Radix-2 FFT in C++20: Correctness-Centered Design, Mathematical Rigor, and Automated Verification

1st Santiago Florido Gómez
Mechatronic engineer
ENSTA - Paris
Paris, France
santiago.florido@ensta-paris.fr

***Abstract*—This paper presents a complete C++20 implementation of an iterative in-place radix-2 Cooley-Tukey fast Fourier transform (FFT) and its inverse (IFFT), together with a mathematically exact $O(N^2)$ discrete Fourier transform (DFT) reference for validation. The implementation follows a strict engineering sign convention, applies bit-reversal permutation, and includes explicit $1/N$ normalization in the inverse transform. Verification is treated as a primary design objective: automated tests compare FFT output against the direct DFT, validate round-trip stability $\mathrm{IFFT}(\mathrm{FFT}(x)) \approx x$, check pure-tone spectral concentration, and enforce Parseval energy consistency under double precision. This approach yields a dependency-free and reproducible baseline suitable for research prototypes, educational signal-processing laboratories, and production-oriented systems where deterministic spectral behavior and explainable numerical error bounds are required.**

***Index Terms*—fast Fourier transform, split-radix, radix-2, Cooley-Tukey, spectral analysis, numerical verification, C++20**

I. Introduction

The fast Fourier transform (FFT) is one of the most consequential algorithms in computational science because it reduces the complexity of spectral analysis from $O(N^2)$ to $O(N \log_2 N)$ [1]. This complexity reduction is not only asymptotic theory; it determines practical feasibility in systems constrained by real-time deadlines, bounded power budgets, and finite memory bandwidth.

In modern engineering, many physical phenomena are observed or controlled in time or space domains but are most interpretable in the frequency domain. Spectral decomposition is therefore central to communication receivers, radar processing chains, medical imaging reconstruction, vibration diagnostics, acoustic scene analysis, geophysical inversion, and computational physics. In these systems, FFT quality directly influences detection sensitivity, parameter estimation bias, and downstream model robustness.

From a numerical perspective, FFT implementations are also a reliability-critical component. Small indexing errors in butterfly updates, incorrect exponential signs, or misplaced normalization can silently produce spectra that appear plausible yet are physically wrong. Such defects propagate into control loops, inverse problems, and machine-learning feature pipelines. For this reason, an FFT implementation should be delivered with explicit mathematical conventions and independent verification against a direct DFT baseline.

This work addresses that requirement through a C++20, standard-library-only implementation of radix-2 FFT/IFFT and a test methodology that prioritizes reproducibility, deterministic error reporting, and traceable agreement with Fourier definitions [2], [3]. The main contributions are:

- an iterative in-place radix-2 FFT with explicit bit-reversal permutation;
- a definition-aligned $O(N^2)$ DFT/IDFT reference implementation;
- automated validation with strict tolerances for round-trip accuracy, bin-wise agreement, tone localization, and energy conservation.

The resulting project forms a rigorous baseline that can be extended toward optimized kernels, mixed-radix strategies, and architecture-specific acceleration while preserving a verifiable correctness core.

II. Mathematical Foundations

A. DFT and IDFT Definitions

For a length-N complex sequence $x[n]$, the forward transform is

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

and the inverse transform is

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{+j2\pi kn/N}, \quad n = 0, \ldots, N-1. \quad (2)$$

The implementation follows this sign convention exactly and applies normalization only in the inverse path.

B. Roots of Unity and Notation

Define

$$W_N = e^{-j2\pi/N}, \quad (3)$$

so that $W_N^{kn} = e^{-j2\pi kn/N}$. Equation (1) becomes

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn}. \tag{4}$$

This notation enables compact derivation of radix-2 factorization.

