Companion Notes for `landau_hermite_jax.py`: Fast Landau–Hermite Collisions via SOE→MPO/TT (JAX-first)

(auto-generated companion for the standalone script)

February 4, 2026

Abstract

This document explains the numerics, implementation, and test suite used by the standalone script `landau_hermite_jax.py`. The script implements a fast evaluation of the spatially homogeneous Landau (Fokker–Planck) collision operator in a 3D tensor-product Hermite basis, using a separable quadrature ("SOE" style separation), tensor-product operator factorizations ("MPO" viewpoint), and optional tensor-train (TT) compression (currently used in the NumPy debug path; the JAX path relies on JIT+vectorized contractions).

The goal is pedagogic completeness: what the code computes, how it is normalized, how it is time-stepped, what "linearized" means in this context, why the tests are designed the way they are, and how to interpret the produced figures (Fig. 1/Fig. 2 panels and the test plots).

Contents

1 Overview: equations, goals, and where the cost lives

This companion is meant to read like a guided tour from first principles to the specific fast implementation in `landau_hermite_jax.py`. The structure is:

1. What equation we solve and what is measured (physics).
2. How we discretize velocity with a Hermite basis (numerics).
3. Why the nonlinear Landau operator is expensive (bottlenecks).
4. How separability (SOE), tensor-product contractions (MPO), and optional compression (TT) remove the bottlenecks.
5. How the script time-steps, tests itself, and produces Fig. 1/Fig. 2 panels.

1.1 The equations solved by the script

The Landau collision operator describes collisional relaxation in velocity space. In this project we consider the *spatially homogeneous* setting, where the distribution functions depend only on velocity $\boldsymbol{v} \in \mathbb{R}^3$ and time t.

One species. For a single species with distribution $f(\boldsymbol{v}, t)$ we solve the ODE/PDE

$$\partial_t f = \nu \, Q(f, f), \tag{1}$$

where ν is a collision frequency (a scalar in this script), and Q is the nonlinear Landau collision operator.

Two species. For two species a and b we solve

$$\partial_t f_a = \nu_{ab} \, Q_{ab}(f_a, f_b), \tag{2}$$

$$\partial_t f_b = \nu_{ba} \, Q_{ba}(f_b, f_a), \tag{3}$$

with parameters chosen so that the *total* invariants (summed across species) are conserved.

1.2 What we are trying to compute

The script does *not* discretize physical space; it only evolves velocity-space dynamics. Concretely, it:

- represents each $f(\boldsymbol{v}, t)$ in a tensor-product Hermite basis in the normalized velocity variables $(x, y, z) = (v_x/v_{\text{th}}, v_y/v_{\text{th}}, v_z/v_{\text{th}})$;
- evolves the corresponding coefficient tensor in time using an explicit ODE integrator;
- computes diagnostics (density, momentum, energy, temperatures, anisotropy) and produces the Fig. 1/Fig. 2 panels;
- optionally (`--run_tests`) runs a correctness/performance suite that produces additional plots and timing data.

1.3 Why this is hard: the nonlinear Landau bottleneck

At a high level, the nonlinear Landau operator is expensive for two reasons:

1. **Nonlocality in velocity:** $Q(f, f)$ involves an integral over $\boldsymbol{v}'$ with a long-range Coulomb kernel. Naively, evaluating this for many velocity degrees of freedom is comparable to a dense integral operator.

2. **Mode coupling:** in a spectral representation, each output coefficient couples to many input coefficients. A naive coefficient-space implementation resembles a high-rank tensor contraction with $\mathcal{O}(p^6)$–$\mathcal{O}(p^7)$ work (depending on formulation), where $p = n_{\max}+1$ per velocity dimension.

The core purpose of the "SOE→MPO/TT" machinery is to replace these naive dense couplings by a sum of *separable* tensor-product contractions, so the cost is dominated by repeated 1D contractions that are friendly to JIT compilation and cache/accelerator hardware.

1.4 Roadmap of the fast algorithm (one paragraph)

In the Hermite formulation used here, the coefficient formulas can be reorganized so that many factors depend only on *one-dimensional* indices in x, y, and z, while the Coulomb kernel moments enter through a scalar factor that can be written as an integral over $s \in [0,1]$. Approximating that scalar factor by Gauss–Legendre quadrature yields a *finite sum of separable terms* (called "SOE" in this code base). Each term becomes a tensor-product operator acting along x, y, and z, which can be applied by three successive 1D contractions (an "MPO" viewpoint). For larger $n_{\max}$, some intermediate tensors can optionally be compressed using a tensor-train (TT) format (currently in the NumPy debug path).

2 How to run the code (quickstart and reproducibility)

2.1 What the script produces

Running `python landau_hermite_jax.py` (defaults) writes:

- `Fig1_panel.pdf` and `Fig1_panel.png`: a 2×2 diagnostic panel for a one-species relaxation problem and a two-species equilibration problem.
- Optionally (if `--run_tests`), a timestamped directory under `tests_landau_hermite/run_YYYYmmdd_HHMMSS/` containing plots, CSV, and JSON summaries. The latest run is also linked as `tests_landau_hermite/latest`.

2.2 Backend

The script has two backends:

- **JAX backend** (`--backend jax`, default): uses `jax.jit`, `vmap/einsum`, and `lax.scan` time stepping for performance and scalability.
- **NumPy backend** (`--backend numpy`): for debugging and for certain explicit checks.

Both backends use the *same* precomputed coefficient tables and the same fast SOE→MPO contraction structure; the difference is whether contractions are executed with NumPy or JAX/XLA.

2.3 CLI options

The script self-documents its CLI via `-h`. For day-to-day use, the most important options are:

- `--nmax`: Hermite truncation ($p = n_{\max} + 1$ modes per dimension).
- `--backend {jax,numpy}`: performance vs debugging.
- `--Q`, `--maxK`: accuracy/cost knobs for the separated kernel evaluation.
- `--linearized on/off`: include linearized overlays (computed within the same script).
- `--run_tests`: generate the test suite plots and timing summaries in `tests_landau_hermite/`.

