference are available.

4.1.3 Quality measuring methods

As a measure of denoising effectiveness, I used SNR improvement expressed in decibels. SNR is described in §3.0.2, while SNR improvement compares how the signal-to-noise ratio changed as a result of processing. It was computed as

$$\Delta\mathrm{SNR}_{\mathrm{dB}} = \mathrm{SNR}_{\mathrm{dB,\,out}} - \mathrm{SNR}_{\mathrm{dB,\,in}}, \tag{4.1}$$

where

$$\mathrm{SNR}_{\mathrm{dB,\,in}} = 10\log_{10}\Big(\frac{\sum_{n=0}^{N-1} d_n^2}{\sum_{n=0}^{N-1}(d_n - x_n)^2}\Big), \quad \mathrm{SNR}_{\mathrm{dB,\,out}} = 10\log_{10}\Big(\frac{\sum_{n=0}^{N-1} d_n^2}{\sum_{n=0}^{N-1}(d_n - y_n)^2}\Big),$$

and x is the input (mixed) signal, y is the output (processed) signal, d is the clean (desired) signal.

Interpretation of SNR improvement

- A positive $\Delta\text{SNR}_{\text{dB}}$ indicates that the processed signal has a higher SNR than the original noisy signal (i.e., noise was reduced).
- A negative $\Delta\text{SNR}_{\text{dB}}$ indicates degradation (i.e., the processing introduced more distortion or noise than it removed).
- If the noise power was reduced by a factor of two, then $\Delta\text{SNR}_{\text{dB}} = 10\log_{10}(2) \approx 3\,\text{dB}$.
- For human perception, an increase of about 10 dB is perceived as "twice as loud."
- A poor SNR improvement result calculated in this way does not necessarily imply poor perceptual quality. For example, if the output has the same frequency content but is phase-shifted relative to the clean signal, the term $(d_n - y_n)$ may be large - yielding a low SNR improvement - while perceptually the processed signal still sounds very similar, since phase differences are generally inaudible.

In addition to numeric evaluation, I listened to all the outputs. Under each plot, brief notes describe my subjective impressions of the processed audio.

4.1.4 Implemented methods and parameters

Spectral subtraction

Both the *power* and *magnitude* variants of spectral subtraction were implemented. Initially, I planned to sweep over multiple values of α (keeping all other parameters constant), but the differences were very subtle. Therefore, I present results only for $\alpha = 1$ and $\alpha = 2$. The case $\alpha = 1$ can be considered the default/basic setting, while $\alpha = 2$ represents an extreme "oversubtraction" scenario. I also experimented with $\alpha > 2$, but in my test cases this always resulted in audible attenuation (either a quieter output or loss of timbral content) of the desired signal. In practice, this does not necessarily mean a worse subjective result – indeed, in the final test I was satisfied with $\alpha = 3$ because the noise was effectively removed, even though the desired signal was somewhat degraded – however, higher α consistently worsened the numerical SNR.

A Hanning window was used in all tests. This choice allowed a straightforward STFT implementation (as described in §2.2.5). I did not explore other window types or window-parameter tuning here, as that is a large subject worthy of a separate study.

After preliminary experiments with various frame sizes, I decided to present results for three frame lengths (all powers of two to match the FFT algorithm; other lengths could be zero-padded before FFT, but that was not investigated):

- *Small* frame: 1024 samples. Smaller frames produced audible distortion (likely due to increased spectral leakage, windowing artifacts, and lower frequency resolution).
- *Medium* frame: 4096 samples. (2048 and 8192 are also reasonable intermediate choices if one wants something between small and medium, or between medium and large.)
- *Large* frame: 16 384 samples (2^{14}), hereafter referred to as "16k". Larger frames did not produce a perceptible improvement over 16k in tests where larger frames were already advantageous (e.g., for stationary but complex signals).

The spectral floor parameter β was set to 10^{-5}.

