NuFast: Efficient Three-Flavor Neutrino Oscillation Probabilities in Rust

Baalateja Kataru

Plankeon Labs
baalateja@planckeon.org

February 2026

Abstract. We present `nufast`, a Rust implementation of the NuFast algorithm for computing three-flavor neutrino oscillation probabilities in vacuum and matter. Our implementation achieves performance competitive with optimized C++ code, with 61 ns for vacuum and 95 ns for matter calculations per energy point —approximately 27% faster than C++ for matter effects. The crate is published on crates.io and includes WebAssembly bindings for browser-based applications. This work enables high-performance neutrino physics calculations in modern software ecosystems.

1 Introduction

Neutrino oscillation is a quantum mechanical phenomenon where neutrinos change flavor as they propagate through space. Accurate and efficient computation of oscillation probabilities is essential for analyzing data from neutrino experiments such as DUNE, Hyper-Kamiokande, and JUNO.

The NuFast algorithm, developed by Denton and Parke [1], provides a computationally efficient method for calculating three-flavor oscillation probabilities including matter effects. The original implementation was provided in C++, Python, and Fortran.

We present `nufast`, a Rust port of this algorithm, designed to leverage Rust's memory safety guarantees and zero-cost abstractions while achieving performance comparable to or better than the original implementations.

2 Algorithm Overview

The NuFast algorithm computes the full 3×3 oscillation probability matrix $P_{\alpha\beta}$ for neutrino flavor transitions $\nu_\alpha \to \nu_\beta$ where $\alpha, \beta \in \{e, \mu, \tau\}$.

2.1 Vacuum Oscillations

In vacuum, the oscillation probability depends on:
- Mixing angles: $\theta_{12}, \theta_{13}, \theta_{23}$
- CP-violating phase: δ
- Mass-squared differences: $\Delta m^2_{21}, \Delta m^2_{31}$
- Baseline L and energy E

The algorithm computes exact probabilities using trigonometric identities without matrix diagonalization.

2.2 Matter Effects

For propagation through matter with constant density ρ and electron fraction Y_e, the Mikheyev-Smirnov-Wolfenstein (MSW) effect modifies the effective mixing parameters. NuFast uses:

1. An initial estimate from the DMP (Denton-Minakata-Parke) approximation
2. Optional Newton-Raphson iterations to improve precision

The number of Newton iterations N_{Newton} controls the trade-off between speed and precision. For most long-baseline experiments, $N_{\text{Newton}} = 0$ or 1 is sufficient.

3 Implementation

Our Rust implementation provides a simple API:

```rust
use nufast::{VacuumParameters, MatterParameters};
use nufast::{probability_vacuum_lbl, probability_matter_lbl};

// Vacuum oscillation
let params = VacuumParameters::nufit52_no(1300.0, 2.5);
let probs = probability_vacuum_lbl(&params);
println!("P(μ→e) = {}", probs.Pme);

// Matter oscillation
let params = MatterParameters::nufit52_no(1300.0, 2.5, 3.0, 0.5, 0);
let probs = probability_matter_lbl(&params);
```

The crate is published on crates.io at https://crates.io/crates/nufast.

3.1 WebAssembly Support

For web applications, we provide `nufast-wasm`, which compiles the core physics engine to WebAssembly. This enables browser-based neutrino physics tools with near-native performance. The compiled WASM module is approximately 32 KB gzipped.

4 Benchmark Methodology

All benchmarks were performed on a system with AMD Ryzen CPU running WSL2 on Windows. Each benchmark:

1. Iterates over 10 million (10^7) calculations
2. Varies energy from 0.5–5.0 GeV (DUNE-like range)
3. Uses standard oscillation parameters (NuFIT 5.2)
4. Includes a sink variable to prevent compiler optimization

The Rust benchmarks used the Criterion library with statistical analysis. C++, Fortran, and Python benchmarks used high-resolution timing.

