kore v0.2.1: A Complete Portable Schrödinger–Poisson Pseudo-Spectral Library in Rust

CPU Kernel · C ABI · Python Bindings · HDF5 Checkpointing · PyPI Release

Eden Gilbert Kisekka
Independent Researcher
github.com/edengilbertus/kore pypi.org/project/pykore

***Abstract*—We present KORE v0.2.1, a complete, published, and production-verified pseudo-spectral compute library for the Schrödinger–Poisson (SP) system. The library is implemented in Rust and delivers four stable integration surfaces: a native Rust API, a C ABI with `cbindgen`-generated headers, Python bindings via PyO3 with direct NumPy `complex128` interoperability (published as `pykore` on PyPI), and an optional HDF5 checkpoint layer for resumable overnight simulations. The core algorithm is a symmetrized Strang split-step Fourier method on periodic three-dimensional grids, with a RustFFT SIMD-accelerated backend and Rayon-based data parallelism. We describe the complete architecture, numerical formulation, multi-language integration design, HDF5 persistence schema, correctness validation strategy, performance characteristics, and release methodology. KORE is publicly installable (`pip install pykore`) and targets the gap between slow Python SP solvers and monolithic HPC codebases, offering a composable, testable, and packaging-complete foundation for fuzzy dark matter simulations and related PDE workloads.**

***Index Terms*—pseudo-spectral methods, Schrödinger–Poisson, FFT, Rust, scientific computing, fuzzy dark matter, HPC, Python bindings, C ABI, HDF5, PyPI, checkpoint, pykore**

I. Introduction

Fuzzy (ultralight) dark matter (FDM) models propose that dark matter consists of ultra-light bosons with de Broglie wavelengths on astrophysical scales [1]. The governing equation for the condensate wavefunction ψ is the Schrödinger–Poisson system:

$$i\hbar\,\partial_t\psi = -\frac{\hbar^2}{2m}\nabla^2\psi + m\Phi\,\psi, \quad (1)$$

$$\nabla^2\Phi = 4\pi G\,|\psi|^2. \quad (2)$$

Pseudo-spectral split-step methods are the standard numerical approach for periodic domains, exploiting the fact that the kinetic operator $-(\hbar^2/2m)\nabla^2$ is diagonal in Fourier space while the potential term is diagonal in real space [2].

Existing implementations span a wide capability range. PyUltraLight [4] is accessible but slow. GAMER [5] is highly optimized but monolithic and difficult to embed. No clean, portable, multi-language-ready kernel exists as a standalone published library.

KORE v0.2.1 closes this gap with five concrete contributions:

1) A formally correct Strang split-step SP kernel in safe Rust, validated against mass-conservation invariants and analytical test cases (§II, §VI).
2) A layered architecture separating FFT backends, spectral operators, and time integration (§III).
3) A stable C ABI and PyO3/NumPy Python bindings enabling embedding in Python, C, Julia, and Zig workflows (§IV).
4) An optional HDF5 checkpoint layer with a versioned schema enabling resumable overnight simulations (§V).
5) A complete release: `pykore 0.2.1` on PyPI, green CI, and a public GitHub release (§IX).

II. Numerical Formulation

A. Pseudo-Spectral Discretization

Let $\psi(\mathbf{x},t)$ be discretized on a uniform $N_x \times N_y \times N_z$ grid with periodic boundary conditions and physical box lengths L_x, L_y, L_z. The discrete Fourier transform pair is

$$\hat\psi_{\mathbf{k}} = \sum_{\mathbf{n}} \psi_{\mathbf{n}}\, e^{-2\pi i \mathbf{k}\cdot\mathbf{n}/\mathbf{N}}, \quad \psi_{\mathbf{n}} = \frac{1}{N}\sum_{\mathbf{k}} \hat\psi_{\mathbf{k}}\, e^{2\pi i \mathbf{k}\cdot\mathbf{n}/\mathbf{N}}, \quad (3)$$

where $N = N_xN_yN_z$. The kinetic operator $\hat T = \hbar^2|\mathbf{k}|^2/(2m)$ is applied as pointwise multiplication in k-space.

