Advanced Real-Time FFT Analysis and Visualization: A Comprehensive Implementation Guide based on Metrological Standards

Executive Summary

The democratization of digital signal processing (DSP) through high-level languages like Python has bridged the gap between academic theory and practical instrumentation. However, transitioning from a basic script that computes a Fast Fourier Transform (FFT) to a robust, real-time spectrum analyzer involves a complex interplay of architectural decisions and mathematical corrections. This report serves as a definitive guide to implementing professional-grade FFT analysis, specifically aligning with the methodologies outlined in metrological standards and the technical documentation provided by Dewesoft.

The user's objective—to implement the techniques exposed in the Dewesoft FFT guide into a custom Python visualizer—requires a departure from standard hobbyist approaches. A simple call to numpy.fft.fft is insufficient for accurate measurement. Professional analysis necessitates a rigorous handling of spectral leakage through windowing, precise amplitude recovery via correction factors, psychoacoustic weighting (A-weighting), and temporal stabilization through advanced averaging modes (Exponential and Peak Hold).

This document deconstructs these requirements into an implementation roadmap. It begins by establishing the theoretical constraints of the Discrete Fourier Transform (DFT) in real-time systems, specifically addressing the Heisenberg-Gabor limit of time-frequency resolution. It then proceeds to a granular analysis of signal conditioning, detailing the mathematical derivation and implementation of specific window functions (Hann vs. Flat Top) and their requisite correction factors. The report further explores the architecture of real-time systems, advocating for a ring-buffer topology to manage the asynchronous nature of audio streams and processing threads. Finally, it surveys the existing open-source landscape, critiquing current GitHub repositories against professional standards to identify reusable patterns and pitfalls.

---

1. The Physics and Mathematics of Digital Spectral Analysis

To construct a visualizer that not only "looks" correct but "measures" correctly, one must first

understand the mathematical transformations occurring within the signal chain. The FFT is not merely a sorting algorithm; it is a projection of a time-domain signal onto a set of orthogonal basis functions (complex sinusoids).

1.1 The Discrete Fourier Transform (DFT)

At the heart of the analyzer is the DFT, which transforms a finite sequence of equally spaced samples of a function into a same-length sequence of equally spaced samples of the discrete-time Fourier transform (DTFT), which is a complex-valued function of frequency.

For a signal $x[n]$ with N samples, the DFT is defined as:

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

Where:

- $X[k]$ is the complex spectral coefficient for the k-th frequency bin.
- N is the frame size (block size).
- k ranges from 0 to $N-1$.

The output $X[k]$ describes the amplitude and phase of the sinusoidal component at frequency $f_k = k \cdot \frac{F_s}{N}$, where F_s is the sampling rate.[1]

1.1.1 The Fast Fourier Transform (FFT) Optimization

Direct computation of the DFT requires $O(N^2)$ operations, which is prohibitively expensive for real-time audio (e.g., $N = 4096$ at 60 Hz refresh rate). The FFT, popularized by Cooley and Tukey, reduces this complexity to $O(N \log N)$ by exploiting the symmetry and periodicity of the complex exponential basis functions.[2] In Python, numpy.fft and scipy.fft leverage optimized C and Fortran libraries (like FFTPACK or PocketFFT) to perform these operations within microseconds, making Python a viable environment for real-time analysis despite its interpreted nature.[1]

1.2 The Time-Frequency Uncertainty Principle

A fundamental constraint in FFT analysis, often a source of confusion for developers, is the trade-off between time resolution and frequency resolution. This is analogous to the Heisenberg Uncertainty Principle in quantum mechanics.

- **Frequency Resolution (Δf):** Defined as $\Delta f = \frac{F_s}{N}$. To distinguish between two closely spaced frequencies (e.g., 50 Hz and 55 Hz), one needs a Δf smaller than their difference. This requires a large N.
- **Time Resolution (Δt):** Defined as the duration of the signal block, $T = \frac{N}{F_s}$. A large N means the system waits longer to collect data, reducing the temporal responsiveness of the display.

Implication for Implementation:

For a standard audio sampling rate of 48 kHz:

- An FFT size of $N = 1024$ yields $\Delta f \approx 46.9$ Hz and latency ≈ 21.3 ms. This is responsive but coarse.
- An FFT size of $N = 8192$ yields $\Delta f \approx 5.8$ Hz but latency ≈ 170 ms. This provides detail but feels "sluggish."

