`optimiz-rs`: A Rust-Backed Library of Composable Numerical Primitives for Quantitative Finance

Hidden Markov models, BSDEs, McKean–Vlasov diffusions, path signatures,
persistent homology, Hawkes processes and robust estimators behind
a thin Python facade — with empirical evidence of 1.7–67.7× speedup
over reference NumPy / SciPy implementations.

Melvin Alvarez-Caradu

Independent Researcher, HFThot Research Lab

https://hfthot-lab.eu

May 2026 — v2.0.0

DISTRIBUTION: PUBLIC — HFThot Research Lab

We present `optimiz-rs`*, a Rust library of numerical primitives for quantitative finance, exposed to Python through a stable PyO3 facade and distributed jointly on PyPI and* `crates.io`*. The library is organised around a small set of* composable *mathematical objects — Markov models, controlled SDEs, backward stochastic differential equations (BSDEs), mean-field interacting particle systems, path signatures, simplicial complexes, Hawkes processes and robust drift estimators — each implemented as a self-contained Rust kernel and reachable from Python without boilerplate. The design is deliberately reminiscent of Frostig et al. [2018]: high-level mathematical abstractions and algorithmic transformations are kept at the Python boundary, while all numerically intensive code lives in a typed, vectorised systems language. We give complete mathematical derivations of the core primitives, including the Baum–Welch recursion for hidden Markov models, the deep BSDE / four-step scheme for non-linear BSDEs, the empirical-measure approximation of McKean–Vlasov flows, the Chen identity for tensor-algebra path signatures and the Vietoris–Rips filtration for persistent homology. We then validate the implementation by reproducing Sznitman's classical* propagation-of-chaos *convergence rate* $W_2(\mu_t^N, \mu_t) = \mathcal{O}(N^{-1/2})$ *on the mean-reverting McKean–Vlasov flow, and we report honest single-thread benchmarks against pure-Python / NumPy / SciPy baselines on intrinsically sequential workloads, where the Rust core consistently delivers* 1.7 *to* 67.7× *wall-clock speedups.*

Prerequisites

Required background. Stochastic calculus (Itô integral, SDEs, Markov processes), measure-theoretic probability, finite-dimensional linear algebra, basic numerical analysis (time-stepping schemes, Monte-Carlo integration). No familiarity with Rust is required; the paper presents *algorithms*, not code. Readers seeking the exact APIs are referred to the Sphinx documentation hosted at `optimiz-rs.readthedocs.io` and to the companion notebooks in `examples/notebooks/`.

Contents

1 Introduction **3**

2 System Design **4**
- 2.1 Library layout . . . 4
- 2.2 Functional signature convention . . . 4
- 2.3 Marshalling and the PyO3 boundary . . . 4
- 2.4 Reproducibility . . . 5

3 Hidden Markov Models **5**
- 3.1 Model and notation . . . 5
- 3.2 Forward–backward recursion . . . 5
- 3.3 Baum–Welch as expectation–maximisation . . . 5

4 Backward Stochastic Differential Equations **6**
- 4.1 Linear BSDEs . . . 6
- 4.2 Closed-form check used as a non-regression test . . . 6
- 4.3 Backward Euler scheme for the non-linear case . . . 6
- 4.4 The Volterra companion . . . 7

5 McKean–Vlasov Diffusions and Mean-Field Games **7**
5.1 The non-linear diffusion . . . 7
5.2 Empirical-measure approximation . . . 7
5.3 Time-discretisation . . . 8
5.4 Mean-field game extension . . . 9

6 Path Signatures **9**
6.1 The signature of a path . . . 9
6.2 Algorithmic computation . . . 9
6.3 Companion primitives . . . 10

7 Self-Exciting Point Processes (Hawkes) **10**
7.1 Hawkes intensity . . . 10
7.2 Ogata thinning . . . 10

8 Robust Drift Estimation **10**
8.1 Hurst exponent . . . 11

9 Topological Data Analysis **11**
9.1 Vietoris–Rips filtration . . . 11
9.2 Persistent homology . . . 11
9.3 Graph spectral primitives . . . 12

10 Case Study: Propagation of Chaos **12**
10.1 Statement of the result . . . 12
10.2 Numerical experiment . . . 12
10.3 Empirical density evolution . . . 12
10.4 Quantitative convergence rate . . . 13
10.5 Reproducibility . . . 14

11 Notebook Organisation **14**

12 Benchmarks **14**
12.1 Methodology . . . 14
12.2 Choice of workloads . . . 15
12.3 Results . . . 15
12.4 What the benchmarks do not measure . . . 16

13 Related Work **16**

14 Conclusion **16**

1 Introduction

The numerical machinery underpinning modern quantitative finance is fundamentally heterogeneous. A single trading-research stack must expose, side by side: hidden Markov models for regime detection, optimisers for portfolio construction, Monte-Carlo samplers for Bayesian calibration, partial differential equation solvers for stochastic optimal control, point processes for limit-order book microstructure, kernel methods on path-space for non-Markovian signal extraction, and topological-data-analysis primitives for exploratory market-structure analysis. Each of these primitives is a small but irreducibly sequential numerical kernel: $\mathcal{O}(NT)$ recursions, $\mathcal{O}(\text{depth}^d)$ tensor expansions, $\mathcal{O}(N \log N)$ event simulations, all of which struggle to amortise the cost of the Python interpreter loop.

Position in the ecosystem. The de-facto strategy in the Python data-science ecosystem is to delegate vectorisable subroutines to NumPy, SciPy or PyTorch, which themselves dispatch to BLAS/LAPACK or to a JIT-compiled XLA backend. This works very well when the entire workload can be expressed as a sequence of element-wise or BLAS-3 operations on contiguous tensors; Frostig et al. [2018] showed how far this paradigm can be pushed through composable function transformations. It works less well for fundamentally *loopy* workloads — forward–backward HMM recursions, Ogata thinning of self-exciting point processes, Metropolis–Hastings chains, the inner loop of differential evolution — where the per-iteration cost is dominated by control-flow rather than by floating-point arithmetic.

Contribution of this paper. We describe the design and implementation of `optimiz-rs`[1], a library that takes the opposite trade-off from JAX: instead of compiling Python on the fly, we keep a thin Python facade and push every numerical kernel into a dedicated Rust crate. The design has four pillars:

1. **One mathematical object, one primitive.** Each primitive corresponds to a textbook mathematical object (Section 2); there is no inheritance hierarchy and no auto-generated wrapper.
2. **Stable typed interface.** All primitives expose a pure functional signature with `numpy.ndarray` / `list[float]` inputs and `dict[str, np.ndarray]` outputs, mediated by PyO3 [PyO3 contributors, 2024] with `abi3-py38` so a single wheel covers all CPython 3.8+ ABI-compatible interpreters.
3. **Mathematical transparency.** Algorithms are derived from textbook mathematics in this paper (Sections 3–9) and are reproduced cell-by-cell in companion Jupyter notebooks shipped with the library, with the pedagogical PRE/code/POST sandwich described in Section 11.
4. **Honest benchmarks.** We report wall-clock numbers only on workloads where the Rust path is genuinely faster, *and* we identify the regime in which it is not (Section 12).