Wiener filtering

Two variants of Wiener filtering were implemented (ones corresponding to noise PSD estimation methods mentioned in §3.1.3):

- *"Instant-Noise" variant*: This is a direct implementation from the Wiener filter definition. For each frame, both the signal and noise spectra are estimated via periodograms. In effect, this behaves like a sequence of "small Wiener filters": a separate optimal filter is computed for each frame. For stationary signals, longer frames yield better estimates of the ideal filter, since one approaches the assumption of an infinitely long, stationary process. However, if the signal is non-stationary, shorter frames better satisfy the stationarity assumption within each frame. In this variant, a noise-only reference of the same length as the noisy input (recorded in parallel) is required – similar to LMS – so that each frame's noise PSD can be estimated.
- *"Average-Noise" variant*: A more real-world scenario is that one has only a short noise sample that was not recorded synchronously with the noisy signal (a scenario like in spectral subtraction denoising). In this hybrid approach, the noise PSD is estimated by averaging periodograms across all available noise-only frames, just as in spectral subtraction. We expect the results to resemble those of spectral subtraction (power) with $\alpha = 1$, as discussed in §3.3.6.

Frame lengths for both Wiener variants matched those used in spectral subtraction. The regularization parameter ϵ was set to 10^{-5} (same as β in spectral subtraction).

LMS adaptive filtering

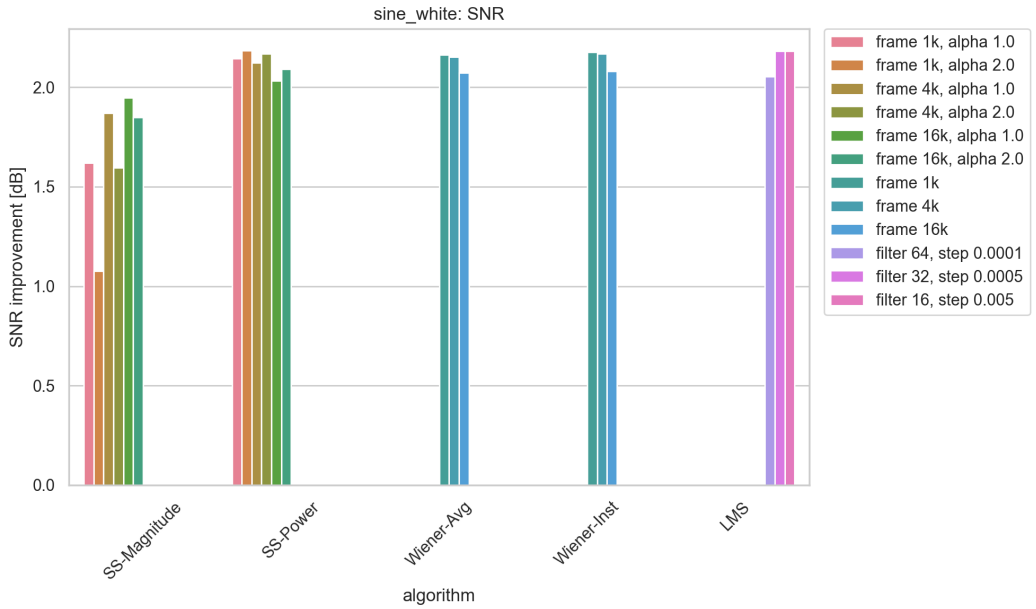
For LMS, the expectation is that a larger filter (more taps) better fits complex signals, while a smaller filter converges faster and runs more quickly but only suits

simpler signals. Similarly, a larger step size yields faster convergence, but a smaller step size obtains a better final fit once convergence is achieved. Consequently, I chose three parameter sets:

- *Fast convergence*: filter size = 16 , step size = 0.005.
- *Balanced*: filter size = 32 , step size = 0.0005.
- *Precise*: filter size = 64 , step size = 0.0001.

