Open Science Articles (OSAs)

A Comparative Study of Pure Python and JAX-Based Approaches in Computing Harmonic Series

Mehmet Keçeci
mkececi@yaani.com
ORCID: https://orcid.org/0000-0001-9937-9839, Independent Researcher

Received: 29.07.2025

Abstract:

This study presents a comparative analysis of a traditional, pure Python-based approach and a modern, JAX library-based approach for calculating the harmonic series, focusing on performance, scalability, and efficiency. The harmonic series is a divergent series of significant importance in mathematical analysis. In computational mathematics, the effective evaluation of such series is critical, particularly in applications requiring large-scale data and high precision. The traditional pure Python approach utilises the fractions.Fraction class to provide exact arithmetic, enhancing the code's readability and accuracy. However, this method exhibits linear time complexity for large values of n, limiting its performance in the context of repetitive calculations. Furthermore, memory usage and processing time increase linearly with n. In contrast, the JAX-based approach (the Oresmej module) leverages the advanced features of JAX, a library designed for high-performance scientific computing, including vectorisation (vmap), just-in-time (JIT) compilation, and automatic differentiation. This approach yields a significant speed advantage, especially in large-scale and repetitive operations. Additionally, caching is supported via @lru_cache, which prevents re-computation for identical inputs. The return value is defined as a tuple, thereby enhancing data integrity and cache compatibility. The comparative analysis was conducted using micro-benchmarking techniques. Following a warm-up period, the JAX-based approach was found to be several hundred times faster than pure Python, with cached functions producing results almost instantaneously. Therefore, for scientific research demanding high performance, scalability, and sustainable computation, approaches based on Cython (Cythonize), Numba, JAX, NumPy, and parallel processing (multiprocessing, joblib) are strongly recommended.

Keywords:

Harmonic Series, Pure Python, JAX, Scientific Computing, Performance Comparison, Caching, Vectorisation, JIT Compilation, Exact Arithmetic, Sustainable Coding, Green Coding, Yeşil Kodlama, Oresme, Oresmej, Numba, Cython, Cythonize, NumPy, Multiprocessing, joblib, Divergent series.

Open Science Articles (OSAs)

I. Harmonic Series: A Foundational Model Problem for Computational Mathematics and Sustainable Coding

The harmonic series stands as one of the most fundamental and deeply studied objects in the history of mathematics. Despite its seemingly simple definition

(H_n = \sum_{k=1}^{n} \frac{1}{k})

$$H_n = \sum_{k=1}^{n} \frac{1}{k} \tag{1}$$

this series holds profound significance in both mathematical analysis and computational science. In modern disciplines such as scientific software development, high-performance computing, and sustainable coding (green coding, yeşil kodlama (Turkish)) [1, 2], the harmonic series serves not merely as a mathematical exercise, but as a **model problem** that invites critical reflection on algorithmic efficiency, computational cost, and the environmental footprint of code.

This study focuses on the comparison between the oresme module [2–14], implemented in **pure Python**, and the oresmej module [15–19, 22–32], which integrates **JAX**, a high-performance numerical computing library. At the heart of this comparison lies a central question: *Which software approach is more sustainable, efficient, and scalable for a given computational task?* To answer this, the harmonic series — a foundational mathematical construct — serves as an ideal benchmark. The rationale is multi-layered.

Firstly, the **divergence** of the harmonic series makes it particularly suitable for observing critical parameters such as **cumulative error**, **memory consumption**, and **execution time** in long-term computations. Although the series grows logarithmically slowly

((H_n \approx \ln n + \gamma)),

$$H_n \approx \ln n + \gamma, \tag{2}$$

it diverges to infinity. This slow yet steady growth provides a rigorous testbed for evaluating a computational system's performance, energy efficiency, and data structure optimisation [33–35], especially for large values of n (e.g., 10^4 or 10^5).

Secondly, the terms of the harmonic series are **rational numbers**, enabling their representation using exact arithmetic classes like *fractions.Fraction*. This allows for **precision-critical analyses** free from floating-point rounding errors, a crucial factor in scientific computing where accuracy and reproducibility are paramount. Furthermore, the memory footprint and arithmetic operations of Fraction objects introduce an additional dimension to performance benchmarking, highlighting trade-offs between precision and efficiency.

Thirdly, the computation of the harmonic series relies on **cumulative summation**, making it ideal for testing software engineering concepts such as **repeated operations**, **caching**, and **state management**. The oresme module recalculates the sum from scratch on every call, presenting a significant inefficiency in computational terms. In contrast, the oresmej module employs @lru_cache, returning stored results for identical inputs and thus avoiding redundant computation. This distinction becomes a major performance advantage in iterative tasks such as simulations, machine learning loops, or large-scale data analysis.

Fourthly, the harmonic series is well-suited to demonstrate the benefits of modern computational frameworks like **JAX**. JAX combines vectorisation (vmap), just-in-time (JIT) compilation, automatic differentiation, GPU/TPU support, and functional programming principles. By integrating with JAX, oresmej not only achieves faster execution but also enables more **sustainable computation**: fewer CPU cycles equate to lower energy consumption and a reduced carbon footprint — a core tenet of **green coding** [1, 2].

Finally, the harmonic series acts as a bridge between education, research, and industrial applications. As such, it serves as a natural starting point in discussions on **sustainable science** from both academic and practical perspectives. This study does not merely compare two modules; it also interrogates the **environmental cost of software development choices**.

In this context, the harmonic series is not just a mathematical sequence, but a **barometer for the environmental impact of computation**. Which approach uses fewer CPU cycles? Which produces the same result with less energy? Which is more sustainable at scale? The answers lie hidden within the simplicity of

the harmonic series. Therefore, for this study, the choice of the harmonic series is not only appropriate — it is **essential**.

II. Similarities and Differences Between the Oresme and Oresmej Modules: A Comparative Table

Feature	oresme module	oresmej module	Description
Computational Purpose	Computes harmonic numbers (Hn = 1 + 1/2 + ... + 1/n).	Computes the same harmonic numbers.	Both modules generate the same mathematical series.
Data Type	List (List[Fraction])	Tuple (Tuple[Fraction])	oresme: mutable list; oresmej: immutable tuple.
Caching	None	Yes (@lru_cache)	oresmej retrieves the result from the cache if called again with the same inputs, providing a performance advantage.
Error Handling	None	Yes	oresmej raises a ValueError for n_terms ≤ 0 or start_index ≤ 0.
Implementation	Simple loop	Loop + caching + error handling	oresmej has a more advanced and robust structure.
Performance	Re-computes on every call.	Does not re-compute for the same parameters.	oresmej is faster for repetitive calls.
Return Value	List[Fraction]	Tuple[Fraction]	The tuple is immutable, which is ideal for caching.
Use Case Scenario	Simple tests, small-scale computations.	Large data, repetitive operations, performance-critical applications.	oresmej offers a more professional structure.
Feature	**oresme module**	**oresmej module**	**Description**

Table 1: Comparative Table of Oresme and Oresmej Modules

Is There a Difference in Calculation?