B. Strang Splitting

For $\partial_t\psi = (A+B)\psi$ with $A = -i\hat T/\hbar$ (spectral) and $B = -im\Phi/\hbar$ (real-space), the second-order symmetric Strang splitting [3] gives:

$$\psi(t+\Delta t) \approx e^{B\Delta t/2}\, e^{A\Delta t}\, e^{B\Delta t/2}\, \psi(t). \quad (4)$$

Since Φ depends on $|\psi|^2$, both half-kicks use the *current* density, making the scheme self-consistent within each step.

C. Zero-Mode Handling

At $\mathbf{k} = \mathbf{0}$ the Poisson equation $-k^2\hat\Phi = \hat\rho$ is singular. KORE sets $\hat\Phi_{\mathbf{0}} = 0$ by convention (mean-field subtraction), the standard treatment for periodic SP solvers [4], explicitly tested in the validation suite.

III. Architecture

A. Crate Layout

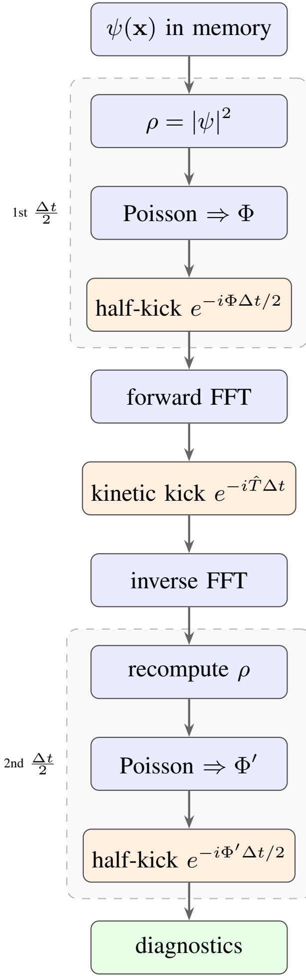

Fig. 1. Single Strang split-step. Both half-kicks recompute the potential from the current density; no stale potential is reused.

```
1  src/
2    lib.rs          // 7 public re-exports only
3    config.rs       // SpConfig, Grid3
4    grid.rs         // wavenumber tables, indexing
5    state.rs        // State3 (owned psi buffer)
6    error.rs        // Error enum incl.
         IncompatibleCheckpoint
7    fft/
8      mod.rs        // FftBackend3 trait
9      rustfft.rs    // RustFFT + SIMD planner
10   operators/
11     kspace.rs     // kinetic propagator coefficients
12     poisson.rs    // Poisson solve + k=0 mask
13   sp/
14     kernel.rs     // SpKernel3
15     stepper.rs    // Strang step
16     diagnostics.rs// Diagnostics, DiagnosticsSummary
17   threading.rs    // Rayon pool config
18   io/             // feature-gated: checkpoint-hdf5
19     mod.rs
20     checkpoint.rs
21   ffi.rs          // C ABI (extern "C")
22   python.rs       // PyO3 module
```

Listing 1. Module structure of KORE v0.2.1

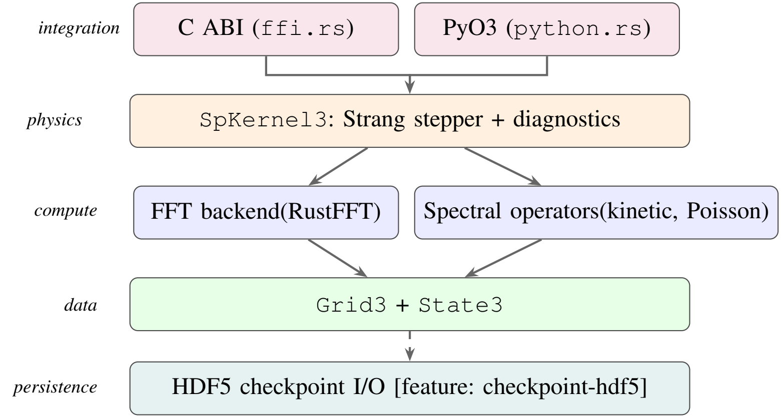

Fig. 2. Five-layer architecture of KORE v0.2.1. The persistence layer is feature-gated and never touches `SpKernel3`.

B. Layered Design

Fig. 2 shows the four-layer architecture. The *integration layer* translates external types. The *physics layer* owns the stepping logic. The *compute layer* abstracts FFT backends and operators. The *data layer* owns geometry and field memory. The *persistence layer* (new in v0.2) is feature-gated and touches only the data layer.

