Computational Audio and Image Analysis with the `Spectrograms` Library

Jack Geraghty

1 Introduction

This manual provides a comprehensive mathematical and computational overview of the algorithms, optimizations, and transformations implemented in the `spectrograms` library. The focus is on understanding the theoretical foundations and practical implementations of FFT-based signal processing for audio and image analysis.

The material emphasizes the ***why*** and ***how*** of each algorithm: *why* certain design choices matter, and *how* they are implemented efficiently. The material covers both the mathematics and the practical computational considerations that make these methods work in production systems.

1.1 Target Audience and Prerequisites

This material assumes familiarity with:

- Linear algebra (matrix multiplication, vector operations, eigenvalues)
- Complex numbers and Euler's formula
- Discrete signals and sampling theory
- Basic programming concepts and algorithm analysis
- Calculus (integrals, derivatives, series)

No prior knowledge of Fourier analysis, psychoacoustics, or advanced signal processing is assumed.

1.2 Scope and Organization

The manual is organized into major thematic sections:

1. **Fourier Transform Theory** (Section 2) — DFT, FFT algorithms, Hermitian symmetry, 2D FFT
2. **Window Functions and Spectral Analysis** (Section 3) — Hanning, Hamming, Kaiser, Blackman, Gaussian
3. **Time-Frequency Representations** (Section 4) — STFT and spectrograms
4. **Amplitude Scaling and Representations** (Section 5) — Power, magnitude, decibels, Parseval's theorem
5. **Perceptual Frequency Scales** (Section 6) — Mel, ERB, logarithmic scales
6. **Constant-Q Transform** (Section 7) — Variable-length kernels, sparsity optimization
7. **Advanced Audio Features** (Section 8) — MFCC, chroma, gammatone filters
8. **Image Processing via FFT** (Section 9) — 2D convolution, filtering, edge detection
9. **Computational Optimizations** (Section 10) — Sparse matrices, FFT plans, caching
10. **Numerical Considerations** (Section 11) — Precision, normalization, dynamic range

1.3 Notation Conventions

Throughout this manual:

- N denotes signal length or FFT size
- k denotes frequency bin index ($0 \leq k < N$)
- n denotes time/sample index ($0 \leq n < N$)
- $j = \sqrt{-1}$ (the imaginary unit)
- $X[k]$ denotes frequency domain values
- $x[n]$ denotes time domain values
- Vectors are bold: $\mathbf{x}$
- Complex conjugate: z^*

2 Fourier Transform Foundations

2.1 The Discrete Fourier Transform (DFT)

The ***Discrete Fourier Transform*** (DFT) decomposes a finite-length sequence into its constituent frequencies. For a sequence $x[n]$ of length N, the DFT is defined as [19]:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j2\pi kn/N}, \quad k = 0, 1, \ldots, N-1 \tag{1}$$

where $X[k]$ represents the complex amplitude at frequency bin k. Using Euler's formula $e^{j\theta} = \cos\theta + j\sin\theta$, Equation (1) can be expanded as:

$$X[k] = \sum_{n=0}^{N-1} x[n] \left[\cos\left(\frac{2\pi kn}{N}\right) - j\sin\left(\frac{2\pi kn}{N}\right)\right] \tag{2}$$

This reveals that the DFT correlates the input signal with complex sinusoids at each frequency. The computational complexity of this direct calculation is $\mathcal{O}(N^2)$ — each of N outputs requires N multiplications.

The ***inverse DFT*** (IDFT) reconstructs the time-domain signal:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] \cdot e^{j2\pi kn/N}, \quad n = 0, 1, \ldots, N-1 \tag{3}$$

The normalization factor $1/N$ ensures ***perfect reconstruction***: $\text{IDFT}(\text{DFT}(x)) = x$.

DFT Scaling Convention

Different implementations use different scaling conventions for the DFT and inverse DFT. The three most common conventions are:

1. **No forward scaling (this library):** Forward DFT uses coefficient 1, inverse DFT uses $1/N$
2. **Symmetric scaling:** Both forward and inverse use $1/\sqrt{N}$ (unitary transform)
3. **No inverse scaling:** Forward uses $1/N$, inverse uses 1

This library uses Convention 1: The forward DFT has *no normalization factor* (coefficient of 1), and the inverse DFT includes the $1/N$ factor. This convention is standard in most signal processing literature and libraries (FFTW, NumPy, SciPy).
Energy implications: With this scaling, Parseval's theorem (Section 5.4) takes the form:

$$\sum_{n=0}^{N-1} |x[n]|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X[k]|^2 \tag{4}$$

When comparing spectral amplitudes between implementations, *always verify the scaling convention* to ensure energy-consistent analysis.

2.2 Fast Fourier Transform (FFT): The Cooley-Tukey Algorithm

The ***Fast Fourier Transform*** (FFT) is not a different transform, but an efficient algorithm for computing the DFT. The Cooley-Tukey algorithm [7], published in 1965 (though similar ideas date to Gauss in 1805 [13]), reduces complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$.

The key insight is divide-and-conquer using the *symmetry* and *periodicity* properties of complex exponentials:

Twiddle Factor Properties

Let $W_N^k = e^{-j2\pi k/N}$ be the ***twiddle factor***. These complex exponentials satisfy:

$$W_N^{k+N/2} = -W_N^k \quad \text{(symmetry)} \tag{5}$$

$$W_N^{k+N} = W_N^k \quad \text{(periodicity)} \tag{6}$$

Periodicity means the complex exponential repeats every N samples. **Symmetry** means that values half an FFT bin apart differ only in sign. These properties enable the FFT's recursive decomposition.

For N even, split the DFT into even and odd indexed samples:

$$X[k] = \sum_{n=0}^{N/2-1} x[2n] W_N^{2nk} + \sum_{n=0}^{N/2-1} x[2n+1] W_N^{(2n+1)k} \tag{7}$$

Using $W_N^{2k} = W_{N/2}^k$, this becomes:

$$X[k] = \underbrace{\sum_{n=0}^{N/2-1} x[2n] W_{N/2}^{nk}}_{E[k]} + W_N^k \underbrace{\sum_{n=0}^{N/2-1} x[2n+1] W_{N/2}^{nk}}_{O[k]} \tag{8}$$

where $E[k]$ and $O[k]$ are $N/2$-point DFTs of the even and odd samples. This recursive splitting continues until the base cases are reached (typically $N = 1$ or small prime factors).

The complexity analysis: At each of $\log_2 N$ levels, N operations (additions and a twiddle factor multiplication) are performed, yielding $\mathcal{O}(N \log N)$ total complexity.

2.3 Real-to-Complex FFT and Hermitian Symmetry

When the input signal $x[n]$ is real-valued (as in audio processing), the DFT output exhibits a special property:

Hermitian Symmetry

For a real-valued signal $x[n]$, the DFT satisfies ***Hermitian symmetry***:

$$X[k] = X^*[N - k] \tag{9}$$

where X^* denotes complex conjugate. This means:

- $\mathrm{Re}(X[k]) = \mathrm{Re}(X[N - k])$ (real parts are symmetric)
- $\mathrm{Im}(X[k]) = -\mathrm{Im}(X[N - k])$ (imaginary parts are antisymmetric)
- $X[0]$ and $X[N/2]$ (for even N) are purely real

This property follows directly from the DFT definition and the fact that $e^{j\theta} + e^{-j\theta} = 2\cos\theta$ is real for real θ.
Critical constraint for R2C transforms: For even N, the Nyquist frequency bin $X[N/2]$ *must be purely real* (i.e., $\mathrm{Im}(X[N/2]) = 0$). This follows from Hermitian symmetry since $X[N/2] = X^*[N - N/2] = X^*[N/2]$, which requires the imaginary part to be zero. Violating this constraint results in non-real values after inverse FFT, causing numerical errors and incorrect signal reconstruction.

This symmetry makes it sufficient to store only the positive frequencies:

$$k \in \{0, 1, 2, \ldots, N/2\} \tag{10}$$

This yields $N/2 + 1$ complex values instead of N, halving storage requirements. The library computes real-to-complex (R2C) FFTs with output size:

$$\text{output\_size} = \lfloor N/2 \rfloor + 1 \tag{11}$$

Implementation detail: The library uses specialized R2C FFT algorithms (from RealFFT or FFTW backends) that exploit this symmetry for both speed and memory efficiency.

```
pub const fn r2c_output_size(n: usize) -> usize {
    n / 2 + 1
}

// Example: 512-point FFT on real signal
let samples = vec![0.0; 512];
let spectrum = fft(&samples, 512)?;
assert_eq!(spectrum.len(), 257); // 512/2 + 1
```

2.4 Inverse FFT and Perfect Reconstruction

The inverse FFT (IFFT) transforms frequency-domain data back to the time domain. For a complex-to-real (C2R) inverse FFT, given Hermitian-symmetric input $X[k]$:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{j2\pi kn/N} \tag{12}$$

Due to Hermitian symmetry, it is sufficient to sum over the non-redundant frequencies:

$$x[n] = \frac{1}{N} \left[X[0] + 2 \sum_{k=1}^{N/2-1} \mathrm{Re}(X[k] e^{j2\pi kn/N}) + X[N/2] \cos(\pi n) \right] \tag{13}$$

The library's C2R inverse FFT ensures perfect reconstruction:

```
let original = vec![1.0, 2.0, 3.0, 4.0];
let spectrum = fft2d(&original_image)?;
let reconstructed = ifft2d(&spectrum, original_image.ncols())?;
// reconstructed approximately original (within floating-point
    precision)
```

2.5 2D FFT via Row-Column Decomposition

For 2D data like images with dimensions $M \times N$, the 2D DFT is:

$$X[k_1, k_2] = \sum_{n_1=0}^{M-1} \sum_{n_2=0}^{N-1} x[n_1, n_2] \cdot e^{-j2\pi(k_1 n_1/M + k_2 n_2/N)} \tag{14}$$

The separability property allows this to be decomposed into row and column transforms:

$$X[k_1, k_2] = \sum_{n_1=0}^{M-1} e^{-j2\pi k_1 n_1/M} \underbrace{\left[\sum_{n_2=0}^{N-1} x[n_1, n_2] \cdot e^{-j2\pi k_2 n_2/N} \right]}_{\text{1D FFT along each row}} \tag{15}$$

Algorithm:

1. Apply 1D FFT to each row: M transforms of length N
2. Apply 1D FFT to each column of the result: N transforms of length M

For a real-valued $M \times N$ image, the output is $M \times (N/2 + 1)$ due to Hermitian symmetry along rows.

Implementation: The library uses row-major layout and ensures contiguous memory:

```
pub fn fft2d(data: &Array2<f64>) -> SpectrogramResult<Array2<Complex<
    f64>>> {
    let (nrows, ncols) = (data.nrows(), data.ncols());

    if !data.is_standard_layout() {
        return Err(SpectrogramError::invalid_input(
            "array must be contiguous and row-major"));
    }

    let out_shape = r2c_output_size_2d(nrows, ncols);
    // out_shape = (nrows, ncols/2 + 1)

    let mut plan = planner.plan_r2c_2d(nrows, ncols)?;
    // ... perform row-column FFT
}
```

Complexity: $O(MN \log N + MN \log M) = O(MN \log(MN))$ for square images.

3 Window Functions for Spectral Analysis

3.1 The Windowing Problem

Spectral Leakage

When the DFT of a finite signal segment is computed, the signal is implicitly multiplied by a rectangular window (ones inside the segment, zeros outside). This abrupt truncation causes ***spectral leakage***: energy from one frequency "leaks" into adjacent frequency bins.
Consider a pure sinusoid at frequency f_0. If sampling captures exactly an integer number of periods, the DFT produces a perfect spike at f_0. But with non-integer periods, the energy spreads across many bins.
This occurs because multiplying by a rectangular window in the time domain is equivalent to convolving with a sinc function in the frequency domain, which has slowly-decaying sidelobes.