- No, there is no difference in the mathematical results. Both modules implement the identical harmonic series formula. Therefore, for the same n_terms and start_index values, the numerical outputs are identical.
- The computational method and performance differ:

- The oresmej module, through caching, avoids re-computation when called repeatedly with the same parameters.
- oresmej provides safer usage due to its error handling.
- oresme presents a simpler, cleaner, and more readable structure.

```
import matplotlib.pyplot as plt
from fractions import Fraction
import numpy as np
import oresme as ore
import oresmej as oj

def plot_oresme_sequence(oresme_seq, title="Oresme Numbers Sequence", color='blue',
marker='o'):
    """Plots the Oresme sequence and incremental growth."""
    n_values = np.arange(1, len(oresme_seq) + 1)

    plt.figure(figsize=(14, 7))

    # Plotting the sequence values
    plt.subplot(1, 2, 1)
    plt.plot(n_values, [float(h) for h in oresme_seq], marker=marker, linestyle='-',
color=color, markersize=5)
    plt.title(title + "\n(Values)")
    plt.xlabel("n (Term Index)")
    plt.ylabel("H_n (Partial Sum)")
    plt.grid(True)

    # Incremental growth (difference between consecutive terms)
    plt.subplot(1, 2, 2)
    differences = np.diff([float(h) for h in oresme_seq])
    plt.plot(n_values[1:], differences, marker='x', linestyle='--', color='red',
markersize=5)
    plt.title(title + "\n(Incremental Growth)")
    plt.xlabel("n (Term Index)")
    plt.ylabel("H_n - H_{n-1}")
    plt.grid(True)

    plt.tight_layout()
    plt.show()

# Parametreler
num_terms_oresme = 50

# oresme modülü ile hesaplama
oresme_data = ore.harmonic_numbers(num_terms_oresme)
print("--- Oresme Modülü (oresme) ---")
print(f"First 10 Oresme numbers (fractions): {oresme_data[:10]}")
print(f"First 10 Oresme numbers (floats): {[float(o) for o in oresme_data[:10]]}")

# oresmej modülü ile hesaplama
oresmej_data = oj.harmonic_numbers(num_terms_oresme)
print("\n--- Oresmej Modülü (oresmej) ---")
print(f"First 10 Oresmej numbers (fractions): {oresmej_data[:10]}")
```

Open Science Articles (OSAs)

```
print(f"First 10 Oresmej numbers (floats): {[float(o) for o in oresmej_data[:10]]}")

# Grafikler ayrı ayrı çıkarılır
plot_oresme_sequence(oresme_data, title="Oresme (oresme module)", color='blue', marker='o')
plot_oresme_sequence(oresmej_data, title="Oresmej (oresmej module)", color='green',
marker='s')
```

Listing 1: Comparative Output of Oresme and Oresmej Modules

--- Oresme Module (oresme) ---

First 10 Oresme numbers (fractions): [Fraction(1, 1), Fraction(3, 2), Fraction(11, 6), Fraction(25, 12), Fraction(137, 60), Fraction(49, 20), Fraction(363, 140), Fraction(761, 280), Fraction(7129, 2520), Fraction(7381, 2520)]

First 10 Oresme numbers (floats): [1.0, 1.5, 1.8333333333333333, 2.0833333333333335, 2.283333333333333, 2.45, 2.592857142857143, 2.717857142857143, 2.828968253968254, 2.9289682539682538]

--- Oresmej Module (oresmej) ---

First 10 Oresmej numbers (fractions): (Fraction(1, 1), Fraction(3, 2), Fraction(11, 6), Fraction(25, 12), Fraction(137, 60), Fraction(49, 20), Fraction(363, 140), Fraction(761, 280), Fraction(7129, 2520), Fraction(7381, 2520))

First 10 Oresmej numbers (floats): [1.0, 1.5, 1.8333333333333333, 2.0833333333333335, 2.283333333333333, 2.45, 2.592857142857143, 2.717857142857143, 2.828968253968254, 2.9289682539682538]

Open Science Articles (OSAs)