4.2 Results

Sine wave with white noise (`sine_white`)



The SNR plot does not accurately reflect what is heard. In this example, the difference between $\alpha = 1$ and $\alpha = 2$ was most pronounced:

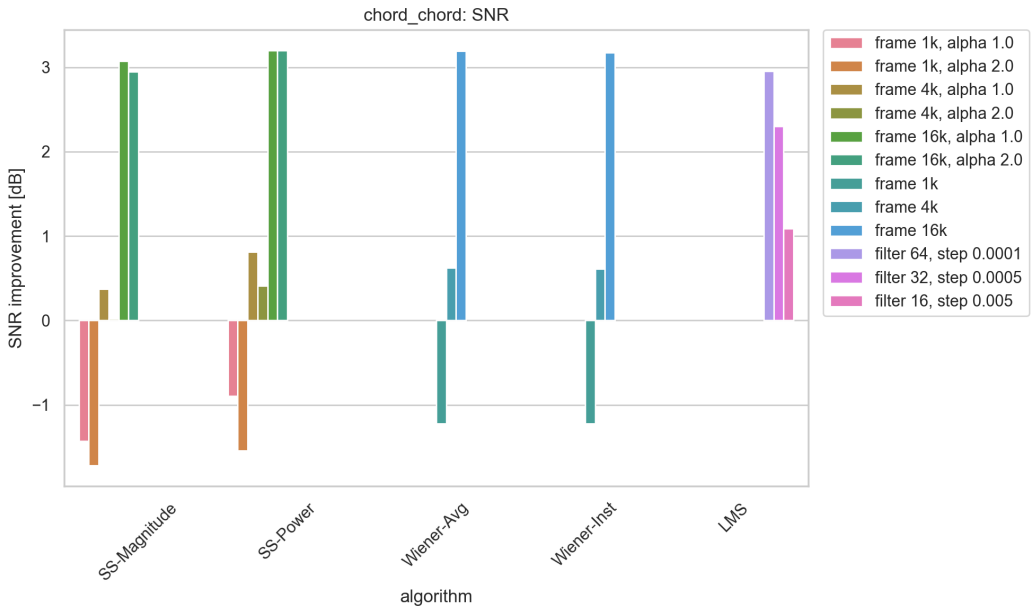
- Spectral subtraction (magnitude and power) with $\alpha = 1$ left some noise that was oddly distorted.
- Spectral subtraction (magnitude) with $\alpha = 2$ performed best of all algorithms, completely eliminating the noise.
- Spectral subtraction (power) with $\alpha = 2$ exhibited the *musical noise* effect.

A frame length of 1024 samples introduced audible chattering; 4096 samples was sufficient. A frame of 16 384 sounded marginally better but not by much.

Both Wiener variants performed worse than spectral subtraction (magnitude) with $\alpha = 2$, also leaving some residual noise. The instant-noise Wiener removed slightly more noise than the averaged-noise variant.

In LMS, it is audible that the filter adapts: at first the noise is present, then it diminishes. For filter size 64 it starts with slowly decaying "puff" and noise vanishes completely at about one third of the recording (i.e.,after ≈ 2 seconds), which is very slowly. The filter size 32 performed best – at the beginning there is a short shuffle, then the noise is mostly gone for the rest of the playback. Filter size 16 adapts quickly, but fluctuates, leaving some noise.

Separating two chords (`chord_chord`)



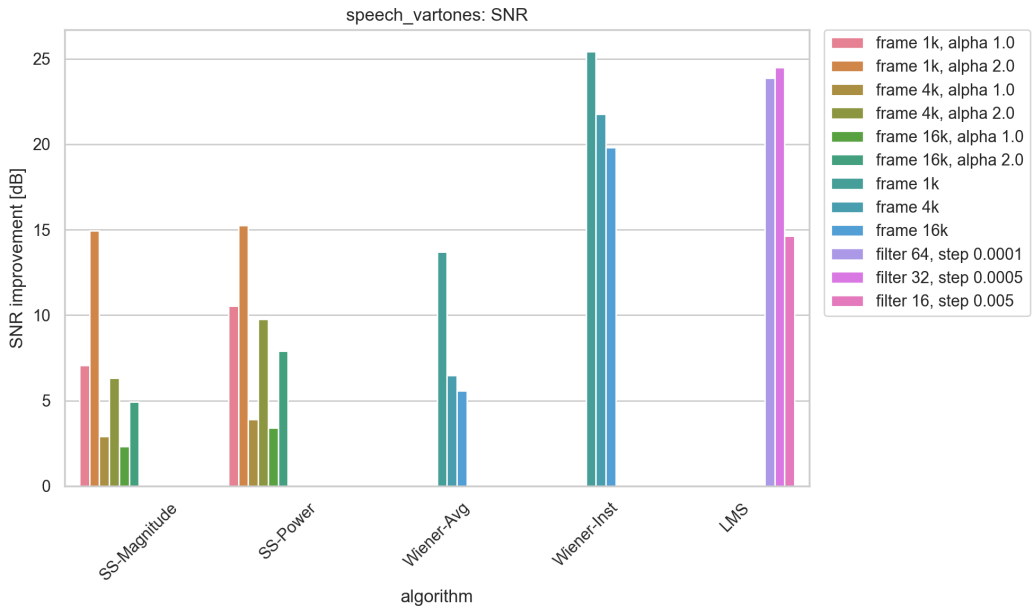
For LMS with filter size 64 the output sounds closer to the expected chord than the mixed signal, but it is still not the same as the clean signal. The remaining LMS setups did not produce satisfactory result.