5 Results

Language	**Vacuum**	**N=0**	**N=1**	**N=2**	**N=3**
Rust	61 ns	95 ns	106 ns	113 ns	117 ns
C++	49 ns	130 ns	143 ns	154 ns	164 ns
Fortran	51 ns	107 ns	123 ns	146 ns	167 ns
Python	14,700 ns	21,900 ns	21,200 ns	18,500 ns	16,300 ns

Table 1: Single-point oscillation probability computation times (ns per call). N refers to the number of Newton-Raphson iterations.

5.1 Key Findings

Comparison	**Vacuum**	**Matter (N=0)**	**Notes**
Rust vs C++	+24%	**−27%**	Rust faster for matter
Rust vs Fortran	+20%	−11%	Rust faster for matter
Rust vs Python	×241	×230	Compiled vs interpreted

Table 2: Relative performance (negative = Rust is faster). Rust shows a significant advantage for matter calculations.

5.1.1 Rust is Faster for Matter Calculations

The most surprising result is that Rust outperforms C++ by **27%** for matter oscillations. This is likely due to:

1. **Better loop optimization**: LLVM's optimization of the Newton iteration
2. **Stricter aliasing rules**: Rust's ownership model enables more aggressive optimization
3. **Modern code patterns**: The Rust implementation uses idiomatic patterns that optimize well

5.1.2 Vacuum Performance

For vacuum calculations, C++ and Fortran are 20% faster than Rust. This is a smaller and simpler computation where traditional numerical languages have an edge.

5.1.3 Python Performance

As expected, Python is approximately 240× slower than compiled languages. However, for interactive exploration or small-scale calculations, this overhead is acceptable.

5.2 Throughput

Language	Vacuum	Matter (N=0)
Rust	17.5 M/s	10.5 M/s
C++	20.3 M/s	7.7 M/s
Fortran	19.7 M/s	9.4 M/s
Python	0.07 M/s	0.05 M/s

Table 3: Throughput in millions of probability calculations per second.

6 Applications

6.1 Interactive Visualization

The `nufast-wasm` module powers "Imagining the Neutrino," an interactive web-based visualization of neutrino oscillations: https://planckeon.github.io/itn/

With WASM, the visualization computes 400-point energy spectra in real-time as users adjust parameters.

6.2 Integration with Analysis Frameworks

The crate can be integrated with Rust-based physics analysis pipelines, or called from Python via PyO3 bindings (future work).

7 Conclusion

We have demonstrated that Rust provides a viable and performant platform for computational neutrino physics. The `nufast` crate achieves:

- **27% speedup** over C++ for matter calculations
- **Competitive performance** for vacuum calculations
- **Memory safety** guarantees without runtime overhead
- **WebAssembly support** for browser applications
- **Published on crates.io** for easy integration

The combination of performance, safety, and modern tooling makes Rust an attractive choice for future neutrino physics software development.

Acknowledgments

This work builds on the NuFast algorithm by Peter B. Denton and Stephen J. Parke. The original implementations are available at https://github.com/PeterDenton/NuFast.

Code Availability

The `nufast` crate is open source under the MIT license:
- Crates.io: https://crates.io/crates/nufast
- GitHub: https://github.com/planckeon/nufast
- Documentation: https://docs.rs/nufast

All benchmark implementations (Rust, C++, Fortran, Python) are included in the repository under `benchmarks/`.

References

1. P. B. Denton and S. J. Parke, “Simple and precise factorization of the Jarlskog invariant for neutrino oscillations in matter,” Phys. Rev. D 100, 053004 (2019). arXiv:1902.07185
2. DUNE Collaboration, “Long-Baseline Neutrino Facility (LBNF) and Deep Underground Neutrino Experiment (DUNE) Conceptual Design Report,” arXiv:1601.05471
3. Hyper-Kamiokande Proto-Collaboration, “Hyper-Kamiokande Design Report,” arXiv:1805.04163
4. JUNO Collaboration, “Neutrino Physics with JUNO,” J. Phys. G 43, 030401 (2016). arXiv:1507.05613