The full help text is included in Appendix 14.

3 Physical model: Landau collision operator (first principles)

3.1 Continuous operator (context)

For species a colliding with species b, the Landau collision operator can be written (in one common form) as

$$Q_{ab}(f_a, f_b)(\boldsymbol{v}) = \frac{\partial}{\partial v_i} \int_{\mathbb{R}^3} U_{ij}(\boldsymbol{u}) \left[f_b(\boldsymbol{v}') \frac{\partial f_a(\boldsymbol{v})}{\partial v_j} - \frac{m_a}{m_b} f_a(\boldsymbol{v}) \frac{\partial f_b(\boldsymbol{v}')}{\partial v_j'} \right] \mathrm{d}\boldsymbol{v}', \qquad \boldsymbol{u} = \boldsymbol{v} - \boldsymbol{v}', \quad (4)$$

where repeated indices are summed over $i, j \in \{1, 2, 3\}$ and, for Coulomb interactions,

$$U_{ij}(\boldsymbol{u}) = \frac{\delta_{ij}|\boldsymbol{u}|^2 - u_i u_j}{|\boldsymbol{u}|^3} = \frac{1}{|\boldsymbol{u}|} \left(\delta_{ij} - \frac{u_i u_j}{|\boldsymbol{u}|^2} \right). \quad (5)$$

In this standalone script we absorb all dimensional prefactors (charges, Coulomb logarithm, etc.) into the scalar collision frequencies ν_{ab} and ν_{ba} that multiply Q_{ab} in time. This operator is:

- **bilinear** in (f_a, f_b),
- **conservative**: it preserves total number, total momentum, and total energy (appropriate combinations across species),
- **entropy producing**: for the *nonlinear* operator, relative entropy to a local Maxwellian is non-increasing in time (H-theorem).

Standard references include Landau (1936), Rosenbluth–MacDonald–Judd (1957), and modern expositions such as Helander & Sigmar (2002) and Villani (2002).

3.2 Rosenbluth potentials viewpoint (intuition)

One way to interpret the Landau operator is via *Rosenbluth potentials*, which rewrite the nonlocal integral in terms of two scalar convolution-type potentials G and H of f_b. This viewpoint makes clear why the operator is simultaneously:

- **nonlocal** (it depends on velocity-space potentials), and
- **differential** in $\boldsymbol{v}$ (it is a divergence of drift+diffusion terms).

The present code does not explicitly solve Poisson equations for potentials; instead it evaluates the same mathematical content directly in Hermite space via structured coefficient formulas. Still, it is helpful to keep the Rosenbluth picture in mind: "build potentials of f_b" then "apply them to f_a".

3.3 What the script discretizes

The code discretizes *velocity* in a tensor-product Hermite basis and evolves the corresponding expansion coefficients in time for spatially homogeneous relaxation. All spatial dependence is absent: the ODE is in coefficient space.

4 From the continuous equation to a coefficient-space ODE

4.1 Step 1: nondimensionalize velocity

For each species we pick a thermal speed v_{th} and define dimensionless velocity coordinates $(x, y, z) = \boldsymbol{v}/v_{\text{th}}$. This isolates temperature/mass dependence into a few scalars and makes the basis and diagnostics simple.

4.2 Step 2: expand f in a tensor-product Hermite basis

We represent the distribution in a truncated basis:

$$f(x,y,z,t) \approx \sum_{\alpha,\beta,\gamma=0}^{n_{\max}} f[\alpha,\beta,\gamma](t)\,\psi_\alpha(x)\psi_\beta(y)\psi_\gamma(z), \tag{6}$$

and evolve the coefficient tensor $f[\alpha,\beta,\gamma](t)$.

4.3 Step 3: project $Q(f_a, f_b)$ onto the basis

Applying Galerkin projection (take inner products with test functions) yields a closed ODE system:

$$\frac{\mathrm{d}}{\mathrm{d}t} f_k(t) = \sum_{k_a,k_b} C^{ab}_{k,k_a,k_b}\; f_{a,k_a}(t)\; f_{b,k_b}(t), \tag{7}$$

where k is a multi-index for (α,β,γ) and C^{ab} is a (very large) dense bilinear coefficient tensor encoding the collision physics.

What is C^{ab}_{k,k_a,k_b} *exactly*? The script ultimately implements the coefficient-level expression obtained by projecting the Landau form onto the (species-dependent) Hermite basis. Using normalized velocities $\overline{\boldsymbol{v}}_a = \boldsymbol{v}/v_{\mathrm{th},a}$ and $\overline{\boldsymbol{v}}'_b = \boldsymbol{v}'/v_{\mathrm{th},b}$, and letting $\psi^a_k(\overline{\boldsymbol{v}}_a)$ denote the 3D tensor-product basis functions for species a (with the 1D definition in Sec. 5), one convenient coefficient expression is:

$$C^{ab}_{k,k_a,k_b} = -\frac{\nu_{ab}}{2}\int_{\mathbb{R}^3}\!\int_{\mathbb{R}^3} \frac{\partial \psi^{a,k}}{\partial \overline{v}_{i,a}}(\overline{\boldsymbol{v}}_a)\,\overline{U}_{ij}(\overline{\boldsymbol{u}}_a)\,\left[\psi^b_{k_b}(\overline{\boldsymbol{v}}'_b)\,\frac{\partial \psi^a_{k_a}}{\partial \overline{v}_{j,a}}(\overline{\boldsymbol{v}}_a) - \frac{m_a}{m_b}\frac{v_{\mathrm{th},a}}{v_{\mathrm{th},b}}\,\psi^a_{k_a}(\overline{\boldsymbol{v}}_a)\,\frac{\partial \psi^b_{k_b}}{\partial \overline{v}'_{j,b}}(\overline{\boldsymbol{v}}'_b)\right] \mathrm{d}\overline{\boldsymbol{v}}_a\,\mathrm{d}\overline{\boldsymbol{v}}'_b, \tag{8}$$

where $\overline{\boldsymbol{u}}_a = \overline{\boldsymbol{v}}_a - (v_{\mathrm{th},b}/v_{\mathrm{th},a})\,\overline{\boldsymbol{v}}'_b$ and $\overline{U}_{ij}$ is the same Landau tensor kernel written in the normalized variables (so it depends only on the relative velocity argument). Equation (8) is the "source of truth" equation we refer back to when explaining every fast step (SOE, MPO contractions, TT compression) below: all of the algorithmic machinery is about evaluating (8) *without* forming a dense 6D coupling tensor.