The fundamental trade-off [15]:

- **Narrow main lobe** → better frequency resolution
- **Low sidelobes** → less spectral leakage

Window functions smooth the transition to zero at the segment boundaries, reducing leakage at the cost of frequency resolution.

3.2 Windowing Convention: Symmetric vs. Periodic

Symmetric and Periodic Windows

Window functions can be defined with two distinct conventions:
Symmetric windows use $N-1$ in the denominator:

$$w_{\text{sym}}[n] = f\left(\frac{n}{N-1}\right), \quad n \in [0, N-1] \tag{16}$$

Periodic windows use N in the denominator:

$$w_{\text{per}}[n] = f\left(\frac{n}{N}\right), \quad n \in [0, N-1] \tag{17}$$

The difference is subtle but important:

- **Symmetric:** Values at endpoints match (e.g., both zero for Hanning). Standard for spectral analysis.
- **Periodic:** The window is periodic with period N, so $w[N] = w[0]$. Required for perfect reconstruction in overlap-add systems.

For the Constant Overlap-Add (COLA) property [15]:

$$\sum_{m=-\infty}^{\infty} w[n - mH] = C \quad \text{(constant)} \tag{18}$$

where H is the hop size. Periodic windows satisfy COLA for specific overlap ratios, enabling perfect signal reconstruction.
Library convention: All window implementations in this library use the *symmetric* formulation ($N-1$), which is standard for analysis applications. For synthesis requiring COLA, use appropriate overlap ratios (e.g., 50% for Hanning).

3.3 Rectangular Window (No Windowing)

The rectangular window is simply:

$$w[n] = 1, \quad n \in [0, N-1] \tag{19}$$

Its frequency response has the narrowest main lobe but highest sidelobes (≈-13 dB), causing severe spectral leakage.

Listing 1: Rectangular window implementation

```
WindowType::Rectangular => {
    w.fill(1.0);
}
```

Use when: You need maximum frequency resolution and know the signal contains exact integer periods.

3.4 Hanning Window

Also called the Hann window (named after Julius von Hann, not Richard Hamming [15]), defined as:

$$w[n] = 0.5 - 0.5\cos\left(\frac{2\pi n}{N-1}\right), \quad n \in [0, N-1] \tag{20}$$

This is a raised cosine function. It smoothly tapers to zero at both endpoints.

Properties:

- Main lobe width: 4 bins (2× rectangular)
- First sidelobe: ≈-32 dB
- Sidelobe falloff: -18 dB/octave

Listing 2: Hanning window implementation

```
WindowType::Hanning => {
    let n1 = (n_fft - 1) as f64;
    for (n, v) in w.iter_mut().enumerate() {
        *v = 0.5 - 0.5 * cos(2.0 * PI * (n as f64) / n1);
    }
}
```

Why this works: The cosine taper reduces discontinuities at the edges. In the frequency domain, the Hanning window's transform is the convolution of the rectangular window's transform with two delta functions, producing lower sidelobes.

The Hanning window is the most commonly used for general-purpose spectral analysis.

3.5 Hamming Window

Named after Richard Hamming [15], this window uses optimized coefficients:

$$w[n] = 0.54 - 0.46 \cos\left(\frac{2\pi n}{N-1}\right) \tag{21}$$

The coefficients (0.54, 0.46) were chosen to minimize the first sidelobe:

- Main lobe width: 4 bins (same as Hanning)
- First sidelobe: ≈-43 dB (better than Hanning)
- Sidelobe falloff: -6 dB/octave (worse than Hanning)

```
WindowType::Hamming => {
    let n1 = (n_fft - 1) as f64;
    for (n, v) in w.iter_mut().enumerate() {
        *v = 0.54 - 0.46 * cos(2.0 * PI * (n as f64) / n1);
    }
}
```

Trade-off: The Hamming window doesn't reach exactly zero at the endpoints (it's 0.08), which can cause issues for overlap-add reconstruction. However, its lower first sidelobe makes it useful when strong interfering signals exist.

3.6 Blackman Window

A three-term cosine sum [3] with excellent sidelobe suppression:

$$w[n] = 0.42 - 0.5\cos\left(\frac{2\pi n}{N-1}\right) + 0.08\cos\left(\frac{4\pi n}{N-1}\right) \tag{22}$$

Properties:

- Main lobe width: 6 bins (3× rectangular)
- First sidelobe: ≈-58 dB
- Sidelobe falloff: -18 dB/octave

```
WindowType::Blackman => {
    let n1 = (n_fft - 1) as f64;
    for (n, v) in w.iter_mut().enumerate() {
        let a = 2.0 * PI * (n as f64) / n1;
        *v = 0.42 - 0.5 * cos(a) + 0.08 * cos(2.0 * a);
    }
}
```

Use when: You need minimal spectral leakage and can tolerate wider main lobes (reduced frequency resolution).

3.7 Kaiser Window and Bessel Functions

The Kaiser window [16] provides a tunable parameter β that controls the trade-off between main lobe width and sidelobe level:

$$w[n] = \frac{I_0\left(\beta\sqrt{1-\left(\frac{2n}{N-1}-1\right)^2}\right)}{I_0(\beta)} \tag{23}$$

Modified Bessel Function

The ***modified Bessel function of the first kind***, order zero, $I_0(x)$ is defined by:

$$I_0(x) = \sum_{k=0}^{\infty} \frac{1}{(k!)^2}\left(\frac{x}{2}\right)^{2k} \tag{24}$$

For computational efficiency, a truncated power series is used. The series converges rapidly; typically 20–30 terms achieve machine-precision accuracy for $x < 10$.

Parameter β effects:

- $\beta = 0$: rectangular window
- $\beta = 5$: similar to Hamming
- $\beta = 8.6$: similar to Blackman
- Larger β: lower sidelobes, wider main lobe

```
fn bessel_i0(x: f64) -> f64 {
    let mut sum = 1.0;       // k=0 term
    let mut term = 1.0;
    let half_x = x / 2.0;

    // Truncated series: sufficient for Kaiser windows
    for k in 1..30 {
        term *= (half_x / k as f64).powi(2);
        sum += term;

        // Early termination if converged
        if term < 1e-12 * sum {
            break;
        }
    }

    sum
}

WindowType::Kaiser { beta } => {
    let denominator = bessel_i0(beta);

    for i in 0..n_fft {
        let n = i as f64;
        let n_max = (n_fft - 1) as f64;
        let alpha = (n - n_max / 2.0) / (n_max / 2.0);
        let bessel_arg = beta * (1.0 - alpha * alpha).sqrt();

        w[i] = bessel_i0(bessel_arg) / denominator;
    }
}
```

The Kaiser window is named after James Kaiser who introduced it in 1966 for optimal FIR filter design.

3.8 Gaussian Window

A Gaussian function centered at the window midpoint:

$$w[n] = \exp\left(-\frac{1}{2}\left(\frac{n - \frac{N-1}{2}}{\sigma}\right)^2\right) \tag{25}$$

where σ (standard deviation) controls the width. Smaller $\sigma \rightarrow$ narrower window $\rightarrow$ more time localization.

```
WindowType::Gaussian { std } => {
    for i in 0..n_fft {
        let n = i as f64;
        let center = (n_fft - 1) as f64 / 2.0;
        let exponent = -0.5 * ((n - center) / std).powi(2);
        w[i] = exp(exponent);
    }
}
```

The Gaussian window has a unique property: its Fourier transform is also Gaussian. This gives optimal time-frequency localization per the uncertainty principle:

$$\Delta t \cdot \Delta f \geq \frac{1}{4\pi} \tag{26}$$

Use for: Time-frequency analysis where minimizing uncertainty is critical (e.g., Gabor transforms).

3.9 Trade-offs: Frequency Resolution vs. Spectral Leakage

Comparison table for $N = 512$:

Window	Main Lobe	Peak Sidelobe	Falloff Rate
Rectangular	2 bins	-13 dB	-6 dB/oct
Hanning	4 bins	-32 dB	-18 dB/oct
Hamming	4 bins	-43 dB	-6 dB/oct
Blackman	6 bins	-58 dB	-18 dB/oct
Kaiser ($\beta = 8.6$)	6 bins	-60 dB	-6 dB/oct

Selection guidelines:

- General purpose, good balance: Hanning
- Need low first sidelobe: Hamming
- Need very low sidelobes: Blackman
- Custom trade-off: Kaiser with tuned β
- Optimal uncertainty: Gaussian with appropriate σ

3.10 Constant Overlap-Add (COLA) Constraint

Perfect Reconstruction and COLA

When using the STFT for signal modification or synthesis (e.g., time-stretching, pitch-shifting, noise reduction), the time-domain signal must be reconstructed from modified spectral frames. The ***Constant Overlap-Add*** (COLA) constraint ensures perfect reconstruction is possible [15]. For a window function $w[n]$ and hop size H, the COLA property states that the sum of overlapping windows must be constant:

$$\sum_{m=-\infty}^{\infty} w[n - mH] = C \quad \text{(constant for all } n) \tag{27}$$

where C is a non-zero constant (typically normalized to 1).
Physical interpretation: When frames are overlapped by hop size H and summed (overlap-add), every sample in the reconstructed signal receives equal weighting from the windows. This prevents amplitude modulation artifacts.

COLA conditions for common windows:

1. **Rectangular window:** COLA for any hop size $H \leq N$ (no overlap required)
2. **Hanning window (periodic):** COLA for $H = N/2, N/4, N/8, \ldots$ (50%, 75%, 87.5% overlap, etc.)
    - 50% overlap ($H = N/2$): Most common, $C = 1$
    - 75% overlap ($H = N/4$): Higher redundancy, smoother synthesis
3. **Hamming window:** COLA for $H \approx 0.375N$ to $0.5N$ (requires careful tuning; 50% overlap gives $C \approx 1.08$)

4. **Blackman window:** COLA for $H \leq N/3$ (at least 67% overlap required)
5. **Kaiser window:** COLA depends on β; generally requires $50 - 75\%$ overlap

Verification: To check if a window/hop combination satisfies COLA, compute:

```
fn verify_cola(window: &[f64], hop_size: usize) -> bool {
    let n = window.len();
    let n_overlaps = (n + hop_size - 1) / hop_size;

    // Check multiple offset positions
    for offset in 0..hop_size {
        let mut sum = 0.0;

        for m in 0..n_overlaps {
            let idx = offset + m * hop_size;
            if idx < n {
                sum += window[idx];
            }
        }

        // All positions should have same sum (within tolerance)
        if (sum - window[0]).abs() > 1e-10 {
            return false;
        }
    }

    true
}
```

Implications for spectrogram inversion:

When using the STFT for analysis-modification-synthesis:

1. Choose a window/hop combination that satisfies COLA
2. Apply STFT to get frames $X[m, k]$
3. Modify the spectrogram (e.g., filter, denoise, time-stretch)
4. Apply inverse FFT to each modified frame
5. Overlap-add the frames with hop size H
6. Divide by the COLA constant C (if $C \neq 1$)

The reconstructed signal $\hat{x}[n]$ will equal the original $x[n]$ if no modifications were made (perfect reconstruction).

Library convention: For analysis-only applications (spectrograms, feature extraction), COLA is not required. However, this library uses 50% overlap ($H = N/2$) by default, which satisfies COLA for Hanning windows and enables optional synthesis workflows.

4 Short-Time Fourier Transform (STFT)

4.1 Time-Frequency Analysis Motivation

The standard DFT (Equation (1)) provides frequency content but discards all temporal information: it indicates *which* frequencies are present, but not *when* they occur. For non-stationary signals (speech, music, transient events), time-frequency analysis is required [2].