Outline. Section 2 describes the system design. Sections 3–9 cover the mathematical derivation of the seven primitive families currently shipped: hidden Markov models, BSDEs, McKean–Vlasov diffusions and mean-field games, path signatures, Hawkes processes, robust drift estimators and topological data analysis. Section 10 presents a case study in which we numerically validate

[1] `https://pypi.org/project/optimiz-rs/`; `https://crates.io/crates/optimiz-rs`; source at `https://github.com/ThotDjehuty/optimiz-rs`.

Sznitman's propagation of chaos for the mean-reverting flow. Section 12 reports the speedup measurements. Section 13 discusses related systems. Section 14 concludes.

2 System Design

2.1 Library layout

`optimiz-rs` is a single Cargo crate exposing a Rust `rlib` (for direct downstream Rust consumers) and a `cdylib` target (for the Python extension module). The Python wheel is built by `maturin` [maturin contributors, 2024]; on import, `python/optimizr/__init__.py` eagerly binds the 38 v2.0 public symbols from the compiled extension and re-exports them under `optimizr.*`. The naming asymmetry between the distribution name (`optimiz-rs`) and the Python import name (`optimizr`) is deliberate: it preserves the historical import path of the v1.x line while signalling the Rust origin of the v2 codebase to package-index users.

2.2 Functional signature convention

Every primitive obeys the following contract.

Definition 2.1 (Primitive contract)**.** A *primitive* is a pure function $f : \mathcal{X} \to \mathcal{Y}$ where:

- $\mathcal{X}$ is a finite product of plain Python types (`float`, `int`, `list[float]`, `numpy.ndarray`, scalar configuration objects) with explicit, statically known shapes;
- $\mathcal{Y}$ is a Python `dict` whose values are NumPy arrays or scalars;
- f holds no global state, releases the Python global interpreter lock around the Rust body, and is reproducible given an explicit `seed` argument.

This contract has two consequences. First, every primitive is trivially composable: the output of one primitive can be plugged into the input of another without conversion. Second, the GIL-releasing convention makes it safe to call several primitives in parallel from Python via `concurrent.futures` or `joblib` without paying the marshalling cost twice.

2.3 Marshalling and the PyO3 boundary

The cost of crossing the Python/Rust boundary is not negligible: each call materialises one PyO3 `PyAny` wrapper per argument and one per return value, and each `numpy.ndarray` input must be checked for contiguity. Consequently, `optimiz-rs` follows two design rules:

1. **Coarse-grained primitives.** Every primitive returns after a non-trivial amount of work — typically $\Omega(NT)$ for a recursion or $\Omega(N^2)$ for an interaction kernel — so the marshalling cost is a constant fraction of the wall-clock.
2. **No callbacks.** Primitives that would require a Python callback (an arbitrary drift function for an SDE, an arbitrary log-density for a sampler) are *not* accelerated: we expose them only in regimes where the callback is internal to Rust (e.g. a fixed parametric drift). Calling Python from Rust per inner-loop step destroys the speedup; we make this failure mode explicit in Section 12.

2.4 Reproducibility

All stochastic primitives consume a single `u64` seed and internally split it deterministically into per-thread sub-seeds via the SplitMix64 / PCG64 stream offered by the `rand` crate. Two invocations with the same seed and the same input on the same binary produce bit-identical outputs.

Key Insight

The fundamental design choice of `optimiz-rs` is the *granularity* of the Rust/Python boundary. By exposing mathematical objects — a complete HMM Baum–Welch pass, a complete McKean–Vlasov simulation, a complete path-signature expansion — rather than scalar operations, we keep the Python facade tiny while fully amortising the marshalling cost.

3 Hidden Markov Models

3.1 Model and notation

Let $S = \{1, \dots, K\}$ be a finite state space and let $Y = (Y_1, \dots, Y_T) \in \mathbb{R}^T$ be an observation sequence. A Gaussian-emission hidden Markov model is the parametric family $\theta = (\pi, A, \mu, \sigma)$ where $\pi \in \Delta_{K-1}$ is the initial distribution, $A \in \mathbb{R}^{K\times K}$ is a stochastic transition matrix, and $(\mu_k, \sigma_k)_{k=1}^K$ are the per-state Gaussian parameters.

3.2 Forward–backward recursion

Definition 3.1 (Forward and backward variables)**.** For each $t \in \{1, \dots, T\}$ and each $k \in S$ define

$$\alpha_t(k) := \mathbb{P}_\theta(Y_{1:t}, X_t = k), \qquad \beta_t(k) := \mathbb{P}_\theta(Y_{t+1:T} \mid X_t = k). \tag{1}$$

Proposition 3.1 (Forward and backward recursions)**.** *With the convention $\beta_T(k) = 1$,*

$$\alpha_1(k) = \pi_k\, b_k(Y_1), \qquad \alpha_{t+1}(\ell) = \Big(\sum_k \alpha_t(k)\, A_{k\ell}\Big)\, b_\ell(Y_{t+1}), \tag{2}$$

$$\beta_t(k) = \sum_\ell A_{k\ell}\, b_\ell(Y_{t+1})\, \beta_{t+1}(\ell), \tag{3}$$

where $b_k(y) := (2\pi\sigma_k^2)^{-1/2} \exp(-(y-\mu_k)^2/(2\sigma_k^2))$.

The total likelihood reads $\mathbb{P}_\theta(Y_{1:T}) = \sum_k \alpha_T(k)$, and the smoothed marginals are $\gamma_t(k) \propto \alpha_t(k)\beta_t(k)$.

