Autocorrelation Benchmark Report (AutoCorr1 vs AutoCorr2)

This report documents two Google Colab notebooks (`AutoCorr1` and `AutoCorr2`) that implement and benchmark autocorrelation estimation methods in C and Python. Autocorrelation can be computed by two mathematically equivalent FFT-based approaches: (1) *explicit cross-correlation* via FFT convolution and (2) *spectral* (Wiener–Khinchin) method. The cross-correlation approach computes the autocorrelation by convolving the signal with a time-reversed copy; the Wiener–Khinchin approach computes it as the inverse FFT of the power spectral density (the magnitude-squared FFT). Both methods require an FFT and inverse FFT, leading to $O(N\log N)$ complexity, and they share common limitations (floating-point precision, zero-padding)]. Conceptually, AutoCorr1 emphasizes the time-domain cross-correlation interpretation, whereas AutoCorr2 emphasizes the frequency-domain interpretation. The notebooks benchmark *Original*, *Optimized*, and *Superfast* implementations of each method in both C (using the FFTW library) and Python (using NumPy/SciPy and ctypes).

AutoCorr1 Notebook (Cross-Correlation Method)

The **AutoCorr1** notebook is titled “Benchmarking autocorrelation methods in C and Python” and focuses on the direct cross-correlation implementation. It defines three C functions for autocorrelation: **Original** (explicit FFT-based convolution emulating `scipy.signal.fftconvolve`), **Optimized** (using the Wiener–Khinchin theorem by multiplying FFT with its complex conjugate), and **Superfast** (similar to Optimized but normalized by the zero-lag value). These C functions use FFTW single-precision (`fftwf`) routines. The notebook then compiles these into a shared library and provides equivalent Python implementations and wrappers. Finally, it runs performance benchmarks over various signal sizes and tests a single test signal. The code blocks below are organized by component.

C Implementation – Original

```
%%writefile autocorr_c.c

#include <fftw3.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>

/*
Versión "Original" en C: Emula scipy.signal.fftconvolve.
*/
void autocorr_original_c(float* x, int n, float* rxx_out) {
    int n_fft = 1 << (int)(ceil(log2(2 * n - 1)));
    int n_out = n / 2;

    fftwf_complex* in_x =
(fftwf_complex*)fftwf_malloc(sizeof(fftwf_complex) * n_fft);
    fftwf_complex* out_x =
(fftwf_complex*)fftwf_malloc(sizeof(fftwf_complex) * n_fft);
    fftwf_complex* in_rev =
(fftwf_complex*)fftwf_malloc(sizeof(fftwf_complex) * n_fft);
    fftwf_complex* out_rev =
(fftwf_complex*)fftwf_malloc(sizeof(fftwf_complex) * n_fft);

    // Prepare original and reversed signals with zero-padding
    for (int i = 0; i < n; i++) {
        in_x[i][0] = x[i];
        in_x[i][1] = 0.0;
        in_rev[i][0] = x[n - 1 - i];
        in_rev[i][1] = 0.0;
    }
    for (int i = n; i < n_fft; i++) {
        in_x[i][0] = in_x[i][1] = 0.0;
        in_rev[i][0] = in_rev[i][1] = 0.0;
    }

    // Create and execute FFT plans
    fftwf_plan plan_fwd_x   = fftwf_plan_dft_1d(n_fft, in_x,  out_x,
FFTW_FORWARD, FFTW_ESTIMATE);
    fftwf_plan plan_fwd_rev = fftwf_plan_dft_1d(n_fft, in_rev, out_rev,
FFTW_FORWARD, FFTW_ESTIMATE);
    fftwf_execute(plan_fwd_x);
    fftwf_execute(plan_fwd_rev);

    // Frequency-domain complex multiplication (cross-spectrum)
    for (int i = 0; i < n_fft; i++) {
        float re_x = out_x[i][0];
        float im_x = out_x[i][1];
        out_x[i][0] = re_x * out_rev[i][0] - im_x * out_rev[i][1];
        out_x[i][1] = re_x * out_rev[i][1] + im_x * out_rev[i][0];
    }

    // Inverse FFT to obtain convolution result
    fftwf_plan plan_bwd = fftwf_plan_dft_1d(n_fft, out_x, in_x,
FFTW_BACKWARD, FFTW_ESTIMATE);
    fftwf_execute(plan_bwd);

    // Biased normalization and copy relevant part of the result
    for (int i = 0; i < n_out; i++) {
        rxx_out[i] = (in_x[n - 1 + i][0] / n_fft) / (float)(n - i);
    }
    // Cleanup
    fftwf_destroy_plan(plan_fwd_x);
    fftwf_destroy_plan(plan_fwd_rev);
    fftwf_destroy_plan(plan_bwd);
    fftwf_free(in_x);
    fftwf_free(out_x);
    fftwf_free(in_rev);
    fftwf_free(out_rev);
}
```