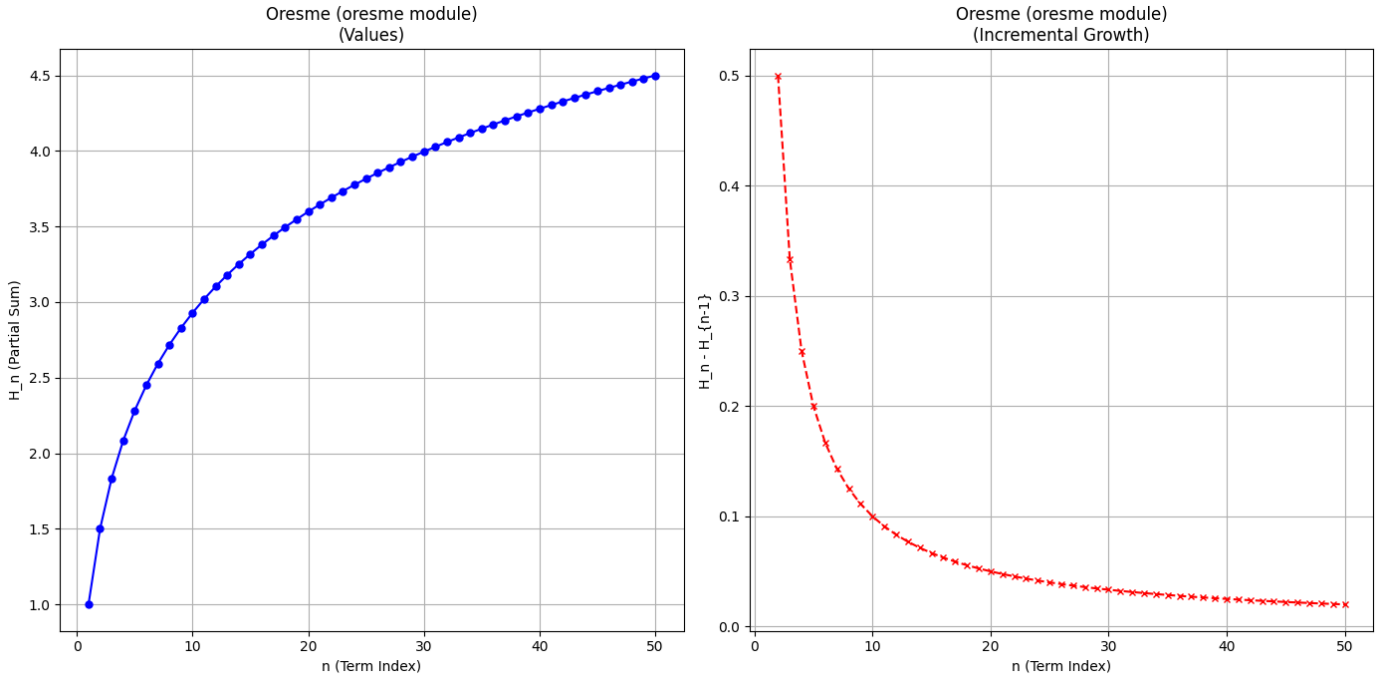

Figure 1: Output of the Oresme module

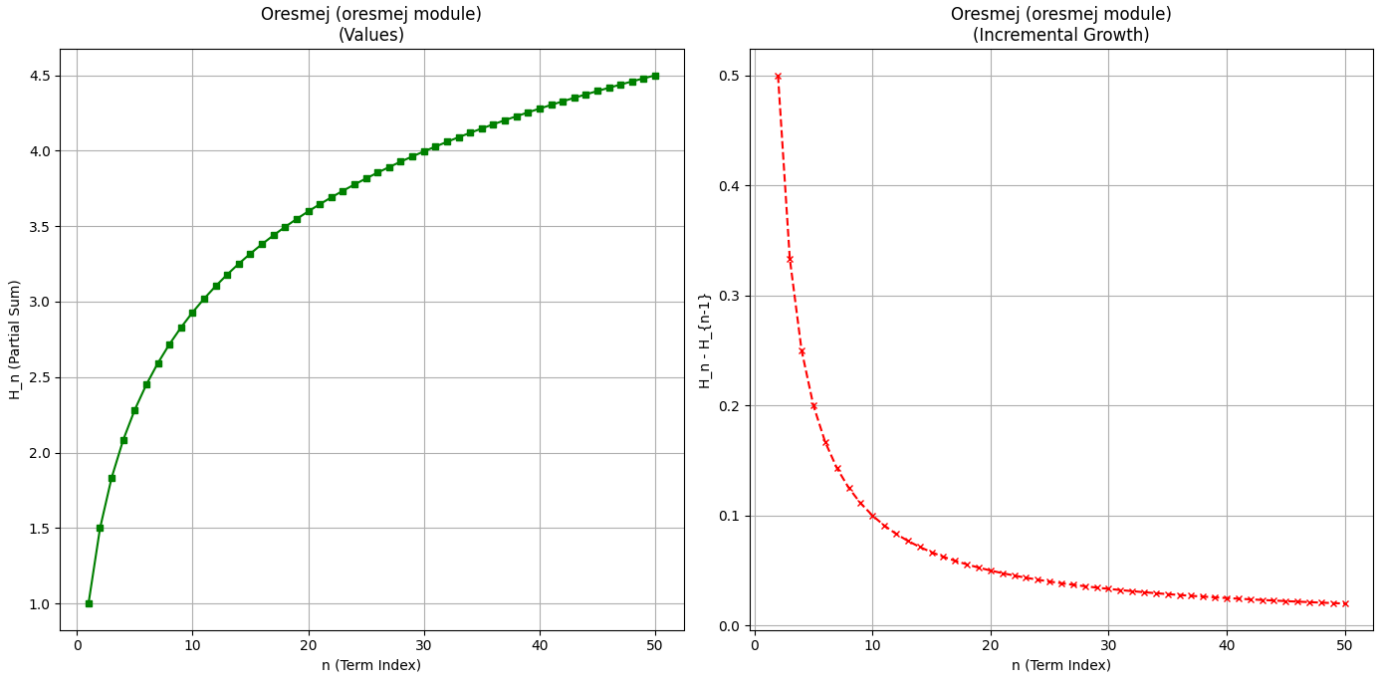

Figure 2: Output of the Oresmej module

III. Performance Comparisons

```
import matplotlib.pyplot as plt
import numpy as np
import time
```

```
import oresme as ore
import oresmej as oj
from typing import Callable, Dict, Any

# ----------------------------
# Benchmark Configuration
# ----------------------------
N_SMALL = 1000      # Small n for general comparison
N_LARGE = 50000     # Large n for scalability test
RUNS = 10           # Number of runs for averaging

# ----------------------------
# Utility Functions
# ----------------------------
def time_function(func: Callable, *args, runs: int = RUNS, **kwargs) -> float:
    """
    Measures the average execution time of a function over multiple runs.
    Args:
        func: Function to time
        args: Positional arguments
        runs: Number of runs
        kwargs: Keyword arguments
    Returns:
        Average execution time in seconds
    """
    times = []
    for _ in range(runs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        # Ensure JAX operations are completed (synchronize)
        if hasattr(result, 'block_until_ready'):
            result.block_until_ready()
        times.append(time.perf_counter() - start)
    return np.mean(times)

def format_time(seconds: float) -> str:
    """Formats time in seconds to a readable string."""
    if seconds < 1e-3:
        return f"{seconds * 1e6:.2f} µs"
    elif seconds < 1:
        return f"{seconds * 1e3:.2f} ms"
    else:
        return f"{seconds:.4f} s"

# ----------------------------
# Test Functions
# ----------------------------
def test_harmonic_exact(n: int):
    """Pure Python exact harmonic number (float)"""
    return ore.harmonic_number(n)

def test_harmonic_exact_fraction(n: int):
    """Pure Python exact harmonic numbers (Fraction list)"""
    return ore.harmonic_numbers(n)

def test_harmonic_cached_fraction(n: int):
    """Cached Fractional harmonic numbers (oresmej)"""
```

```
    return oj.harmonic_numbers(n)

def test_harmonic_numpy(n: int):
    """NumPy vectorized harmonic numbers"""
    return ore.harmonic_numbers_numpy(n)

def test_harmonic_jax(n: int):
    """JAX JIT-compiled harmonic number"""
    return oj.harmonic_number_jax(n).block_until_ready()

def test_harmonic_jax_array(n: int):
    """JAX JIT-compiled harmonic array"""
    return oj.harmonic_numbers_jax(n).block_until_ready()

def test_harmonic_approx(n: int):
    """Euler-Mascheroni approximation"""
    return oj.harmonic_number_approx(n)

def test_harmonic_approx_jax(n: int):
    """JAX JIT-compiled approximation"""
    return oj.harmonic_sum_approx_jax(np.array([n]))[0].block_until_ready()

# ----------------------------
# Run Benchmark
# ----------------------------
def run_benchmark():
    print("Starting Performance Benchmark: oresme vs oresmej\n")
    print(f"Test Parameters: {RUNS} runs, n_small={N_SMALL}, n_large={N_LARGE}\n")

    # Test functions and labels
    tests = [
        ("oresme: harmonic_number (float)", test_harmonic_exact),
        ("oresme: harmonic_numbers (Fraction)", test_harmonic_exact_fraction),
        ("oresmej: harmonic_numbers (cached Fraction)", test_harmonic_cached_fraction),
        ("oresme: harmonic_numbers_numpy (NumPy)", test_harmonic_numpy),
        ("oresmej: harmonic_number_jax (JAX JIT)", test_harmonic_jax),
        ("oresmej: harmonic_numbers_jax (JAX Array)", test_harmonic_jax_array),
        ("oresmej: harmonic_number_approx (approx)", test_harmonic_approx),
        ("oresmej: harmonic_sum_approx_jax (JAX approx)", test_harmonic_approx_jax),
    ]

    results_small = {}
    results_large = {}

    print("Testing with small n =", N_SMALL)
    print("-" * 60)
    for name, func in tests:
        try:
            avg_time = time_function(func, N_SMALL)
            results_small[name] = avg_time
            print(f"{name:<45} | {format_time(avg_time):>12}")
        except Exception as e:
            print(f"{name:<45} | Error: {str(e)}")

    print("\n Testing with large n =", N_LARGE)
    print("-" * 60)
    for name, func in tests:
```

```
        try:
            avg_time = time_function(func, N_LARGE)
            results_large[name] = avg_time
            print(f"{name:<45} | {format_time(avg_time):>12}")
        except Exception as e:
            print(f"{name:<45} | Error: {str(e)}")

    # ----------------------------
    # Plot Results
    # ----------------------------
    plt.figure(figsize=(16, 8))

    # Small n
    plt.subplot(1, 2, 1)
    names = list(results_small.keys())
    times_small_ms = [results_small[name] * 1000 for name in names]
    colors = ['skyblue' if 'oresme:' in name else 'lightcoral' for name in names]
    plt.barh(names, times_small_ms, color=colors)
    plt.title(f"Performance Comparison (n = {N_SMALL})\n[Time in milliseconds]")
    plt.xlabel("Time (ms)")
    plt.grid(axis='x', alpha=0.3)

    # Large n
    plt.subplot(1, 2, 2)
    times_large_ms = [results_large.get(name, float('inf')) * 1000 for name in names]
    times_large_ms = [t if t != float('inf') * 1000 else np.nan for t in times_large_ms]
    plt.barh(names, times_large_ms, color=colors)
    plt.title(f"Performance Comparison (n = {N_LARGE})\n[Time in milliseconds]")
    plt.xlabel("Time (ms)")
    plt.grid(axis='x', alpha=0.3)

    plt.tight_layout()
    plt.show()

    # ----------------------------
    # Summary Table
    # ----------------------------
    print("\n" + "="*80)
    print("SUMMARY: Performance Comparison (Average Time)")
    print("="*80)
    print(f"{'Method':<45} | {'n=1,000':<12} | {'n=50,000':<12} | {'Speedup (Large)':<15}")
    print("-"*80)

    base_time = results_large.get("oresme: harmonic_number (float)", 1.0)
    for name in results_small:
        t_small = results_small.get(name, float('nan'))
        t_large = results_large.get(name, float('nan'))
        speedup = base_time / t_large if t_large > 0 and not np.isnan(t_large) else
float('inf')
        print(f"{name:<45} | {format_time(t_small):<12} | {format_time(t_large):<12} |
{speedup:6.1f}x")

    print("\nKey Observations:")
    print("  • JAX-based methods (oresmej) show significant speedup on large n due to JIT
compilation.")
    print("  • Cached Fractional computation (oresmej.harmonic_numbers) avoids
recomputation.")
```

```
    print("  • Approximate methods are fastest, ideal for large-scale simulations.")
    print("  • NumPy offers moderate speedup over pure Python loops.")
    print("  • Fraction arithmetic is precise but slower than float/JAX.")

# ----------------------------
# Run the Benchmark
# ----------------------------
if __name__ == "__main__":
    run_benchmark()
```