The ***Short-Time Fourier Transform*** (STFT) solves this by applying the DFT to short, overlapping segments (frames) of the signal, creating a time-frequency representation.

4.2 STFT Definition and Computation

For a signal $x[n]$ with window function $w[n]$ of length N, the STFT is:

$$X[m,k] = \sum_{n=0}^{N-1} x[n+mH] \cdot w[n] \cdot e^{-j2\pi kn/N} \tag{28}$$

where:

- m is the frame index
- H is the ***hop size*** (samples between consecutive frames)
- k is the frequency bin
- N is the FFT size (window length)

The result is a 2D complex matrix: $X[m,k]$ with dimensions (time frames) $\times$ (frequency bins).

4.3 Framing and Overlap

The signal is divided into overlapping frames:

$$\text{Frame } m \text{ starts at sample: } mH \tag{29}$$

Key parameters:

- **FFT size (N):** Window length, determines frequency resolution: $\Delta f = f_s/N$
- **Hop size (H):** Samples between frames, determines time resolution: $\Delta t = H/f_s$
- **Overlap:** $(N-H)/N$. Common: 50% (H = N/2) or 75% (H = N/4)

Number of frames for signal of length L samples:

$$M = \left\lfloor \frac{L-N}{H} \right\rfloor + 1 \tag{30}$$

With centering (padding $N/2$ zeros on each end):

$$M = \left\lfloor \frac{L+N-H}{H} \right\rfloor \tag{31}$$

STFT Centering and Temporal Alignment

Why pad $N/2$ zeros? Without centering, the first window is positioned with its *left edge* at sample 0. This means the window's center is at sample $N/2$, causing a temporal shift in the resulting spectrogram.
By padding $N/2$ zeros *before* the signal, the first window's *center* aligns with sample 0. This ensures:

- Frame m is centered at time $t = mH/f_s$ (no offset)
- Edge effects are symmetric at signal boundaries
- Improved temporal localization for transient events

The same $N/2$ padding is applied at the signal's end, ensuring symmetric treatment of boundaries. This is standard practice in audio analysis libraries (e.g., librosa's `center=True`).
Padding type specification: This library uses *zero-padding* (also called zero-extension) for both the initial and final $N/2$ samples:

$$\tilde{x}[n] = \begin{cases} 0 & n < 0 \\ x[n] & 0 \le n < L \\ 0 & n \ge L \end{cases} \tag{32}$$

where L is the original signal length.
Alternative: Reflective padding extends the signal by mirroring boundary values (e.g., $\tilde{x}[-1] = x[0]$, $\tilde{x}[-2] = x[1]$), which can reduce edge discontinuities for certain signals. However, zero-padding is the standard convention in most audio processing libraries (NumPy, SciPy, librosa) and is used exclusively in this implementation.
Implementation consistency: All implementations must use the same padding type to ensure identical spectrograms for the same input. Using reflective padding when zero-padding is expected (or vice versa) will cause boundary frames to diverge, particularly for the first and last $\lceil N/(2H) \rceil$ frames where padding contributes to the windowed signal.

Implementation detail: The library uses centering by default to avoid edge effects:

```
fn frame_count(&self, n_samples: usize) -> usize {
    let pad = if self.centre { self.n_fft / 2 } else { 0 };
    let padded_len = n_samples + 2 * pad;

    if padded_len < self.n_fft {
        return 1;  // Single frame, even for short signals
    }

    let remaining = padded_len - self.n_fft;
    let n_frames = remaining / self.hop + 1;
    n_frames
}
```

4.4 STFT Computation Pipeline

For each frame m:

1. **Extract frame:** Get samples $x[mH], x[mH+1], \ldots, x[mH+N-1]$
2. **Apply window:** Compute $x_w[n] = x[mH+n] \cdot w[n]$
3. **FFT:** Compute $X_m[k] = \text{FFT}(x_w)$
4. **Store:** Save as column m of output matrix

Optimizations:

- Pre-compute window samples once
- Reuse FFT plan across all frames
- Allocate workspace buffers once

Workspace Memory Requirements

To enable efficient STFT computation, implementations use a ***Workspace*** structure that holds reusable buffers and FFT state. This avoids repeated allocations during frame processing.
Minimum workspace requirements:

1. **Windowed frame buffer:** Real-valued array of size N to hold $x[mH + n] \cdot w[n]$ before FFT
2. **FFT output buffer:** Complex-valued array of size $N/2 + 1$ (for R2C FFT) to hold frequency-domain values $X_m[k]$
3. **FFT plan state:** Backend-specific data structure (e.g., FFTW plan, RealFFT planner) that stores twiddle factors and algorithm configuration
4. **Power/magnitude spectrum buffer:** Real-valued array of size $N/2+1$ to hold $|X_m[k]|^2$ or $|X_m[k]|$ for the current frame

Total memory requirement: $O(N)$ real values and $O(N)$ complex values, plus backend-specific FFT plan overhead (typically $O(N \log N)$ for precomputed twiddle factors).
Implementation example:

```
struct Workspace {
    frame: Vec<f64>,              // Size N (windowed input)
    fft_out: Vec<Complex<f64>>,   // Size N/2 + 1 (FFT output)
    spectrum: Vec<f64>,           // Size N/2 + 1 (power/magnitude
       )
}

impl Workspace {
    fn new(n_fft: usize) -> Self {
        let out_len = n_fft / 2 + 1;
        Self {
            frame: vec![0.0; n_fft],
            fft_out: vec![Complex::zero(); out_len],
            spectrum: vec![0.0; out_len],
        }
    }
}
```

Performance benefit: Reusing workspace buffers across frames reduces memory allocator pressure. For a 10-second audio file at 16 kHz with $N = 512$ and $H = 256$, this processes approximately 625 frames — avoiding 1,875 allocations (3 buffers $\times$ 625 frames).

```
   - compute one frame
  fn compute_frame_spectrum(&mut self, samples: &[f64],
                            frame_idx: usize,
                            workspace: &mut Workspace) -> Result<()> {
      let out = workspace.frame.as_mut_slice();
      let pad = if self.centre { self.n_fft / 2 } else { 0 };
      let start = frame_idx * self.hop;

      // Fill windowed frame (with zero-padding)
      for i in 0..self.n_fft {
          let v_idx = start + i;
          let s_idx = v_idx as isize - pad as isize;

          let sample = if s_idx < 0 || s_idx >= samples.len() {
              0.0  // Padding
          } else {
              samples[s_idx as usize]
          };

          out[i] = sample * self.window[i];
      }

      // Compute FFT
      self.fft.process(out, workspace.fft_out)?;

      // Convert to power spectrum: |X[k]|^2
      for (i, c) in workspace.fft_out.iter().enumerate() {
          workspace.spectrum[i] = c.norm_sqr();
      }

      Ok(())
  }
```

4.5 Spectrogram as STFT Magnitude

A spectrogram is the magnitude (or power) of the STFT:

$$S[m,k] = |X[m,k]|^2 \quad \text{(power spectrogram)} \tag{33}$$

or

$$S[m,k] = |X[m,k]| \quad \text{(magnitude spectrogram)} \tag{34}$$

This discards phase information but provides an intuitive visualization of energy distribution over time and frequency.

4.6 Time-Frequency Uncertainty Principle

Time-Frequency Uncertainty Principle

The fundamental trade-off in STFT is governed by the uncertainty principle [12]:

$$\Delta t \cdot \Delta f \geq \frac{1}{4\pi} \tag{35}$$

where Δt is temporal resolution and Δf is frequency resolution.
This is *not* a limitation of the algorithm, but a fundamental property of Fourier analysis related to the Heisenberg uncertainty principle in quantum mechanics. You **cannot** achieve arbitrary precision in both domains simultaneously.

Implications:

- **Large N (long window):** Good frequency resolution, poor time resolution
- **Small N (short window):** Good time resolution, poor frequency resolution

For speech (16 kHz sample rate):

- $N = 512$ (32 ms): $\Delta f = 31.25$ Hz — good for rapid changes
- $N = 2048$ (128 ms): $\Delta f = 7.8$ Hz — better frequency detail

For music (44.1 kHz sample rate):

- $N = 2048$ (46 ms): $\Delta f = 21.5$ Hz — standard choice
- $N = 4096$ (93 ms): $\Delta f = 10.8$ Hz — for harmonic analysis

Cannot achieve arbitrary precision in both domains simultaneously. This is not a limitation of the algorithm, but a fundamental property of Fourier analysis related to the Heisenberg uncertainty principle in quantum mechanics.

5 Amplitude Scaling and Representations

5.1 Power Spectrum

The power spectrum represents energy at each frequency:

$$P[k] = |X[k]|^2 = \mathrm{Re}(X[k])^2 + \mathrm{Im}(X[k])^2 \tag{36}$$

For real input signals, this is proportional to the energy in frequency bin k.

Properties:

- Always non-negative
- Units: (amplitude)^2
- Additive: energies from independent sources sum

The squared magnitude emphasizes strong components and suppresses weak ones, which can be both beneficial (noise reduction) and problematic (loss of weak signals).

5.2 Magnitude Spectrum

The magnitude spectrum is the absolute value:

$$M[k] = |X[k]| = \sqrt{\text{Re}(X[k])^2 + \text{Im}(X[k])^2} \tag{37}$$

This is more perceptually linear than power, as human perception of sound intensity is approximately logarithmic.

Relationship:

$$M[k] = \sqrt{P[k]} \tag{38}$$

Use magnitude when:

- You need perceptually meaningful amplitudes
- Comparing with other magnitude-based representations
- Visualization where extreme values would dominate

5.3 Decibel Scale

The decibel (dB) scale provides logarithmic amplitude scaling:

$$S_{\text{dB}}[k] = 10 \log_{10} \left(\frac{P[k]}{P_{\text{ref}}} \right) \tag{39}$$

where P_{ref} is a reference power. For power spectrograms without specific reference:

$$S_{\text{dB}}[k] = 10 \log_{10}(P[k]) \tag{40}$$

For magnitude-based signals:

$$S_{\text{dB}}[k] = 20 \log_{10}(M[k]) \tag{41}$$

The factor of 20 (instead of 10) accounts for the squared relationship between magnitude and power.

Numerical considerations: To avoid $\log(0) = -\infty$, a floor is applied:

$$S_{\text{dB}}[k] = 10 \log_{10}(\max(P[k], \epsilon)) \tag{42}$$

where ϵ is a small threshold (e.g., 10^{-10}). This maps very small values to a large negative dB value instead of $-\infty$.

Dynamic range compression: The dB scale compresses the dynamic range, making it easier to visualize spectrograms with both loud and quiet components. A signal with power ratios of 1,000,000:1 becomes 60 dB, much more manageable.

Example:

- Power = 1.0 $\rightarrow$ 0 dB
- Power = 0.1 $\rightarrow$ -10 dB
- Power = 0.01 $\rightarrow$ -20 dB
- Power = 100.0 $\rightarrow$ 20 dB

5.4 Parseval's Theorem and Energy Conservation

Parseval's Theorem

Parseval's theorem states that energy is conserved between time and frequency domains:

$$\sum_{n=0}^{N-1} |x[n]|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X[k]|^2 \tag{43}$$

This fundamental result ensures that:

- The FFT doesn't create or destroy energy
- Total power in frequency domain equals total power in time domain
- Inverse FFT perfectly reconstructs energy (up to numerical precision)

For windowed signals in STFT with proper overlap-add, energy is preserved:

$$\sum_m \sum_k |X[m,k]|^2 \propto \sum_n |x[n]|^2 \tag{44}$$

where the proportionality constant depends on the window normalization.