C Implementation – Optimized

```
/*
Versión "Optimized" en C: Emula la lógica de NumPy (Wiener-Khinchin
theorem).
*/
void autocorr_optimized_c(float* x, int n, float* rxx_out) {
    int n_fft = 1 << (int)(ceil(log2(2 * n - 1)));
    int n_out = n / 2;

    fftwf_complex* in_fft  =
(fftwf_complex*)fftwf_malloc(sizeof(fftwf_complex) * n_fft);
    fftwf_complex* out_fft =
(fftwf_complex*)fftwf_malloc(sizeof(fftwf_complex) * n_fft);

    for (int i = 0; i < n; i++) {
        in_fft[i][0] = x[i];
        in_fft[i][1] = 0.0;
    }
    for (int i = n; i < n_fft; i++) {
        in_fft[i][0] = 0.0;
        in_fft[i][1] = 0.0;
    }

    fftwf_plan plan_fwd = fftwf_plan_dft_1d(n_fft, in_fft,  out_fft,
FFTW_FORWARD, FFTW_ESTIMATE);
    fftwf_plan plan_bwd = fftwf_plan_dft_1d(n_fft, out_fft, in_fft,
FFTW_BACKWARD, FFTW_ESTIMATE);
    fftwf_execute(plan_fwd);

    // Compute |X(k)|^2 in frequency domain
    for (int i = 0; i < n_fft; i++) {
        float re = out_fft[i][0];
        float im = out_fft[i][1];
        out_fft[i][0] = re * re + im * im;
        out_fft[i][1] = 0.0;
    }
    fftwf_execute(plan_bwd);
    for (int i = 0; i < n_out; i++) {
        rxx_out[i] = (in_fft[i][0] / n_fft) / (float)(n - i);
    }

    fftwf_destroy_plan(plan_fwd);
    fftwf_destroy_plan(plan_bwd);
    fftwf_free(in_fft);
    fftwf_free(out_fft);
}
```

C Implementation – Superfast

```
/*
Versión "Superfast" en C (normalized by zero-lag).
*/
void autocorr_superfast_c(float* x, int n, float* rxx_out) {
    int n_fft = 1 << (int)(ceil(log2(2 * n - 1)));
    int n_out = n / 2;

    fftwf_complex* in_fft  =
(fftwf_complex*)fftwf_malloc(sizeof(fftwf_complex) * n_fft);
    fftwf_complex* out_fft =
(fftwf_complex*)fftwf_malloc(sizeof(fftwf_complex) * n_fft);

    for (int i = 0; i < n; i++) {
        in_fft[i][0] = x[i];
        in_fft[i][1] = 0.0;
    }
    for (int i = n; i < n_fft; i++) {
        in_fft[i][0] = 0.0;
        in_fft[i][1] = 0.0;
    }

    fftwf_plan plan_fwd = fftwf_plan_dft_1d(n_fft, in_fft,  out_fft,
FFTW_FORWARD, FFTW_ESTIMATE);
    fftwf_plan plan_bwd = fftwf_plan_dft_1d(n_fft, out_fft, in_fft,
FFTW_BACKWARD, FFTW_ESTIMATE);
    fftwf_execute(plan_fwd);

    for (int i = 0; i < n_fft; i++) {
        float re = out_fft[i][0];
        float im = out_fft[i][1];
        out_fft[i][0] = re * re + im * im;
        out_fft[i][1] = 0.0;
    }
    fftwf_execute(plan_bwd);

    float norm_factor = in_fft[0][0];  // Rxx[0] before normalization
    for (int i = 0; i < n_out; i++) {
        rxx_out[i] = in_fft[i][0] / norm_factor;
    }

    fftwf_destroy_plan(plan_fwd);
    fftwf_destroy_plan(plan_bwd);
    fftwf_free(in_fft);
    fftwf_free(out_fft);
}
```

FFTW Compilation Command

The notebook installs FFTW and compiles the above C code into a shared library (`libautocorr.so`) as follows:

```
!apt-get update && apt-get install -y libfftw3-dev
!gcc -shared -o libautocorr.so autocorr_c.c -lfftw3f -lm -O3 -fPIC
```

This creates `libautocorr.so` with the three functions `autocorr_original_c`, `autocorr_optimized_c`, and `autocorr_superfast_c` using FFTW (single-precision).