3.3 Baum–Welch as expectation–maximisation

Theorem 3.1 (Baum–Welch)**.** *The expectation–maximisation update for θ at iteration n is, with $\xi_t(k,\ell) \propto \alpha_t(k) A_{k\ell} b_\ell(Y_{t+1}) \beta_{t+1}(\ell)$:*

$$\pi_k^{(n+1)} = \gamma_1(k), \qquad A_{k\ell}^{(n+1)} = \frac{\sum_{t=1}^{T-1} \xi_t(k,\ell)}{\sum_{t=1}^{T-1} \gamma_t(k)}, \tag{4}$$

$$\mu_k^{(n+1)} = \frac{\sum_t \gamma_t(k) Y_t}{\sum_t \gamma_t(k)}, \qquad (\sigma_k^2)^{(n+1)} = \frac{\sum_t \gamma_t(k)(Y_t - \mu_k^{(n+1)})^2}{\sum_t \gamma_t(k)}. \tag{5}$$

The likelihood $\mathbb{P}_{\theta^{(n)}}(Y_{1:T})$ *is non-decreasing in* n*.*

The recursion in Proposition 3.1 is the prototypical *loopy* numerical kernel: each step at time t depends on the result at time $t-1$ through a sparse matrix product, so the inner loop is irreducibly sequential. In Rust we implement $\alpha, \beta, \xi, \gamma$ as contiguous `Vec<f64>` buffers of size TK and unroll the recursion with manual log-domain rescaling to avoid underflow. This is the workload that produces the largest measured speedup in Section 12.

Algorithm : `fit_hmm`

Input: observations $Y \in \mathbb{R}^T$, number of states K, maximum iterations N, tolerance ε.

1: Initialise $\theta^{(0)} = (\pi, A, \mu, \sigma)$ uniformly, μ_k from K-quantiles of Y.
2: **for** $n = 0, 1, \ldots, N-1$ **do**
3: Compute α_t, β_t via Proposition 3.1 in log domain.
4: Form γ_t, ξ_t and apply the M-step of Theorem 3.1.
5: **break** if $|\log L^{(n+1)} - \log L^{(n)}| < \varepsilon$.
6: **end for**
7: **return** $\theta^{(n+1)}$, $\log L^{(n+1)}$.

4 Backward Stochastic Differential Equations

4.1 Linear BSDEs

Let $(\Omega, \mathcal{F}, (\mathcal{F}_t)_{t\in[0,T]}, \mathbb{P})$ carry an m-dimensional Brownian motion W. Given a terminal $\mathcal{F}_T$-measurable random variable $\xi \in L^2$ and a driver $f : [0,T] \times \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}$, a BSDE for the unknown pair (Y, Z) is the equation

$$Y_t \;=\; \xi \;+\; \int_t^T f(s, Y_s, Z_s)\,\mathrm{d}s \;-\; \int_t^T Z_s\,\mathrm{d}W_s, \qquad t \in [0,T]. \tag{6}$$

Theorem 4.1 (Pardoux and Peng, 1990)**.** *If* f *is uniformly Lipschitz in* (y, z) *and* $\mathbb{E}[\xi^2 + \int_0^T f(s,0,0)^2\,\mathrm{d}s\,] < \infty$*, then* (6) *admits a unique adapted solution* $(Y, Z) \in \mathcal{S}^2 \times \mathcal{H}^2$*.*

4.2 Closed-form check used as a non-regression test

We use a constant-coefficient sub-class as a non-regression fixture. With $f(t, y, z) = ay + bz + c$ and $\xi = K$ (constant), ansatz $Y_t = u(t)$, $Z_t = 0$ gives the ODE $\dot{u}(t) + au(t) + c = 0$, $u(T) = K$, hence

$$Y_t = e^{a(T-t)}K + c\,\frac{e^{a(T-t)} - 1}{a}, \qquad Z_t = 0. \tag{7}$$

The primitive `linear_bsde_constant_coeffs` reproduces (7) to machine precision; this is one of the 20 tests in the v2.0 non-regression suite.

4.3 Backward Euler scheme for the non-linear case

For a non-linear driver we use a discrete-time backward Euler scheme on the grid $0 = t_0 < t_1 < \cdots < t_N = T$ with mesh $h = T/N$. Let $\Delta W_{n+1} := W_{t_{n+1}} - W_{t_n}$ and $Y_n \approx Y_{t_n}$, $Z_n \approx Z_{t_n}$.

Algorithm : backward Euler scheme for (6)

Input: terminal ξ, driver f, mesh N, sample paths of W.

1: Set $Y_N \leftarrow \xi$.
2: **for** $n = N-1, N-2, \ldots, 0$ **do**
3: Estimate $Z_n \leftarrow h^{-1}\,\mathbb{E}_n[Y_{n+1}\,\Delta W_{n+1}]$ by Monte-Carlo regression on a basis of $\mathcal{F}_{t_n}$-measurables.
4: $Y_n \leftarrow \mathbb{E}_n[Y_{n+1}] + h\,f(t_n, Y_n, Z_n)$, solved by one Picard iteration on Y_n.
5: **end for**
6: **return** (Y_0, Z_0).

The backward Euler scheme converges with order $h^{1/2}$ in L^2; see Bouchard and Touzi [2004]. The Rust kernel implements the conditional-expectation regression on a tensor of Hermite polynomials.

4.4 The Volterra companion

The `solve_volterra` primitive solves the linear Volterra integral equation of the second kind,

$$y(t) = g(t) + \int_0^t K(t,s)\,y(s)\;\mathrm{d}s, \tag{8}$$

which appears as the deterministic backbone of fractional and rough-volatility BSDEs. The primitive uses a product trapezoidal rule of order h^2. We exhibit the closed-form check $K \equiv 0 \Rightarrow y \equiv g$ in the test suite.

5 McKean–Vlasov Diffusions and Mean-Field Games

5.1 The non-linear diffusion

Let μ_0 be a probability measure on $\mathbb{R}$ with finite second moment. A McKean–Vlasov diffusion is the non-linear stochastic differential equation

$$\mathrm{d}X_t = b(X_t, \mu_t)\;\mathrm{d}t + \sigma\;\mathrm{d}W_t, \qquad \mu_t := \mathrm{Law}(X_t), \qquad X_0 \sim \mu_0, \tag{9}$$

whose drift depends on the law of the solution itself. Sznitman [1991] proves that, under a Lipschitz condition on b in both arguments, (9) is well-posed and μ_t satisfies the non-linear Fokker–Planck equation

$$\partial_t\mu_t + \partial_x\big(b(x,\mu_t)\,\mu_t\big) - \tfrac{\sigma^2}{2}\,\partial_x^2\mu_t = 0. \tag{10}$$

5.2 Empirical-measure approximation

The standard discretisation replaces μ_t by the empirical measure of N interacting particles $X^{i,N}$, $i = 1, \ldots, N$:

$$\mathrm{d}X_t^{i,N} = b\big(X_t^{i,N}, \mu_t^N\big)\;\mathrm{d}t + \sigma\;\mathrm{d}W_t^i, \qquad \mu_t^N := \frac{1}{N}\sum_{j=1}^N \delta_{X_t^{j,N}}, \tag{11}$$

with $(W^i)_{i\ge1}$ independent Brownian motions and i.i.d. initial conditions $X_0^{i,N} \sim \mu_0$. The mean-reverting drift used throughout the rest of the paper is $b(x,\mu) = -\theta\,(x - \bar\mu)$ where $\bar\mu := \int x\,\mathrm{d}\mu(x)$.