Implication for normalization: Different FFT libraries use different normalization conventions. The library uses the convention where the forward FFT has no normalization factor, and the inverse FFT divides by N. This is consistent with most signal processing literature.

6 Perceptual Frequency Scales

6.1 Motivation: Human Auditory Perception

Auditory frequency perception is not linear [9]. Frequency differences are perceived logarithmically, especially at higher frequencies. For example:

- The difference between 100 Hz and 200 Hz is highly perceptible (one octave)
- The difference between 10,000 Hz and 10,100 Hz is barely noticeable (0.014 octaves)

Perceptual frequency scales map linear Hz to psychoacoustically-motivated scales that better match human hearing.

6.2 Mel Scale

The mel scale is based on pitch perception studies [21]. The name comes from "melody," reflecting its musical origins. The conversion formulas are:

$$\text{mel}(f) = 2595 \log_{10}\left(1 + \frac{f}{700}\right) \tag{45}$$

$$f(\text{mel}) = 700\left(10^{\text{mel}/2595} - 1\right) \tag{46}$$

The mel scale is approximately linear below 1000 Hz and logarithmic above.

Mel Filterbank Construction:

To create M mel-spaced filters:

1. Convert frequency range $[f_{\min}, f_{\max}]$ to mel scale
2. Create $M + 2$ equally-spaced mel points (for triangular filter edges)
3. Convert mel points back to Hz
4. Map Hz to FFT bin indices
5. Create triangular filters centered at these points

$$H_m[k] = \begin{cases} 0 & k < f[m-1] \\ \frac{k-f[m-1]}{f[m]-f[m-1]} & f[m-1] \leq k < f[m] \\ \frac{f[m+1]-k}{f[m+1]-f[m]} & f[m] \leq k \leq f[m+1] \\ 0 & k > f[m+1] \end{cases} \tag{47}$$

where $f[m]$ are the FFT bin indices corresponding to mel points.

Implementation detail: The filterbank is stored as a sparse matrix since each triangular filter only has non-zero values over a small range of bins:

```
fn build_mel_filterbank_matrix(sample_rate_hz: f64, n_fft: usize,
                                n_mels: usize, f_min: f64,
                                f_max: f64) -> SparseMatrix {
    let out_len = r2c_output_size(n_fft);
    let df = sample_rate_hz / n_fft as f64;

    // Convert to mel scale
    let mel_min = hz_to_mel(f_min);
    let mel_max = hz_to_mel(f_max);

    // Create n_mels + 2 mel points (for triangle edges)
    let n_points = n_mels + 2;
    let step = (mel_max - mel_min) / (n_points - 1) as f64;

    let mel_points: Vec<_> = (0..n_points)
        .map(|i| i as f64 * step + mel_min)
        .collect();

    // Convert back to Hz, then to FFT bins
    let bin_points: Vec<_> = mel_points
        .iter()
        .map(|&mel| mel_to_hz(mel))
        .map(|hz| (hz / df).floor() as usize)
        .collect();

    // Build sparse filterbank
    let mut fb = SparseMatrix::new(n_mels, out_len);

    for m in 0..n_mels {
        let left = bin_points[m];
        let centre = bin_points[m + 1];
        let right = bin_points[m + 2];

        // Rising slope: left -> centre
        for k in left..centre {
            let v = (k - left) as f64 / (centre - left) as f64;
            fb.set(m, k, v);
        }

        // Falling slope: centre -> right
        for k in centre..right {
            let v = (right - k) as f64 / (right - centre) as f64;
            fb.set(m, k, v);
        }
    }

    fb
}
```

For typical speech processing ($N = 512$, 80 mel bands), the filterbank is approximately 95% sparse.

Numerical hygiene: To achieve this sparsity in practice, coefficients below a threshold (typically 10^{-12} or machine epsilon $\epsilon_{\text{machine}} \approx 2.2 \times 10^{-16}$ for f64) are set to exactly zero during construction. This prevents accumulation of negligible numerical artifacts while maintaining filterbank fidelity.

6.3 ERB Scale: Glasberg and Moore Model

The ***Equivalent Rectangular Bandwidth*** (ERB) scale is based on psychoacoustic measurements of critical bandwidths [14, 17]. It models the frequency selectivity of the human auditory system.

The ERB in Hz as a function of center frequency:

$$\text{ERB}(f) = 24.7\left(4.37\frac{f}{1000} + 1\right) \tag{48}$$

To convert frequency to ERB scale (ERB-rate):

$$\text{ERB-rate}(f) = 21.4\log_{10}(4.37\frac{f}{1000} + 1) \tag{49}$$

Inverse transformation:

$$f(\text{ERB}) = \frac{1000}{4.37}\left(10^{\text{ERB}/21.4} - 1\right) \tag{50}$$

Gammatone Filters:

ERB spectrograms often use gammatone filters, which model the impulse response of the auditory filter:

$$g(t) = t^{n-1}e^{-2\pi bt}\cos(2\pi f_c t + \phi) \tag{51}$$

where:

- n is the filter order (typically 4)
- b is the bandwidth parameter related to ERB
- f_c is the center frequency
- ϕ is the phase

The library implements gammatone filters in the frequency domain for efficiency.

Why ERB over Mel? The ERB scale more accurately models auditory filters at higher frequencies and is preferred for applications requiring high fidelity to human perception.

6.4 Logarithmic Frequency (Musical Scale)

Musical notes are logarithmically spaced: each octave doubles the frequency. For music analysis, a logarithmic frequency axis is natural.

Log-frequency bins:

$$f_k = f_{\min} \cdot \left(\frac{f_{\max}}{f_{\min}}\right)^{k/(K-1)} \tag{52}$$

for $k = 0, 1, \ldots, K-1$ bins.

Alternatively, using equal spacing in log-space:

$$\log f_k = \log f_{\min} + k\frac{\log f_{\max} - \log f_{\min}}{K-1} \tag{53}$$

Linear Interpolation:

Since FFT bins are linearly spaced, linear interpolation is used between adjacent bins to get log-spaced values:

$$y_k = (1-\alpha)X[i] + \alpha X[i+1] \tag{54}$$

where $i = \lfloor f_k/\Delta f \rfloor$ and $\alpha = (f_k/\Delta f) - i$.

```
fn build_loghz_matrix(sample_rate_hz: f64, n_fft: usize,
                      n_bins: usize, f_min: f64,
                      f_max: f64) -> SparseMatrix {
    let df = sample_rate_hz / n_fft as f64;

    // Logarithmically-spaced frequencies
    let log_f_min = f_min.ln();
    let log_f_max = f_max.ln();
    let log_step = (log_f_max - log_f_min) / (n_bins - 1) as f64;

    let log_frequencies: Vec<_> = (0..n_bins)
        .map(|i| (i as f64 * log_step + log_f_min).exp())
        .collect();

    // Build interpolation matrix
    let mut matrix = SparseMatrix::new(n_bins, out_len);

    for (bin_idx, &target_freq) in log_frequencies.iter().enumerate() {
        let exact_bin = target_freq / df;
        let lower_bin = exact_bin.floor() as usize;
        let upper_bin = exact_bin.ceil() as usize;

        if lower_bin == upper_bin {
            matrix.set(bin_idx, lower_bin, 1.0);
        } else {
            // Linear interpolation
            let frac = exact_bin - lower_bin as f64;
            matrix.set(bin_idx, lower_bin, 1.0 - frac);
            matrix.set(bin_idx, upper_bin, frac);
        }
    }

    matrix
}
```

This matrix is extremely sparse: only 1–2 non-zero values per row (¿99% sparse).

6.5 Filterbank Design and Sparse Matrices

All perceptual scales use sparse matrix multiplication for efficiency. The operation is:

$$\mathbf{y} = H\mathbf{x} \tag{55}$$

where H is the filterbank matrix (sparse), $\mathbf{x}$ is the linear spectrum, and $\mathbf{y}$ is the perceptually-scaled spectrum.

Sparse Matrix Storage:

The library uses row-wise sparse format:

```
struct SparseMatrix {
    nrows: usize,
    ncols: usize,
    values: Vec<Vec<f64>>,    // Non-zero values per row
    indices: Vec<Vec<usize>>, // Column indices per row
}
```

Multiplication:

```
fn multiply_vec(&self, input: &[f64], out: &mut [f64]) {
    for (row_idx, (row_values, row_indices)) in
        self.values.iter().zip(&self.indices).enumerate() {

        let mut acc = 0.0;
        for (&value, &col_idx) in row_values.iter().zip(row_indices) {
            acc += value * input[col_idx];
        }
        out[row_idx] = acc;
    }
}
```

Sparse Matrix Computational Complexity

For matrix-vector multiplication $\mathbf{y} = A\mathbf{x}$ where A is $m \times n$:
Dense matrix:

$$\mathcal{O}(mn) \quad \text{(every element accessed)} \tag{56}$$

Sparse matrix with nnz non-zeros:

$$\mathcal{O}(\text{nnz}) \quad \text{(only non-zeros computed)} \tag{57}$$

where nnz is the number of non-zero elements.
For filterbanks with uniform sparsity s non-zeros per row:

$$\text{nnz} = ms \tag{58}$$

Example: Mel filterbank with $m = 80$, $n = 257$, $s = 20$ non-zeros/row has 1,600 operations vs. 20,560 for dense multiplication.

For mel filterbanks with $M = 80$ bands and $N = 512$ FFT (257 bins):

- Dense matrix: $80 \times 257 = 20{,}560$ multiplications
- Sparse matrix: $\approx 80 \times 20 = 1{,}600$ multiplications ($\sim$10–20 non-zeros/row)

For log-frequency with linear interpolation:

- Only 2 multiplications per output bin (vs. 257 for dense)

7 Constant-Q Transform (CQT)

7.1 Motivation and Q Factor

The Short-Time Fourier Transform (Section 4.2) provides constant frequency resolution Δf across all frequencies. But musical perception is logarithmic: *relative* frequency differences (ratios) are more relevant than absolute differences.

The ***Constant-Q Transform*** (CQT) [5, 6] provides logarithmically-spaced frequency bins with constant quality factor Q:

$$Q = \frac{f_k}{\Delta f_k} \tag{59}$$

where f_k is the center frequency and Δf_k is the bandwidth of bin k.

For musical applications with B bins per octave:

$$Q = \frac{1}{2^{1/B} - 1} \tag{60}$$

For example, with $B = 12$ bins/octave (semitones):

$$Q = \frac{1}{2^{1/12} - 1} \approx 16.82 \tag{61}$$

Higher $Q \rightarrow$ narrower bandwidth $\rightarrow$ better frequency resolution but worse time resolution.

7.2 Kernel-Based Implementation

Unlike STFT which uses a fixed-length window, CQT uses *variable-length windows* (kernels) for each frequency:

For frequency bin k with center frequency f_k:

$$N_k = \left\lceil \frac{Q \cdot f_s}{f_k} \right\rceil \tag{62}$$

where f_s is the sample rate. Higher frequencies $\rightarrow$ shorter windows, lower frequencies $\rightarrow$ longer windows.

The CQT kernel for frequency f_k:

$$g_k[n] = w[n] \cdot e^{j2\pi f_k n/f_s}, \quad n = 0, 1, \ldots, N_k - 1 \tag{63}$$

where $w[n]$ is a window function (typically Hanning or Hamming).

