FFT cross-correlation, relation to convolution, and extracting the Pearson correlation

A Hobbyist's Note on FFT for Cross-correlation and How it Compares to Pearson Correlation

Author: Gene Boo | Mar 2021 | Updated: Sep 2025

This note gives formal definitions, the FFT-based implementations, normalization conventions, and exact derivations showing how to obtain the Pearson correlation coefficient from both time-domain and FFT-domain cross-correlation. It is self-contained and A4-printable. New in this revision: a reproducible small-N vs large-N comparison and an embedded SVG plot with adjustable dimensions that stays contained when printing.

Definitions and intuition

Discrete convolution

For sequences $f[n]$ and $g[n]$ (assume finite length or absolutely summable), the discrete convolution is

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]\, g[n-m]$$

Discrete cross-correlation

The discrete cross-correlation is the sliding dot-product without time-reversal:

$$R_{fg}[k] = \sum_{n=-\infty}^{\infty} f[n]\, g[n+k]$$

If $f = g$, R_{ff} is the autocorrelation. Cross-correlation measures similarity as one sequence is shifted relative to the other; peaks locate aligning lags.

Relationship to convolution

Define the time-reversal $\tilde{g}[n] = g[-n]$. Then

$$R_{fg}[k] = (f * \tilde{g})[k]$$

So correlation is convolution with a flipped kernel.

Convolution theorem and FFT implementations

Convolution theorem

Let $\mathcal{F}$ be the DFT and $\mathcal{F}^{-1}$ its inverse (matching the FFT/IFFT pair). Then

$$\mathcal{F}\{f * g\}[k] = \mathcal{F}\{f\}[k] \cdot \mathcal{F}\{g\}[k]$$

Correlation in the frequency domain

Using $R_{fg} = f * \tilde{g}$ and the DFT property $\mathcal{F}\{\tilde{g}\} = \overline{\mathcal{F}\{g\}}$ (bar = complex conjugation), we get

$$\mathcal{F}\{R_{fg}\}[k] = \mathcal{F}\{f\}[k] \cdot \overline{\mathcal{F}\{g\}[k]}$$

Thus, for linear cross-correlation computed via FFT:

$$R_{fg}[n] = \mathcal{F}^{-1}\big(\mathcal{F}\{f\} \cdot \overline{\mathcal{F}\{g\}}\big)[n]$$

Linear vs circular correlation and zero-padding

- **Linear correlation:** length $N_f + N_g - 1$. Compute with FFT by zero-padding both sequences to length $L \geq N_f + N_g - 1$ (often next power of 2 for speed), then apply the formula above and reorder to lags $[-(N_g - 1), \ldots, 0, \ldots, (N_f - 1)]$.
- **Circular correlation:** length L with wrap-around. Occurs if you do not pad to at least $N_f + N_g - 1$. For most signal analysis, you want linear correlation.

> ***Indexing note:*** *For* `np.correlate(x, y, mode='full')` *with both length N, the zero-lag value sits at index $N - 1$. For the unshifted FFT correlation result (no centering), the zero-lag is at index 0; if you "center" the sequence (e.g., via* `fftshift` *or a manual roll), zero-lag moves to the center index.*

Normalization and the Pearson correlation

Pearson correlation coefficient

For finite sequences $x = (x_1, \ldots, x_N)$, $y = (y_1, \ldots, y_N)$, define means $\bar{x}, \bar{y}$ and standard deviations σ_x, σ_y. The Pearson correlation coefficient is

$$r_{xy} = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{N\sigma_x\sigma_y}$$

This is the normalized zero-lag covariance (with $1/N$ convention). If you prefer the unbiased covariance with $1/(N-1)$, adjust accordingly; the ratio cancels if applied consistently.