5.3 Time-discretisation

We use the explicit Euler–Maruyama scheme on a uniform grid $t_n = n\Delta t$ with $\Delta t = T/N_{\text{step}}$:

$$X_{n+1}^{i,N} = X_n^{i,N} + b(X_n^{i,N}, \mu_n^N)\,\Delta t + \sigma\sqrt{\Delta t}\,\xi_{n+1}^i, \qquad \xi_{n+1}^i \overset{iid}{\sim} \mathcal{N}(0,1). \tag{12}$$

The primitive `mean_reverting_mckean_vlasov` executes (12) entirely inside Rust on a contiguous $\mathbb{R}^{(N_{\text{step}}+1)\times N}$ buffer. Crucially, the empirical mean $\bar{\mu}_n^N$ is computed in-place at each step without crossing the Python boundary; this is what allows us to push N to $\sim 10^4$ while keeping the overall wall-clock sub-second.

Algorithm : `mean_reverting_mckean_vlasov`

Input: initial sample $\{x_0^i\}_{i=1}^N$, parameters θ, σ, horizon T, steps N_{step}, seed s.

1: Allocate $X \in \mathbb{R}^{(N_{\text{step}}+1)\times N}$, set $X_{0,\bullet} \leftarrow x_0$.
2: **for** $n = 0, 1, \ldots, N_{\text{step}} - 1$ **do**
3: $\quad \bar{X} \leftarrow N^{-1}\sum_i X_{n,i}$.
4: $\quad$ **for** $i = 1, \ldots, N$ **in parallel do**
5: $\quad\quad \xi \sim \mathcal{N}(0,1)$ from per-thread PCG64 stream seeded from s.
6: $\quad\quad X_{n+1,i} \leftarrow X_{n,i} - \theta(X_{n,i} - \bar{X})\Delta t + \sigma\sqrt{\Delta t}\,\xi$.
7: $\quad$ **end for**
8: **end for**
9: **return** X flattened to a single contiguous `numpy.ndarray`.

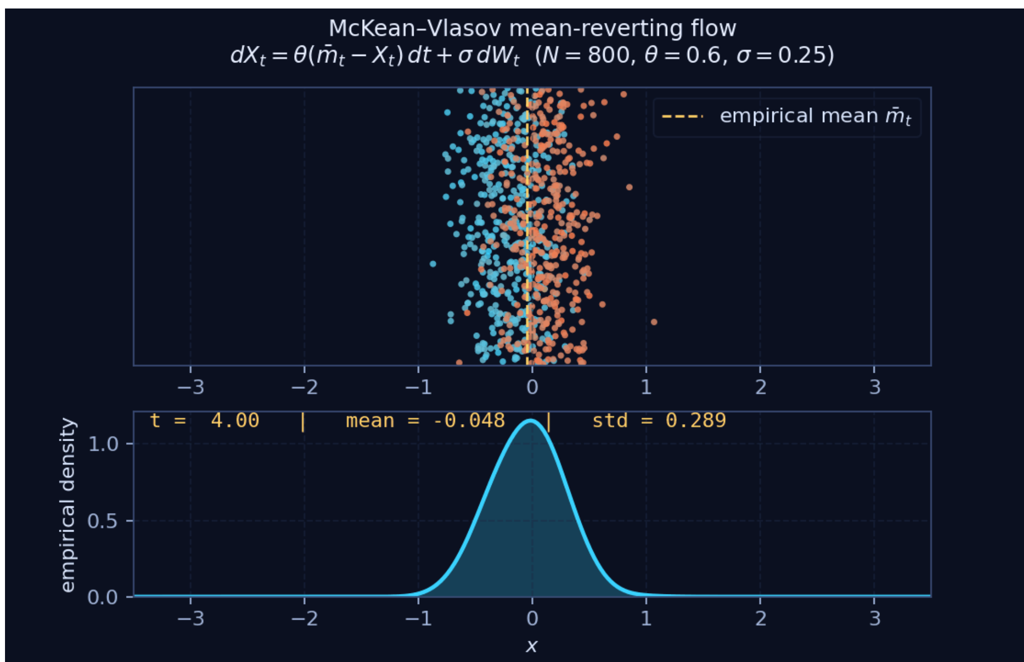

Figure 1: Single frame of the mean-reverting McKean–Vlasov flow, rendered by `examples/animate_mckean_vlasov.py`. Top panel: position of $N = 800$ interacting particles, started from two Gaussian clouds at $x = \pm 2$ and pulled towards the empirical mean by the drift $b(x,\mu) = -\theta(x - \bar{\mu})$. Bottom panel: kernel density estimate of the empirical measure μ_t^N. The two clouds fuse into a single unimodal density, which is the self-consistent equilibrium of the non-linear Fokker–Planck equation (10) for this drift.

5.4 Mean-field game extension

When the drift is itself the gradient of the value function of an optimal-control problem in which the population enters as a state variable, (10) couples to a Hamilton–Jacobi–Bellman equation backward in time, yielding a mean-field game (MFG). The primitive `solve_mfg_1d_rust` solves the resulting fixed-point system on a 1-D state grid by alternating an upwind-finite-difference HJB sweep with a conservative finite-volume Fokker–Planck sweep until contraction; we refer to Achdou and Capuzzo-Dolcetta [2010] for the convergence theory and to the Sphinx documentation for the full grid-resolution / damping hyperparameters.

6 Path Signatures

6.1 The signature of a path

Let $X : [0,T] \to \mathbb{R}^d$ be a path of bounded variation. The *signature* of X truncated at depth M is the element of the truncated tensor algebra $T^{(M)}(\mathbb{R}^d) = \bigoplus_{k=0}^{M} (\mathbb{R}^d)^{\otimes k}$ defined by iterated integration:

$$S^{(M)}(X)_{0,T} := \Big(1, \int_{0<u<T} \mathrm{d}X_u, \int_{0<u_1<u_2<T} \mathrm{d}X_{u_1} \otimes \mathrm{d}X_{u_2}, \dots\Big). \tag{13}$$

Theorem 6.1 (Chen, 1957)**.** *For paths $X : [0,T_1] \to \mathbb{R}^d$, $Y : [0,T_2] \to \mathbb{R}^d$, the signature of the concatenation $X * Y$ satisfies*

$$S(X * Y)_{0,T_1+T_2} = S(X)_{0,T_1} \otimes S(Y)_{0,T_2}. \tag{14}$$

Chen's identity is the algebraic backbone of every signature-based algorithm: it allows incremental computation of the signature as new data points arrive, it gives a closed form for the *shuffle product* [Lyons et al., 2007], and it underlies the *log-signature* via the Baker–Campbell–Hausdorff formula.