Spectral methods, as expected, require a large frame, since the tested chord frequencies are close together:

- Frame 1024: Poor performance, introducing very intense chattering.
- Frame 4096: Better, but artifacts are still audible.
- Frame 16k: Acceptable - only the desired chord is heard.

For spectral subtraction, $\alpha = 1$ performs slightly better here, but the difference is subtle; all four spectral variants behave very similarly.

Speech with varying-tone noise (`speech_vartones`)

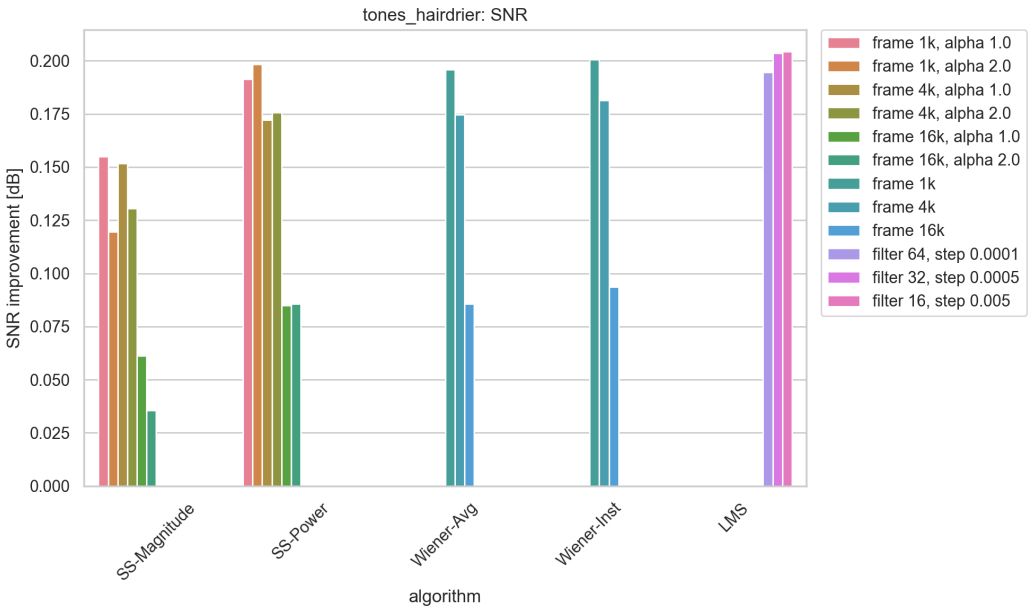


LMS clearly performs best (and the hight bar on the SNR plot reflects that): the noise is uniformly suppressed and almost inaudible. Short fragments of residual noise appear when the noise changed frequency, but they are quickly suppressed. The perceptually best configuration was filter size 64. The medium configuration (filter size 32) – which got higher SNR result – also sounded quite well. It left less artifacts at frequency changes at the cost of leaving a little bit of noise everywhere.

According to the plot, the instant-noise Wiener also does well, but the processed signal exhibits characteristic "clicks" (probably on the noise frequency or level changes, same as lms, though they appear lauder here), and occasional brief residual noise (as if the lauder part remained where one noise frequency changed it's level).