Professional analyzers, as noted in the Dewesoft guide, often allow users to adjust the "line resolution" (number of bins), which implicitly changes the block size N.[4] The implementation must therefore support variable buffer sizes dynamically.

1.3 Aliasing and the Nyquist-Shannon Theorem

The Nyquist-Shannon sampling theorem dictates that a digital system can only accurately represent frequencies up to half the sampling rate ($F_s/2$). Frequencies above this limit "fold over" (alias) into the baseband, creating phantom signals.

While the hardware analog-to-digital converter (ADC) typically includes an anti-aliasing filter, the software implementation must respect the Nyquist limit. The FFT output is symmetric for real-valued signals; the upper half of the array (from $N/2$ to N) is a complex conjugate mirror of the lower half and contains no new information.

Implementation Constraint: When using numpy.fft.fft, the output array has length N. The visualizer must discard the second half (indices $N/2$ to N) and multiply the amplitude of the first half (indices 1 to $N/2 - 1$) by 2 to conserve energy (Parseval's theorem). The DC component (index 0) and Nyquist component (index $N/2$) are unique and should not be doubled in the same manner.[5]

2. The Architecture of Real-Time Analysis Systems

Building a script that plots a static wav file is trivial; building a system that visualizes live audio without stuttering (under-runs) or freezing the GUI (blocking) requires a sophisticated concurrent architecture. The Dewesoft guide implies a continuous stream of analysis, which necessitates a decoupled producer-consumer model.

2.1 The Python Global Interpreter Lock (GIL) and Audio Threads

Python's GIL prevents multiple native Python threads from executing bytecodes simultaneously. However, signal acquisition libraries like sounddevice (based on PortAudio) operate their callback functions in a separate C-thread.[6]

Crucial Warning: Operations inside the audio callback must be "real-time safe." They must not allocate memory, wait on locks, or perform heavy computation. Performing an FFT or a plot update inside the audio callback will inevitably cause audio glitches ("dropouts").

2.2 The Ring Buffer (Circular Buffer) Strategy

To manage the continuous flow of data, a **Ring Buffer** is the standard architectural component. It serves as a reservoir that absorbs the jitter between the precise clock of the audio hardware and the variable scheduling of the operating system's video refresh.[8]

2.2.1 Data Flow Topology

1. **Hardware Producer:** The audio interface (via sounddevice) writes small chunks (e.g., 512 samples) into the Ring Buffer's write head.
2. **Analysis Consumer:** A separate processing thread wakes up periodically (e.g., every 30ms for 33 FPS), reads the *last* N samples from the buffer, processes them, and pushes the result to a display queue.
3. **Visualization Consumer:** The GUI thread (Main Thread) pops processed spectra from the display queue and updates the graphics.

This decoupling allows the FFT size (N) to be independent of the audio hardware's buffer size. For instance, the hardware might deliver data in 512-sample chunks, but the FFT might process 4096-sample windows overlapping by 50%. The Ring Buffer abstracts this mismatch.[10]

2.3 Overlap Processing (Overlap-Add/Save logic)

The Dewesoft guide and advanced DSP literature emphasize "Overlap." In spectrum analysis, this typically refers to processing overlapping time windows.

Why Overlap? Most window functions (like Hann) attenuate the signal to zero at the edges. If we process contiguous blocks (0-1024, 1024-2048), the data at sample 1024 is effectively ignored. By overlapping blocks (e.g., 0-1024, 512-1536), we ensure that the center of the next window covers the edge of the previous one, ensuring all data contributes to the spectral estimate.[4]

Implementation Detail:

With a Ring Buffer, implementing overlap is implicit. If the write pointer advances by 512 samples (Hop Size), but the read operation requests the "most recent 1024 samples," the system automatically creates a 50% overlap with the previous frame.

---

3. Signal Conditioning: Windowing Strategies and Correction Factors

This section directly addresses the "Windowing" techniques highlighted in the Dewesoft guide. The raw FFT assumes the signal is periodic within the window. Since real-world signals are not, "Spectral Leakage" occurs, smearing energy across the spectrum.