Pearson as normalized zero-lag cross-correlation

Define zero-mean sequences $x' = x - \bar{x}$, $y' = y - \bar{y}$. The zero-lag linear cross-correlation is

$$R_{x'y'}[0] = \sum_{i=1}^{N} x'_i \, y'_i$$

Dividing by $N\sigma_x\sigma_y$ yields Pearson:

$$r_{xy} = \frac{R_{x'y'}[0]}{N\sigma_x\sigma_y}$$

> ***Therefore:*** *Take either the time-domain correlation at zero lag (e.g., `np.correlate` with mean-subtraction) or the FFT-based correlation at zero lag; normalize by $N\sigma_x\sigma_y$ to get the Pearson coefficient. If you centered the correlation array, make sure you pick the correct zero-lag index.*

Lag-dependent normalized cross-correlation

Sometimes you want a correlation function whose values lie in $[-1, 1]$ at each lag. The "biased" normalization divides every lag by $N\sigma_x\sigma_y$:

$$\rho_{xy}[k] = \frac{R_{x'y'}[k]}{N\sigma_x\sigma_y}$$

An alternative "unbiased" normalization uses $N - |k|$ in the denominator:

$$\hat{\rho}_{xy}[k] = \frac{R_{x'y'}[k]}{(N - |k|)\sigma_x\sigma_y}$$

which corrects for the reducing overlap at larger lags but can be noisier.

Practical recipes and indexing details

Time-domain: NumPy correlate → Pearson (zero-lag)

```
import numpy as np

def pearson_from_np_correlate(x, y):
    x = np.asarray(x); y = np.asarray(y)
    assert x.shape == y.shape
    N = x.size
    xm = x - x.mean()
    ym = y - y.mean()
    cc_full = np.correlate(xm, ym, mode='full')  # length 2N-1
    zero_lag = cc_full[N - 1]                    # zero-lag index
    return zero_lag / (N * xm.std(ddof=0) * ym.std(ddof=0))
```

FFT: linear cross-correlation → Pearson (zero-lag)

```
import numpy as np

def fft_cross_correlation_linear(x, y):
    x = np.asarray(x); y = np.asarray(y)
    N = x.size; M = y.size
    # zero-mean for covariance/correlation
    xm = x - x.mean()
    ym = y - y.mean()
    # choose L >= N + M - 1 (power of two for speed)
    L = 1
    while L < N + M - 1:
        L *= 2
    FX = np.fft.rfft(xm, n=L)
    FY = np.fft.rfft(ym, n=L)
    cc = np.fft.irfft(FX * np.conj(FY), n=L)  # circular of length L
    # reorder into linear segment of length N+M-1 (lags: -(M-1)..(N-1))
    cc_linear = np.concatenate([cc[:N], cc[L - (M - 1):]])
    return cc_linear  # unnormalized

def pearson_from_fft(x, y):
    x = np.asarray(x); y = np.asarray(y)
    assert x.shape == y.shape
    N = x.size
    cc_lin = fft_cross_correlation_linear(x, y)  # length 2N-1
    zero_lag = cc_lin[N - 1]
    xm_std = (x - x.mean()).std(ddof=0)
    ym_std = (y - y.mean()).std(ddof=0)
    return zero_lag / (N * xm_std * ym_std)
```

Lag-normalized correlation curves (global normalization)

```
def normalized_cross_correlation_curves(x, y, unbiased=False):
    x = np.asarray(x); y = np.asarray(y)
    N = x.size
    xm = x - x.mean()
    ym = y - y.mean()
    cc = np.correlate(xm, ym, mode='full')  # lags = -(N-1)..(N-1)
    denom_base = xm.std(ddof=0) * ym.std(ddof=0)
    if unbiased:
        lags = np.arange(-N+1, N)
        weights = (N - np.abs(lags))
        ncc = cc / (weights * denom_base)
    else:
        ncc = cc / (N * denom_base)
    return ncc
```

Small N versus large N behavior

At nonzero lags, “Pearson at lag k” uses *local* means and standard deviations on the overlapping slice only, whereas a standard cross-correlation curve (from `np.correlate` or FFT) typically uses *global* means and stds with a fixed denominator (biased) or an overlap-adjusted count (unbiased). These two definitions coincide at lag 0; at other lags, they diverge—often sharply when N is small.