4.4 Where the cost comes from (the bottleneck, explicitly)

If we denote $p = n_{\max} + 1$ and $N = p^3$ coefficients per species, then a direct evaluation of the above dense bilinear form would be:

- **memory-prohibitive**: storing C^{ab} is $\mathcal{O}(N^3)$;
- **compute-prohibitive**: contracting it is also $\mathcal{O}(N^3)$ operations.

Even if one never explicitly forms C^{ab}, a naive implementation of the nested sums typically scales worse than $\mathcal{O}(p^6)$ in practice. The entire purpose of the fast machinery in this project is to compute the same RHS using only:

- precomputed *1D tables* (cheap to build and reuse), and
- a small number of repeated *tensor-product contractions* (cheap to evaluate when fused).

5 Normalization, basis, and coefficient tensor representation

5.1 Thermal normalization

Each species has a reference thermal speed v_{th} and mass m. The code uses normalized velocity coordinates

$$x = v_x/v_{\mathrm{th}}, \qquad y = v_y/v_{\mathrm{th}}, \qquad z = v_z/v_{\mathrm{th}}. \tag{9}$$

The convention used throughout is

$$v_{\mathrm{th}} = \sqrt{\frac{2T_{\mathrm{eq}}}{m}} \quad \Longleftrightarrow \quad T_{\mathrm{eq}} = \frac{m v_{\mathrm{th}}^2}{2}, \tag{10}$$

where T_{eq} is the equilibrium temperature corresponding to the chosen normalization.

5.2 Hermite basis used in the code

The script uses a Hermite–Gaussian basis in each dimension:

$$\psi_n(x) = \frac{1}{\sqrt{\pi}} \frac{H_n(x)}{\sqrt{2^n n!}} e^{-x^2}, \tag{11}$$

where H_n is the physicists' Hermite polynomial. The code implements ψ_n in `psi_1d`.

Why Hermite? (intuition) Hermite–Gaussian bases are well matched to collisional kinetic problems because:

- Maxwellians are Gaussians, so the equilibrium state is captured very compactly (in this script, the equilibrium Maxwellian corresponds to a single nonzero coefficient $f[0,0,0]$ for each species).
- Multiplication by x and differentiation in x act by *ladder relations* that shift n by ±1, making many velocity-moment and derivative operations sparse in coefficient space.
- A tensor-product basis makes 3D operations separable across (x, y, z), which is essential for the MPO-style contractions used later.

Orthogonality and normalization. With the above convention, $\{\psi_n\}_{n\geq 0}$ are orthonormal on $\mathbb{R}$:

$$\int_{\mathbb{R}} \psi_n(x)\, \psi_m(x)\, \mathrm{d}x = \delta_{nm}. \tag{12}$$

The 3D tensor basis $\psi_\alpha(x)\psi_\beta(y)\psi_\gamma(z)$ is then orthonormal on $\mathbb{R}^3$, so inner products and many diagnostics reduce to sums over low-order coefficients.

Ladder identities (used implicitly). The Hermite functions satisfy identities of the schematic form

$$x\,\psi_n(x) \propto \sqrt{n+1}\,\psi_{n+1}(x) + \sqrt{n}\,\psi_{n-1}(x), \tag{13}$$

$$\partial_x \psi_n(x) \propto \sqrt{n+1}\,\psi_{n+1}(x) - \sqrt{n}\,\psi_{n-1}(x), \tag{14}$$

up to fixed constants determined by the chosen normalization. In coefficient space, these become sparse "shift" operators. The fast Landau operator construction uses these structured shifts when building the 1D tables.

5.3 Tensor coefficient representation

For a truncation parameter $n_{\max}$, define $p = n_{\max} + 1$. The coefficient tensor is

$$f[\alpha, \beta, \gamma], \quad \alpha, \beta, \gamma \in \{0, \ldots, n_{\max}\}, \tag{15}$$

so that

$$f(x, y, z) \approx \sum_{\alpha,\beta,\gamma=0}^{n_{\max}} f[\alpha, \beta, \gamma]\, \psi_\alpha(x)\psi_\beta(y)\psi_\gamma(z). \tag{16}$$

The code keeps this 3D "cube" representation (shape (p, p, p)) throughout to avoid flattening overhead and to make tensor contractions explicit and JIT-friendly.

5.4 Moment diagnostics and invariants

From the Hermite coefficients, the script extracts:

- density n,
- momentum $\boldsymbol{P}$,
- total kinetic energy W,
- temperature $T = \frac{2}{3}\frac{W}{n}$,
- anisotropy measure $A = (T_z - T_x)/T_{\text{avg}}$.

These are computed from low-order coefficients (see `invariants_from_tensor` and `temperature_components_hat_fr`
Conservation of $(n, \boldsymbol{P}, W)$ (or total invariants for 2 species) is one of the most stringent correctness checks for collision operators.

6 Fast collision evaluation: SOE→MPO/TT

6.1 What we are accelerating: the coefficient-space collision RHS

Equation (8) is a double velocity integral with products and derivatives of basis functions and a nonlocal kernel. After Galerkin projection, this becomes a dense bilinear form in coefficient space (Sec. 4), but the *physics* structure has not disappeared; it is encoded in very specific combinatorial factors coming from Hermite identities and a small family of kernel moments.

6.2 Acronyms and the high-level idea

We will use three acronyms throughout:

- **SOE** ("separated sum" / "separable quadrature"): replace one blocking scalar factor by a finite sum of separable terms using quadrature; see [16, 11].
- **MPO** (matrix product operator): apply a tensor-product (Kronecker) operator to a tensor by successive 1D contractions; see [9, 10].
- **TT** (tensor train): compress 3D tensors via low-rank cores (same format as MPS); see [7, 8].