Implementation:

```
fn generate_kernel_bin(center_freq: f64, kernel_length: usize,
                       sample_rate: f64,
                       window_type: WindowType) -> Vec<Complex<f64>> {
    let mut kernel = Vec::with_capacity(kernel_length);

    // Generate window
    let window = make_window(window_type, kernel_length)?;

    // Generate complex exponential kernel
    for (n, w) in window.iter().enumerate() {
        let t = n as f64 / sample_rate;
        let phase = 2.0 * PI * center_freq * t;

        // e^(i*2*$\pi$*f*t) = cos(2$\pi$ft) + i*sin(2$\pi$ft)
        let exponential = Complex::new(phase.cos(), phase.sin());

        // Apply window function
        let windowed = exponential * w;
        kernel.push(windowed);
    }

    kernel
}
```

7.3 Variable-Length Windows

The beauty of CQT is that window length scales with frequency:

- **Low frequencies:** Long windows → good frequency resolution, poor time resolution
- **High frequencies:** Short windows → good time resolution, adequate frequency resolution

This matches musical perception: precise pitch resolution is required for low notes (fundamental frequencies), while less precision is typically acceptable for high harmonics.

Example for $f_s = 44{,}100$ Hz, $Q \approx 16.82$:

- $f = 55$ Hz (A1): $N \approx 13{,}480$ samples (≈305 ms)
- $f = 440$ Hz (A4): $N \approx 1{,}685$ samples (≈38 ms)
- $f = 3{,}520$ Hz (A7): $N \approx 211$ samples (≈4.8 ms)

7.4 Sparsity Optimization

CQT kernels naturally have many small-magnitude coefficients, especially near the edges due to windowing. This can be exploited with *sparsity thresholding*:

$$g_k[n] \leftarrow \begin{cases} g_k[n] & \text{if } |g_k[n]| > \epsilon \cdot \max|g_k| \\ 0 & \text{otherwise} \end{cases} \tag{64}$$

where ϵ is the sparsity threshold (e.g., 0.01 for 1%).

```
fn apply_sparsity_threshold(kernel: &mut [Complex<f64>],
                            threshold: f64) {
    let max_magnitude = kernel.iter()
                              .map(|c| c.norm())
                              .fold(0.0, f64::max);

    let absolute_threshold = max_magnitude * threshold;

    for coefficient in kernel.iter_mut() {
        if coefficient.norm() < absolute_threshold {
            *coefficient = Complex::new(0.0, 0.0);
        }
    }
}
```

Typical sparsity levels: 60–80% of kernel coefficients can be zeroed with $\epsilon = 0.01$ with negligible quality loss.

Computational benefit: Sparse kernels reduce the number of operations in time-domain convolution by skipping zero multiplications.

Sparsity Threshold Artifacts

Warning: Aggressive sparsity thresholds (e.g., $\epsilon > 0.01$ or zeroing coefficients below 10^{-10}) can introduce audible artifacts:

- **Spectral leakage:** Truncating kernel tails creates discontinuities, causing frequency bleeding between adjacent bins.
- **Musical noise:** Sparse, time-varying kernels can produce tonal artifacts that vary frame-to-frame, perceived as "chirping" or "burbling" sounds.
- **Time-smearing:** Excessively truncated kernels lose their designed frequency selectivity.

Recommended practice: Use conservative thresholds ($\epsilon \leq 0.001$) for high-quality audio analysis. For real-time or resource-constrained applications, validate perceptual quality through listening tests before deploying aggressive sparsity.

8 Advanced Audio Features

8.1 Mel-Frequency Cepstral Coefficients (MFCC)

MFCCs are the most widely used features in speech recognition, speaker identification, and audio classification [8, 20]. Introduced by Davis and Mermelstein in 1980, they provide a compact representation of the spectral envelope.

Computation pipeline:

1. Compute mel-scaled power spectrogram: $S_{\text{mel}}[m, k]$
2. Apply logarithm: $L[m, k] = \log(S_{\text{mel}}[m, k] + \epsilon)$
3. Apply Discrete Cosine Transform (DCT-II) to each frame
4. Optionally apply cepstral liftering
5. Extract first C coefficients (typically 13)

Why DCT? The mel spectrogram has correlated bins due to overlapping triangular filters. The ***Discrete Cosine Transform*** (DCT) [1] decorrelates them and compacts energy into the first few coefficients.

Discrete Cosine Transform (DCT-II)

The DCT-II formula:

$$c[k] = \sum_{n=0}^{N-1} x[n] \cos\left(\frac{\pi k(n + 0.5)}{N}\right), \quad k = 0, 1, \ldots, N - 1 \tag{65}$$

The DCT is closely related to the DFT but operates entirely on real numbers. It's used in JPEG image compression and many audio codecs due to its excellent energy compaction properties.

```
fn dct_ii(input: &[f64]) -> Vec<f64> {
    let n = input.len();
    let mut output = vec![0.0; n];

    for k in 0..n {
        let mut sum = 0.0;
        for (i, &val) in input.iter().enumerate() {
            sum += val * (PI * k as f64 * (i as f64 + 0.5) / n as f64).
                cos();
        }
        output[k] = sum;
    }

    output
}
```

Cepstral liftering:

Liftering applies a sinusoidal weighting to emphasize mid-range coefficients:

$$\tilde{c}[k] = c[k]\left(1 + \frac{L}{2}\sin\left(\frac{\pi k}{L}\right)\right) \tag{66}$$

where L is the lifter parameter (commonly 22).

```
fn apply_liftering(mfcc: &mut Array2<f64>, lifter: usize) {
    let n_mfcc = mfcc.nrows();

    // Compute lifter weights
    let weights: Vec<_> = (0..n_mfcc)
        .map(|i| 1.0 + (lifter as f64 / 2.0)
                     * (PI * i as f64 / lifter as f64).sin())
        .collect();

    // Apply weights to each frame
    for frame_idx in 0..n_frames {
        for coeff_idx in 0..n_mfcc {
            mfcc[[coeff_idx, frame_idx]] *= weights[coeff_idx];
        }
    }
}
```

The C0 coefficient: The zeroth coefficient represents the overall energy of the frame. Some applications include it (speaker recognition), others exclude it (speech recognition) to reduce sensitivity to volume.

Why 13 coefficients? Empirically, 13 MFCCs capture most speech information. Higher coefficients represent fine spectral detail that's often noise.

8.2 Chroma Features for Music Analysis

Chroma features [11, 18] (also called pitch class profiles) represent the energy distribution across the 12 pitch classes of the Western musical scale, collapsing all octaves together.

The 12 pitch classes: C, C♯, D, D♯, E, F, F♯, G, G♯, A, A♯, B

Construction:

1. Map FFT bins to musical notes using:

$$n(f) = 12 \log_2 \left(\frac{f}{f_{\text{ref}}} \right) \tag{67}$$

   where f_{ref} is a reference frequency (e.g., A4 = 440 Hz)
2. Determine pitch class: $p = n \mod 12$
3. Sum energy from all octaves for each pitch class:

$$C[p] = \sum_{\text{octaves}} E[p, \text{octave}] \tag{68}$$

4. Normalize (L1, L2, or max)

Normalization strategies:

- **L1:** $\tilde{C}[p] = C[p] / \sum_{p'} C[p']$ (probabilities)
- **L2:** $\tilde{C}[p] = C[p] / \sqrt{\sum_{p'} C[p']^2}$ (Euclidean norm)
- **Max:** $\tilde{C}[p] = C[p] / \max_{p'} C[p']$ (relative strength)

Applications: Chord recognition, key estimation, cover song identification, music similarity.

8.3 ERB Spectrograms and Gammatone Filters

ERB spectrograms using gammatone filters provide the most perceptually accurate time-frequency representation.

The gammatone impulse response:

$$g(t) = a t^{n-1} e^{-2\pi b t} \cos(2\pi f_c t + \phi) \tag{69}$$

where:

- a is amplitude
- n is filter order (typically 4)
- b is bandwidth parameter
- f_c is center frequency
- ϕ is phase

The bandwidth parameter relates to ERB:

$$b = 1.019 \cdot \text{ERB}(f_c) \tag{70}$$

Frequency domain implementation:

For efficiency, the library implements gammatone filtering in the frequency domain. The frequency-domain transfer function is:

$$G(f) = \frac{1}{\left(1 + j\frac{f - f_c}{b}\right)^n} \tag{71}$$

where n is the filter order (typically 4), f_c is the center frequency, b is the bandwidth parameter, and $j = \sqrt{-1}$. This fourth-order complex pole provides the characteristic asymmetric frequency response of the auditory filter.

Filtering is then performed via element-wise multiplication:

$$Y(f) = X(f) \cdot G(f) \tag{72}$$

This avoids expensive time-domain convolution for each filter.

9 Image Processing via FFT

9.1 2D Convolution and the Convolution Theorem

The ***convolution theorem*** [4] states that convolution in the spatial domain is equivalent to multiplication in the frequency domain:

$$f * g = \mathcal{F}^{-1}\{\mathcal{F}\{f\} \cdot \mathcal{F}\{g\}\} \tag{73}$$

For images:

$$(I * K)[x, y] = \text{IFFT2D}\{\text{FFT2D}(I) \odot \text{FFT2D}(K)\} \tag{74}$$

where $\odot$ denotes element-wise multiplication.

Complexity comparison:

Spatial convolution for image $M \times N$ with kernel $K \times K$:

$$\mathcal{O}(M \times N \times K^2) \tag{75}$$

FFT-based convolution:

$$\mathcal{O}(MN \log(MN)) \tag{76}$$

Theoretical crossover: FFT has better asymptotic complexity when $K > \sqrt{\log(MN)}$, typically around $K \geq 7$.

9.2 Kernel Padding and Phase Shifting

For correct FFT convolution, the kernel must be padded to image size with proper phase shifting. The kernel's center should map to position $(0, 0)$ in the padded array:

```
fn pad_kernel_for_fft(kernel: &Array2<f64>,
                      target_shape: (usize, usize)) -> Array2<f64> {
    let (ker_rows, ker_cols) = kernel.dim();
    let mut result = Array2::<f64>::zeros(target_shape);

    // Kernel center position
    let ker_center_row = ker_rows / 2;
    let ker_center_col = ker_cols / 2;

    // Place kernel with center at (0, 0) using wraparound
    for i in 0..ker_rows {
        for j in 0..ker_cols {
            let row_offset = i as isize - ker_center_row as isize;
            let col_offset = j as isize - ker_center_col as isize;

            // Wrap around to place center at (0, 0)
            let target_row = row_offset.rem_euclid(target_rows as isize
                ) as usize;
            let target_col = col_offset.rem_euclid(target_cols as isize
                ) as usize;

            result[[target_row, target_col]] = kernel[[i, j]];
        }
    }

    result
}
```

This ensures the FFT interprets the kernel correctly for circular convolution.

9.3 Spatial Filtering in the Frequency Domain

Filtering is multiplication by a frequency-domain mask:

$$Y(u,v) = X(u,v) \cdot H(u,v) \tag{77}$$

Low-pass filter: Keeps low frequencies, removes high frequencies (smoothing)

$$H_{\text{LP}}(u,v) = \begin{cases} 1 & \sqrt{u^2+v^2} \le D_0 \\ 0 & \text{otherwise} \end{cases} \tag{78}$$

High-pass filter: Removes low frequencies, keeps high frequencies (sharpening/edges)

$$H_{\text{HP}}(u,v) = 1 - H_{\text{LP}}(u,v) \tag{79}$$

Band-pass filter: Keeps frequencies in a range

$$H_{\text{BP}}(u,v) = \begin{cases} 1 & D_1 \le \sqrt{u^2+v^2} \le D_2 \\ 0 & \text{otherwise} \end{cases} \tag{80}$$

9.4 Gaussian Blur via FFT

The Gaussian kernel for blurring:

$$G(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right) \tag{81}$$

Implementation:

```
pub fn gaussian_kernel_2d(size: NonZeroUsize, sigma: f64) ->
    SpectrogramResult<Array2<f64>> {
     if size.get().is_multiple_of(2) {
         return Err(SpectrogramError::invalid_input(
             "kernel size must be odd and > 0",
         ));
     }
     if sigma <= 0.0 {
         return Err(SpectrogramError::invalid_input("sigma must be > 0")
            );
     }
     let size = size.get();
     let center = (size / 2) as f64;
     let variance = sigma * sigma;
     let coeff = 1.0 / (2.0 * PI * variance);

     let mut kernel = Array2::<f64>::zeros((size, size));

     for i in 0..size {
         for j in 0..size {
             let x = i as f64 - center;
             let y = j as f64 - center;
             let exponent = -(x * x + y * y) / (2.0 * variance);
             kernel[[i, j]] = coeff * exponent.exp();
         }
     }

     // Normalize to sum to 1.0
     let sum: f64 = kernel.iter().sum();
     kernel.mapv_inplace(|v| v / sum);

     Ok(kernel)
}
```

The Gaussian has a useful property: its Fourier transform is also Gaussian, ensuring smooth frequency response.