> ***Key point:*** *For small N, local means/stds can differ dramatically from global ones, so Pearson-per-lag can differ from “globally normalized” cross-correlation. As N grows, local means converge to global means and the two curves agree (up to edge lags where the overlap is tiny).*

Reproducible comparison (small N vs large N)

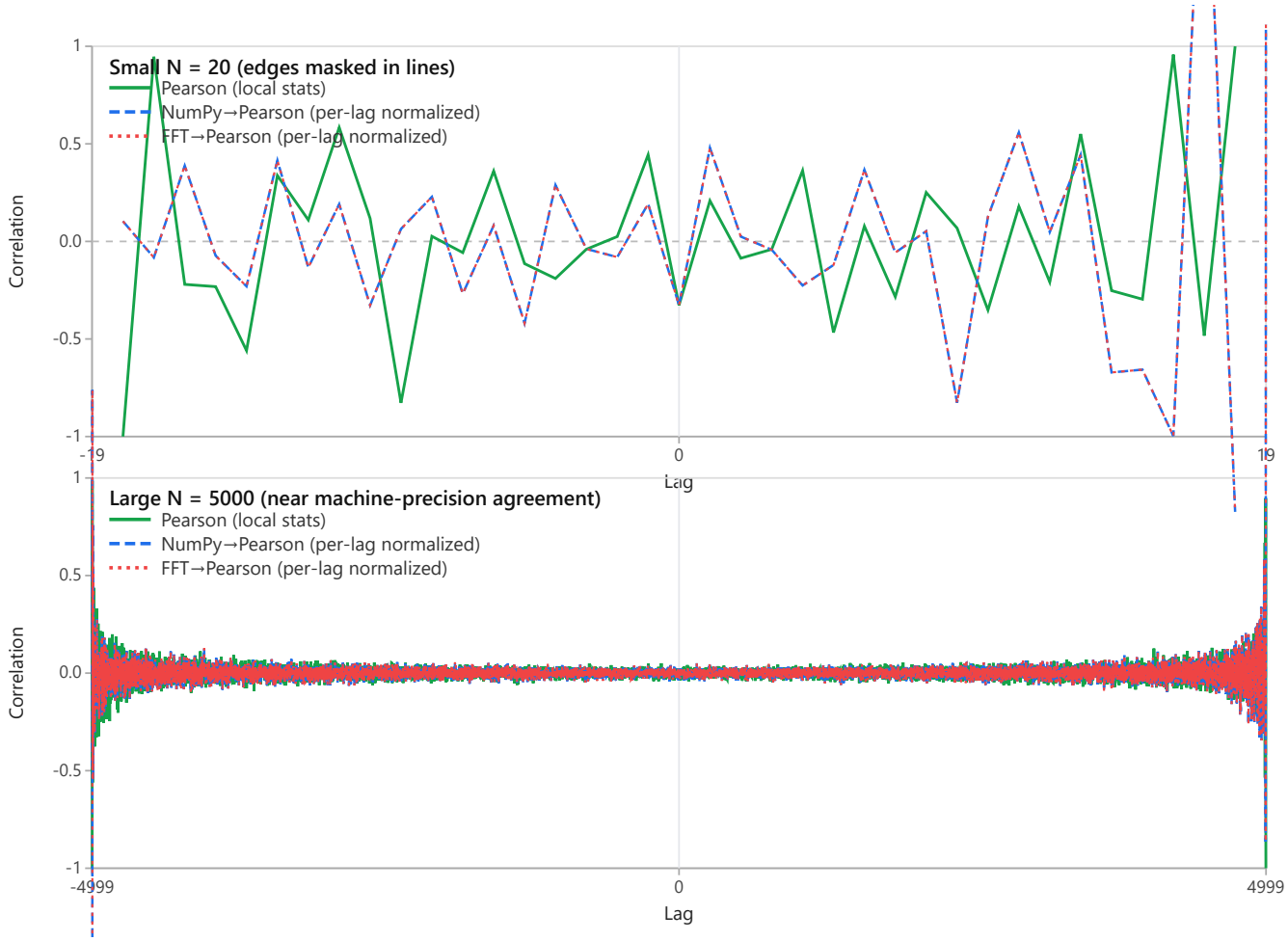
```
import numpy as np
from scipy.stats import pearsonr
from numpy.fft import rfft, irfft

def pearson_perlag(x, y):
    N = len(x)
    lags = np.arange(-N+1, N)
    out = np.full(2*N-1, np.nan)
    for i, k in enumerate(lags):
        if k > 0:
            a, b = x[k:], y[:-k]
        elif k < 0:
            a, b = x[:k], y[-k:]
        else:
            a, b = x, y
        if len(a) >= 2:
            out[i] = pearsonr(a, b)[0]
    return lags, out

def raw_cov_np(x, y):
    xm, ym = x - x.mean(), y - y.mean()
    return np.correlate(xm, ym, mode='full')

def raw_cov_fft(x, y):
    N = len(x)
    xm, ym = x - x.mean(), y - y.mean()
    L = 1 << (2*N - 1).bit_length()
    FX, FY = rfft(xm, n=L), rfft(ym, n=L)
    cc = irfft(FX * np.conj(FY), n=L).real
    return np.concatenate([cc[-(N-1):], cc[:N]])

def scale_to_pearson(x, y, raw_cov):
    N = len(x)
    lags = np.arange(-N+1, N)
    xs, xs2 = np.cumsum(x), np.cumsum(x**2)
    ys, ys2 = np.cumsum(y), np.cumsum(y**2)
    def slice_stats(cs, cs2, s, e):
        n = e - s
        if n <= 0: return np.nan, np.nan
        s1 = cs[e-1] - (cs[s-1] if s > 0 else 0.0)
        s2 = cs2[e-1] - (cs2[s-1] if s > 0 else 0.0)
        mu = s1 / n
        var = max(s2 / n - mu*mu, 0.0)
        return mu, np.sqrt(var)