The overall goal is: turn the dense multi-index couplings into *(i)* precomputed 1D tables and *(ii)* a small number of repeated tensor-product contractions.

6.3 Where the main bottleneck lives: one scalar factor that couples x, y, z

Using ladder identities (derivatives) and Hermite product expansions (products), most of the basis algebra in Eq. (8) reduces to 1D tables such as $P_{n,m}^{k'}$ (stored as `P1D` in the script). What remains is the kernel moment structure. In the derivation used by the original project, the Coulomb kernel produces scalar moments

$$I_K = \int_{\mathbb{R}^3} \frac{\psi_K(\boldsymbol{u})}{|\boldsymbol{u}|}\,\mathrm{d}\boldsymbol{u}, \qquad K = (2a, 2b, 2c), \tag{17}$$

that can be written in closed form as

$$I_K = \frac{(-1)^A}{\sqrt{\pi}(2A+1)}\,\frac{\sqrt{(2a)!(2b)!(2c)!}}{2^{A-1}\,a!\,b!\,c!}, \qquad A = a+b+c. \tag{18}$$

Everything in (18) is separable across dimensions *except* the single rational factor $1/(2A+1)$, because $A = a+b+c$ is an *additive* multi-index. Unlocking separability of this factor is the SOE step.

6.4 SOE (separable quadrature): separating the kernel factor

The key identity is the Beta-function integral

$$\frac{1}{2A+1} = \int_0^1 s^{2A}\,\mathrm{d}s. \tag{19}$$

With $A = a+b+c$,

$$s^{2A} = s^{2a}\,s^{2b}\,s^{2c}, \tag{20}$$

so for each fixed s the dependence is separable in (a, b, c). Approximating (19) by Q-point Gauss–Legendre quadrature on $[0, 1]$ yields a finite separated sum:

$$\frac{1}{2(a+b+c)+1} \approx \sum_{q=1}^{Q} w_q \left(s_q^{2a}\right)\left(s_q^{2b}\right)\left(s_q^{2c}\right). \tag{21}$$

This is the **SOE** step: one coupled 3D dependence becomes a sum of Q fully separable terms. In this code base, `Q` is small (default 8), so the q-sum is efficient and can be vectorized (`vmap`) or looped with compiled control flow (`lax.fori_loop`) in JAX.

Accuracy knob. At fixed $n_{\max}$, increasing Q improves the kernel-moment accuracy (hence Maxwellian residuals). The script includes a small "auto-Q" heuristic (enabled by default) and the test suite includes (Q, `maxK`) convergence plots for reviewer-facing confidence.

6.5 Hankel-like tables: precomputing the remaining 1D dependence

After SOE separation, what remains are purely 1D combinatorial/lattice factors. A particularly important family of factors depends on a combined index K (a sum of intermediate 1D indices), and behaves like a discrete Hankel kernel. The code packages this dependence into tables

$$F_q[n] = \begin{cases} 0, & n \text{ odd}, \\ (-1)^{n/2}\,t_{n/2}\,s_q^n, & n \text{ even}, \end{cases} \tag{22}$$

with

$$t_n = \frac{\sqrt{(2n)!}}{2^n n!}. \tag{23}$$

The practical takeaway: rather than recomputing factorial-heavy expressions inside the RHS, we precompute `Fq[q,n]` once and reuse it for every RHS evaluation. The truncation parameter `maxK` caps the maximum n needed; in practice `maxK` should comfortably exceed a few multiples of $n_{\max}$ (the code uses a conservative default of `256`).

6.6 MPO viewpoint: tensor-product operator contraction

After SOE separation, the operator can be evaluated through a sequence of *1D* mode products along the x, y, and z axes. In quantum many-body language this is a matrix-product operator (MPO); in numerical linear algebra it is a Kronecker-structured operator.

A simple 2D analogy. Suppose $X \in \mathbb{R}^{p\times p}$ and we want to apply a separable linear map $Y = A\,X\,B^\top$ with $A, B \in \mathbb{R}^{p\times p}$. In index notation,

$$Y_{ij} = \sum_{m=0}^{p-1}\sum_{n=0}^{p-1} A_{im}\,X_{mn}\,B_{jn}, \tag{24}$$

which looks like a dense p^4 contraction, but it can be implemented as two p^3 matrix multiplications: first $Z = AX$, then $Y = ZB^\top$. This idea generalizes to 3D tensor products.

3D tensor-product action (mode products). For a 3D coefficient tensor $f \in \mathbb{R}^{p\times p\times p}$ and three 1D matrices $M^{(x)}, M^{(y)}, M^{(z)} \in \mathbb{R}^{p'\times p}$, a separable contraction takes the form

$$(Mf)_{abc} = \sum_{\alpha,\beta,\gamma} M^{(x)}_{a\alpha}\,M^{(y)}_{b\beta}\,M^{(z)}_{c\gamma}\,f_{\alpha\beta\gamma}. \tag{25}$$

Rather than forming the 3D operator explicitly, we compute this by three successive 1D contractions:

$$f^{(1)}_{a\beta\gamma} = \sum_{\alpha} M^{(x)}_{a\alpha}\,f_{\alpha\beta\gamma}, \tag{26}$$

$$f^{(2)}_{ab\gamma} = \sum_{\beta} M^{(y)}_{b\beta}\,f^{(1)}_{a\beta\gamma}, \tag{27}$$

$$(Mf)_{abc} = \sum_{\gamma} M^{(z)}_{c\gamma}\,f^{(2)}_{ab\gamma}. \tag{28}$$

Each step is a batched matrix multiplication. In the JAX backend, these are written as `einsum/tensordot` and compiled/fused by XLA. The cost scales like a small constant times $p'p^3$ per term, rather than the naive p^6–p^7 dense couplings.

Concretely, the code precomputes (for each quadrature node q, tensor component indices i, j, and term type) three 1D matrices that act along each dimension. Applying the operator to a 3D tensor then reduces to three successive contractions (a Kronecker product action), implemented via `einsum/tensordot` and fused by XLA in the JAX path.