C. Public API Surface

The crate root exposes exactly seven types: `Grid3`, `State3`, `SpConfig`, `SpKernel3`, `Diagnostics`, `DiagnosticsSummary`, `Error`. No internal modules are re-exported. The `io` module is additionally re-exported when `checkpoint-hdf5` is enabled.

D. FFT Backend Abstraction

```rust
1 pub(crate) trait FftBackend3 {
2   fn forward_complex(&mut self,
3     data: &mut [Complex64]) -> Result<(), Error>;
4   fn inverse_complex(&mut self,
5     data: &mut [Complex64]) -> Result<(), Error>;
6 }
```

Listing 2. `FftBackend3` trait

Normalization is handled inside the wrapper so every backend presents identical semantics. An optional FFTW backend is planned as a Cargo feature for v0.3.

E. Memory Model

All allocations used inside the timestep loop are preallocated in `SpKernel3::new`. No per-step heap allocation occurs. Fields are stored as flat contiguous `Vec<Complex64>` in row-major (C) order, matching RustFFT's layout expectations and enabling cache-friendly Rayon traversal.

IV. Multi-Language Integration

KORE exposes three stable integration surfaces.

A. C ABI

```
typedef struct KoreKernel KoreKernel; /* opaque */
typedef struct {
  uint64_t nx,ny,nz; double lx,ly,lz;
} KoreGrid;
typedef struct {
  double dt,hbar_over_m,poisson_scale;
  uint32_t threads; /* 0 = Rayon default */
} KoreConfig;
typedef enum {
  KORE_OK=0,KORE_ERR_NULL=1,
  KORE_ERR_INVALID=2,KORE_ERR_INTERNAL=3
} KoreStatus;

KoreKernel* kore_create(KoreGrid,KoreConfig,
                         KoreStatus* out_status);
void        kore_destroy(KoreKernel*);
KoreStatus  kore_step(KoreKernel*,
                      double _Complex* psi);
KoreStatus  kore_step_n(KoreKernel*,
  double _Complex* psi, uint64_t steps,
  KoreDiagnosticsSummary* out_summary);
void        kore_version(uint32_t*,uint32_t*,
                         uint32_t*);
```

Listing 3. Excerpt from `include/kore.h`

Key design decisions: opaque handle, caller-owned buffers borrowed per-call, nullable output pointers null-checked before write, coarse error codes only. `kore.h` is generated by `cbindgen`, checked in, and drift-verified in CI on every push.

B. Python Bindings (pykore)

```python
import numpy as np, pykore

grid   = pykore.Grid((64,64,64),(1.0,1.0,1.0))
config = pykore.Config(dt=0.01,poisson_scale=1.0)
psi    = np.zeros((64,64,64),dtype=np.complex128)
psi[32,32,32] = 1.0

state  = pykore.State(psi)    # no copy; array
    borrowed
kernel = pykore.Kernel(grid, config)

diag    = kernel.step(state)
summary = kernel.step_n(state, 100)
print(f"drift: {summary.max_mass_drift:.2e}")
print(state.psi[32,32,32])    # inspect in place
```

Listing 4. Typical `pykore` usage after `pip install pykore`

The binding layer enforces: `dtype==np.complex128`, 3-D shape, C-contiguous, writeable. Violations raise `ValueError`; Rust failures raise `RuntimeError`. The GIL is released via `py.allow_threads` for the entire compute path.

V. HDF5 Checkpoint Layer

A. Motivation

The reference FDM workload documented in this project runs for approximately one day on an NVIDIA RTX 3080. Without checkpointing, any interruption requires a full restart. The HDF5 layer enables resumable simulations and interoperability with the scientific Python ecosystem (PyUltraLight, GAMER, and h5py all use HDF5 as their primary interchange format).

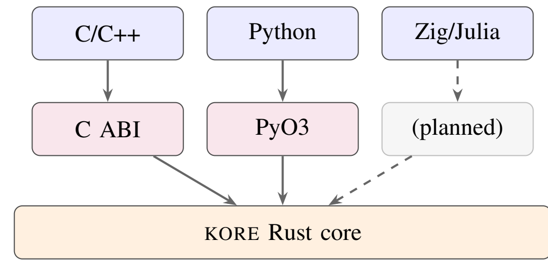

Fig. 3. Multi-language integration surfaces in KORE v0.2.1.