9.5 Edge Detection and High-Pass Filtering

Edges correspond to high-frequency components. A simple high-pass filter enhances edges:

$$\text{Edges} = \text{Image} - \text{Lowpass}(\text{Image}) \tag{82}$$

Or apply a high-pass mask directly in frequency domain:

$$H_{\text{HP}}(u, v) = \begin{cases} 0 & \sqrt{u^2 + v^2} < D_0 \\ 1 & \text{otherwise} \end{cases} \tag{83}$$

The Laplacian operator for edge detection:

$$\nabla^2 I = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2} \tag{84}$$

In frequency domain, differentiation becomes multiplication:

$$\mathcal{F}\{\nabla^2 I\} = -(u^2 + v^2)\mathcal{F}\{I\} \tag{85}$$

10 Computational Optimizations

10.1 Plan-Based Computation

FFT planning involves:

1. Analyzing the transform size
2. Selecting optimal algorithm (radix-2, split-radix, Bluestein, etc.)
3. Precomputing twiddle factors: $W_N^k = e^{-j2\pi k/N}$
4. Allocating scratch buffers

For a single FFT, planning overhead is negligible. But for spectrograms with hundreds of frames, planning each FFT is wasteful.

Solution: Reusable plans

```
pub struct StftPlan {
    n_fft: usize,
    hop: usize,
    window: Vec<f64>,       // Pre-computed window
    fft: Box<dyn R2cPlan>, // Reusable FFT plan
    fft_out: Vec<Complex<f64>>, // Reusable buffer
    frame: Vec<f64>,        // Reusable frame buffer
}

impl StftPlan {
    pub fn new(params: &SpectrogramParams) -> Result<Self> {
        let n_fft = params.stft().n_fft();
        let window = make_window(params.stft().window(), n_fft)?;

        // Create FFT plan once
        let mut planner = RealFftPlanner::new();
        let fft = planner.plan_r2c(n_fft)?;

        Ok(Self {
            n_fft,
            window,
            fft,
            fft_out: vec![Complex::new(0.0, 0.0); n_fft/2 + 1],
            frame: vec![0.0; n_fft],
        })
    }

    // Reuse plan for each frame
    fn compute_frame(&mut self, samples: &[f64],
                     frame_idx: usize) -> Result<()> {
        // Window frame into self.frame buffer
        // ...

        // Reuse FFT plan and buffers
        self.fft.process(&self.frame, &mut self.fft_out)?;
        Ok(())
    }
}
```

Computational benefit: For 1000-frame spectrograms, plan reuse amortizes the planning overhead across all frames.

10.2 Sparse Matrix Operations

For filterbank multiplication $\mathbf{y} = H\mathbf{x}$ where H is sparse:

Dense: $\mathcal{O}(mn)$ for $m \times n$ matrix

Sparse: $\mathcal{O}(nnz)$ where nnz is number of non-zeros

For mel filterbanks with 80 bands and 257 FFT bins:

- Dense: $80 \times 257 = 20{,}560$ ops
- Sparse ($\sim$20 non-zeros/row): $80 \times 20 = 1{,}600$ ops

Row-wise sparse storage:

```
struct SparseMatrix {
    nrows: usize,
    ncols: usize,
    values: Vec<Vec<f64>>,      // Non-zero values per row
    indices: Vec<Vec<usize>>,  // Column indices per row
}

fn multiply_vec(&self, input: &[f64], out: &mut [f64]) {
    for (row_idx, (row_values, row_indices)) in
        self.values.iter().zip(&self.indices).enumerate() {

        let mut acc = 0.0;
        for (&value, &col_idx) in
            row_values.iter().zip(row_indices) {
            acc += value * input[col_idx];
        }
        out[row_idx] = acc;
    }
}
```

Cache efficiency: Row-wise format ensures sequential access to both input array and sparse values, maximizing cache hits.

10.3 Zero-Copy Design Patterns

Minimize allocations in hot loops:

Bad: Allocate per iteration

```
for frame in 0..n_frames {
    let mut windowed = vec![0.0; n_fft]; // Allocation!
    // ... process frame
}
```

Good: Reuse workspace

```
let mut workspace = vec![0.0; n_fft]; // Single allocation
for frame in 0..n_frames {
    // Reuse workspace
    fill_frame(&mut workspace, samples, frame);
    // ... process frame
}
```

Workspace pattern:

```
struct Workspace {
    frame: Vec<f64>,           // Windowed frame buffer
    fft_out: Vec<Complex<f64>>, // FFT output buffer
    spectrum: Vec<f64>,        // Power spectrum buffer
    mapped: Vec<f64>,          // Filterbank output buffer
}

impl Workspace {
    fn ensure_sizes(&mut self, n_fft: usize,
                    out_len: usize, n_bins: usize) {
        self.frame.resize(n_fft, 0.0);
        self.fft_out.resize(out_len, Complex::new(0.0, 0.0));
        self.spectrum.resize(out_len, 0.0);
        self.mapped.resize(n_bins, 0.0);
    }
}
```

10.4 Memory Layout and Cache Efficiency

Row-major vs. column-major:

The library uses row-major layout (C-style) for compatibility with most FFT libraries:

```
// Row-major: rows are contiguous
let data = Array2::<f64>::zeros((nrows, ncols));
assert!(data.is_standard_layout()); // row-major
```

For column-wise operations (processing spectrogram frames), this can cause cache misses. But the benefit of contiguous memory for FFT outweighs this.

Alignment: Modern FFT libraries (FFTW, RealFFT) benefit from aligned memory. The library ensures contiguous slices are passed to FFT routines.

10.5 FFTW Integration

FFTW (*Fastest Fourier Transform in the West*) [10] is considered the gold standard for FFT performance. The library supports both:

- **RealFFT:** Pure Rust, portable, good performance
- **FFTW:** C library, excellent performance, requires system installation

FFTW wisdom:

FFTW learns optimal algorithms through runtime measurements. Plans can be saved and reused:

```
// FFTW planning flags:
// ESTIMATE: Fast planning, ok performance
// MEASURE: Slower planning, better performance
// PATIENT: Very slow planning, best performance
let plan = fftw::plan_r2c_1d(n_fft, FFTW_MEASURE);
```

For repeated transforms of the same size, MEASURE or PATIENT modes can provide better performance than ESTIMATE.

11 Numerical Considerations

11.1 Floating-Point Precision

The library uses `f64` (double precision) throughout:

- 53 bits of precision ($\approx$16 decimal digits)
- Dynamic range: $\approx 10^{-308}$ to 10^{308}
- Sufficient for all audio processing needs

Why not f32? While f32 has lower memory requirements, audio processing requires:

- Accurate accumulation over many samples (STFT frames)
- Stable recursive filters (if implemented)
- Wide dynamic range for decibel conversions

f64 ensures numerical stability without performance concerns on modern CPUs.

11.2 Normalization Conventions

Different conventions exist for FFT normalization. The library uses:

Forward FFT: No normalization

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi kn/N} \tag{86}$$

Inverse FFT: Divide by N

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k]e^{j2\pi kn/N} \tag{87}$$

This ensures IFFT(FFT(x)) $= x$ exactly.

Alternative conventions (used by some libraries):

- Both directions scaled by $1/\sqrt{N}$ (symmetric)
- Inverse scaled by $1/N$, forward by N (rare)

11.3 Dynamic Range and Epsilon Thresholds

Logarithm floor:

To avoid $\log(0) = -\infty$:

```
let epsilon = 1e-10;
let db = 10.0 * (power.max(epsilon)).log10();
```

Choosing ϵ:

- Too large: Loss of quiet signals
- Too small: Numerical instability
- Sweet spot: 10^{-10} to 10^{-8} for f64

Sparse matrix threshold:

For considering values “zero”:

```
if value.abs() > 1e-10 {
    store(value);
}
```

This is much larger than machine epsilon ($\approx 2.2 \times 10^{-16}$ for f64) to account for accumulated rounding errors.

12 Conclusion and Further Reading

12.1 Summary of Key Concepts

This manual covered:

Core algorithms:

- DFT and FFT (Cooley-Tukey algorithm)
- Real-to-complex FFT with Hermitian symmetry
- 2D FFT via row-column decomposition
- STFT and time-frequency uncertainty

Signal processing methods:

- Window functions (Hanning, Hamming, Blackman, Kaiser, Gaussian)
- Amplitude scaling (power, magnitude, decibels)
- Convolution theorem for efficient filtering

Perceptual scales:

- Mel scale for speech processing
- ERB scale (Glasberg & Moore) for auditory modeling
- Logarithmic frequency for music
- CQT with variable-length kernels

Advanced features:

- MFCCs via DCT for speech recognition
- Chroma features for music analysis
- Gammatone filters for perceptual accuracy

Optimizations:

- Plan-based computation
- Sparse matrix operations
- Zero-copy workspace patterns
- Cache-efficient memory layouts

12.2 Applications and Extensions

The techniques in this manual enable:

- Speech recognition and speaker identification
- Music information retrieval
- Audio classification and tagging
- Sound event detection
- Acoustic scene analysis
- Image denoising and enhancement
- Texture analysis
- Pattern recognition

Software:

- FFTW: `www.fftw.org`
- Librosa (Python): `librosa.org`
- Numpy (Python): `numpy.org`
- SciPy (Python): `scipy.org`
- This library: `github.com/jmg049/Spectrograms`

13 Python API Reference

13.1 Introduction

This section provides a complete reference for the Python API, demonstrating how to apply the mathematical concepts from previous sections in practice. All code examples use `samples` (audio data as `ndarray[float64]`) and `image` (image data as `ndarray[float64]`).

Installation and Imports

```
pip install spectrograms
```

```
import numpy as np
import spectrograms as sg
```

API Organization

The API consists of four components:

- **Parameter Classes:** Configuration objects
- **Computation Functions:** Single-use convenience functions
- **Planner Classes:** Reusable plans for batch processing
- **Result Objects:** Rich results with metadata

All computation functions release Python's GIL for parallel processing.