3.1 Spectral Leakage and the Window Function

Spectral leakage manifests as "skirts" around a spectral peak. A pure sine wave, which should appear as a single thin line, typically appears as a wide mountain. This obscures nearby smaller signals and reduces the effective dynamic range of the analyzer.[12]

To mitigate this, we multiply the time-domain signal frame by a "Window Function" that tapers the edges to zero, smoothing the discontinuity.

3.2 Comparison of Window Types

The choice of window is a trade-off between Amplitude Accuracy and Frequency Selectivity. The Dewesoft guide typically highlights the following standard windows:

Window Type	Main Lobe Width	Max Side Lobe Level	Application	Dewesoft Recommendation
Rectangular	Narrowest	-13 dB	Transients, separations of close tones	No (unless for transients)

Hann (Hanning)	Medium	-32 dB	General purpose, random noise	**Default / Standard**
Hamming	Medium	-43 dB	Narrowband signals	Alternative to Hann
Flat Top	Widest	-93 dB	Calibration, Exact Amplitude measurement	**Amplitude Accuracy**
Blackman	Wide	-58 dB	High dynamic range	Low noise floor analysis

Analytical Insight:

- **Hann:** The industry workhorse. It offers a good balance. If the user listens to music or analyzes noise, Hann is the correct default.[4]
- **Flat Top:** Essential for "voltmeter-like" accuracy. If the input is a 1V sine wave, a Hann window might report 0.85V or 0.9V depending on exact frequency alignment (scalloping loss). A Flat Top window will consistently report 1.0V (within <0.1% error) but will blur the frequency peak across several bins.[15]

3.3 Window Correction Factors

A critical omission in many open-source visualizers is the **Correction Factor**. When a window function tapers the signal, it removes energy. If you sum the squared amplitudes of the windowed FFT, the result will be less than the energy of the original signal. To restore the correct units, we must apply a scaling factor.[17]

There are two distinct types of correction factors mandated by professional standards:

1. **Amplitude Correction Factor (ACF):** Used for periodic (deterministic) signals like sine waves. It compensates for the coherent gain reduction.

$$ACF = \frac{1}{\frac{1}{N}\sum_{n=0}^{N-1} w[n]} = \frac{1}{\text{mean}(w)}$$

   - For Hann: $ACF = 2.0$
   - For Flat Top: $ACF \approx 4.18$ (varies by implementation)
2. **Energy Correction Factor (ECF):** Used for random (stochastic) signals like noise. It

compensates for the incoherent power loss.

$$ECF = \frac{1}{\sqrt{\frac{1}{N}\sum_{n=0}^{N-1} w^2[n]}} = \frac{1}{\text{RMS}(w)}$$

- For Hann: $ECF \approx 1.63$

Implementation Requirement:

The visualizer must allow the user to toggle between "Spectrum" (Periodic, use ACF) and "Spectral Density" (Noise, use ECF) modes, or default to ACF for general audio visualization.

Formulas for Flat Top Coefficients (ISO Standard) [17]:

$$w[n] = a_0 - a_1 \cos\left(\frac{2\pi n}{N-1}\right) + a_2 \cos\left(\frac{4\pi n}{N-1}\right) - a_3 \cos\left(\frac{6\pi n}{N-1}\right) + a_4 \cos\left(\frac{8\pi n}{N-1}\right)$$

- $a_0 = 1.0, a_1 = 1.93, a_2 = 1.29, a_3 = 0.388, a_4 = 0.028$

---

4. Advanced Averaging and Detector Modes

Raw FFT data is chaotic. To make it readable and useful for analysis, Dewesoft and other professional tools implement averaging logic. This is not just for smoothing; it defines what physical quantity is being displayed.[4]

4.1 Linear Averaging

Linear averaging calculates the arithmetic mean of the power spectra over a fixed number of frames (M).

$$P_{avg}[k] = \frac{1}{M}\sum_{i=1}^{M} |X_i[k]|^2$$

This is useful for static signals but introduces a "block update" delay, making the display look jerky (updating only once every M frames).