Averaging-based methods (spectral subtraction magnitude/power and averaged-noise Wiener) are not suited for time-varying noise; this is clearly seen here. In the denoised signal, there are moments where only speech is present, then very loud noise appears briefly, then disappears but part of the speech is also lost. Although the SNR plot shows some improvement for these methods, the perceptual outcome is poor.

Varying-tone signal with hairdryer noise (`tones_hairdrier`)

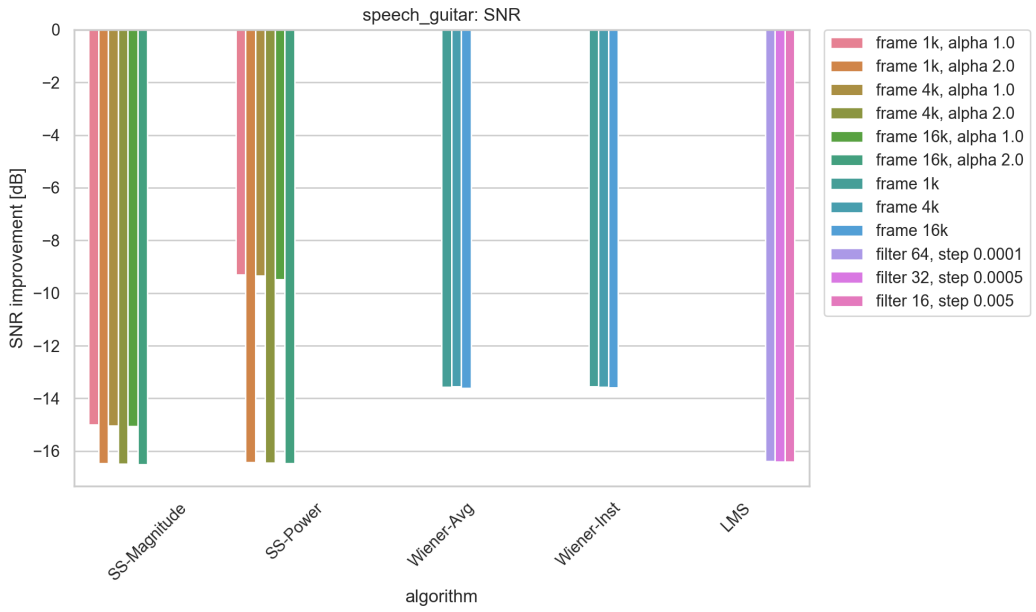


This is another test with time-varying noise. Here LMS resulted in highest SNR improvement, but perceptually did not perform so well. The quickly-adapting configuration (size 16) performed best, and introduces audible improvement, but did not remove the noise completely.

The instant-noise Wiener performed best of the algorithms – noise is almost entirely gone.

Spectral subtraction (magnitude/power) and averaged-noise Wiener only reduce one of the hairdryer modes; the other mode remains.

Speech with guitar noise (`speech_guitar`)