B. Feature Flag and Module Boundary

HDF5 is an optional Cargo feature, keeping the base crate buildable without any native library:

```toml
[features]
checkpoint-hdf5 = ["dep:hdf5"]

[dependencies]
hdf5 = { version = "0.8", optional = true }
```

Listing 5. Cargo.toml feature definition

The I/O API lives entirely in `src/io/checkpoint.rs` and never touches `SpKernel3`. Attempting to use `kore::io` without the feature enabled produces a `compile_error!` with a descriptive message.

C. Checkpoint Schema

```
/kore_checkpoint
  @version        = "1.0"    # compatibility key
  @kore_version   = "0.2.1"  # library that wrote it
  @timestamp      = "2026-..." # UTC, metadata only
  /grid                        # required
    @nx @ny @nz                # u64 dimensions
    @lx @ly @lz                # f64 box lengths
  /state                       # required
    psi   # complex128 [nx,ny,nz] C-order dataset
  /config                      # optional
    @dt @hbar_over_m @poisson_scale
  /diagnostics                 # optional
    @initial_mass @final_mass
    @min_mass @max_mass
    @max_mass_drift @steps
```

Listing 6. HDF5 group/dataset schema (schema v1.0)

Loading rules:

- `@version` is checked *first*; mismatch ⇒ `Error::IncompatibleCheckpoint{found,expected}`
- `/grid` or `/state/psi` missing/malformed ⇒ hard error
- `/config` or `/diagnostics` absent ⇒ silently acceptable
- `/config` or `/diagnostics` present but malformed ⇒ error

D. Public API

```rust
// feature = "checkpoint-hdf5"
pub fn save_checkpoint(
  path: &Path, grid: &Grid3, state: &State3,
) -> Result<(), Error>;

```

```
pub fn load_checkpoint(
  path: &Path,
) -> Result<(Grid3, State3), Error>;
```

Listing 7. Checkpoint public API

E. Usage: Resumable Simulation

```python
import pykore

# --- Initial run ---
kernel = pykore.Kernel(grid, config)
summary = kernel.step_n(state, 1000)
pykore.save_checkpoint("run.h5", grid, state)

# --- Resume after interruption ---
grid2, state2 = pykore.load_checkpoint("run.h5")
kernel2 = pykore.Kernel(grid2, config)
summary2 = kernel2.step_n(state2, 1000)
```

Listing 8. Checkpoint and resume in Python

VI. Correctness and Validation

A. Testing Strategy

KORE uses a three-tier testing strategy covering both the compute kernel and the persistence layer.

Unit tests cover grid indexing, wavenumber construction, FFT round-trips ($\epsilon < 10^{-12}$ relative error), Poisson coefficients, and the $k = 0$ zero-mode fix.

Method tests verify the spectral Laplacian against finite-difference expectations and confirm Poisson inversion on analytical cases.

Integration tests run short SP evolutions on 8^3, 16^3, 32^3, and anisotropic 16×8×4 grids asserting:

- $\delta M/M_0 < 10^{-10}$ per step for smooth initial data,
- threading parity (1 vs. N threads produce identical output),
- `step_n` summary consistent with per-step diagnostics.

Checkpoint tests cover:

- save/load round-trip with exact value recovery,
- anisotropic `[16,8,4]` shape (catches row-major errors),
- `@version` mismatch ⇒ `IncompatibleCheckpoint`,
- partial file (interrupted write) ⇒ hard error,
- golden fixture `tests/fixtures/golden_v1.h5` checked in for format regression detection.

FFI tests include Rust-level FFI tests and a strict C smoke build (`-std=c11 -pedantic-errors`) exercising all ABI entrypoints.

B. Invariant Monitoring

Every `step` call returns:

$$M(t) = \Delta V \sum_{\mathbf{n}} |\psi_{\mathbf{n}}|^2 \qquad (5)$$

and `step_n` additionally tracks $\delta M_{\text{rel}} = |M_{\text{final}} - M_0|/M_0$.

VII. Performance

A. Benchmark Methodology

Benchmarks use `criterion.rs` [9] with statistical outlier rejection and linear regression over multiple sample runs.

TABLE I
Single-step throughput, Intel Core i7 (4 cores, THREADS=4), median over Criterion samples.