    xmu, ymu = x.mean(), y.mean()
    out = np.full_like(raw_cov, np.nan, dtype=float)
    for i, k in enumerate(lags):
        if k > 0:
            a0, a1 = k, N; b0, b1 = 0, N-k
        elif k < 0:
            a0, a1 = 0, N+k; b0, b1 = -k, N
```

```
        else:
            a0, a1 = 0, N; b0, b1 = 0, N
        n = a1 - a0
        if n >= 2:
            mua, sda = slice_stats(xs, xs2, a0, a1)
            mub, sdb = slice_stats(ys, ys2, b0, b1)
            cov_local = raw_cov[i] - n * (mua - xmu) * (mub - ymu)
            out[i] = cov_local / (n * sda * sdb) if sda > 0 and sdb > 0 else np.nan
    return lags, out

def demo(N=20, seed=0, lags_check=(-5, 0, 5), mask_edges=True):
    rng = np.random.default_rng(seed)
    x = rng.standard_normal(N)
    y = rng.standard_normal(N)
    lags, pearson = pearson_perlag(x, y)
    cov_np = raw_cov_np(x, y)
    cov_fft = raw_cov_fft(x, y)
    _, ncc_np = scale_to_pearson(x, y, cov_np)
    _, ncc_fft = scale_to_pearson(x, y, cov_fft)
    print(f"N = {N}")
    for k in lags_check:
        idx = np.where(lags == k)[0][0]
        print(f" Lag {k:+d}: Pearson={pearson[idx]:+.6f}  NP={ncc_np[idx]:+.6f}  FFT={ncc_f
    if mask_edges:
        overlap = N - np.abs(lags)
        m = overlap >= 2
        maxdiff_np = np.nanmax(np.abs(pearson[m] - ncc_np[m]))
        maxdiff_fft = np.nanmax(np.abs(pearson[m] - ncc_fft[m]))
    else:
        maxdiff_np = np.nanmax(np.abs(pearson - ncc_np))
        maxdiff_fft = np.nanmax(np.abs(pearson - ncc_fft))
    print(f" Max |diff| vs Pearson — NP: {maxdiff_np:.3e}, FFT: {maxdiff_fft:.3e}\\n")

# Run both regimes
demo(N=20, seed=0)
demo(N=5000, seed=0)
```