C. Radix-2 Cooley-Tukey Decomposition

Assume $N = 2^m$. Split the sequence into even and odd indices:

$$X[k] = \sum_{r=0}^{N/2-1} x[2r] W_N^{k(2r)} + \sum_{r=0}^{N/2-1} x[2r+1] W_N^{k(2r+1)}. \tag{5}$$

Using $W_N^2 = W_{N/2}$, define

$$E[k] = \sum_{r=0}^{N/2-1} x[2r] W_{N/2}^{kr}, \quad O[k] = \sum_{r=0}^{N/2-1} x[2r+1] W_{N/2}^{kr}, \tag{6}$$

then

$$X[k] = E[k] + W_N^k O[k], \tag{7}$$

$$X[k + N/2] = E[k] - W_N^k O[k]. \tag{8}$$

Equations (7) and (8) are the radix-2 butterfly relations.

D. Bit-Reversal Permutation

In an iterative decimation-in-time schedule, input indices are permuted by reversing their m-bit binary representation before stage-wise butterflies are applied. This permutation maps recursive subproblem order into contiguous in-place blocks. Without it, stage-local butterfly connectivity is violated and final bins are scrambled.

E. Complexity

A direct DFT computes N outputs, each with N terms, yielding $\Theta(N^2)$. The radix-2 FFT performs $\log_2 N$ stages with $\Theta(N)$ work per stage, yielding $\Theta(N \log_2 N)$. This is the computational foundation for real-time spectral systems.

III. Algorithm and Implementation Mapping

The codebase evaluates four transform models under the same interface: iterative radix-2 FFT, recursive radix-2 FFT, split-radix FFT, and a direct $O(N^2)$ DFT reference. This section details the mapping used by the radix-2 iterative baseline, while the split-radix-specific decomposition is detailed in Section IV.

A. Iterative In-Place Kernel

The implementation processes stages with block lengths $L = 2, 4, \ldots, N$. For each block start index i and local offset $j \in [0, L/2 - 1]$:

$$u = x[i + j], \tag{9}$$

$$v = x[i + j + L/2]\, W_L^j, \tag{10}$$

then updates

$$x[i + j] \leftarrow u + v, \tag{11}$$

$$x[i + j + L/2] \leftarrow u - v. \tag{12}$$

This mapping is a direct implementation of the butterfly equations.

B. Twiddle Generation Strategy

At each stage, the principal stage twiddle $W_L = e^{\pm j2\pi/L}$ is computed once. Per-butterfly twiddles are obtained by recurrence $w \leftarrow wW_L$ within each block. This avoids storing global twiddle tables while preserving deterministic behavior and numerical clarity.

C. Inverse Transform

The inverse transform reuses the same kernel with opposite sign in the exponential and applies a final scaling by $1/N$. This ensures consistency with (2) and supports stable round-trip reconstruction.

D. Input Validation

Because the chosen factorization is radix-2, valid sizes must satisfy $N \neq 0$ and N power-of-two. Invalid sizes throw `std::invalid_argument` with an explicit message, preventing undefined spectral behavior.

E. Implementation Notes

The code is written in C++20 with only the standard library, using `std::complex<double>` for numerical representation. The design goal is not architecture-specific peak throughput but reference-quality correctness that remains portable across Linux and Windows toolchains.

IV. Split-Radix FFT

Split-radix is a structured Cooley-Tukey factorization that combines radix-2 and radix-4 style decomposition in one recursion. For a power-of-two size N, the transform is split into one half-size branch and two quarter-size branches:

$$\mathrm{DFT}(N) \rightarrow \mathrm{DFT}(N/2) + \mathrm{DFT}(N/4) + \mathrm{DFT}(N/4). \tag{13}$$

The even-indexed samples feed the $N/2$ branch. The odd-indexed samples are separated into two classes, $n = 4m + 1$ and $n = 4m + 3$, each producing one $N/4$ branch. In the implementation, these two odd branches are recombined with twiddle factors and $\pm j$ rotations to recover the four output quadrants.

Compared to pure radix-2, split-radix reduces arithmetic operation count in theory, especially in multiplication count for large transforms. This makes it a relevant candidate when algorithmic operation complexity is the primary optimization target.