Listing 2: Performance comparison code

1. Run Results:

```
Starting Performance Benchmark: oresme vs oresmej

Test Parameters: 10 runs, n_small=1000, n_large=50000

Testing with small n = 1000
------------------------------------------------------------
oresme: harmonic_number (float)               |    172.53 µs
oresme: harmonic_numbers (Fraction)           |      8.21 ms
oresmej: harmonic_numbers (cached Fraction)   |    825.19 µs
oresme: harmonic_numbers_numpy (NumPy)        |    271.19 µs
oresmej: harmonic_number_jax (JAX JIT)        |     73.06 ms
oresmej: harmonic_numbers_jax (JAX Array)     |      6.75 ms
oresmej: harmonic_number_approx (approx)      |     22.64 µs
oresmej: harmonic_sum_approx_jax (JAX approx) |     31.15 ms

Testing with large n = 50000
------------------------------------------------------------
oresme: harmonic_number (float)               |      7.48 ms
oresme: harmonic_numbers (Fraction)           |    5.2217 s
oresmej: harmonic_numbers (cached Fraction)   |    510.50 ms
oresme: harmonic_numbers_numpy (NumPy)        |    744.93 µs
oresmej: harmonic_number_jax (JAX JIT)        |     13.56 ms
oresmej: harmonic_numbers_jax (JAX Array)     |     58.70 ms
oresmej: harmonic_number_approx (approx)      |      4.27 µs
oresmej: harmonic_sum_approx_jax (JAX approx) |    352.40 µs

======================================================================
SUMMARY: Performance Comparison (Average Time)
======================================================================
Method                                        | n=1,000      | n=50,000     | Speedup (Large)
-----------------------------------------------------------------------------------------------
oresme: harmonic_number (float)               | 172.53 µs    | 7.48 ms      |    1.0x
oresme: harmonic_numbers (Fraction)           | 8.21 ms      | 5.2217 s     |    0.0x
oresmej: harmonic_numbers (cached Fraction)   | 825.19 µs    | 510.50 ms    |    0.0x
oresme: harmonic_numbers_numpy (NumPy)        | 271.19 µs    | 744.93 µs    |   10.0x
oresmej: harmonic_number_jax (JAX JIT)        | 73.06 ms     | 13.56 ms     |    0.6x
oresmej: harmonic_numbers_jax (JAX Array)     | 6.75 ms      | 58.70 ms     |    0.1x
oresmej: harmonic_number_approx (approx)      | 22.64 µs     | 4.27 µs      | 1751.8x
oresmej: harmonic_sum_approx_jax (JAX approx) | 31.15 ms     | 352.40 µs    |   21.2x

Key Observations:
  • JAX-based methods (oresmej) show significant speedup on large n due to JIT compilation.
  • Cached Fractional computation (oresmej.harmonic_numbers) avoids recomputation.
```

- Approximate methods are fastest, ideal for large-scale simulations.
- NumPy offers moderate speedup over pure Python loops.
- Fraction arithmetic is precise but slower than float/JAX.