4.2 Exponential Averaging (Video Filtering)

For real-time visualizers, exponential averaging (often called "Video Averaging" in hardware analyzers) is preferred. It acts as a recursive low-pass filter on the spectral bins.

$$Y_{new}[k] = \alpha \cdot |X_{curr}[k]| + (1 - \alpha) \cdot Y_{old}[k]$$

- α (alpha) controls the decay rate.
- $\alpha = 1.0$: No averaging (Instantaneous).
- $\alpha = 0.1$: Slow, smooth response (Heavy averaging).

This creates the fluid, water-like motion seen in high-end audio software. It stabilizes the noise floor while allowing strong signals to rise and fall smoothly.[20]

4.3 Peak Hold (Max Hold)

Peak hold retains the maximum amplitude observed at each frequency bin over the measurement duration.

$$Y_{hold}[k] = \max(Y_{hold}[k], |X_{curr}[k]|)$$

This is crucial for:

1. Detecting transient spikes that disappear quickly.
2. Measuring the "envelope" of a varying signal (e.g., the total bandwidth usage of a frequency-hopping transmitter). A "decaying peak hold" is a common variation where the peak value slowly decrements over time, providing a visual "memory" of recent loud events.[22]

4.4 RMS vs. Magnitude Averaging

A subtle distinction exists between averaging magnitudes and averaging power.

- **RMS Averaging:** $\sqrt{\text{Mean}(|X|^2)}$. This is physically correct for power estimation.
- **Magnitude Averaging:** $\text{Mean}(|X|)$. For Gaussian noise, this results in a systematic error of -1.05 dB (underestimation) because the magnitude of Gaussian noise follows a Rayleigh distribution.[24] **Recommendation:** Implementing RMS averaging is safer for analytical accuracy, though Magnitude averaging is computationally cheaper.

---

5. Psychoacoustic Weighting (A-Weighting)

The user query specifically references techniques in the Dewesoft guide, which often includes sound level measurement. Human hearing is not linear; we are much less sensitive to bass frequencies than to mid-range frequencies. To correlate FFT data with human perception,

A-Weighting filters are applied.

5.1 Frequency Domain Implementation

While A-weighting is traditionally an analog filter, in FFT analysis, it is most efficiently implemented as a spectral multiplication. We calculate the A-weighting gain curve for the specific frequency bins of the FFT and multiply the magnitude spectrum.[25]

The Standard Formula (IEC 61672:2003): The weighting $R_A(f)$ is given by [26]:

$$R_A(f) = \frac{12194^2 f^4}{(f^2 + 20.6^2)\sqrt{(f^2 + 107.7^2)(f^2 + 737.9^2)}\,(f^2 + 12194^2)}$$

The gain in decibels is:

$$A(f)_{dB} = 20\log_{10}(R_A(f)) + 2.00$$

The $+2.00$ dB constant normalizes the curve to 0 dB at 1000 Hz (1 kHz).

Integration Steps:

1. Generate frequency array freqs for the FFT bins.
2. Compute weights = A_weighting_function(freqs).
3. Compute FFT_magnitudes.
4. Weighted_Spectrum = FFT_magnitudes * weights.
5. Convert to dB.

This technique allows the visualizer to display "dB(A)" units, which are standard for environmental noise analysis.[26]

---

6. High-Performance Visualization Methodologies

The computational load of the FFT is often dwarfed by the cost of rendering the graphics. Python's standard plotting library, matplotlib, is generally too slow for real-time audio (capping at 10-15 FPS for complex plots).

6.1 Acceleration with PyQtGraph

For real-time performance (30-60 FPS), **PyQtGraph** is the industry standard in the Python scientific community.[27] It sits on top of Qt's QGraphicsView framework and uses optimized NumPy array handling.

Key Advantages:

- **Downsampling:** Automatically reduces the number of points drawn if the screen resolution is lower than the data resolution.
- **Hardware Acceleration:** partial use of OpenGL for line drawing.
- **Direct Buffer Updates:** Allows updating plot data without object overhead via .setData().

6.2 Logarithmic Scaling and Display

Audio spectrums should almost always be displayed with a logarithmic X-axis (Frequency) and logarithmic Y-axis (Amplitude in dB).