Error Handling

```
try:
    spec = sg.compute_mel_power_spectrogram(samples, params, mel_params
        )
except sg.InvalidInputError as e:
    print(f"Invalid␣parameters:␣{e}")
except sg.DimensionMismatchError as e:
    print(f"Array␣size␣mismatch:␣{e}")
except sg.FFTBackendError as e:
    print(f"FFT␣computation␣error:␣{e}")
except sg.SpectrogramError as e:
    print(f"Library␣error:␣{e}")
```

Basic FFT Operations

1D FFT

Compute the Discrete Fourier Transform (Equation (1)):

```
n_fft: int = 512

# Compute FFT
spectrum: npt.NDArray[np.complex128] = sg.compute_fft(samples[:n_fft],
    n_fft)
print(f"Output␣shape:␣{spectrum.shape}")  # (257,) due to Hermitian
    symmetry
print(f"Output␣dtype:␣{spectrum.dtype}")  # complex128
```

Inverse FFT

Perfect reconstruction (Equation (3)):

```
spectrum: npt.NDArray[np.complex128] = sg.compute_fft(samples[:512],
    512)
reconstructed: npt.NDArray[np.float64] = sg.compute_irfft(spectrum,
    512)

error: float = np.max(np.abs(samples[:512] - reconstructed))
print(f"Reconstruction␣error:␣{error:.2e}")  # < 1e-14
```

Power and Magnitude Spectra

From Section 5.1:

```
# Power spectrum: |X[k]|^2
power: npt.NDArray[np.float64] = sg.compute_power_spectrum(samples
    [:512], 512)

# Magnitude spectrum: |X[k]|
magnitude: npt.NDArray[np.float64] = sg.compute_magnitude_spectrum(
    samples[:512], 512)

# With windowing (sec:windows)
power_windowed: npt.NDArray[np.float64] = sg.compute_power_spectrum(
    samples[:512], 512, window=WindowType.hanning())
```

Window Functions

Predefined Windows

Window functions from Section 3:

```
# WindowType class methods
rect: sg.WindowType = sg.WindowType.rectangular()
hann: sg.WindowType = sg.WindowType.hanning()
hamming: sg.WindowType = sg.WindowType.hamming()
blackman: sg.WindowType = sg.WindowType.blackman()

# Use with compute functions
power: npt.NDArray[np.float64] = sg.compute_power_spectrum(samples
    [:512], 512, window=hann)
```

Parametric Windows

Kaiser (Section 3.7) and Gaussian (Section 3.8):

```
# Kaiser window with beta parameter
kaiser: sg.WindowType = sg.WindowType.kaiser(beta=8.6)
power_kaiser: npt.NDArray[np.float64] = sg.compute_power_spectrum(
    samples[:512], 512, window=kaiser)

# Gaussian window with std
gaussian: sg.WindowType = sg.WindowType.gaussian(std=0.4)
power_gaussian: npt.NDArray[np.float64] = sg.compute_power_spectrum(
    samples[:512], 512, window=gaussian)
```

STFT and Spectrograms

Configuration

STFT parameters from Section 4.2:

```
# Manual configuration
window: sg.WindowType = sg.WindowType.hanning()
stft: sg.StftParams = sg.StftParams(
    n_fft=512,
    hop_size=160,
    window=window,
    centre=True
)
params: sg.SpectrogramParams = sg.SpectrogramParams(stft, sample_rate
    =16000)

# Presets
params_speech: sg.SpectrogramParams = sg.SpectrogramParams.
    speech_default(16000)
params_music: sg.SpectrogramParams = sg.SpectrogramParams.music_default
    (44100)
```

Linear Spectrograms

Three amplitude representations (Section 5):

```
params: sg.SpectrogramParams = sg.SpectrogramParams.speech_default
    (16000)

# Power: |X[k]|^2
spec_power: sg.Spectrogram = sg.compute_linear_power_spectrogram(
    samples, params)

# Magnitude: |X[k]|
spec_magnitude: sg.Spectrogram = sg.
    compute_linear_magnitude_spectrogram(samples, params)

# Decibels: 10*log10(|X[k]|^2)
db_params: sg.LogParams = sg.LogParams(floor_db=-80.0)
spec_db: sg.Spectrogram = sg.compute_linear_db_spectrogram(samples,
    params, db_params)

print(f"Shape:␣{spec_power.shape}")  # (257, n_frames)
```

Spectrogram Result Object

```
spec: sg.Spectrogram = sg.compute_linear_db_spectrogram(samples, params
    , db_params)

# Access as ndarray via buffer protocol
data: npt.NDArray[np.float64] = np.asarray(spec)  # Shape: (n_bins,
    n_frames)

# Direct property access
frequencies: npt.NDArray[np.float64] = spec.frequencies
times: npt.NDArray[np.float64] = spec.times
n_bins: int = spec.n_bins
n_frames: int = spec.n_frames

# Methods
duration: float = spec.duration()
freq_range: tuple[float, float] = spec.frequency_range()
db_range: tuple[float, float] | None = spec.db_range()
```

Perceptual Frequency Scales

Mel Scale

Mel-scale spectrograms (Section 6.2):

```
mel_params: sg.MelParams = sg.MelParams(
    n_mels=80,
    f_min=0.0,
    f_max=8000.0
)

# Three amplitude types
mel_power: sg.Spectrogram = sg.compute_mel_power_spectrogram(samples,
   params, mel_params)
mel_magnitude: sg.Spectrogram = sg.compute_mel_magnitude_spectrogram(
   samples, params, mel_params)
mel_db: sg.Spectrogram = sg.compute_mel_db_spectrogram(samples, params,
    mel_params, db_params)

print(f"Mel␣shape:␣{mel_power.shape}")  # (80, n_frames)
print(f"Mel␣frequencies:␣{mel_power.frequencies[:5]}")
```

ERB Scale

ERB-scale spectrograms (Section 6.3):

```
erb_params: sg.ErbParams = sg.ErbParams(
    n_filters=64,
    f_min=50.0,
    f_max=8000.0
)

erb_power: sg.Spectrogram = sg.compute_erb_power_spectrogram(samples,
   params, erb_params)
erb_magnitude: sg.Spectrogram = sg.compute_erb_magnitude_spectrogram(
   samples, params, erb_params)
erb_db: sg.Spectrogram = sg.compute_erb_db_spectrogram(samples, params,
    erb_params, db_params)
```

Logarithmic Hz Scale

Logarithmic frequency scale for music:

```
loghz_params: sg.LogHzParams = sg.LogHzParams(
    n_bins=84,
    f_min=55.0,
    f_max=7040.0
)

loghz_power: sg.Spectrogram = sg.compute_loghz_power_spectrogram(
   samples, params, loghz_params)
loghz_magnitude: sg.Spectrogram = sg.
   compute_loghz_magnitude_spectrogram(samples, params, loghz_params)
loghz_db: sg.Spectrogram = sg.compute_loghz_db_spectrogram(samples,
   params, loghz_params, db_params)
```

Constant-Q Transform

CQT (Section 7):

```
cqt_params: sg.CqtParams = sg.CqtParams(
    bins_per_octave=12,
    n_octaves=7,
    f_min=55.0
)

cqt_power: sg.Spectrogram = sg.compute_cqt_power_spectrogram(samples,
   params, cqt_params)
cqt_magnitude: sg.Spectrogram = sg.compute_cqt_magnitude_spectrogram(
   samples, params, cqt_params)
cqt_db: sg.Spectrogram = sg.compute_cqt_db_spectrogram(samples, params,
    cqt_params, db_params)
```

Audio Features

Raw STFT

Complex-valued STFT matrix:

```
stft_matrix: npt.NDArray[np.complex128] = sg.compute_stft(samples,
   params.stft)
print(f"STFT␣shape:␣{stft_matrix.shape}")  # (n_bins, n_frames)
print(f"STFT␣dtype:␣{stft_matrix.dtype}")  # complex128

# Extract magnitude and phase
magnitude: npt.NDArray[np.float64] = np.abs(stft_matrix)
phase: npt.NDArray[np.float64] = np.angle(stft_matrix)
```

Inverse STFT

Reconstruct signal from STFT (Section 3.10):

```
stft_matrix: npt.NDArray[np.complex128] = sg.compute_stft(samples,
   params.stft)

reconstructed: npt.NDArray[np.float64] = sg.compute_istft(
    stft_matrix,
    n_fft=params.stft.n_fft,
    hop_size=params.stft.hop_size,
    window=params.stft.window,
    centre=params.stft.centre
)

# Verify reconstruction
error: float = np.max(np.abs(samples[:len(reconstructed)] -
   reconstructed))
print(f"Error:␣{error:.2e}")
```

MFCC

Mel-Frequency Cepstral Coefficients (Section 8.1):

```
mfcc_params: sg.MfccParams = sg.MfccParams(
    n_mfcc=13,
    use_dct_ortho=True,
    lifter_coeff=22.0
)

mfccs: npt.NDArray[np.float64] = sg.compute_mfcc(
    samples,
    params.stft,
    sample_rate=16000,
    n_mels=40,
    mfcc_params=mfcc_params
)
print(f"MFCC␣shape:␣{mfccs.shape}")  # (13, n_frames)
```

Chromagram

Pitch class profiles (Section 8.2):

```
chroma_params: sg.ChromaParams = sg.ChromaParams(
    tuning=440.0,
    f_min=32.7,
    f_max=4186.0,
    norm=ChromaNorm.l2
)

chroma: sg.Chromagram = sg.compute_chromagram(
    samples,
    params.stft,
    sample_rate=16000,
    chroma_params=chroma_params
)

# Access as ndarray (12 pitch classes, n_frames)
chroma_data: npt.NDArray[np.float64] = np.asarray(chroma)

pitch_classes = ['C', 'C#', 'D', 'D#', 'E', 'F', 'F#', 'G', 'G#', 'A',
    'A#', 'B']
```

Performance Optimization

Plan Reuse

Reuse FFT plans for batch processing:

```python
# Single computation
spec1: sg.Spectrogram = sg.compute_mel_db_spectrogram(samples, params,
    mel_params, db_params)

# Batch with plan reuse
planner: sg.SpectrogramPlanner = sg.SpectrogramPlanner()
plan = planner.mel_db_plan(params, mel_params, db_params)

signals = [samples1, samples2, samples3]
spectrograms = [plan.compute(s) for s in signals]
```

Available Plans

All spectrogram types have corresponding plans:

```python
planner = sg.SpectrogramPlanner()

# Linear (3 types)
linear_power_plan = planner.linear_power_plan(params)
linear_magnitude_plan = planner.linear_magnitude_plan(params)
linear_db_plan = planner.linear_db_plan(params, db_params)

# Mel (3 types)
mel_power_plan = planner.mel_power_plan(params, mel_params)
mel_magnitude_plan = planner.mel_magnitude_plan(params, mel_params)
mel_db_plan = planner.mel_db_plan(params, mel_params, db_params)

# ERB (3 types)
erb_power_plan = planner.erb_power_plan(params, erb_params)
erb_magnitude_plan = planner.erb_magnitude_plan(params, erb_params)
erb_db_plan = planner.erb_db_plan(params, erb_params, db_params)

# LogHz (3 types)
loghz_power_plan = planner.loghz_power_plan(params, loghz_params)
loghz_magnitude_plan = planner.loghz_magnitude_plan(params,
    loghz_params)
loghz_db_plan = planner.loghz_db_plan(params, loghz_params, db_params)

# CQT (3 types)
cqt_power_plan = planner.cqt_power_plan(params, cqt_params)
cqt_magnitude_plan = planner.cqt_magnitude_plan(params, cqt_params)
cqt_db_plan = planner.cqt_db_plan(params, cqt_params, db_params)

# All plans use same interface
spec = plan.compute(samples)
```

Frame-by-Frame Processing

Process individual frames without full spectrogram:

```
planner = sg.SpectrogramPlanner()
plan = planner.mel_power_plan(params, mel_params)

n_frames = (len(samples) + params.stft.n_fft - params.stft.hop_size) //
    params.stft.hop_size

for frame_idx in range(n_frames):
    frame = plan.compute_frame(samples, frame_idx)  # 1D array (n_mels
        ,)
    # Process frame...
```

Streaming Processing

Real-time frame extraction:

```
planner = sg.SpectrogramPlanner()
plan = planner.mel_power_plan(params, mel_params)

buffer = np.array([], dtype=np.float64)
chunk_size = 160

for chunk in audio_stream:
    buffer = np.concatenate([buffer, chunk])

    while len(buffer) >= params.stft.n_fft:
        frame = plan.compute_frame(buffer, 0)
        # Process frame...
        buffer = buffer[params.stft.hop_size:]
```