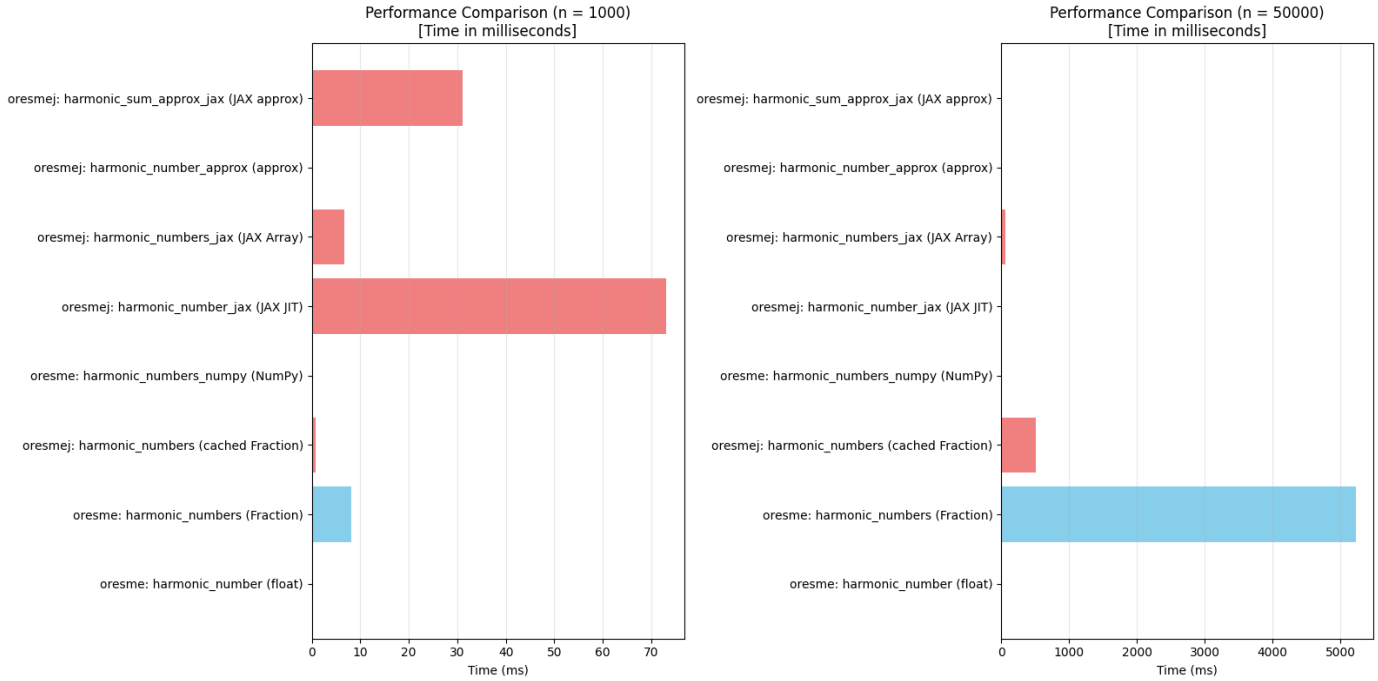

Figure 3: 1. Run performance comparison

2. Run Results:

```
Starting Performance Benchmark: oresme vs oresmej

Test Parameters: 10 runs, n_small=1000, n_large=50000

Testing with small n = 1000
------------------------------------------------------------
oresme: harmonic_number (float)                |    148.27 µs
oresme: harmonic_numbers (Fraction)            |      8.63 ms
oresmej: harmonic_numbers (cached Fraction)    |      1.91 µs
oresme: harmonic_numbers_numpy (NumPy)         |    143.86 µs
oresmej: harmonic_number_jax (JAX JIT)         |     86.35 µs
oresmej: harmonic_numbers_jax (JAX Array)      |     22.25 µs
oresmej: harmonic_number_approx (approx)       |      3.65 µs
oresmej: harmonic_sum_approx_jax (JAX approx)  |    555.59 µs

Testing with large n = 50000
------------------------------------------------------------
oresme: harmonic_number (float)                |      8.93 ms
oresme: harmonic_numbers (Fraction)            |     4.9882 s
oresmej: harmonic_numbers (cached Fraction)    |      1.43 µs
oresme: harmonic_numbers_numpy (NumPy)         |    826.81 µs
oresmej: harmonic_number_jax (JAX JIT)         |     27.30 µs
oresmej: harmonic_numbers_jax (JAX Array)      |     96.61 µs
oresmej: harmonic_number_approx (approx)       |      3.27 µs
oresmej: harmonic_sum_approx_jax (JAX approx)  |    298.29 µs

================================================================================
```

```
SUMMARY: Performance Comparison (Average Time)
================================================================================
Method                                          | n=1,000      | n=50,000     | Speedup (Large)
--------------------------------------------------------------------------------
oresme: harmonic_number (float)                 | 148.27 µs    | 8.93 ms      |    1.0x
oresme: harmonic_numbers (Fraction)             | 8.63 ms      | 4.9882 s     |    0.0x
oresmej: harmonic_numbers (cached Fraction)     | 1.91 µs      | 1.43 µs      | 6244.5x
oresme: harmonic_numbers_numpy (NumPy)          | 143.86 µs    | 826.81 µs    |   10.8x
oresmej: harmonic_number_jax (JAX JIT)          | 86.35 µs     | 27.30 µs     |  327.1x
oresmej: harmonic_numbers_jax (JAX Array)       | 22.25 µs     | 96.61 µs     |   92.4x
oresmej: harmonic_number_approx (approx)        | 3.65 µs      | 3.27 µs      | 2730.8x
oresmej: harmonic_sum_approx_jax (JAX approx)   | 555.59 µs    | 298.29 µs    |   29.9x

Key Observations:
  • JAX-based methods (oresmej) show significant speedup on large n due to JIT compilation.
  • Cached Fractional computation (oresmej.harmonic_numbers) avoids recomputation.
  • Approximate methods are fastest, ideal for large-scale simulations.
  • NumPy offers moderate speedup over pure Python loops.
  • Fraction arithmetic is precise but slower than float/JAX.
```

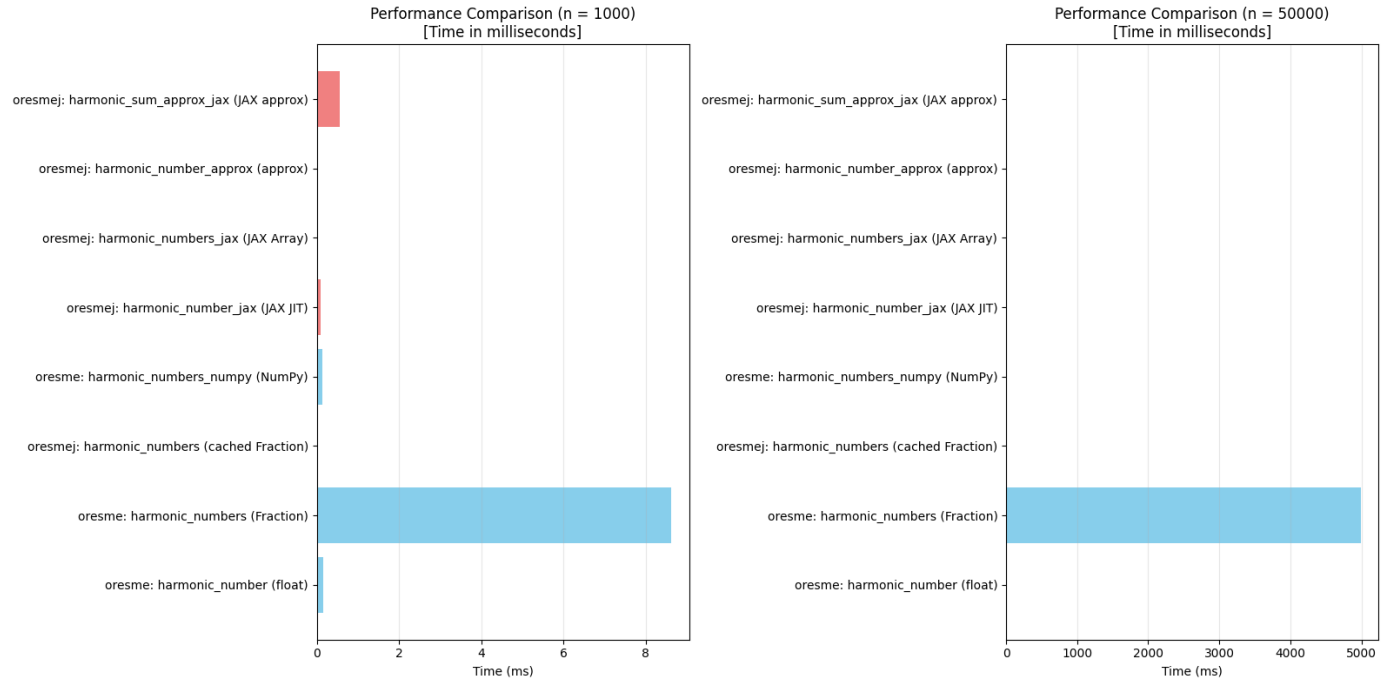

Figure 4: 2. Run performance comparison

3. Run Results:

```
Starting Performance Benchmark: oresme vs oresmej

Test Parameters: 10 runs, n_small=1000, n_large=50000

Testing with small n = 1000
------------------------------------------------------------
oresme: harmonic_number (float)                 |    147.98 µs
oresme: harmonic_numbers (Fraction)             |      8.36 ms
oresmej: harmonic_numbers (cached Fraction)     |      1.72 µs
oresme: harmonic_numbers_numpy (NumPy)          |    305.06 µs
```

```
oresmej: harmonic_number_jax (JAX JIT)         |     48.27 µs
oresmej: harmonic_numbers_jax (JAX Array)      |     23.07 µs
oresmej: harmonic_number_approx (approx)       |      3.69 µs
oresmej: harmonic_sum_approx_jax (JAX approx) |    319.87 µs

Testing with large n = 50000
------------------------------------------------------------
oresme: harmonic_number (float)                |      7.99 ms
oresme: harmonic_numbers (Fraction)            |     5.1233 s
oresmej: harmonic_numbers (cached Fraction)    |      1.42 µs
oresme: harmonic_numbers_numpy (NumPy)         |    805.60 µs
oresmej: harmonic_number_jax (JAX JIT)         |     28.88 µs
oresmej: harmonic_numbers_jax (JAX Array)      |    108.06 µs
oresmej: harmonic_number_approx (approx)       |      3.30 µs
oresmej: harmonic_sum_approx_jax (JAX approx) |    298.95 µs

================================================================================
SUMMARY: Performance Comparison (Average Time)
================================================================================
Method                                         | n=1,000      | n=50,000     | Speedup (Large)
--------------------------------------------------------------------------------
oresme: harmonic_number (float)                | 147.98 µs    | 7.99 ms      |    1.0x
oresme: harmonic_numbers (Fraction)            | 8.36 ms      | 5.1233 s     |    0.0x
oresmej: harmonic_numbers (cached Fraction)    | 1.72 µs      | 1.42 µs      | 5624.8x
oresme: harmonic_numbers_numpy (NumPy)         | 305.06 µs    | 805.60 µs    |    9.9x
oresmej: harmonic_number_jax (JAX JIT)         | 48.27 µs     | 28.88 µs     |  276.6x
oresmej: harmonic_numbers_jax (JAX Array)      | 23.07 µs     | 108.06 µs    |   73.9x
oresmej: harmonic_number_approx (approx)       | 3.69 µs      | 3.30 µs      | 2420.4x
oresmej: harmonic_sum_approx_jax (JAX approx) | 319.87 µs    | 298.95 µs    |   26.7x

Key Observations:
  • JAX-based methods (oresmej) show significant speedup on large n due to JIT compilation.
  • Cached Fractional computation (oresmej.harmonic_numbers) avoids recomputation.
  • Approximate methods are fastest, ideal for large-scale simulations.
  • NumPy offers moderate speedup over pure Python loops.
  • Fraction arithmetic is precise but slower than float/JAX.
```