- **X-Axis:** A linear FFT has constant bandwidth per bin (e.g., 20 Hz). On a log scale, this means high frequencies look "crowded" and low frequencies look "sparse."
- **Y-Axis (Decibels):**

$$L_{dB} = 20 \log_{10} \left(\frac{A}{A_{ref}} \right)$$

  Where A is the measured amplitude and A_{ref} is the reference (usually 1.0 for Full Scale, or $20\mu Pa$ for sound pressure).

Safety Check: The log of zero is undefined ($-\infty$). The implementation must clip the data to a minimum floor (e.g., 10^{-6}) before logging to prevent math errors.[28]

---

7. Comparative Analysis of Open Source Tools

A review of existing GitHub repositories identifies several "archetypes" of visualizers. Analyzing these helps identify what to adopt and what to avoid.

7.1 The "Minimalist" (e.g., Realtime_PyAudio_FFT)

[29]

- **Architecture:** Simple while loop reading audio and updating matplotlib.
- **Pros:** Easy to understand.
- **Cons:** High latency, low frame rate, blocks the audio thread (potential glitches), lacks windowing correction.
- **Verdict:** Good for learning, bad for analysis.

7.2 The "Streamer" (e.g., python-fft-streamer)

[30]

- **Architecture:** Splits analysis and display over TCP.
- **Pros:** Decouples processing load. Great for IoT (Raspberry Pi capturing, PC displaying).
- **Verdict:** Excellent architectural pattern for remote sensing.

7.3 The "Professional" Prototype (e.g., Python-Realtime-Audio-Visualizer)

[31]

- **Architecture:** Uses sounddevice (modern PortAudio wrapper) and pyqtgraph. Implements a ring buffer.
- **Pros:** High frame rate, non-blocking audio, proper library choices.
- **Cons:** Often lacks the specific metrological corrections (Flat Top window, A-weighting) unless manually added.
- **Verdict:** This is the best foundation for the user's request. It provides the skeleton; the user must inject the "Dewesoft math" (Windowing + Weighting + Averaging) into the processing loop.

---

8. Synthesized Implementation Guide

Based on the theoretical requirements and the analysis of existing tools, here is the comprehensive implementation logic for a Dewesoft-style analyzer in Python.

8.1 Dependencies

Bash

```
pip install numpy scipy sounddevice pyqtgraph PyQt6
```

8.2 The Processing Class (The "Dewesoft" Engine)

This class encapsulates the DSP logic: buffering, windowing, weighting, and averaging.

Python

```
import numpy as np
import scipy.signal

class SpectralAnalyzer:
    def __init__(self, sample_rate=48000, fft_size=2048, window_type='hann'):
        self.fs = sample_rate
```

```
        self.fft_size = fft_size
        self.freqs = np.fft.rfftfreq(fft_size, 1/sample_rate)

        # 1. Window Setup & Correction Factors
        if window_type == 'flattop':
            self.window = scipy.signal.windows.flattop(fft_size)
            # Amplitude Correction Factor (ISO 18431)
            self.acf = 1 / np.mean(self.window)
        else:
            self.window = scipy.signal.windows.hann(fft_size)
            self.acf = 1 / np.mean(self.window) # approx 2.0

        # 2. A-Weighting Curve Pre-calculation
        self.a_curve = self._compute_a_weighting(self.freqs)

        # 3. Detector State (for Averaging)
        self.prev_spectrum = np.zeros(len(self.freqs))
        self.peak_hold_spectrum = np.zeros(len(self.freqs))
        self.alpha = 0.2  # Smoothing factor (0.0 - 1.0)

    def _compute_a_weighting(self, f):
        """ Computes the A-weighting gain in linear scale. """
        # Limit f to avoid divide by zero at DC
        f = np.maximum(f, 1e-5)
        f2 = f**2

        const = 12194**2 * f**4
        denom = ((f2 + 20.6**2) * np.sqrt((f2 + 107.7**2) * (f2 + 737.9**2)) * (f2 + 12194**2))

        # Result is Magnitude Gain.
        # Normalize to 1.0 at 1kHz (approx +2.0 dB shift handled here or in log)
        ra = const / denom
        ra_1k = 12194**2 * 1000**4 / ((1000**2 + 20.6**2) * np.sqrt((1000**2 + 107.7**2) * (1000**2
+ 737.9**2)) * (1000**2 + 12194**2))
        return ra / ra_1k

    def process(self, audio_frame, weighting=True, avg_mode='exp'):
        """
        Input: audio_frame (numpy array of size fft_size)
        Returns: spectrum in dB
        """
        # DC Removal
        audio_frame = audio_frame - np.mean(audio_frame)
```

```
        # Apply Window
        windowed_frame = audio_frame * self.window

        # FFT (Real)
        # Scale by 2/N (for one-sided) and multiply by ACF
        spectrum_mag = np.abs(np.fft.rfft(windowed_frame)) * (2 / self.fft_size) * self.acf

        # Apply A-Weighting
        if weighting:
            spectrum_mag *= self.a_curve

        # Averaging Logic
        if avg_mode == 'exp':
            self.prev_spectrum = self.alpha * spectrum_mag + (1 - self.alpha) * self.prev_spectrum
            result = self.prev_spectrum
        elif avg_mode == 'peak':
            self.peak_hold_spectrum = np.maximum(self.peak_hold_spectrum, spectrum_mag)
            result = self.peak_hold_spectrum
        else: # Instantaneous
            result = spectrum_mag

        # Convert to dB (Reference 1.0 for Full Scale)
        # Clamp to -120dB floor
        return 20 * np.log10(np.maximum(result, 1e-6))
```