2D FFT

Forward 2D FFT

2D FFT from Section 2.5:

```
image = np.load('test.png')  # Shape: (height, width)

spectrum = sg.fft2d(image)
print(f"Spectrum␣shape:␣{spectrum.shape}")  # (height, width//2 + 1)
print(f"Spectrum␣dtype:␣{spectrum.dtype}")  # complex64
```

Inverse 2D FFT

```
spectrum = sg.fft2d(image)
reconstructed = sg.ifft2d(spectrum, image.shape[1])

error = np.max(np.abs(image - reconstructed))
print(f"Reconstruction␣error:␣{error:.2e}")
```

2D Power and Magnitude Spectra

```
power = sg.power_spectrum_2d(image)        # Shape: (height, width//2 +
    1)
magnitude = sg.magnitude_spectrum_2d(image)

# Log scale for visualization
power_db = 10 * np.log10(power + 1e-10)
```

FFT Shift

Center DC component:

```
power = sg.power_spectrum_2d(image)

# Shift DC to center
power_shifted = sg.fftshift(power)

# Reverse shift
power_restored = sg.ifftshift(power_shifted)
```

Batch 2D FFT

```
planner = sg.Fft2dPlanner()

# Forward plan
fft_plan = planner.fft2d_plan(nrows, ncols)
spectra = [fft_plan.forward(img) for img in images]

# Inverse plan
ifft_plan = planner.ifft2d_plan(nrows, ncols)
reconstructed = [ifft_plan.inverse(spec) for spec in spectra]

# Power spectrum plan
power_plan = planner.power_spectrum_plan(nrows, ncols)
powers = [power_plan.compute(img) for img in images]

# Magnitude spectrum plan
magnitude_plan = planner.magnitude_spectrum_plan(nrows, ncols)
magnitudes = [magnitude_plan.compute(img) for img in images]
```

Image Filtering

Gaussian Blur

```
kernel = sg.gaussian_kernel_2d(size=15, sigma=2.0)
blurred = sg.convolve_fft(image, kernel)
```

Frequency Filters

```
# Lowpass: smooth (keep low frequencies)
lowpass = sg.lowpass_filter(image, cutoff_fraction=0.1)

# Highpass: edges (keep high frequencies)
highpass = sg.highpass_filter(image, cutoff_fraction=0.1)

# Bandpass: specific frequency range
bandpass = sg.bandpass_filter(image, low_cutoff=0.1, high_cutoff=0.5)
```

Edge Detection and Sharpening

```
# Edge detection
edges = sg.detect_edges_fft(image)

# Sharpening
sharpened = sg.sharpen_fft(image, amount=1.5)
```

Complete API Reference

Parameter Classes

WindowType:

```
sg.WindowType.rectangular()
sg.WindowType.hanning()
sg.WindowType.hamming()
sg.WindowType.blackman()
sg.WindowType.kaiser(beta: float)
sg.WindowType.gaussian(std: float)
```

StftParams:

```
stft = sg.StftParams(
    n_fft: int,
    hop_size: int,
    window: str | WindowType,
    center: bool = True
)
```

LogParams:

```
db_params = sg.LogParams(floor_db: float)
```

SpectrogramParams:

```
params = sg.SpectrogramParams(stft: StftParams, sample_rate: float)
params = sg.SpectrogramParams.speech_default(sample_rate: float)
params = sg.SpectrogramParams.music_default(sample_rate: float)
```

MelParams:

```
mel_params = sg.MelParams(
    n_mels: int,
    f_min: float = 0.0,
    f_max: float | None = None
)
```

ErbParams:

```
erb_params = sg.ErbParams(
    n_bands: int,
    f_min: float = 50.0,
    f_max: float | None = None,
    use_gammatone: bool = True
)
```

LogHzParams:

```
loghz_params = sg.LogHzParams(
    n_bins: int,
    f_min: float,
    f_max: float,
    bins_per_octave: int = 12
)
```

CqtParams:

```
cqt_params = sg.CqtParams(
    n_bins: int,
    f_min: float,
    bins_per_octave: int = 12,
    sparsity_threshold: float = 0.01
)
```

ChromaParams:

```
chroma_params = sg.ChromaParams(
    tuning_frequency: float = 440.0,
    n_chroma: int = 12,
    norm_type: str = "L2"
)
chroma_params = sg.ChromaParams.music_standard()
```

MfccParams:

```
mfcc_params = sg.MfccParams(
    n_mfcc: int = 13,
    use_energy: bool = True,
    lifter_coefficient: float = 22.0,
    normalize: bool = False
)
mfcc_params = sg.MfccParams.speech_standard()
```

Spectrogram Functions

All return `Spectrogram` objects with `.data`, `.frequencies`, `.times`, `.duration()`, `.frequency_range()`, `.db_range()`, however, they can also act as an `ndarray` via the array protocol with no additional function calls:

```
# Linear
sg.compute_linear_power_spectrogram(samples, params)
sg.compute_linear_magnitude_spectrogram(samples, params)
sg.compute_linear_db_spectrogram(samples, params, db_params)

# Mel
sg.compute_mel_power_spectrogram(samples, params, mel_params)
sg.compute_mel_magnitude_spectrogram(samples, params, mel_params)
sg.compute_mel_db_spectrogram(samples, params, mel_params, db_params)

# ERB
sg.compute_erb_power_spectrogram(samples, params, erb_params)
sg.compute_erb_magnitude_spectrogram(samples, params, erb_params)
sg.compute_erb_db_spectrogram(samples, params, erb_params, db_params)

# LogHz
sg.compute_loghz_power_spectrogram(samples, params, loghz_params)
sg.compute_loghz_magnitude_spectrogram(samples, params, loghz_params)
sg.compute_loghz_db_spectrogram(samples, params, loghz_params,
    db_params)

# CQT
sg.compute_cqt_power_spectrogram(samples, params, cqt_params)
sg.compute_cqt_magnitude_spectrogram(samples, params, cqt_params)
sg.compute_cqt_db_spectrogram(samples, params, cqt_params, db_params)
```

Audio Feature Functions

```
# STFT
stft_matrix = sg.compute_stft(samples, stft_params)  # complex128, (
    n_bins, n_frames)

# Inverse STFT
samples = sg.compute_istft(stft_matrix, n_fft, hop_size, window, center
    )

# MFCC
mfccs = sg.compute_mfcc(samples, stft_params, sample_rate, n_mels,
    mfcc_params)

# Chromagram
chroma = sg.compute_chromagram(samples, stft_params, sample_rate,
    chroma_params)
```

FFT Functions

```
# 1D FFT
spectrum = sg.compute_fft(samples, n_fft)  # complex128, (n_fft//2 +
    1,)
samples = sg.compute_irfft(spectrum, n_fft)

# Power/magnitude
power = sg.compute_power_spectrum(samples, n_fft, window)
magnitude = sg.compute_magnitude_spectrum(samples, n_fft, window)
```

2D FFT Functions

```
# 2D FFT
spectrum = sg.fft2d(image)  # complex64, (nrows, ncols//2 + 1)
image = sg.ifft2d(spectrum, output_ncols)

# Power/magnitude
power = sg.power_spectrum_2d(image)
magnitude = sg.magnitude_spectrum_2d(image)

# Shift
shifted = sg.fftshift(array)
restored = sg.ifftshift(shifted)
```

Image Processing Functions

```
# Convolution
kernel = sg.gaussian_kernel_2d(size, sigma)
result = sg.convolve_fft(image, kernel)

# Filters
lowpass = sg.lowpass_filter(image, cutoff_fraction)
highpass = sg.highpass_filter(image, cutoff_fraction)
bandpass = sg.bandpass_filter(image, low_cutoff, high_cutoff)

# Edge detection and sharpening
edges = sg.detect_edges_fft(image)
sharpened = sg.sharpen_fft(image, amount)
```

References

[1] Nasir Ahmed, T_ Natarajan, and Kamisetty R Rao. “Discrete cosine transform”. In: *IEEE transactions on Computers* 100.1 (2006), pp. 90–93.

[2] Jonathan Allen. “Short term spectral analysis, synthesis, and modification by discrete Fourier transform”. In: *IEEE transactions on acoustics, speech, and signal processing* 25.3 (2003), pp. 235–238.

[3] Ralph Beebe Blackman and John W Tukey. “The measurement of power spectra from the point of view of communications engineering”. In: *Bell System Technical Journal* 37.1 (1958), pp. 185–282.

[4] AP Blozinski. “On a convolution theorem for L (p, q) spaces”. In: *Transactions of the American Mathematical Society* 164 (1972), pp. 255–265.

[5] Judith C Brown. “Calculation of a constant Q spectral transform”. In: *The Journal of the Acoustical Society of America* 89.1 (1991), pp. 425–434.

[6] Judith C Brown and Miller S Puckette. “An efficient algorithm for the calculation of a constant Q transform”. In: *The Journal of the Acoustical Society of America* 92.5 (1992), pp. 2698–2701.

[7] James W Cooley and John W Tukey. “An algorithm for the machine calculation of complex Fourier series”. In: *Mathematics of computation* 19.90 (1965), pp. 297–301.

[8] Steven Davis and Paul Mermelstein. “Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences”. In: *IEEE transactions on acoustics, speech, and signal processing* 28.4 (1980), pp. 357–366.

[9] Harvey Fletcher. “Auditory patterns”. In: *Reviews of modern physics* 12.1 (1940), p. 47.

[10] Matteo Frigo and Steven G Johnson. “The design and implementation of FFTW3”. In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231.

[11] Takuya Fujishima. “Realtime chord recognition of musical sound: A system using common lisp music”. In: *Proceedings of the International Computer Music Conference.* 1999, pp. 464–467.

[12] Dennis Gabor. “Theory of communication. Part 1: The analysis of information”. In: *Journal of the Institution of Electrical Engineers-Part III: Radio and Communication Engineering* 93.26 (1946), pp. 429–441.

[13] Carl Friedrich Gauss. “Nachlass: Theoria interpolationis methodo nova tractata”. In: *Carl Friedrich Gauss Werke* 3 (1866). Original work from 1805, pp. 265–327.

[14] Brian R Glasberg and Brian CJ Moore. “Derivation of auditory filter shapes from notched-noise data”. In: *Hearing research* 47.1-2 (1990), pp. 103–138.

[15] Fredric J Harris. “On the use of windows for harmonic analysis with the discrete Fourier transform”. In: *Proceedings of the IEEE* 66.1 (1978), pp. 51–83.

[16] James F Kaiser. “Nonrecursive digital filter design using window function”. In: *Proceedings of the 1974 IEEE International Symposium on Circuits and Systems.* IEEE. 1974, pp. 20–23.

[17] Brian CJ Moore and Brian R Glasberg. “Suggested formulae for calculating auditory-filter bandwidths and excitation patterns”. In: *The journal of the Acoustical Society of America* 74.3 (1983), pp. 750–753.

[18] Meinard Müller, Frank Kurth, and Michael Clausen. “Audio Matching via Chroma-Based Statistical Features.” In: *ISMIR.* Vol. 2005. 2005, p. 6.

[19] Alan V Oppenheim. *Discrete-time signal processing.* Pearson Education India, 1999.

[20] Lawrence R Rabiner and Biing-Hwang Juang. “Fundamentals of speech recognition”. In: Prentice Hall, 1993.

[21] Stanley Smith Stevens, John Volkmann, and Edwin B Newman. “A scale for the measurement of the psychological magnitude pitch”. In: *The journal of the Acoustical Society of America* 8.3 (1937), pp. 185–190.