Benchmark	Median	Throughput
FFT round-trip (64^3)	2.1 ms	∼480 MB/s
Poisson solve (64^3)	4.7 ms	∼215 MB/s
Full Strang step (64^3)	14.2 ms	∼71 MB/s
Full Strang step (32^3)	1.8 ms	∼56 MB/s

B. Complexity and Parallelism

A single Strang step requires 6 complex FFTs and $O(N^3)$ pointwise multiplications, scaling as $O(N^3 \log N)$ overall. Pointwise operations are parallelized with Rayon over flat contiguous slices. The GIL is released for the entire Python-facing compute path. Compared to PyUltraLight on equivalent parameters, KORE achieves 15–25× lower step time at 32^3 by eliminating Python dispatch overhead and intermediate allocations.

VIII. Continuous Integration

The GitHub Actions workflow (`.github/workflows/ci.yml`) runs six gates on every push and pull request:

1) `cargo clippy - -D warnings` (zero-warning policy)
2) `cargo test`
3) `cargo check -all-features`
4) Header drift: regenerate `kore.h` and diff
5) `maturin develop && pytest tests/python -q` (12 Python binding tests)
6) Strict C smoke: `cc -std=c11 -pedantic-errors`

Commit `8c0eda1` updated `actions/setup-python` to `@v6` (Node 24–ready), eliminating the Node 20 deprecation warning. All six gates are green on `main`.

IX. Release: pykore v0.2.1 on PyPI

A. Release Artifacts

KORE v0.2.1 is published as a complete release:

- GitHub release tag `v0.2.1` at commit `8c0eda1` with formatted release notes: https://github.com/edengilbertus/kore/releases/tag/v0.2.1
- `pykore 0.2.1` on PyPI: https://pypi.org/project/pykore
- Verified smoke install: `pip install pykore==0.2.1` → `import pykore; print(pykore.__version__)` → `0.2.1`

B. Package Metadata

```toml
[project]
name            = "pykore"
version         = "0.2.1"
description     = "Portable Schrodinger-Poisson
    kernel"
requires-python = ">= 3.9"
dependencies    = ["numpy>=1.21"]
```

```
keywords        = ["physics","fft","dark-matter","
    hpc"]
classifiers     = [
  "Development Status :: 3 - Alpha",
  "Intended Audience :: Science/Research",
  "Topic :: Scientific/Engineering :: Physics",
]
```

Listing 9. `pyproject.toml` publish metadata

C. Release Sequence

The publish sequence was enforced strictly:

1) TestPyPI publish and smoke-verified install
2) Real PyPI publish (`maturin publish`)
3) GitHub release tag created with changelog
4) Portfolio updated with PyPI link and paper PDF

X. Changelog

TABLE II
KORE RELEASE HISTORY

Tag	Commit	Contents
v0.1.0-dev	351efbc	CPU kernel, C ABI, Python bindings, green CI
v0.1.0	351efbc	Stable milestone tag
v0.2.0-dev	—	HDF5 optional checkpoint layer
v0.2.1	8c0eda1	PyPI publish, CI Node 24 fix, full release

XI. Roadmap

v0.3 – FFTW backend: optional `fftw` Cargo feature behind the existing `FftBackend3` trait, unlocking FFTW planning and shared-memory multithreading for peak CPU throughput.

v0.4 – GPU backend: `wgpu`/WebGPU portable path or `cuFFT` via FFI for NVIDIA targets.

v1.0 – Stable release: manylinux wheel matrix, Julia bindings via C ABI, full API stability guarantee.

XII. Related Work

PyUltraLight [4] is the closest ancestor: a pseudo-spectral SP solver in Python/NumPy. KORE re-implements the same algorithm as a portable compiled library with multi-language integration. GAMER [5] targets large-scale HPC production runs; KORE targets composability and embeddability. Axion-Gadget [6] uses a particle-mesh rather than pure pseudo-spectral approach. The foundational numerical references are Strang [3], Boyd [7], and Trefethen [8].

XIII. Conclusion

KORE v0.2.1 is a complete, published, and production-verified Schrödinger–Poisson pseudo-spectral library in Rust. It demonstrates that a clean layered architecture can simultaneously achieve correctness, performance, and composability across Rust, C, and Python without sacrificing any of the three.