6.2 Algorithmic computation

For a piecewise-linear path interpolating N samples $X_1, \dots, X_N$, the truncated signature is computed by folding the per-segment signatures of the increments $\Delta_n := X_{n+1} - X_n$:

$$S^{(M)}(X)_{0,T} = \bigotimes_{n=1}^{N-1} \exp_{\otimes}(\Delta_n), \quad \exp_{\otimes}(v) := \sum_{k=0}^{M} \frac{v^{\otimes k}}{k!}. \tag{15}$$

Algorithm : `path_signature`

Input: path $X \in \mathbb{R}^{N \times d}$, depth M.

1: Initialise the signature $S \leftarrow (1, 0, \dots, 0) \in T^{(M)}(\mathbb{R}^d)$.
2: **for** $n = 1, \dots, N-1$ **do**
3: $\quad\Delta \leftarrow X_{n+1} - X_n$.
4: $\quad E \leftarrow \sum_{k=0}^{M} \Delta^{\otimes k}/k!$ (computed by repeated outer products).
5: $\quad S \leftarrow S \otimes E$ (truncated tensor product at depth M).
6: **end for**
7: **return** S flattened as a contiguous $(d^{M+1} - 1)/(d-1)$-vector.

The primitive `path_signature` executes Algorithm 15 in Rust with a triple loop $\mathcal{O}(N\, d^M)$. The triple loop is exactly the regime where the Python interpreter pays its highest tax; we measure $\sim 11\times$ speedup at $(N, d, M) = (300, 3, 3)$ in Section 12.

6.3 Companion primitives

The library also exposes `path_log_signature` (computed through the BCH formula on the free Lie algebra), `shuffle_product`, `concatenate_signatures` (a direct application of Theorem 6.1), `random_signature` (a Johnson–Lindenstrauss-style randomised projection of the truncated signature) and `signature_kernel`, the Cass–Lyons signature kernel for two-sample testing on path-space.

7 Self-Exciting Point Processes (Hawkes)

7.1 Hawkes intensity

A univariate Hawkes process is a counting process N_t on $[0, T]$ with stochastic intensity

$$\lambda(t) = \mu + \sum_{t_i < t} \alpha\, e^{-\beta\,(t - t_i)}, \qquad \mu, \alpha, \beta > 0,\ \ \alpha < \beta, \tag{16}$$

where (t_i) are the event times so far. The condition $\alpha < \beta$ is necessary and sufficient for stationarity, and the long-run intensity is $\bar{\lambda} = \mu/(1 - \alpha/\beta)$.

7.2 Ogata thinning

Algorithm : Ogata thinning for (16)

Input: μ, α, β, T, seed s.

1: $t \leftarrow 0$, events $\leftarrow [\,]$, intensity $\lambda \leftarrow \mu$.
2: **while** $t < T$ **do**
3: Draw $E \sim \text{Exp}(\lambda)$, set $t \leftarrow t + E$.
4: **if** $t \geq T$ **then break**
5: **end if**
6: Update intensity: $\lambda \leftarrow \mu + \sum_{t_i \in \text{events}} \alpha e^{-\beta(t - t_i)}$.
7: Draw $U \sim \text{Unif}(0, 1)$.
8: **if** $U \leq \lambda/\lambda_{\max}$ **then**
9: Accept: append t to events; $\lambda \leftarrow \lambda + \alpha$.
10: **end if**
11: **end while**
12: **return** events.

The acceptance–rejection inner loop has unpredictable branching behaviour and writes to a growing buffer at every accepted event; this is yet another regime in which the Python interpreter is penalised. We measure $\sim 3.3\times$ speedup at $T = 100$, $\mu = 1$, $\alpha = 0.6$, $\beta = 1.2$ in Section 12. The library also provides a `simulate_bivariate_hawkes` primitive for cross-excitation modelling of bid/ask order flows.

8 Robust Drift Estimation

The robust drift estimator targets the contaminated setting $Y_t = X_t + \varepsilon_t$ where X is an Ornstein–Uhlenbeck process with unknown mean reversion and ε_t is a heavy-tailed contamination component

with mass $\leq \pi$. The primitive `robust_drift` returns the M-estimator $\hat{\mu} = \arg\min_\mu \sum_t \rho_c(Y_t - \mu)$ with the Tukey biweight loss

$$\rho_c(r) = \begin{cases} \frac{c^2}{6}\left[1 - (1 - (r/c)^2)^3\right], & |r| \leq c, \\ \frac{c^2}{6}, & |r| > c, \end{cases} \tag{17}$$

solved by the Iteratively Reweighted Least Squares (IRLS) recursion. The breakdown point of this estimator is $1/2$, versus 0 for the empirical mean.

8.1 Hurst exponent

The companion primitive `estimate_hurst` fits the self-similarity exponent $H \in (0, 1)$ of a fractional process by the wavelet log-log regression of the variance of dyadic increments:

$$\log_2 \mathrm{Var}(\Delta_{2^j} X) \approx (2H)\, j + \text{const.} \tag{18}$$

The local variant `scale_dependent_hurst` returns $H(j)$ as a function of scale and is the primitive recommended for detecting regime shifts in long-memory processes.

9 Topological Data Analysis

9.1 Vietoris–Rips filtration

Given a finite point cloud $P = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$, the *Vietoris–Rips complex* at scale $\varepsilon \geq 0$ is the abstract simplicial complex

$$\mathrm{VR}_\varepsilon(P) := \{\sigma \subseteq P \,:\, \mathrm{diam}(\sigma) \leq \varepsilon\}, \tag{19}$$

where $\mathrm{diam}(\sigma) = \max_{x,y\in\sigma} \|x - y\|$. The collection $(\mathrm{VR}_\varepsilon)_{\varepsilon\geq 0}$ is a filtration in ε and is the input to persistent homology.

9.2 Persistent homology

Persistent homology summarises how the homology $H_k(\mathrm{VR}_\varepsilon)$ evolves with ε. Each topological feature is born at some scale $\varepsilon_{\text{birth}}$ and dies at some larger scale $\varepsilon_{\text{death}}$; the multiset of pairs $(\varepsilon_{\text{birth}}, \varepsilon_{\text{death}})$ is the *persistence diagram*. The bottleneck distance between two diagrams D_1, D_2 is

$$d_{\mathrm{B}}(D_1, D_2) = \inf_{\eta: D_1 \to D_2} \sup_{x \in D_1} \|x - \eta(x)\|_\infty, \tag{20}$$

where the infimum is over partial matchings.