Python Implementation – Original

```
def AutoCorr1_original(X):
    N = len(X)
    # Explicit FFT-based convolution (fftconvolve)
    fft_cor = scipy.signal.fftconvolve(X, X[::-1], mode='full')[N-1:]
    fft_cor /= (N - np.arange(N))
    return fft_cor[:len(X)//2]
```

This Python function mimics the original C method by using SciPy's FFT-based convolution of `X` with its reversed copy, then applying the same biased normalization.

Python Implementation – Optimized

```
def AutoCorr1_optimized(X):
    N = len(X)
    Nfft = 2 ** int(np.ceil(np.log2(2*N - 1)))
    Xf = np.fft.fft(X, Nfft)
    Rxx = np.fft.ifft(Xf * np.conjugate(Xf)).real
    Rxx = Rxx[:N] / (N - np.arange(N))
    return Rxx[:len(X)//2]
```

This Python function applies the Wiener–Khinchin theorem: it FFTs the signal, multiplies by its complex conjugate, IFFT's the result, and normalizes similarly. It is functionally equivalent to the C optimized version.

Python Implementation – Superfast

```
def AutoCorr1_superfast(X):
    Xf = np.array(X, dtype=np.float32)
    N = Xf.size
    Nfft = 1 << (int(np.ceil(np.log2((N << 1) - 1))))
    F = np.fft.fft(Xf, Nfft)
    Rxx = np.fft.ifft(F * np.conjugate(F)).real[:N]
    Rxx /= Rxx[0]
    return Rxx[:len(X)//2]
```

This function is similar to the optimized version but normalizes the autocorrelation by its zero-lag value `Rxx[0]`, matching the "Superfast" C implementation.

ctypes Configuration

The notebook uses Python's `ctypes` to interface with the C library:

```
lib_c = ctypes.CDLL('./libautocorr.so')
arg_types = [
    np.ctypeslib.ndpointer(dtype=np.float32, flags='C_CONTIGUOUS'),
    ctypes.c_int,
    np.ctypeslib.ndpointer(dtype=np.float32, flags='C_CONTIGUOUS')
]
lib_c.autocorr_original_c.argtypes = arg_types;
lib_c.autocorr_original_c.restype = None
lib_c.autocorr_optimized_c.argtypes = arg_types;
lib_c.autocorr_optimized_c.restype = None
lib_c.autocorr_superfast_c.argtypes = arg_types;
lib_c.autocorr_superfast_c.restype = None
```

This sets each C function to accept a float32 array, an integer length, and a float32 output array. No return value is used (`restype=None`).

Python Wrappers for C Functions

Wrapper functions in Python convert NumPy arrays to the required types and call the C routines:

```
def AutoCorr1_C_Original(X):
    n = len(X)
    x_c = np.array(X, dtype=np.float32)
    rxx_out_c = np.empty(n // 2, dtype=np.float32)
    lib_c.autocorr_original_c(x_c, n, rxx_out_c)
    return rxx_out_c

def AutoCorr1_C_Optimized(X):
    n = len(X)
    x_c = np.array(X, dtype=np.float32)
    rxx_out_c = np.empty(n // 2, dtype=np.float32)
    lib_c.autocorr_optimized_c(x_c, n, rxx_out_c)
    return rxx_out_c

def AutoCorr1_C_Superfast(X):
    n = len(X)
    x_c = np.array(X, dtype=np.float32)
    rxx_out_c = np.empty(n // 2, dtype=np.float32)
    lib_c.autocorr_superfast_c(x_c, n, rxx_out_c)
    return rxx_out_c
```

These functions allow the benchmark to call the C implementations from Python, returning NumPy arrays.