8.3 Architecture Overview

1. **Main Thread (GUI):** Initializes pyqtgraph.PlotWidget. Sets up a QTimer to fire at 60 Hz.
2. **Audio Thread:** sounddevice.InputStream callback writes raw audio into a thread-safe Ring Buffer (e.g., a pre-allocated numpy array with a rolling index).
3. **Update Loop (QTimer):**
    - Queries the Ring Buffer for the *latest* fft_size samples (handling the wrap-around).
    - Passes data to SpectralAnalyzer.process().
    - Calls curve.setData() on the graph.

This structure ensures that the GUI remains responsive, the audio never drops out, and the mathematical analysis complies with the rigorous standards outlined in the Dewesoft guide.

9. Conclusion

The transition from a simple FFT script to a professional-grade analyzer lies in the details: the correct application of window correction factors to restore energy lost during tapering, the

implementation of psychoacoustic weighting to match human perception, and the use of sophisticated averaging to stabilize the display. By adopting the Ring Buffer architecture and leveraging the specific mathematical implementations for Flat Top windows and A-weighting provided above, one can build a Python-based tool that rivals commercial software in accuracy, if not in feature breadth. The combination of sounddevice for low-latency acquisition, scipy for rigorous signal processing, and pyqtgraph for accelerated rendering represents the optimal stack for this application.

Works cited

1. fft_python, accessed February 13, 2026, https://raghavchhetri.github.io/scattered.dimes/2021/07/21/Fourier-Transforms-in-Python
2. NumPy for Fast Fourier Transform (FFT) Analysis - GeeksforGeeks, accessed February 13, 2026, https://www.geeksforgeeks.org/numpy/numpy-for-fast-fourier-transform-fft-analysis/
3. numpy.fft.fft — NumPy v2.4 Manual, accessed February 13, 2026, https://numpy.org/doc/stable/reference/generated/numpy.fft.fft.html
4. Guide to FFT Analysis (Fast Fourier Transform) | Dewesoft, accessed February 13, 2026, https://dewesoft.com/blog/guide-to-fft-analysis
5. The Fundamentals of FFT-Based Signal Analysis and Measurement, accessed February 13, 2026, https://www.sjsu.edu/people/burford.furman/docs/me120/FFT_tutorial_NI.pdf
6. python-sounddevice/examples/play_sine.py at master - GitHub, accessed February 13, 2026, https://github.com/spatialaudio/python-sounddevice/blob/master/examples/play_sine.py
7. python-sounddevice/examples/spectrogram.py at master - GitHub, accessed February 13, 2026, https://github.com/spatialaudio/python-sounddevice/blob/master/examples/spectrogram.py
8. real time overlapping buffer for FFT - Signal Processing Stack Exchange, accessed February 13, 2026, https://dsp.stackexchange.com/questions/48768/real-time-overlapping-buffer-for-fft
9. Real time audio processing - buffers and FFT window length : r/DSP - Reddit, accessed February 13, 2026, https://www.reddit.com/r/DSP/comments/50ecea/real_time_audio_processing_buffers_and_fft_window/
10. Convolution and windowing using a buffer - how do I do overlap add?, accessed February 13, 2026, https://dsp.stackexchange.com/questions/29632/convolution-and-windowing-using-a-buffer-how-do-i-do-overlap-add
11. Overlap-Add (OLA) STFT Processing | Spectral Audio Signal Processing -