6.7 TT (tensor train) compression

For larger $n_{\max}$, some intermediate tensors can become costly. A tensor-train (TT) factorization approximates a 3D tensor $X \in \mathbb{R}^{p\times p\times p}$ by low-rank factors (TT-SVD). The script includes an optional TT rounding path (enabled via `--use_tt` in the NumPy backend) to reduce intermediate ranks and memory traffic when exploring higher $n_{\max}$.

What TT means (in one equation). A rank-(r_1, r_2) TT approximation represents a 3D tensor as

$$X_{ijk} \approx \sum_{a=1}^{r_1} \sum_{b=1}^{r_2} G^{(1)}_{ia} \, G^{(2)}_{ajb} \, G^{(3)}_{bk}, \tag{29}$$

where the cores $G^{(1)} \in \mathbb{R}^{p \times r_1}$, $G^{(2)} \in \mathbb{R}^{r_1 \times p \times r_2}$, and $G^{(3)} \in \mathbb{R}^{r_2 \times p}$ are found (approximately) by a sequence of SVDs (TT-SVD). If the ranks stay small, storing and applying such tensors can be much cheaper than dense p^3 storage.

Why TT is optional here. On CPU, JAX/XLA excels at fusing the repeated dense contractions used in the MPO path. TT would require SVDs, which are expensive and (for this project) not always a net win at moderate $n_{\max}$. Therefore TT compression is currently used primarily for the NumPy debug path and for experimentation. The main performance path is "MPO + JIT".

TT-SVD (how the factors are computed). The standard TT-SVD procedure for a 3D tensor $X \in \mathbb{R}^{p \times p \times p}$ is:

1. Reshape X into a matrix $X_{(1)} \in \mathbb{R}^{p \times p^2}$ and compute its (truncated) SVD $X_{(1)} \approx U_1 \Sigma_1 V_1^\top$; set $G^{(1)} = U_1$ and absorb Σ_1 into $V_1^\top$.
2. Reshape $\Sigma_1 V_1^\top$ into a matrix of shape $(r_1 p) \times p$ and compute another SVD to obtain $G^{(2)}$ and $G^{(3)}$.

Truncation tolerances control the ranks (r_1, r_2) and the approximation error. See [7, 8] for details and error bounds.

TT/MPS references: Oseledets (2011), Hackbusch (2012), and related MPO/MPS reviews (Schollwöck 2011; Orús 2014).

6.8 Algorithmic walkthrough (as implemented)

This subsection summarizes the concrete data flow in `landau_hermite_jax.py`.

Inputs and shapes. Fix a truncation $n_{\max}$ and set $p = n_{\max} + 1$. Each distribution is a tensor $f \in \mathbb{R}^{p \times p \times p}$ in the Hermite basis.

Step 0: precompute tables (once per $(n_{\max}, Q, \texttt{maxK})$ and species pair). The function `build_model_tables_np` builds a container of dense arrays:

- Gauss–Legendre nodes/weights (s_q, w_q) on $[0, 1]$ for the separable quadrature.
- 1D mixing coefficients `a1d` encoding relative/COM transforms with the species-dependent angle $(\cos\theta, \sin\theta)$.
- A dense 1D coefficient table `P1D` that maps intermediate indices back to Hermite indices (a 1D "dual↔primal" transform used repeatedly in each dimension).
- The separable Hankel tables `Fq[q,K]` which implement the even-K Coulomb moment factors multiplied by s_q^K (see `Fq_table_np`).
- Preassembled 1D matrices `M_buildS[q,i,j,term,dim]` used to build auxiliary tensors S_1^{ij} and S_2^{ij} from the background distribution (next paragraph).

In the JAX backend these arrays are transferred to device once and treated as constants by JIT.

Step 1: build auxiliary tensors from f_b. For a cross-collision $Q_{ab}(f_a, f_b)$, the algorithm first constructs a small set of intermediate tensors (denoted S_1^{ij} and S_2^{ij} in the code) that play the role of Rosenbluth-potential moments in this Hermite formulation. Each S_ℓ^{ij} has shape (p', p', p') where $p' = 2n_{\max} + 2$ (the intermediate index range required by the Hankel structure). The construction is a sum over quadrature nodes q of Kronecker-structured applications:

$$S_\ell^{ij} \approx \sum_{q=1}^{Q} w_q \left(M_{q,i,j,\ell}^{(x)} \otimes M_{q,i,j,\ell}^{(y)} \otimes M_{q,i,j,\ell}^{(z)} \right) f_b, \tag{30}$$

implemented via three successive 1D contractions. This avoids forming any dense 6D objects.

Step 2: apply S to f_a to produce the RHS. Given S_1, S_2 (from f_b), the RHS for f_a is assembled by applying another Kronecker-structured mapping and then projecting back to the Hermite coefficient cube using the 1D `P1D` table along each axis. This is the "MPO" viewpoint: the full operator factorizes over dimensions, so applying it is a sequence of mode products rather than a giant tensor contraction.

Optional: TT compression (NumPy path). When `--use_tt` is enabled in the NumPy backend, selected intermediate tensors may be TT-rounded to reduce ranks. The JAX backend currently relies primarily on XLA fusion rather than TT-SVD, because SVD-based compression is relatively expensive on CPU for the small-to-moderate p used in the main figures.

Complexity. The dominant cost scales roughly like $O(Q\,p^4)$ for the structured contractions (with a small constant), rather than $O(p^9)$ for a dense bilinear tensor.

7 Nonlinear vs linearized evolution

7.1 Nonlinear operator

The main ODE uses the full bilinear operator:

$$\text{1sp:} \quad \frac{\mathrm{d}f}{\mathrm{d}t} = Q_{11}(f, f), \qquad \text{2sp:} \quad \frac{\mathrm{d}f_a}{\mathrm{d}t} = Q_{ab}(f_a, f_b), \quad \frac{\mathrm{d}f_b}{\mathrm{d}t} = Q_{ba}(f_b, f_a). \tag{31}$$

7.2 Linearization about a Maxwellian

For many purposes it is useful to compare the full nonlinear evolution to a *tangent-linear* approximation. The script can optionally compute such "linearized overlays" with `--linearized on`.