Benchmarking Code

The notebook then measures execution time for all six methods (three Python, three C) over a range of signal sizes. A typical benchmarking loop looks like:

```
sizes = [2**4, 2**5, 2**6, 2**7, 2**8, 2**9, 2**10, 2**11, 2**12, 2**13,
2**14, 2**15, 2**16, 2**17, 2**18]
num_trials = 100
rng = np.random.default_rng()

methods = ['Python Original', 'Python Optimized', 'Python Superfast',
           'C Original', 'C Optimized', 'C Superfast']
mean_times = {m: [] for m in methods}
std_times  = {m: [] for m in methods}

functions_to_test = {
    'Python Original': AutoCorr1_original, 'Python Optimized':
AutoCorr1_optimized,
    'Python Superfast': AutoCorr1_superfast, 'C Original':
AutoCorr1_C_Original,
    'C Optimized': AutoCorr1_C_Optimized, 'C Superfast':
AutoCorr1_C_Superfast
}

print(f"Ejecutando benchmark con {num_trials} pruebas por tamaño...")
for N in sizes:
    times_dict = {m: [] for m in methods}
    signals = rng.standard_normal((num_trials, N)).astype(np.float32)
    for x in signals:
        for name, func in functions_to_test.items():
            start = time.perf_counter_ns()
            _ = func(x)
            times_dict[name].append((time.perf_counter_ns() - start) *
1e-9)

    for m in methods:
        mean_times[m].append(np.mean(times_dict[m]))
        std_times[m].append(np.std(times_dict[m]))
print("Benchmark finalizado.\n")
```

This code runs each method on random signals of increasing length and records mean and standard deviation of execution times (in seconds).

Single Signal Test Execution

Finally, the notebook tests all methods on a single example signal (a noisy square wave) and reports the maximum autocorrelation value and timing:

```
# --- Señal de prueba: onda cuadrada con ruido ---
fs = 1000          # Hz
T  = 2.0           # seconds
t  = np.arange(0, T, 1/fs)
square_wave = np.sign(np.sin(2*np.pi*5*t))
signal = square_wave + 0.2*np.random.randn(len(t))

# --- Compute autocorrelation and time for one run ---
print("\n--- Tiempos para una sola ejecución con señal de prueba ---\n")
results_single_run = {}
for name, func in functions_to_test.items():
    start_time = time.perf_counter()
    r = func(signal)
    elapsed = time.perf_counter() - start_time
    results_single_run[name] = (r, elapsed)
    print(f"{name}: máximo Rxx = {r.max():.3f}, tiempo =
{elapsed*1e3:.2f} ms")
```

This outputs each method's peak autocorrelation value and execution time (in milliseconds) for the test signal. (Plots of the results are omitted here as requested.)

AutoCorr2 Notebook (Spectral Method)

The **AutoCorr2** notebook has essentially the same structure and code as AutoCorr1, again benchmarking the three variants of autocorrelation in C and Python. The main conceptual difference is emphasis: this notebook frames the optimized methods explicitly via the spectral (Wiener–Khinchin) viewpoint, but the implementations are equivalent. All code blocks mirror those in the AutoCorr1 notebook:

- **C implementations** (Original, Optimized, Superfast) – identical to above (see [28–29]).
- **FFTW installation and compilation** – same as in AutoCorr1:

```
!apt-get -qq update && apt-get -qq install -y libfftw3-dev
!gcc -shared -o libautocorr.so autocorr_c.c -lfftw3f -lm -O3 -fPIC
```

- **Python methods (Original, Optimized, Superfast)** – same definitions as in AutoCorr1:

```
def AutoCorr1_original(X):
    N = len(X)
    fft_cor = scipy.signal.fftconvolve(X, X[::-1], mode='full')[N-1:]
    fft_cor /= (N - np.arange(N))
    return fft_cor[:len(X)//2]
```

```
def AutoCorr1_optimized(X):
    N = len(X)
    Nfft = 2 ** int(np.ceil(np.log2(2*N - 1)))
    Xf = np.fft.fft(X, Nfft)
    Rxx = np.fft.ifft(Xf * np.conjugate(Xf)).real
    Rxx = Rxx[:N] / (N - np.arange(N))
    return Rxx[:len(X)//2]

def AutoCorr1_superfast(X):
    Xf = np.array(X, dtype=np.float32)
    N = Xf.size
    Nfft = 1 << (int(np.ceil(np.log2((N << 1) - 1))))
    F = np.fft.fft(Xf, Nfft)
    Rxx = np.fft.ifft(F * np.conjugate(F)).real[:N]
    Rxx /= Rxx[0]
    return Rxx[:len(X)//2]
```