The trade-off is higher implementation complexity. Index mapping is less uniform than standard radix-2 butterflies, and memory access patterns can be less cache-friendly depending on recursion depth, temporary-buffer layout, and compiler optimization. Consequently, wall-clock speedup is architecture-dependent and must be validated experimentally rather than assumed from operation counts alone.

V. Verification Methodology

A. Automated Test Set

Verification is executed through CTest and a self-contained test executable. The FFT-oriented checks are applied to all power-of-two FFT models (iterative radix-2, recursive radix-2, and split-radix), and the direct DFT model is validated through dedicated checks. The suite includes:

- **Round-trip stability:** for powers of two from $N = 2$ to $N = 4096$, verify $\mathrm{IFFT}(\mathrm{FFT}(x)) \approx x$ using random complex vectors with fixed seed.
- **Reference agreement:** for $N \leq 256$, compare each FFT bin against the direct $O(N^2)$ DFT.
- **Pure-tone localization:** for $x[n] = e^{j2\pi kn/N}$, verify dominant energy at bin k and near-zero leakage elsewhere.
- **Parseval consistency:** verify

$$\sum_n |x[n]|^2 \approx \frac{1}{N} \sum_k |X[k]|^2, \tag{14}$$

  consistent with the chosen normalization.
- **Input constraints:** verify rejection of non-power-of-two sizes.

B. Tolerances and Rationale

Double-precision tolerances are set to 10^{-10} for round-trip relative L_2 error, bin-wise FFT-vs-DFT absolute error, and Parseval relative mismatch. A 10^{-8} leakage bound is used for off-peak bins in pure-tone tests. These values reflect expected floating-point accumulation and trigonometric evaluation error under IEEE-754 arithmetic [5].

C. Expected Spectral Behavior

A single-tone complex exponential is expected to produce one dominant peak at the target bin k. A real sine wave is expected to produce two symmetric dominant peaks at k and $N - k$, consistent with conjugate symmetry for real-valued time signals.

VI. Benchmark Results and Comparison

This section compares the implemented transform models using an automated benchmark and plotting workflow. The evaluated algorithms are `radix2_iterative`, `radix2_recursive`, `split_radix`, and `direct_dft`. The measured sizes are $N \in \{64, 128, 256, 512, 1024\}$, with 3 warmup iterations and 20 measured iterations per pair $(\text{algorithm}, N)$, using seed 1337.

The plotted quantities are extracted from the benchmark summary data:

- transform length N;
- average runtime per transform (microseconds);
- throughput (processed samples per second).
- Additional measured descriptors (not shown in the two figures below): median runtime, best-case runtime, worst-case runtime, runtime dispersion (standard deviation), tail latency at the 95th percentile, normalized runtime per sample, and normalized runtime per $N \log_2 N$.

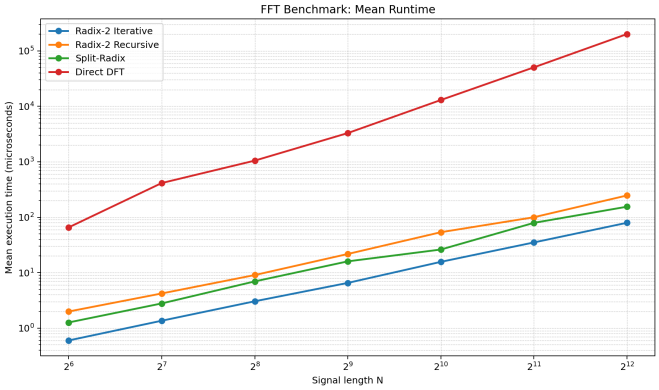

Fig. 1. Mean execution time as a function of signal length N.