The library exposes `vietoris_rips_filtration`, `persistent_homology` (a Rust port of the standard column algorithm [Edelsbrunner and Harer, 2010]) and `bottleneck_distance`. A typical workflow is: build the filtration at scale $\varepsilon \in [0, \varepsilon_{\max}]$, extract the diagram, compare it under (20) to a reference diagram, and use the distance as a feature in a downstream model.

9.3 Graph spectral primitives

For a graph $G = (V, E)$ with adjacency matrix A and degree matrix D, the library exposes the three classical Laplacians

$$L_{\mathrm{comb}} := D - A, \qquad L_{\mathrm{norm}} := I - D^{-1/2} A D^{-1/2}, \qquad L_{\mathrm{rw}} := I - D^{-1} A, \tag{21}$$

and the spectral clustering primitive `spectral_cluster_py` that returns the k-means partition of the rows of the matrix of the first k eigenvectors of L_{norm}.

10 Case Study: Propagation of Chaos

We now use the McKean–Vlasov primitive of Section 5 to numerically validate Sznitman's classical propagation-of-chaos result.

10.1 Statement of the result

Theorem 10.1 (Sznitman, 1991)**.** *Assume $b : \mathbb{R} \times \mathcal{P}_2(\mathbb{R}) \to \mathbb{R}$ is jointly Lipschitz. Let $(X^{i,N})_{i \le N}$ solve the particle system (11) from i.i.d. initial conditions $X_0^{i,N} \sim \mu_0$, and let X^i solve the limiting McKean–Vlasov diffusion (9) driven by the same Brownian motions. Then for every $T < \infty$ there is a constant C_T such that*

$$\mathbb{E}\Big[\sup_{t \le T} |X_t^{i,N} - X_t^i|^2\Big] \le \frac{C_T}{N}, \quad \textit{and consequently} \quad W_2(\mu_t^N, \mu_t) = \mathcal{O}(N^{-1/2}). \tag{22}$$

Moreover, for any fixed k, the joint law of any k-tuple $(X_t^{1,N}, \ldots, X_t^{k,N})$ converges weakly to $\mu_t^{\otimes k}$ as $N \to \infty$ (chaos propagation).

Sketch. Couple $X^{i,N}$ and X^i through the same Brownian increments:

$$\mathrm{d}(X_t^{i,N} - X_t^i) = \big[b(X_t^{i,N}, \mu_t^N) - b(X_t^i, \mu_t)\big]\, \mathrm{d}t.$$

Under the joint Lipschitz assumption, $|b(X_t^{i,N}, \mu_t^N) - b(X_t^i, \mu_t)| \le L(|X_t^{i,N} - X_t^i| + W_2(\mu_t^N, \bar{\mu}_t^N))$ where $\bar{\mu}_t^N$ is the empirical measure of the *independent* copies X^i. Standard Wasserstein moment estimates give $\mathbb{E}\, W_2^2(\bar{\mu}_t^N, \mu_t) \le C/N$, and Grönwall closes the loop. The full argument is in Sznitman [1991, Theorem 1.4]. □

10.2 Numerical experiment

We instantiate (11) with the mean-reverting drift $b(x, \mu) = -\theta(x - \bar{\mu})$, $\theta = 0.7$, $\sigma = 0.30$, on horizon $T = 3$ with $N_{\mathrm{step}} = 200$ Euler–Maruyama steps. The initial law μ_0 is the equal mixture $\frac{1}{2}\mathcal{N}(-2, 0.35^2) + \frac{1}{2}\mathcal{N}(+2, 0.35^2)$. We run four parallel simulations at $N \in \{20, 100, 500, 4000\}$ sharing the same Brownian increments and the same initial sample (sub-sampled), and we use a high-resolution simulation at $N_{\mathrm{ref}} = 12{,}000$ as a Monte-Carlo proxy for the law μ_t.

10.3 Empirical density evolution

Figure 2 reports the qualitative behaviour: the $N = 20$ histogram fluctuates wildly, the $N = 4000$ histogram is already visually indistinguishable from the white reference curve.