- **ctypes configuration and wrappers** – identical to AutoCorr1:

```
lib_c = ctypes.CDLL('./libautocorr.so')
arg_types = [
    np.ctypeslib.ndpointer(dtype=np.float32, flags='C_CONTIGUOUS'),
    ctypes.c_int,
    np.ctypeslib.ndpointer(dtype=np.float32, flags='C_CONTIGUOUS')
]
lib_c.autocorr_original_c.argtypes = arg_types;
lib_c.autocorr_original_c.restype = None
lib_c.autocorr_optimized_c.argtypes = arg_types;
lib_c.autocorr_optimized_c.restype = None
lib_c.autocorr_superfast_c.argtypes = arg_types;
lib_c.autocorr_superfast_c.restype = None

def AutoCorr1_C_Original(X):
    n = len(X)
    x_c = np.array(X, dtype=np.float32)
    rxx_out_c = np.empty(n // 2, dtype=np.float32)
    lib_c.autocorr_original_c(x_c, n, rxx_out_c)
    return rxx_out_c

def AutoCorr1_C_Optimized(X):
    n = len(X)
    x_c = np.array(X, dtype=np.float32)
    rxx_out_c = np.empty(n // 2, dtype=np.float32)
    lib_c.autocorr_optimized_c(x_c, n, rxx_out_c)
    return rxx_out_c

def AutoCorr1_C_Superfast(X):
    n = len(X)
    x_c = np.array(X, dtype=np.float32)
    rxx_out_c = np.empty(n // 2, dtype=np.float32)
    lib_c.autocorr_superfast_c(x_c, n, rxx_out_c)
    return rxx_out_c
```

- **Benchmark loop** – similar to AutoCorr1 but with fewer sample sizes listed (powers of two from $2^4$ to $2^{18}$):

```
sizes = [2**4, 2**6, 2**8, 2**10, 2**12, 2**14, 2**16, 2**18]
num_trials = 100
rng = np.random.default_rng()
methods = ['Python Original', 'Python Optimized', 'Python Superfast',
           'C Original', 'C Optimized', 'C Superfast']
mean_times = {m: [] for m in methods}
std_times  = {m: [] for m in methods}

functions_to_test = {
    'Python Original': AutoCorr1_original, 'Python Optimized':
AutoCorr1_optimized,
    'Python Superfast': AutoCorr1_superfast, 'C Original':
AutoCorr1_C_Original,
    'C Optimized': AutoCorr1_C_Optimized, 'C Superfast':
AutoCorr1_C_Superfast
}

print(f"Ejecutando benchmark con {num_trials} pruebas por tamaño...")
for N in sizes:
    times_dict = {m: [] for m in methods}
    signals = rng.standard_normal((num_trials, N)).astype(np.float32)
    for x in signals:
        for name, func in functions_to_test.items():
            start = time.perf_counter_ns()
            _ = func(x)
            times_dict[name].append((time.perf_counter_ns() - start) *
1e-9)
    for m in methods:
        mean_times[m].append(np.mean(times_dict[m]))
        std_times[m].append(np.std(times_dict[m]))
print("Benchmark finalizado.\n")
```

- **Single-signal test** – also identical structure:

```
# --- Señal de prueba: onda cuadrada con ruido ---
fs = 1000          # Hz
T  = 2.0           # seconds
t  = np.arange(0, T, 1/fs)
square_wave = np.sign(np.sin(2*np.pi*5*t))
signal = square_wave + 0.2*np.random.randn(len(t))

# --- Compute autocorrelation and time for one run ---
print("\n--- Tiempos para una sola ejecución con señal de prueba ---\n")
results_single_run = {}
for name, func in functions_to_test.items():
    start_time = time.perf_counter()
    r = func(signal)
    elapsed = time.perf_counter() - start_time
    results_single_run[name] = (r, elapsed)
    print(f"{name}: máximo Rxx = {r.max():.3f}, tiempo =
{elapsed*1e3:.2f} ms")
```

As with AutoCorr1, this prints each method's peak autocorrelation and timing. All plotting code (graph generation) has been omitted.