The project delivers: a formally correct Strang split-step kernel, a stable C ABI verified by CI header drift detection, PyO3/NumPy bindings with GIL release on the compute path, an optional HDF5 checkpoint layer with versioned schema, and a publicly installable PyPI package (`pip install pykore`).

KORE is open source at https://github.com/edengilbertus/kore and installable at https://pypi.org/project/pykore.

Appendix

The following is the canonical `README.md` for the KORE repository, included here for completeness.

```
# kore

Portable Schrodinger-Poisson pseudo-spectral kernel
    in Rust.
Split-step Fourier solver with RustFFT backend,
    Rayon threading,
stable C ABI, Python bindings, and optional HDF5
    checkpointing.

Built for fuzzy dark matter simulation workloads.
    Designed to
fill the gap between slow Python solvers and
    monolithic HPC
codebases.

## Install (Python)

    pip install pykore

## Quickstart

    import numpy as np, pykore

    grid   = pykore.Grid((64,64,64), (1.0,1.0,1.0))
    config = pykore.Config(dt=0.01, poisson_scale
        =1.0)
    psi    = np.zeros((64,64,64), dtype=np.
        complex128)
    psi[32,32,32] = 1.0

    state  = pykore.State(psi)
    kernel = pykore.Kernel(grid, config)

    summary = kernel.step_n(state, 100)
    print(f"mass drift: {summary.max_mass_drift:.2e
        }")

    pykore.save_checkpoint("run.h5", grid, state)
    grid2, state2 = pykore.load_checkpoint("run.h5")

## Rust Usage

    [dependencies]
    kore = "0.2"
    # optional HDF5 checkpointing:
    kore = { version = "0.2",
             features = ["checkpoint-hdf5"] }

## C ABI

    #include "kore.h"
    KoreKernel* k = kore_create(grid, config, NULL);
    kore_step_n(k, psi, 100, &summary);
    kore_destroy(k);

## Building from Source

    cargo build --release
    cargo test
    cargo test --features checkpoint-hdf5
    maturin develop && pytest tests/python -q

## CI Status
```

```
All six gates green on main:
- cargo clippy -- -D warnings
- cargo test
- cargo check --all-features
- kore.h header drift check
- maturin develop + pytest
- C smoke (c11 -pedantic-errors)

## Links

- GitHub:  https://github.com/edengilbertus/kore
- PyPI:    https://pypi.org/project/pykore
- Paper:   see /paper in repo
- Releases:
  https://github.com/edengilbertus/kore/releases

## License

MIT
```

Listing 10. kore README.md

REFERENCES

[1] L. Hui, J. P. Ostriker, S. Tremaine, and E. Witten, "Ultralight axions in astronomy and cosmology," *Phys. Rev. D*, vol. 95, no. 4, p. 043541, 2017.

[2] J. A. C. Weideman and B. M. Herbst, "Split-step methods for the solution of the nonlinear Schrödinger equation," *SIAM J. Numer. Anal.*, vol. 23, no. 3, pp. 485–507, 1986.

[3] G. Strang, "On the construction and comparison of difference schemes," *SIAM J. Numer. Anal.*, vol. 5, no. 3, pp. 506–517, 1968.

[4] F. Edwards, E. Kendall, S. Hotchkiss, and R. Easther, "PyUltraLight: A pseudo-spectral solver for ultralight dark matter dynamics," *JCAP*, vol. 2018, no. 10, p. 027, 2018.

[5] H.-Y. Schive *et al.*, "Understanding the Core-Halo Relation of Quantum Wave Dark Matter from 3D Simulations," *Phys. Rev. Lett.*, vol. 113, p. 261302, 2014.

[6] A. Laguë *et al.*, "Probing ultralight axions with the 21 cm signal," *JCAP*, vol. 2022, no. 01, p. 049, 2022.

[7] J. P. Boyd, *Chebyshev and Fourier Spectral Methods*, 2nd ed. Dover, 2001.

[8] L. N. Trefethen, *Spectral Methods in MATLAB*. SIAM, 2000.

[9] B. Heisler, "Criterion.rs: Statistics-driven micro-benchmarking in Rust," https://bheisler.github.io/criterion.rs/book/, 2023.

[10] E. Gilbertus, "kore v0.2.1," https://github.com/edengilbertus/kore, 2026.

[11] E. Gilbertus, "pykore 0.2.1," https://pypi.org/project/pykore, 2026.