Which Maxwellian do we linearize about? The correct background is the *equilibrium Maxwellian with the same conserved invariants* as the initial condition. For the 1-species case, this means the unique isotropic Maxwellian M_{eq} such that

$$(n, \boldsymbol{P}, W)[M_{\text{eq}}] = (n, \boldsymbol{P}, W)[f_0]. \tag{32}$$

For the 2-species cross-collision case, the equilibrium is a *pair* $(M_{a,\text{eq}}, M_{b,\text{eq}})$ with:

- each species keeps its own density (n_a and n_b are invariants),
- both share a common drift velocity set by total momentum,

- both share a common temperature set by total energy after removing drift energy.

In the code this is constructed by projecting the analytical Maxwellian onto the truncated Hermite basis (see `build_maxwellian_tensor_from_invariants` and `build_common_equilibrium_maxwellians_2sp_from_inv`

Why this matters (the "plateau" issue). The linearized Landau operator has a nontrivial nullspace corresponding to collision invariants. If one linearizes about a Maxwellian M that does *not* match the IC invariants, then $h_0 = f_0 - M$ contains nullspace components and they will not decay. In that case a quadratic free-energy diagnostic can decrease but plateau above zero. The overlays in this script are therefore always constructed about the invariant-matched equilibrium backgrounds.

Definition of the linearized operator (1 species). For a perturbation h with $f = M_{\mathrm{eq}} + h$ and $\|h\| \ll 1$,

$$Q(M+h, M+h) = Q(M, M) + Q(h, M) + Q(M, h) + O(h^2). \tag{33}$$

Since $Q(M, M) = 0$, the linearized operator is

$$L(h) = Q(h, M) + Q(M, h). \tag{34}$$

For two species, the linearized Jacobian couples δf_a and δf_b through the mixed terms.

How `--linearized_method` works in the script. The CLI flag `--linearized_method` only affects *how we compute* the linearized time histories:

- `tangent` (default): matrix-free tangent-linear evolution using JAX directional derivatives (`jax.jvp`) in the JAX backend, or explicit bilinear expansions in the NumPy backend. This scales to larger $n_{\max}$ because it never forms a dense Jacobian.
- `matrix`: build an explicit dense Jacobian (via `jax.jacfwd` in JAX or a column-by-column fast assembly in NumPy) and then evolve with `expm_multiply`. This is convenient for small $n_{\max}$ but becomes expensive quickly.

7.3 Entropy vs free energy for linearized dynamics

This script uses two complementary Lyapunov-style diagnostics, each matched to the correct physics:

Nonlinear: KL divergence to the instantaneous local Maxwellian. The nonlinear Landau operator satisfies an H-theorem: a suitable relative entropy to the *instantaneous* local Maxwellian is non-increasing in time. The script measures this with a grid-based Kullback–Leibler (KL) divergence

$$\mathcal{D}(t) = \int f \log\left(\frac{f}{M_{\mathrm{local}}}\right) \mathrm{d}v. \tag{35}$$

Here $M_{\mathrm{local}}(\boldsymbol{v}, t)$ is the Maxwellian with the same $(n, \boldsymbol{u}, T)$ as $f(\boldsymbol{v}, t)$, i.e. it depends on time through the low-order moments. For the spatially homogeneous setting, this "local" Maxwellian is global in space but local in moment space.

Explicit form of M_{local} (what the code evaluates). In physical variables,

$$M_{\mathrm{local}}(\boldsymbol{v},t)=\frac{n(t)}{(2\pi T(t)/m)^{3/2}}\exp\left(-\frac{m|\boldsymbol{v}-\boldsymbol{u}(t)|^2}{2T(t)}\right), \tag{36}$$

with $n(t)=\int f\,\mathrm{d}v$, $\boldsymbol{u}(t)=\boldsymbol{P}(t)/(mn(t))$, and $T(t)=\frac{2}{3}\frac{1}{n(t)}\int\frac{1}{2}m|\boldsymbol{v}-\boldsymbol{u}(t)|^2 f\,\mathrm{d}v$. In the code, all of these are computed from low-order Hermite coefficients (Sec. 5), then the integral defining $\mathcal{D}$ is approximated on a fixed tensor-product grid in $(x,y,z)=\boldsymbol{v}/v_{\mathrm{th}}$.

Why this is a strong physics check. For the *nonlinear* Landau operator, the H-theorem implies $\mathcal{D}(t)$ should be non-increasing (up to discretization and truncation error). If $\mathcal{D}$ increases noticeably, it is a red flag: either the operator is incorrect, or parameters ($n_{\max}, Q,$ `maxK`) are too low, or the entropy grid is too coarse. This is why monotonicity of $\mathcal{D}$ is included in `--run_tests`.

However, this KL functional is *not* expected to be monotone for tangent-linear evolution (and can become ill-defined if the linearized solution produces slight negative values due to truncation).

Linearized: quadratic free energy about the fixed equilibrium Maxwellian. For tangent-linear evolution about a fixed Maxwellian background, the natural monotone quantity is the quadratic "free energy" (the second variation of entropy at M):

$$\mathcal{F}(t;M)=\int\frac{(f-M)^2}{M}\,\mathrm{d}v. \tag{37}$$

In the script, M is the invariant-matched equilibrium Maxwellian described above. The script therefore plots $\mathcal{D}/\mathcal{D}_0$ for nonlinear curves and $\mathcal{F}/\mathcal{F}_0$ for linearized curves in the main panels, and it tests monotonicity for each in `--run_tests`.

8 Time stepping and JAX implementation

8.1 Integrator

The script uses explicit SSPRK3 (Shu–Osher) by default, with options for RK2 and RK4. SSPRK3 offers a good stability/accuracy compromise for dissipative problems while using only 3 RHS evaluations per step. See Shu & Osher (1988) and Gottlieb–Shu–Tadmor (2001).