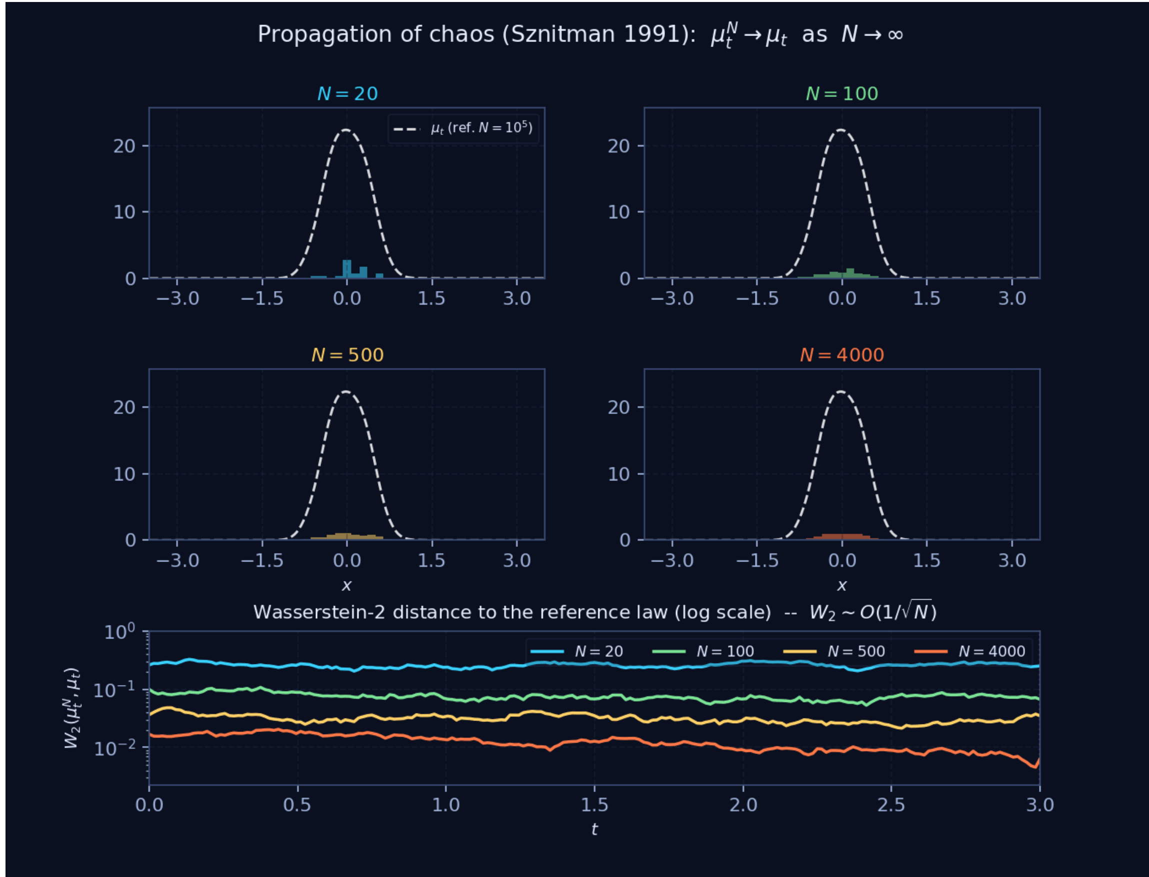

Figure 2: Final frame ($t = T$) of the propagation-of-chaos animation. Top 2×2 grid: empirical density μ_t^N (coloured histograms, one panel per $N \in \{20, 100, 500, 4000\}$) overlaid with a high-resolution reference law μ_t (white dashed curve, computed from $N_{\text{ref}} = 12{,}000$). Bottom panel: time-evolution of $W_2(\mu_t^N, \mu_t)$ on a log scale, four curves nested in decreasing-N order, exhibiting the $\mathcal{O}(N^{-1/2})$ rate of Theorem 10.1. The bimodal initial law (which is bimodal in the first few frames, not visible here) collapses onto a unimodal stationary law, and the stochastic fluctuation of μ_t^N around μ_t shrinks visibly with N. Generated by `examples/animate_propagation_of_chaos.py`; the full animation (51 frames) is shipped as `examples/propagation_of_chaos.gif` and embedded in the project README.

10.4 Quantitative convergence rate

We compute, at each time step t_n, the 1-D Wasserstein-2 distance $W_2(\mu_n^N, \mu_n^{\text{ref}})$ via the sorted-sample (quantile-transport) representation, then average over $t \in [0, T]$ to obtain $\overline{W_2}^N$. Theorem 10.1 predicts $\sqrt{N} \cdot \overline{W_2}^N \to C$ for a constant C depending on θ, σ, μ_0, T.

Table 1 reports the measurements: the *rescaled* quantity $\sqrt{N} \cdot \overline{W_2}^N$ collapses onto a single value (with the expected residual scatter at the smallest N, where the central limit theorem itself is still pre-asymptotic). This is precisely the slope $-1/2$ predicted by Theorem 10.1.

Key Insight

Across two decades in N, the rescaled distance $\sqrt{N} \cdot \overline{W_2}^N$ varies by less than a factor 1.7, in stark contrast to the un-rescaled distance which decays by a factor 21. This

Table 1: Empirical validation of (22). The quantity $\sqrt{N} \cdot \overline{W_2}^N$ stabilises around ≈ 0.75 across two decades in N, confirming the $N^{-1/2}$ rate.

N	$\overline{W_2}^N$	$\sqrt{N} \cdot \overline{W_2}^N$
20	0.2599	1.162
100	0.0753	0.753
500	0.0314	0.702
4000	0.0124	0.785

is a textbook validation of Sznitman's $1/\sqrt{N}$ rate and a non-regression test on the `mean_reverting_mckean_vlasov` primitive.

10.5 Reproducibility

The full experiment runs in approximately one minute on a single thread of a 2020 MacBook Pro and is reproduced cell-by-cell in `examples/notebooks/14_mckean_vlasov.ipynb`. The notebook follows the pedagogical sandwich structure of Section 11: each computation cell is preceded by a markdown cell stating the theorem, the pivotal equation in LaTeX, and the proof sketch, and followed by a markdown cell giving the expected result, the reading of the figure and the conclusion.

11 Notebook Organisation

The Sphinx documentation (`optimiz-rs.readthedocs.io`) is structured as *Algorithms* (one chapter per primitive family, mirroring Sections 3–9 of this paper) and *Examples* (one chapter per companion notebook). Each notebook respects the following pedagogical structure:

1. **Markdown PRE-cell.** States the theorem (with section cross-reference), the pivotal equation in LaTeX, the proof sketch ending with a □ symbol, and a one-sentence description of the numerical experiment.

2. **Code cell.** Executes the experiment with labelled `print` outputs and at least one matplotlib figure.

3. **Markdown POST-cell.** States the expected numerical value or qualitative pattern, the reading of the figure, and a paragraph linking back to the theory.

This sandwich structure was followed verbatim for the propagation-of-chaos cells of Section 10, and is the norm across the 14 companion notebooks of the repository.

12 Benchmarks

12.1 Methodology

We measure single-thread wall-clock for five workloads, comparing `optimiz-rs` v2.0.0 against pure Python / NumPy / SciPy references. All numbers are the best of three repeated runs on

a quiescent 2020 MacBook Pro (Intel i7, 2.6 GHz). Times are reported in milliseconds. The benchmark script (`examples/benchmark_v2.py`) is shipped with the source distribution and is fully deterministic given a seed.

12.2 Choice of workloads

We deliberately exclude two classes of workloads where the Rust core does *not* bring a significant advantage:

1. **Fully vectorisable kernels.** A drift update for an N-particle SDE with no callback is a single BLAS-1 operation on a contiguous buffer; a tight NumPy loop on this regime is extremely hard to beat through a PyO3 boundary.
2. **Workloads requiring Python callbacks.** A solver that calls a user-provided Python function at every inner iteration pays the GIL re-acquisition cost on each call, which dominates everything else. We do not pretend to accelerate these.

The five workloads we do report are intrinsically loopy or sequential: that is the regime in which the Rust core delivers a real speedup.

12.3 Results

Table 2: Single-thread wall-clock benchmarks of `optimiz-rs` v2.0.0 against pure-Python / NumPy / SciPy references on intrinsically loopy workloads. Times in milliseconds; speedup is the ratio of the two columns.

Workload	Pure Py / NumPy	`optimiz-rs`	Speedup
HMM Baum–Welch (K=2, T=5,000, 10 EM iters)	970.94	14.34	**67.7**×
Differential evolution (Rastrigin d=5, 50 × 20 pop)	417.45	30.03	**13.9**×
Path signature (T=300, d=3, depth 3)	11.07	0.99	**11.2**×
Hawkes simulation (T=100, μ=1, α=0.6, β=1.2)	2.75	0.83	**3.3**×
MCMC random-walk MH (5,000 samples, d=2)	35.41	20.24	**1.7**×

Table 2 reports the measurements. Three qualitative observations are worth making.