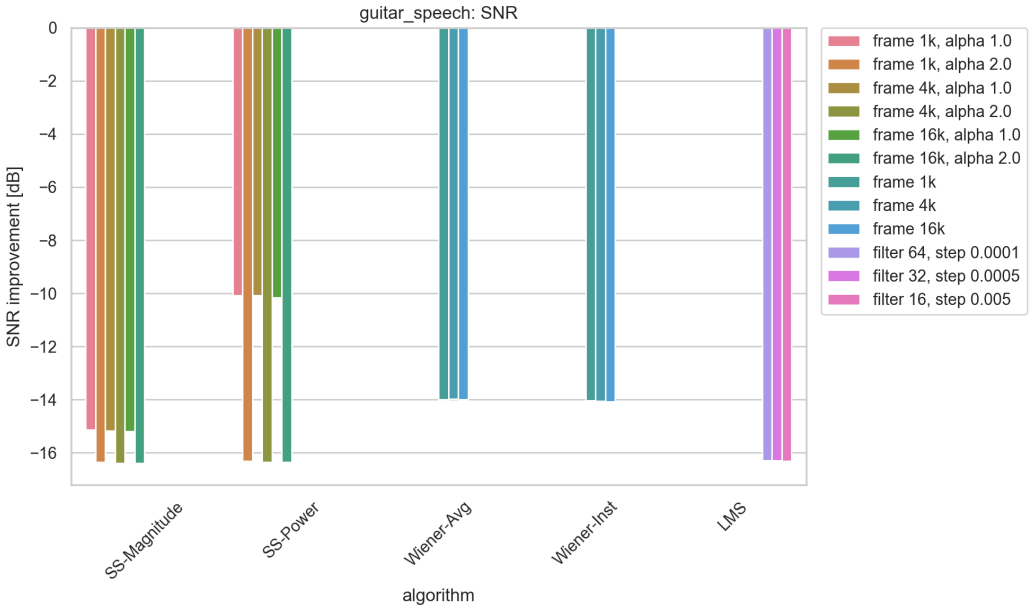


According to the plot, every algorithm degrades the SNR, which is not true perceptually. All spectral methods attenuate the guitar sound, leaving mostly the speech. The best results are for frame 4096:

- Instant-noise Wiener performed best – guitar is inaudible.
- Spectral subtraction (magnitude, $\alpha = 2$) performs almost as well.
- Remaining spectral subtraction setups and averaged-noise Wiener perform slightly worse – guitar is still audible but attenuated.

LMS fails – guitar remains audible.

Guitar with speech noise (`guitar_speech`)



According to the plot, every algorithm degrades the SNR, which again is not true perceptually. Only one setup achieved the desired effect (i.e., leaving only the instrument sound), but several others also introduced slight improvement.

Spectral subtraction (magnitude)

- Frames 1024 and 4096: Speech remains intelligible.
- Frame 16k: Introduces new distortions but almost completely removes speech. A strange residual noise remains, making words indistinguishable.
- For $\alpha = 2$, performance is slightly better, but only noticeable for frame 16k; smaller frames are still comparably poor.

Spectral subtraction (power)

- Frame 4096, $\alpha = 2$: Speech is still faintly audible, though attenuated, but the guitar is slightly distorted.
- Frame 16k, $\alpha = 2$: Only a faint hiss remains after speech, but the guitar is heavily distorted and sounds unnatural.
- Other settings: Speech remains intelligible.

Averaged-noise Wiener

- Frame 16k: Heavily distorts the guitar and speech remains more audible than in any spectral-subtraction variant.

Instant-noise Wiener

- Frame 1024: Speech is completely removed, but a light chattering/buzzing artifact appears.
- Frame 4096: Hardly any indication of modification – one would not suspect speech (i.e., the noise) was ever present.
- Frame 16k: Also sounds very good, but there are slight "wavering" artifacts.

LMS Fails – speech remains audible.

Speech with real microwave noise (`speech_microvave`)

For this test case there is no SNR-improvement plot, as I did not have the clean signal to compare with. Instead, I experimented with various algorithms and parameters, trying to find the setup that best extracts only the speech from the recording.

The most satisfying result in terms of noise removal was obtained with frame 4096 or 2048 and $\alpha = 3$ for spectral subtraction (magnitude). Although the speech was slightly distorted, it remained clearly intelligible and the microwave noise was effectively removed.

Using magnitude spectral subtraction with same frame sizes and $\alpha \in [1.5, 2.0]$ yielded a better outcome in terms of preserving speech quality, but left some residual noise.

In general, spectral subtraction proved easier to tune for such a realistic case. Wiener filtering was attempted, but for no frame size did it match the effectiveness of spectral subtraction.

LMS and instant-noise Wiener performed poorly, since this is not their intended scenario: the noise was recorded at a different time from the noisy signal. Although similar, the noise references were not identical, so these methods fail to converge properly.

4.3 Conclusions

4.4 ...

4.4.1 Other Quality Assessment Methods

- There are also more advanced computational methods for comparing audio quality, see [18].
- Ultimately, audio-processing companies employ so-called critical listeners to judge quality, since no method fully reflects human perception.