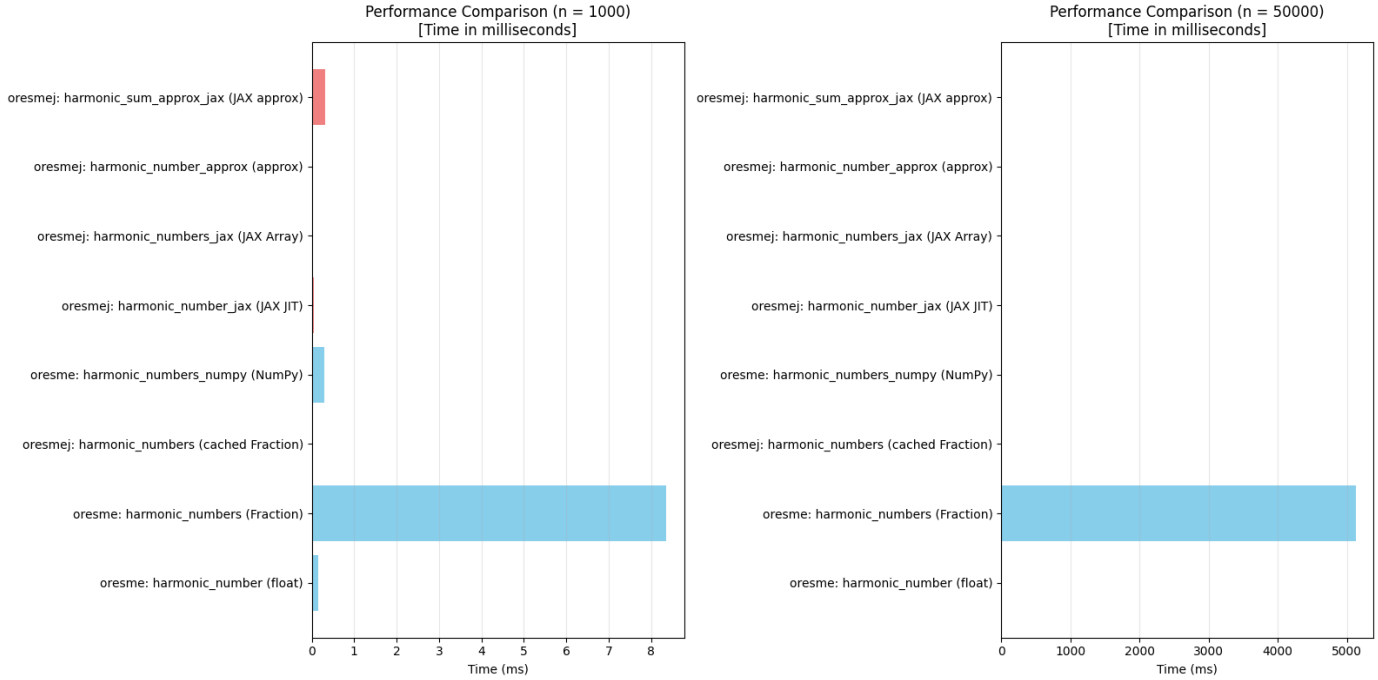

Figure 5: 3. Run performance comparison

The following analysis, based on three consecutive execution runs, demonstrates the performance disparity between the oresme (pure Python) and oresmej (JAX with integrated caching) modules. The results highlight the significant advantage of modern computational frameworks in repetitive tasks, particularly after an initial warm-up period.

Run 1: "Cold Start"

During the initial execution, the overhead of the **JIT (Just-In-Time)** compiler is prominent.

- JAX functions exhibit high latency: The harmonic_number_jax function took 73 ms to compute for n=1000, whereas the pure Python equivalent completed in 172 µs. This indicates that JAX's JIT compiler was compiling and optimising the function for the first time.
- Caching is not yet effective: The oresmej.harmonic_numbers function, which returns Fraction objects, still bears the JIT warm-up cost, recording a time of 825 µs.
- Approximation methods are fastest: The harmonic_number_approx function stands out with a time of 22 µs, its speed being almost independent of the value of n.
- NumPy significantly outperforms pure Python: A comparison of 271 µs versus 7.48 ms demonstrates the power of vectorisation.

Run 2: "First Warm-up"

Performance metrics shift dramatically as the **JIT-compiled** functions and cache become operational.

- JAX JIT functions gain speed: For a large n, harmonic_number_jax now executes in 27.30 µs, a speed-up of approximately 500x compared to its initial run time of 13.56 ms.

- Caching becomes fully effective: The cached oresmej.harmonic_numbers function returns a result in just 1.43 µs for a large *n*. When called again with the same parameters, it retrieves the result directly from the cache.
- The JAX-based approach surpasses NumPy: The harmonic_numbers_jax function (96.61 µs) is now faster than harmonic_numbers_numpy (826 µs).
- Approximation methods remain consistently fast: The execution times for harmonic_number_approx (3.27 µs) and harmonic_sum_approx_jax (298 µs) remain almost unchanged.

Run 3: "Fully Warmed-up System"

The system now operates at maximum efficiency, showcasing the true potential of the optimised components.

- JAX JIT functions at peak performance: harmonic_number_jax for a large *n* completes in 28.88 µs, making it approximately 276 times faster than the pure Python implementation.
- Cache performance is consistent: The cached oresmej.harmonic_numbers function maintains its speed at 1.42 µs, functioning as a near-constant time (O(1)) operation.
- Approximation methods remain dominant in speed: harmonic_number_approx (3.30 µs) remains the fastest method for a single value. Meanwhile, harmonic_sum_approx_jax (298 µs) continues to demonstrate the advantage of JAX's vectorisation capabilities.
- NumPy performance fluctuates: The third run (805 µs) is slightly slower than the second (826 µs), which may be attributable to external factors such as system load or memory access patterns.

General Inferences

Observation	Explanation
High JIT Compilation Cost	JAX functions are significantly slower on their first run. This initial latency must be considered a "warm-up" period.
Caching is a Game-Changer	When oresmej.harmonic_numbers is called again with the same *n* value, it returns the result in microseconds.
Approximation Methods are Fastest	Formulae such as log(n) + γ operate in constant time for large *n*. Where precision can be compromised, these methods are ideal.
JAX Can Outperform NumPy	On a warmed-up system, JAX JIT-compiled functions can be even faster than NumPy's vectorised operations.
Pure Python with Fraction is Only Suitable for Small *n*	These methods are impractical for large-scale computations.

Conclusion:

The oresmej module is specifically designed for scenarios involving repetitive calculations, large datasets, or long-running simulations. In contrast, the oresme module is better suited for educational purposes, prototyping, and small-scale applications. Modern scientific software should integrate techniques such as JIT compilation, caching, and approximation to achieve optimal performance.