DSPRelated.com, accessed February 13, 2026, https://www.dsprelated.com/freebooks/sasp/Overlap_Add_OLA_STFT_Processing.html
12. Window function - Wikipedia, accessed February 13, 2026, https://en.wikipedia.org/wiki/Window_function
13. Understanding FFTs and Windowing - ni, accessed February 13, 2026, https://download.ni.com/evaluation/pxi/Understanding%20FFTs%20and%20Windowing.pdf
14. Dynamic Signal Analysis Basics - Crystal Instruments, accessed February 13, 2026, https://www.crystalinstruments.com/dynamic-signal-analysis-basics
15. flattop — SciPy v1.17.0 Manual, accessed February 13, 2026, https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.windows.flattop.html
16. FFT window function calculation detail - EEVblog, accessed February 13, 2026, https://www.eevblog.com/forum/chat/fft-window-function-calculation-detail/
17. Window Correction Factors - SIEMENS Community, accessed February 13, 2026, https://community.sw.siemens.com/s/article/window-correction-factors
18. Is it customary to correct for the gain of a window? - Signal Processing Stack Exchange, accessed February 13, 2026, https://dsp.stackexchange.com/questions/18164/is-it-customary-to-correct-for-the-gain-of-a-window
19. User note: FFT averaging - OROS Wiki, accessed February 13, 2026, https://wiki.oros.com/index.php?title=User_note:_FFT_averaging
20. Fast Fourier Transformation FFT - NTi Audio Solutions for Audio & Acoustics, accessed February 13, 2026, https://www.nti-audio.com/en/support/know-how/fast-fourier-transformation-fft
21. Frequency Domain | PySDR: A Guide to SDR and DSP using Python, accessed February 13, 2026, https://pysdr.org/content/frequency_domain.html
22. Understanding Data Averaging Modes, accessed February 13, 2026, https://www.emerson.com/documents/automation/white-paper-understanding-data-averaging-modes-en-5545142.pdf
23. Python Spectrum Analyzer 'Max Hold' function - Liquid Instruments Knowledge Base, accessed February 13, 2026, https://knowledge.liquidinstruments.com/python-api-examples/python-spectrum-analyzer-'max-hold'-function
24. dft - SNR of averaging FFT in magnitude - Signal Processing Stack ..., accessed February 13, 2026, https://dsp.stackexchange.com/questions/80643/snr-of-averaging-fft-in-magnitude
25. Using A-Weighting on time signal - python - Stack Overflow, accessed February 13, 2026, https://stackoverflow.com/questions/65842795/using-a-weighting-on-time-signal
26. A-weighting - Wikipedia, accessed February 13, 2026, https://en.wikipedia.org/wiki/A-weighting
27. pyqtgraph/pyqtgraph: Fast data visualization and GUI tools for scientific /

engineering applications - GitHub, accessed February 13, 2026, https://github.com/pyqtgraph/pyqtgraph
28. FFT exponential form ends up in noisy audio - Stack Overflow, accessed February 13, 2026, https://stackoverflow.com/questions/49276041/fft-exponential-form-ends-up-in-noisy-audio
29. aiXander/Realtime_PyAudio_FFT: Realtime audio analysis in Python, using PyAudio and Numpy to extract and visualize FFT features from streaming audio. - GitHub, accessed February 13, 2026, https://github.com/aiXander/Realtime_PyAudio_FFT
30. vladbalmos/python-fft-streamer: A FFT analysis script that streams amplitude data over TCP combined with an audio visualizer - GitHub, accessed February 13, 2026, https://github.com/vladbalmos/python-fft-streamer
31. omega0verride/Python-Realtime-Audio-Visualizer - GitHub, accessed February 13, 2026, https://github.com/omega0verride/Python-Realtime-Audio-Visualizer