4.4.2 Other noise reduction methods

There are also more advanced noise reduction techniques that attempt to combine ideas from Wiener filtering and spectral subtraction to achieve better results. One example is presented in [16], where a two-step noise reduction (TSNR) is followed by harmonic regeneration (HRNR). In the first step, noise is attenuated using an adaptive gain similar to Wiener filtering, but with a "decision-directed" style estimate that reduces musical noise compared to classical spectral subtraction. In the second step, missing harmonic components of speech that were overly suppressed in the first stage are regenerated to restore a more natural sound quality.

More advanced adaptive methods, such as the Recursive Least Squares (RLS) algorithm, attempt to recursively estimate the autocorrelation matrix and the cross-correlation vector required by the Wiener filter, allowing for faster convergence and better tracking of time-varying noise statistics compared to LMS-based approaches [7]. These techniques have been successfully applied in biomedical signal processing; for example, adaptive RLS filtering is used to remove baseline wander and muscle artifacts from electrocardiogram (ECG) recordings [17].

Modern noise reduction methods in audio processing often combine classical DSP models with machine-learning approaches. One such example is the Deep Feature Loss (DFL) framework [15].

In addition, many modern communication platforms employ AI-driven noise suppression that detects and perserves speech while attenuating other sounds.

Bibliography

[1]

[2] R. G. Lyons, *Wprowadzenie do cyfrowego przetwarzania sygnałów*, Wydawnictwo Komunikacji i Łączności sp. z. o. o. Warszawa 1999, 2000, na polski przełożył Jan Zarzycki.

[3] *Cooley-Turkey FFT algorithm* (Wikipedia) [link] (last access 03.06.2025).

[4] YouTube video by 3Blue1Brown, *But what is the Fourier Transform? A visual introduction.* [link] (last access 06.05.2025).

[5] Blog by Elan Ness-Cohn *Developing An Intuition for Fourier Transforms* [link] (last access 06.05.2025).

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest i C. Stein, *Introduction to Algorithms*, Third Edition, MIT Press 2009

[7] S. V. Vaseghi, *Advanced Digital Signal Processing and Noise Reduction*, Fourth Edition, 2008 John Wiley & Sons, Ltd. ISBN: 978-0-470-75406-1

[8] S. W. Smith *The Scientist and Engineer's Guide to Digital Signal Processing*, Second Edition, California Technical Publishing 1999

[9] MIT course materials *Matrix Calculus (for Machine Learning and Beyond)* [link], (last access 03.06.2025)

[10] J. O. Smith III, *Mathematics of the Discrete Fourier Transform (DFT) with Audio Applications*, 2nd ed., Center for Computer Research in Music and Acoustics (CCRMA), Department of Music, Stanford University, Stanford, California, 2007. [online link]

[11] J. O. Smith III, *Spectral Audio Signal Processing*, Center for Computer Research in Music and Acoustics (CCRMA), Department of Music, Stanford University, Stanford, California, 2011. [online link]

[12] Wikipedia: *Round off error* [link] (last access 28.05.2025)

[13] P. Duhamel, M. Vetterli *Fast Fourier Transforms: A Tutorial Review and a State of the Art*, CRC Press LLC, 1999, (from *Signal Processing* 19: 259-299, 1990) [online link]

[14] M. H. Hayes, *Statistical Digital Signal Processing and Modeling*, John Wiley & Sons, Inc., New York, 1996.

[15] F. G. Germain, Q. Chen, V. Koltun, *Speech Denoising with Deep Feature Losses*, arXiv preprint arXiv:1806.10522, 2018. [online link] (last access 03.06.2025)

[16] S. Vihari, A. S. Murthy, P. Soni, and D. C. Naik, "Comparison of Speech Enhancement Algorithms," Procedia Computer Science 89 (2016), [online link] (last access 03.06.2025)

[17] Ahlam Fadhil Mahmood, Safaa N. Awny, Ali Alameer, "RLS adaptive filter co-design for de-noising ECG signal," Results in Engineering, vol. 24, 2024, art. no. 103563, ISSN 2590-1230, [online link]

[18] A. Rix, J. Beerends, D.-S. Kim, P. Kroon, O. Ghitza, "Objective Assessment of Speech and Audio Quality—Technology and Applications," IEEE Transactions on Audio, Speech, and Language Processing, vol. 14, pp. 1890-1901, 2006, [online link] (last access 03.06.2025)