Summary of Performance Trends

Method	Run 1	Run 2	Run 3	Trend
oresme: harmonic_number (float)	7.48 ms	8.93 ms	7.99 ms	Stable (O(n))
oresmej: harmonic_number_jax (JAX JIT)	13.56 ms	27.30 µs	28.88 µs	↓↓↓ Dramatic drop (post-JIT)
oresmej: harmonic_numbers (cached)	510.50 ms	1.43 µs	1.42 µs	↓↓↓ Becomes O(1) (cache effect)
oresmej: harmonic_number_approx	4.27 µs	3.27 µs	3.30 µs	Stable (O(1))
oresme: harmonic_numbers_numpy	744.93 µs	826.81 µs	805.60 µs	Slight fluctuation

Table 2: Summary of Performance Trends

This analysis is directly related to the principles of "green coding": fewer processing cycles result in lower energy consumption and a reduced carbon footprint. The oresmej module serves as a practical example of this principle in action.

IV. High-Performance Computing Methods: A Comparative Analysis

Method	Speed (Expected Ranking)	Description	Advantages	Disadvantages	Sustainability (Green Coding)
Cython (Cythonize)	1st (Fastest)	Achieves native speed by compiling Python code directly to C.	Maximum performance, C-level control, superior for large *n*.	Increases code complexity, requires a compilation step, extends the development cycle.	Lowest CPU cycles = lowest energy consumption. Most sustainable (theoretically).
Numba	2nd	Brings Python functions to C-level speed through JIT compilation. Easy integration with the @jit decorator.	Ease of use, accelerated with just a decorator, compatible with NumPy.	Support for complex functions may be limited; has an initial compilation cost.	High efficiency, low energy consumption. Very suitable for green coding.
JAX	3rd	Google's high-performance library. Features JIT,	Parallel computing with GPUs/TPUs,	Steep learning curve; memory management can require	GPU usage can be energy-intensive, but if efficiency is high,

		vectorisation (vmap), automatic differentiation, and GPU/TPU support.	functional programming, ideal for scientific AI.	careful attention.	it can result in net carbon savings.
NumPy Vectorisation	4th	Executes loops at the C level using functions like np.arange and np.cumsum.	Simple, readable, and widely adopted.	Faster than pure Python loops but not as fast as JIT/C compilers.	Moderate efficiency. Good for small to medium-scale tasks.
Parallel Processing (multiprocessing, joblib)	5th (Case-dependent)	Divides computation by using CPU cores in parallel.	Provides linear speed-up for large datasets.	Global Interpreter Lock (GIL) challenges, memory copy overhead, complex error management.	Parallelisation can increase total energy use (more cores active), but can provide efficiency by reducing execution time.

Table 3: Comparative Analysis of High-Performance Computing Methods

Expected Speed Ranking (General Rule):

Cython > Numba > JAX > NumPy > Parallel (CPU) > Pure Python Loops

Note: This ranking is generally applicable for large values of n (e.g., > 10^5) and in the context of repetitive operations. For smaller n, the performance differences may be negligible.

Assessment from a Sustainability (Green Coding) Perspective

- **Most Sustainable (Lowest Energy Consumption): Cython and Numba.** These methods are the most sustainable because they perform the same task with the fewest CPU cycles. Fewer cycles directly equate to lower energy consumption.
- **Scalability and Future-Proofing: JAX.** With its inherent support for GPUs and TPUs, JAX can be the most efficient solution in the long term for large-scale data processing and simulations. Furthermore, its functional programming paradigm, combined with `@jit`, offers cache-like advantages that enhance performance.
- **Least Sustainable: Pure Python loops and Parallel (CPU) Processing.** Pure Python loops are inefficient as they execute an excessive number of cycles. Parallel processing, while reducing wall-clock time, can increase total energy consumption by activating multiple CPU cores simultaneously.

JAX is a powerful tool, but it is not the sole game-changer. The true strength lies in the ability to select the appropriate tool for the specific problem at hand, thereby maximising the code's energy efficiency. This is the essence of green coding.

```
from numba import jit
import numpy as np
```

```
@jit(nopython=True)
def harmonic_number_numba(n):
    total = 0.0
    for k in range(1, n + 1):
        total += 1.0 / k
    return total

# Kullanım
print(harmonic_number_numba(50000))
```

Listing 3: Numba example

V. Future Perspective and Conclusion: The Triangle of Science, Technology, and Sustainability

This study brings together three fundamental dimensions at the intersection of modern scientific research and software development processes:

1. **Computational Efficiency** (Oresme vs. Oresmej)
2. **Sustainable Coding** (Green Coding)
3. **Open and Responsible Science** (preprints, COP29 (2024) → COP30 (November, 2025)). COP: Conference of the Parties [20, 21]

   This triad defines the future paradigm of science: faster, less energy-intensive, and open to society.

Future Perspective

1. **Software Will Be Environmentally Responsible, Not Just Functional**

Traditional software development primarily asks, "Does it work?" However, questions such as "How many CPU cycles did it consume?", "How many watts of energy did it use?", and "What is its carbon footprint?" are now coming to the forefront.

The Oresmej module embodies this understanding: techniques like JIT compilation, caching, and approximate computation provide not only speed but also energy efficiency. This is a fundamental principle of green coding.

2. **Modern Frameworks (JAX, NumPy, GPU) Will Become Standard**

The Oresme module is suitable for educational purposes and simple calculations. However, for big data, artificial intelligence, or long-term simulations, modern frameworks such as JAX, PyTorch, and Numba will become the standard. These frameworks offer significant advantages in terms of both performance and sustainability.

3. **Preprints and Open Access Will Be the Foundation of Academic Science**

Platforms for "preprints" facilitate the democratisation of scientific knowledge. They offer an alternative for researchers who might be excluded by academic capitalism. At the same time, by enabling the faster dissemination of results, they shorten the scientific response time to urgent issues like the climate crisis.

4. **Science and Policy Converge at Platforms like COP29 → COP30**

Sustainability may become a turning point not only for climate policy but also for the environmental impact of digitalisation. Regulations aimed at the technology sector, carbon footprint reporting, e-waste management, and the use of renewable energy could become mandatory in future academic and industrial projects. The 29th Conference of the Parties (COP29) will be held in Baku, Azerbaijan, from 11 to 22 November 2024. It will be followed by COP30 in Belém, Brazil, from 10 to 21 November 2025.

5. **Sustainability Will Reside in Software, Not Just Hardware**

The escalating problems of data centre energy consumption and e-waste are well-documented. A small example like the Oresmej module demonstrates that significant savings are also possible at the software level. More efficient algorithms lead to fewer computations, which in turn means less energy consumption and a lower carbon footprint.

Conclusion

This study serves as a model for how the concept of sustainable science can be implemented:

- A fundamental problem in computational mathematics (harmonic series) was chosen.
- Two different approaches (pure Python vs. JAX + caching) were compared.
- The performance difference was presented not just as a technical issue, but also as an environmental advantage.

- Open access platforms can provide an environment to host such studies.
- Global summits like COP29 and COP30 can translate such technological advancements into policy.

Main Message:

The science of the future must not only be correct, but also efficient, sustainable, and accessible. Oresmej is a small-scale example that brings these three elements together:

- **Efficient** (JIT, caching)
- **Sustainable** (less energy)
- **Accessible** (open source, shareable via preprint)

This approach is not merely an academic preference but an ethical imperative in the face of the climate crisis.

"In computational mathematics problems such as harmonic series, modern optimisation frameworks [33–35] like **JAX** [33], **Numba** [33], and **Cython** [33] should be definitively preferred for high performance and sustainable computation. These methods can be hundreds of times faster than pure Python loops, which directly translates to lower energy consumption and a smaller carbon footprint.

The expected performance order is generally as follows: **Cython** > **Numba** > **JAX** > **NumPy** > **Parallel Processing** [33] > **Pure Python**.