8.2 JAX performance strategy

Key performance choices:

- Enable float64 (`jax_enable_x64`) for accurate invariants and Maxwellian fixed-point checks.
- Treat all precomputed tables as constants to JIT (single device transfer).
- Use `lax.scan` for the time loop so the full stepper is compiled once.
- Keep shapes static (fixed $p=n_{\max}+1$ and fixed Q per run).
- Avoid Python loops over tensor indices; rely on vectorized contractions.

These are typical "best practices" for XLA-based array programming systems.

9 Why JAX is a good fit for this solver (and how it reduces cost)

9.1 What is JAX (in practical terms)?

JAX is a numerical computing library that looks like NumPy but is built around three "program transformations":

- **automatic differentiation** (forward- and reverse-mode),
- **vectorization** (`vmap`: turn a function into a batched function),
- **compilation** (`jit`: compile array programs with XLA).

The key idea is that, instead of executing Python loops and NumPy calls directly, JAX can *trace* an array program and hand a static computation graph to the XLA compiler, which then generates optimized machine code. That is the main reason this project benefits from JAX: the Landau–Hermite RHS is a large composition of dense contractions with fixed shapes, repeated many times.

9.2 Where JAX helps the most for this code

At the level of the math, the fast RHS derived from Eq. (8) boils down to:

- many **batched dense contractions** (mode products),
- repeated for many time steps,
- with **fixed shapes** once $n_{\max}$, Q, and `maxK` are chosen,
- and with large constant tables reused at every call.

This profile matches JAX/XLA extremely well: it can compile the entire computation graph ahead of time and then execute fused kernels with minimal Python overhead. In contrast, pure Python/NumPy pays per-call dispatch overhead and does not automatically fuse chains of contractions.

9.3 Glossary: mode products, batched contractions, and static shapes

The most common operation in this solver is a *mode product*: contracting a tensor with a matrix along one axis. For a 3D coefficient tensor $f \in \mathbb{R}^{p\times p\times p}$ and a 1D operator $A \in \mathbb{R}^{p'\times p}$, the mode-$x$ product is

$$(f \times_x A)[i, j, k] = \sum_{\alpha=1}^{p} A[i, \alpha]\, f[\alpha, j, k]. \tag{38}$$

Applying a Kronecker-product operator ($A_x \otimes A_y \otimes A_z$) is then just three mode products (first along x, then y, then z), as shown in Sec. 6. In code, these appear as `einsum` calls such as:

```
# schematic: g = (Ax kron Ay kron Az) f
tmp = jnp.einsum("ia,ajk->ijk", Ax, f)      # x-mode product
tmp = jnp.einsum("jb,ibk->ijk", Ay, tmp)    # y-mode product
g   = jnp.einsum("kc,ijc->ijk", Az, tmp)    # z-mode product
```

"Static shapes" means that the sizes p, p', Q are fixed for a run; this allows XLA to plan memory and fuse kernels aggressively.

9.4 How the script uses JAX transformations

`jax.jit`: compile the RHS and the stepper. The JAX backend wraps the expensive kernels in `jax.jit`. Once compiled, subsequent calls run at steady-state speed. This is why the test suite reports both compile time and per-call time.

`vmap`: batch over quadrature nodes. The SOE separation introduces a sum over quadrature nodes $q = 1, \ldots, Q$. In the JAX RHS, the work for all q is expressed as a batched computation using `vmap`, and then reduced by a weighted sum. This removes Python-level loops over q while keeping the math identical.

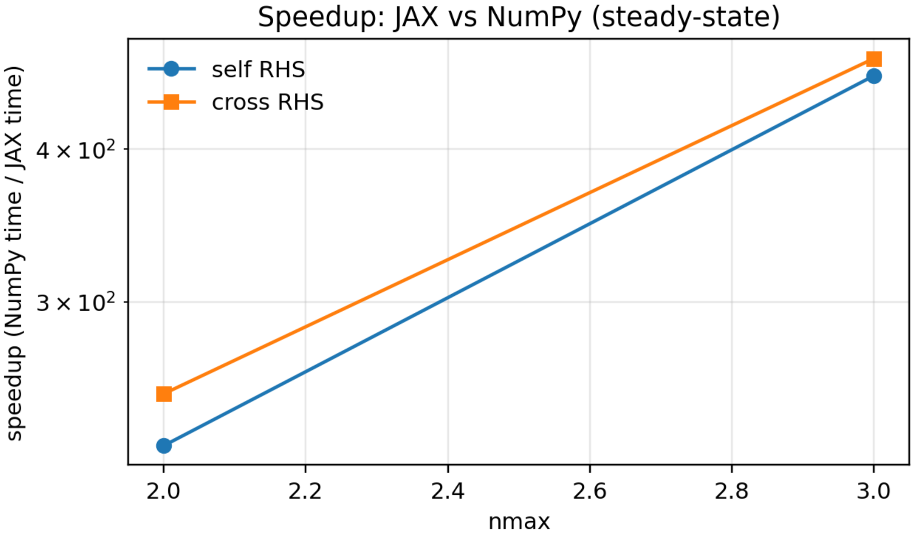


Figure 1: Measured speedup (NumPy time / JAX time) for per-call RHS evaluation at the $n_{\max}$ where NumPy comparisons are enabled. The speedup comes primarily from JIT compilation and reduced Python overhead.

`lax.scan`: compile the time loop. Time integration is a perfect match for `lax.scan`: it turns "for step in range(N)" into a single compiled loop. That reduces overhead and improves fusion opportunities.

9.5 JAX building blocks used here

Constants as compiled inputs. The script builds all coefficient tables once per run (depending on $n_{\max}, Q,$ `maxK` and the species pair). In the JAX path these arrays are transferred once and then treated as constants by the compiled kernels. This avoids repeated allocations and helps XLA reuse memory.