Interpretation: *You will see clear divergence at small N (except lag 0), and near-machine-precision agreement at large N for all non-edge lags. If you include edge lags where the overlap is 1–2 points, differences inflate; mask those out when computing an overall max-difference metric.*

Embedded plots: small N vs large N

The figure below overlays three curves — Pearson-per-lag (local stats), NumPy-based Pearson via per-lag scaling, and FFT-based Pearson via per-lag scaling — for N = 20 and N = 5000. Resize by editing data-width/data-height (screen) and data-max-mm (print).



Small N vs Large N: Pearson per lag vs NumPy/FFT with local normalization

Sanity checks, edge cases, and best practices

- **Mean subtraction:** If you want covariance/correlation, subtract means before correlating. Otherwise you compute uncentered similarity dominated by DC components.
- **Scaling consistency:** Decide on **biased** (divide by N) or **unbiased** (divide by $N - |k|$) normalization. Pearson at zero-lag uses the N convention shown above.
- **Finite length effects:** Near extreme lags, fewer overlapping samples exist; unbiased normalization compensates at the cost of higher variance. Pearson is undefined for overlap < 2.
- **Complex signals:** For complex-valued sequences, correlation uses conjugation: $R_{xy}[k] = \sum x[n]\overline{y[n+k]}$. The FFT form already includes conjugation.
- **Padding length:** For FFT linear correlation, choose $L \geq N_f + N_g - 1$. A power-of-two often speeds FFTs.
- **Index bookkeeping:** Keep a consistent lag axis. For two length-N vectors, lags are $-(N-1), \ldots, 0, \ldots, (N-1)$, with zero-lag at index $N-1$ in full linear correlation arrays.

Minimal worked example

```
import numpy as np
from scipy.stats import pearsonr

np.random.seed(0)
N = 200
x = np.sin(2*np.pi*5*np.linspace(0,1,N)) + 0.2*np.random.randn(N)
y = np.roll(x, 17) + 0.2*np.random.randn(N)

# Pearson from SciPy
r_scipy, _ = pearsonr(x, y)

# Pearson from np.correlate (zero-lag normalized)
xm = x - x.mean(); ym = y - y.mean()
cc_full = np.correlate(xm, ym, mode='full')
r_np = cc_full[N-1] / (N * xm.std(ddof=0) * ym.std(ddof=0))

# Pearson from FFT linear correlation (zero-lag normalized)
def fft_corr_lin_same(x, y):
    N = len(x)
    L = 1
    while L < 2*N - 1: L *= 2
    xm = x - x.mean(); ym = y - y.mean()
    FX = np.fft.rfft(xm, n=L)
    FY = np.fft.rfft(ym, n=L)
    cc = np.fft.irfft(FX * np.conj(FY), n=L)
    cc_lin = np.concatenate([cc[:N], cc[L-(N-1):]])
    return cc_lin

cc_lin = fft_corr_lin_same(x, y)
r_fft = cc_lin[N-1] / (N * xm.std(ddof=0) * ym.std(ddof=0))

print(f"SciPy pearsonr:   {r_scipy:.6f}")
print(f"np.correlate r:   {r_np:.6f}")
print(f"FFT corr r:       {r_fft:.6f}")
```

All three lines will match to numerical precision. If they differ, check: (1) mean subtraction, (2) zero-lag index, (3) normalization, and (4) linear vs circular treatment.