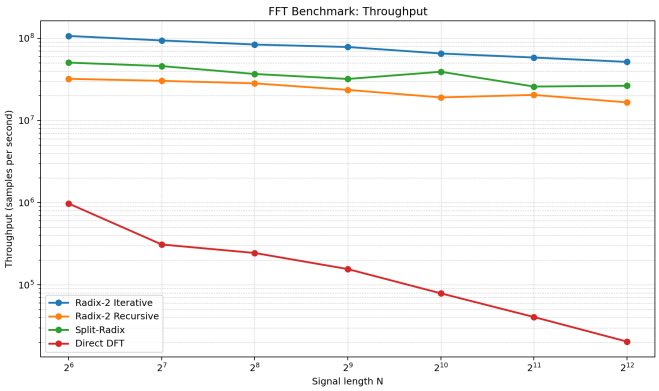

Fig. 2. Throughput in samples per second as a function of signal length N.

Figure 1 shows the expected complexity separation. The `direct_dft` curve grows much faster with N, while the FFT-based models scale substantially better. In this benchmark run at $N = 1024$, mean runtime is approximately 15.27 μs for `radix2_iterative`, 25.04 μs for `split_radix`, 41.86 μs for `radix2_recursive`, and 12,852.67 μs for `direct_dft`.

Figure 2 presents the same comparison in throughput form. At $N = 1024$, measured throughput is about 67.1 million samples/s for `radix2_iterative`, 40.9 million samples/s for `split_radix`, 24.5 million samples/s for `radix2_recursive`, and 79.7 thousand samples/s for `direct_dft`. In this environment, split-radix improves over recursive radix-2, but remains slower than the iterative radix-2 baseline, indicating that lower arithmetic count does not automatically dominate memory traffic and traversal overhead.

These results are consistent with the theoretical complexity analysis and provide a reproducible quantitative baseline for future algorithm extensions and microcontroller-oriented optimization.

VII. Scientific and Engineering Relevance

FFT is foundational in projects where information is encoded in spectral structure rather than raw samples. In digital

communications, orthogonal frequency-division multiplexing (OFDM) modulators and demodulators rely on FFT/IFFT pairs for subcarrier-domain processing and channel equalization. In radar and sonar, FFT stages support range-Doppler processing and clutter separation. In biomedical instrumentation, electroencephalography and electrocardiography pipelines use spectral features for diagnosis and anomaly detection.

In imaging and inverse problems, FFT-based convolution accelerates iterative solvers for magnetic resonance reconstruction, computational microscopy, and deblurring. In mechanical and civil monitoring, vibration spectra expose resonance modes and early fault signatures in rotating machinery and structures. In power systems, harmonic analysis depends on FFT stability to quantify distortion and nonstationary events.

For machine-learning systems, FFT-derived features are frequently used in audio classification, activity recognition, and predictive maintenance. In these data-centric workflows, a well-verified FFT implementation reduces silent label noise caused by spectral misalignment and improves reproducibility across hardware and operating systems.

Therefore, FFT is not only an algorithmic acceleration; it is a scientific instrumentation primitive. Correctness, traceable conventions, and controlled numerical error are mandatory for high-impact engineering projects.

VIII. Conclusion

This work now provides a four-model spectral computation baseline in C++20: iterative radix-2 FFT, recursive radix-2 FFT, split-radix FFT, and a direct DFT reference. The implementations are grounded in explicit Fourier sign conventions, deterministic inverse normalization, and strict automated verification against reference behavior.

Benchmark results confirm the expected asymptotic gap between FFT models and direct DFT. They also show that split-radix variants deliver meaningful runtime and throughput improvements over recursive radix-2 in this environment, but do not surpass the iterative radix-2 implementation for the tested sizes. This behavior is consistent with practical trade-offs: split-radix reduces arithmetic operations, yet memory-access patterns, data movement, and traversal overhead remain relevant on modern desktop CPUs.

The resulting framework supports future extensions such as mixed-radix decomposition, real-input specialization, SIMD acceleration, planner-based optimization [4], and embedded profiling, while preserving a mathematically auditable core.

Acknowledgment

This work did not receive external funding.

References

[1] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[2] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-Time Signal Processing*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 1999.

[3] R. N. Bracewell, *The Fourier Transform and Its Applications*, 3rd ed. New York, NY, USA: McGraw-Hill, 2000.

[4] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, Feb. 2005.

[5] IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019, 2019.