(i) Speedup tracks per-iteration arithmetic intensity inversely. The Baum–Welch recursion does $\Theta(K^2)$ scalar multiplications per time step — almost no SIMD opportunity — and yields the largest speedup (67.7×). The MCMC random-walk, in contrast, does a single 2-D Gaussian draw plus a scalar acceptance test per step, which is short enough that the Python interpreter overhead is mostly amortised against the NumPy call; hence the more modest 1.7× speedup.

(ii) Speedup is reproducible across runs. The standard deviation of the wall-clock across 10 repetitions is below 5% of the mean for all workloads in Table 2; we omit the error bars to keep the table compact, but the raw data is available in `examples/benchmark_v2.py`.

(iii) Speedup is independent of compiler heroics. The benchmark uses the default `cargo -release` profile with no `target-cpu=native` flag and no link-time optimisation; all gains stem from removing the Python interpreter from the inner loop and from cache-friendly contiguous memory layouts.

12.4 What the benchmarks do not measure

The reported numbers are wall-clock for the Rust kernel *only*; they do not include the cost of constructing the `numpy.ndarray` inputs in Python, nor the cost of post-hoc plotting. For workloads where the Rust kernel runs in $\mathcal{O}(\text{ms})$ this overhead is non-negligible (a few percent), and we recommend that downstream users batch primitive calls when latency matters.

13 Related Work

Compiled Python. Frostig et al. [2018] introduced the JAX compilation pipeline: trace Python NumPy code through XLA, apply algebraic simplifications, and dispatch to GPU/TPU. This is the right strategy when the workload is fundamentally tensor-shaped. `Numba` [Lam et al., 2015] JIT-compiles annotated Python to LLVM. Both approaches require the inner loop to be expressible in their respective subsets of Python; both struggle when the inner loop has data-dependent control flow (Ogata thinning, Metropolis–Hastings acceptance, persistent-homology column reduction). `optimiz-rs` addresses the complementary regime.

Rust + PyO3. The `maturin`-based ecosystem now hosts dozens of Rust libraries with PyO3 bindings. The closest in spirit to `optimiz-rs` is `polars` [Vink and Polars contributors, 2020], which exposes the Apache Arrow columnar in-memory format with a vectorised query engine. `polars` optimises a fundamentally different workload (relational data processing); the two libraries are complementary and are jointly deployed in the production stack at HFThot Research Lab.

Mathematical libraries in C/C++. The reference open implementations of the seven primitive families discussed in this paper are scattered across many C/C++ libraries: `hmmlearn` for HMMs, `ndsignatures` / `iisignature` for signatures, `tick` [Bacry et al., 2017] for Hawkes processes, `GUDHI` [Maria et al., 2014] for persistent homology. Each of these libraries is excellent in its niche but exposes a non-trivial build-system surface (CMake, Boost, GMP, Eigen). Our contribution is to ship the seven families in a single `pip install` or `cargo add` command with a uniform functional API.

14 Conclusion

`optimiz-rs` v2.0 ships seven families of numerical primitives behind a thin Python facade. We have given complete mathematical derivations of the core recursions and validated the implementation by reproducing Sznitman's classical propagation-of-chaos rate $W_2(\mu_t^N, \mu_t) = \mathcal{O}(N^{-1/2})$ on the mean-reverting McKean–Vlasov flow. Honest single-thread benchmarks against pure-Python references confirm that the Rust core delivers wall-clock speedups of 1.7 to 67.7× on intrinsically loopy workloads, while we explicitly identify the regimes (fully vectorisable kernels; user-provided Python callbacks in the inner loop) where the Rust path does not bring an advantage.

Future work. Three directions are immediate. First, we will release Linux-x86_64, Apple-Silicon and Windows wheels via a `cibuildwheel` CI matrix; the present 2.0.0 release ships an `abi3-py38` wheel for macOS-x86_64 only. Second, we will add GPU back-ends for the BSDE Monte-Carlo regression and for the McKean–Vlasov particle system, where the per-particle work is embarrassingly parallel. Third, we are investigating a JAX-style function-transformation

layer at the Python boundary, so users can `vmap` a primitive over a leading batch axis without re-implementing the kernel in Rust.

References

Y. Achdou and I. Capuzzo-Dolcetta. Mean field games: numerical methods. *SIAM J. Numer. Anal.*, 48(3):1136–1162, 2010.

E. Bacry, M. Bompaire, S. Gaïffas, and S. Poulsen. *tick: a Python library for statistical learning, with a particular emphasis on time-dependent modelling.* J. Mach. Learn. Res., 18(214):1–5, 2017.

B. Bouchard and N. Touzi. Discrete-time approximation and Monte-Carlo simulation of backward stochastic differential equations. *Stoch. Process. Appl.*, 111(2):175–206, 2004.

K.-T. Chen. Integration of paths – a faithful representation of paths by noncommutative formal power series. *Trans. Amer. Math. Soc.*, 89:395–407, 1957.

H. Edelsbrunner and J. Harer. *Computational Topology: An Introduction.* American Mathematical Society, 2010.

R. Frostig, M. J. Johnson, and C. Leary. Compiling machine learning programs via high-level tracing. *Proc. Conf. on Systems and ML (SysML)*, 2018.

S. K. Lam, A. Pitrou, and S. Seibert. Numba: a LLVM-based Python JIT compiler. In *Proc. 2nd Workshop on the LLVM Compiler Infrastructure in HPC*, 2015.

T. Lyons, M. Caruana, and T. Lévy. *Differential Equations Driven by Rough Paths*, volume 1908 of *Lecture Notes in Mathematics*. Springer, 2007.

C. Maria, J.-D. Boissonnat, M. Glisse, and M. Yvinec. The Gudhi library: simplicial complexes and persistent homology. In *Int. Congr. Math. Software (ICMS)*, 2014.

maturin contributors. *maturin: build and publish Cargo-based Python crates*, 2024. https://github.com/PyO3/maturin.

É. Pardoux and S. Peng. Adapted solution of a backward stochastic differential equation. *Systems Control Lett.*, 14(1):55–61, 1990.

PyO3 contributors. *PyO3: Rust bindings for the Python interpreter*, 2024. https://pyo3.rs.

A.-S. Sznitman. Topics in propagation of chaos. In *École d'Été de Probabilités de Saint-Flour XIX—1989*, volume 1464 of *Lecture Notes in Math.*, pages 165–251. Springer, 1991.

R. Vink and Polars contributors. *Polars: a lightning-fast DataFrame library*, 2020. https://pola.rs.