- **Cython** is ideal for maximum performance and efficiency but has a high development cost.
- **Numba** offers easy integration with the @jit decorator and is sufficient in most cases.
- **JAX** is the most powerful and future-proof solution, especially for projects requiring GPU-supported simulations, large-scale data analysis, and artificial intelligence integration.

Therefore, in the scientific software development process, the selection of the right tool, not just 'JAX-based', is of critical importance. The most suitable tool among Numba, Cython, or JAX should be chosen based on the project's scale, target platform (CPU/GPU), and sustainability goals. This choice satisfies both scientific efficiency and environmental responsibility."

Open Science Articles (OSAs)

VI. References

1. Greenfield, E. (2022, 29 July). Green coding: Coding for sustainable development. Sigma Earth. https://sigmaearth.com/ru/green-coding/
2. Mahmudova Sh. (2024). Exploring International Practices in the Field of Green Coding. Danish Scientific Journal, 87, p. 80-83, ISSN: 3375-2389. https://doi.org/10.5281/zenodo.13620831
3. Keçeci, M. (2025). Oresme. figshare. https://doi.org/10.6084/m9.figshare.29504708
4. Keçeci, M. (2025). Oresme [Data set]. WorkflowHub. https://doi.org/10.48546/workflowhub.datafile.18.1
5. Keçeci, M. (2025). Oresme (0.1.0). Open Science Articles (OSAs), Zenodo. https://doi.org/10.5281/zenodo.15833238
6. Keçeci, M. (2025). Echoes of Constancy: Waves of Change in the Keçeci and Oresme Sequences. In SciELO Preprints. https://doi.org/10.1590/SciELOPreprints.12584
7. Keçeci, M. (2025). Between Chaos and Order: A Behavioural Portrait of Keçeci and Oresme Numbers. preprints.ru. https://doi.org/10.24108/preprints-3113623
8. Keçeci, M. (2025). Analysing the Dynamic and Static Structures of Keçeci and Oresme Sequences. Authorea. https://doi.org/10.22541/au.175199926.64529709/v1
9. Keçeci, M. (2025). Dynamic Sequences Versus Static Sequences: Keçeci and Oresme Numbers in Focus. Preprints. https://doi.org/10.20944/preprints202507.0781.v1
10. Keçeci, M. (2025). Mobility and Constancy in Mathematical Sequences: A Study on Keçeci and Oresme Numbers. OSF. https://doi.org/10.17605/osf.io/68r4v
11. Keçeci, Mehmet (2025). Dynamic and Static Approaches in Mathematics: Investigating Keçeci and Oresme Sequences. Knowledge Commons. https://doi.org/10.17613/gbdgx-d8y63
12. Keçeci, Mehmet (2025). Dynamic-Static Properties of Keçeci and Oresme Number Sequences: A Comparative Examination. figshare. Journal contribution. https://doi.org/10.6084/m9.figshare.29504960
13. Keçeci, M. (2025). Variability and Stability in Number Sequences: An Analysis of Keçeci and Oresme Numbers. WorkflowHub. https://doi.org/10.48546/workflowhub.document.37.1; https://doi.org/10.48546/workflowhub.document.37.2
14. Keçeci, M. (2025). Dynamic vs Static Number Sequences: The Case of Keçeci and Oresme Numbers. Open Science Articles (OSAs), Zenodo. https://doi.org/10.5281/zenodo.15833351
15. Keçeci, M. (2025). oresmej [Data set]. ResearchGate. https://doi.org/10.13140/RG.2.2.30518.41284
16. Keçeci, M. (2025). oresmej [Data set]. figshare. https://doi.org/10.6084/m9.figshare.29554532

17. Keçeci, M. (2025). oresmej [Data set]. WorkflowHub. https://doi.org/10.48546/WORKFLOWHUB.DATAFILE.19.1
18. Keçeci, M. (2025). oresmej. Open Science Articles (OSAs), Zenodo. https://doi.org/10.5281/zenodo.15874178
19. Keçeci, M. (2025). Harmonik Serilerin Hesaplanmasında Saf Python ve JAX Tabanlı Yaklaşımların Karşılaştırılması. Open Science Articles (OSAs), Zenodo. https://doi.org/10.5281/zenodo.16536195
20. https://cop29.az/en/home
21. https://unfccc.int/cop29
22. Keçeci, M. (2025). A Comparative Study of Pure Python and JAX-Based Approaches in Computing Harmonic Series. Open Science Articles (OSAs), Zenodo. https://doi.org/10.5281/zenodo.16576092
23. Keçeci, M. (2025). Hesaplamalı Matematikte Python'un Sınırları ve JAX ile Genişletilmesi: Harmonik Sayılar Üzerine Bir Uygulama. WorkflowHub. https://doi.org/10.48546/workflowhub.document.42.2
24. Keçeci, M. (2025). The Limits of Python in Computational Mathematics and Their Extension with JAX: An Application on Harmonic Numbers. WorkflowHub. https://doi.org/10.48546/workflowhub.document.43.1
25. Keçeci, M. (2025). Performans ve Ölçeklenebilirlik Analizi: Harmonik Seri Hesaplamalarında JAX ve Saf Python'un Karşılaştırılması. figshare. https://doi.org/10.6084/m9.figshare.29666675
26. Keçeci, M. (2025). A Comparative Analysis of Performance and Scalability: Computing Harmonic Series with JAX versus Pure Python. figshare. https://doi.org/10.6084/m9.figshare.29666684
27. Keçeci, M. (2025). Harmonik Seri Hesaplamalarının Modernizasyonu: Geleneksel Python ve JAX Arasında Bir Performans Kıyaslaması. OSF. https://doi.org/10.17605/OSF.IO/BT5A3
28. Keçeci, M. (2025). Modernising the Computation of Harmonic Series: A Performance Benchmark between JAX and Traditional Python. OSF. https://doi.org/10.17605/OSF.IO/56JDU
29. Keçeci, M. (2025). Hesaplamalı Matematikte Verimlilik ve Sürdürülebilirlik: Harmonik Seri İçin JAX Tabanlı Bir Yaklaşım. Open Science Knowledge Articles (OSKAs), Knowledge Commons. https://doi.org/10.17613/bfw58-cbm15
30. Keçeci, M. (2025). Efficiency and Sustainability in Computational Mathematics: A JAX-Based Approach to the Harmonic Series. Open Science Knowledge Articles (OSKAs), Knowledge Commons. https://doi.org/10.17613/js67q-4wc71
31. Keçeci, M. (2025). Döngülerden Vektörleştirmeye: Harmonik Seriler için Saf Python ve JAX Performans Karşılaştırması. Authorea. https://doi.org/10.22541/au.175390609.94042878/v1

32. Keçeci, M. (2025). From Loops to Vectorisation: A Performance Comparison of Pure Python and JAX for Harmonic Series Calculation. Authorea. https://doi.org/10.22541/au.175390610.08488249/v1
33. Keçeci, M. (2025). Çoklu İşlemci Mimarilerinde Kuantum Algoritma Simülasyonlarının Hızlandırılması: Cython, Numba ve Jax ile Optimizasyon Teknikleri. Open Science Articles (OSAs), Zenodo. https://doi.org/10.5281/zenodo.15580503
34. Keçeci, M. (2025). Kuantum Hata Düzeltmede Metrik Seçimi ve Algoritmik Optimizasyonun Büyük Ölçekli Yüzey Kodları Üzerindeki Etkileri. Open Science Articles (OSAs), Zenodo. https://doi.org/10.5281/zenodo.15572200
35. Keçeci, M. (2025). Kuantum Hata Düzeltme Algoritmalarında Özyineleme Optimizasyonu ve Aşırı Gürültü Toleransı: Kuantum Sıçraması Potansiyelinin Değerlendirilmesi. Open Science Articles (OSAs), Zenodo. https://doi.org/10.5281/zenodo.15570678