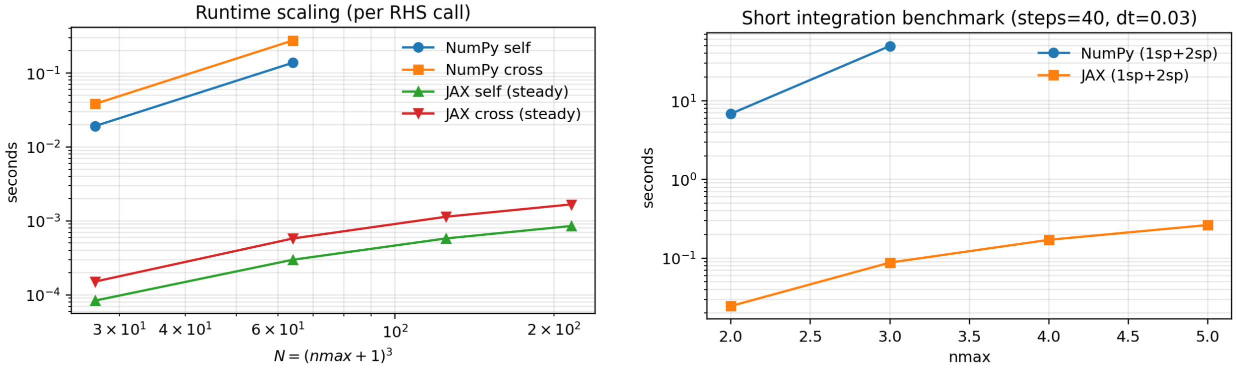
Fusion and dispatch overhead. Many parts of the RHS are "small" elementwise operations surrounding large contractions. In NumPy these would be multiple separate kernel launches and Python dispatches. XLA can fuse chains of elementwise ops into the surrounding contraction kernels, reducing memory traffic and call overhead.

9.6 How much does JAX reduce the cost? (measured in this repository)

The `--run_tests` suite measures both NumPy and JAX steady-state RHS times for small $n_{\max}$ values (NumPy comparisons are disabled at larger $n_{\max}$ because they become slow). The speedups are summarized in Table 1 and in the dedicated speedup plot.

9.7 Why this stays differentiable

All RHS computations in the JAX backend are written in terms of JAX primitives (array operations, `einsum`, `lax` control flow). This means that, in principle, one can differentiate through the RHS and through time stepping using `jax.grad` / `jax.jvp`/`jax.vjp`, subject to the usual considerations (memory cost for reverse-mode through long time horizons, stiffness constraints, etc.). The script



(a) RHS steady-state scaling.

(b) Short integration benchmark.

Figure 2: Performance figures generated by `--run_tests`. These plots are anchored to the exact kernels used in the main figures.

is careful to avoid Python-side control flow inside jitted regions, which is a common pitfall for differentiability.

9.8 Citations and context

For the JAX programming model and transformations, see the JAX reference [20]. JAX relies on the XLA compiler to generate optimized machine code for array programs; XLA is the compilation technology that enables kernel fusion and loop lowering used implicitly by `jax.jit` and `lax.scan` [21].

10 Initial conditions used in Fig. 1

10.1 Fig. 1: one-species IC (positivity-safe)

The script supports three one-species IC families via `--fig1_ic`:

- `poly4` (default): a strongly non-Maxwellian, globally nonnegative "Maxwellian × positive polynomial" IC that excites modes up to $n = 4$ (for $n_{\max} \geq 4$).
- `twostream`: a two-stream-like mixture designed to be nonnegative in physical space; it is projected to the truncated basis and optionally "high-mode damped" to remain nonnegative on diagnostic grids.
- `prl_m2`: the original low-order anisotropy IC (only second moments) used in the reference PRL panel logic.

Equilibrium Maxwellian in this basis. In the normalized coordinate $(x, y, z) = \boldsymbol{v}/v_{\text{th}}$, the equilibrium Maxwellian used by this script is simply the 3D tensor-product ground state

$$M_0(x, y, z) = \psi_0(x)\psi_0(y)\psi_0(z), \tag{39}$$

so a density-normalized Maxwellian is $f(x, y, z) = f_{000}\, M_0(x, y, z)$ with $f_{000} = n/v_{\text{th}}^3$.

`poly4` (default): Maxwellian times a positive polynomial (exact at $n_{\max} \geq 4$). The physical-space target is

$$f_{\text{poly4}}(x,y,z) = Z\,\psi_0(x)\psi_0(y)\psi_0(z)\Big(1 + a_{2x}x^2 + a_{2y}y^2 + a_{2z}z^2 + a_{4x}x^4 + a_{4y}y^4 + a_{4z}z^4\Big), \tag{40}$$

with Z chosen so that $f[0,0,0] = 1/v_{\text{th}}^3$. The coefficients are set *exactly as in the code* (see `build_ic_fig1_1sp_poly4`):

$$\texttt{base2} = 0.55, \qquad \texttt{base4} = 0.32 \ \ (\text{if } n_{\max} \geq 4), \qquad a = \texttt{--amp1}, \tag{41}$$

and then

$$a_{2x} = \texttt{base2}(1 + 0.60\,a), \quad a_{2y} = a_{2z} = \texttt{base2}(1 - 0.30\,a), \tag{42}$$

with the same pattern for the quartic coefficients a_{4x}, a_{4y}, a_{4z}. Because all polynomial coefficients are nonnegative, the target shape is globally nonnegative in physical space, and for $n_{\max} \geq 4$ this IC is exactly representable in the truncated Hermite basis.

`twostream`: shifted-Maxwellian mixture projected to the truncated basis. The *physical-space* target shape is the even mixture

$$g_u(x) = \tfrac{1}{2}\Big(\psi_0(x-u) + \psi_0(x+u)\Big), \tag{43}$$

with $u = \texttt{--amp1}$ (stream separation in v_x/v_{th}). The 3D IC is then

$$f_{\text{ts}}(x,y,z) = \left(\sum_{n=0}^{n_{\max}} c_n\,\psi_n(x)\right)\psi_0(y)\psi_0(z), \tag{44}$$

where the coefficients c_n are obtained by a weighted L^2 projection of g_u onto $\text{span}\{\psi_0, \dots, \psi_{n_{\max}}\}$ on a dense grid (see `build_ic_fig1_1sp_twostream`). Finally, the coefficients are scaled so that the resulting tensor satisfies

$$f[0,0,0] = \frac{1}{v_{\text{th}}^3} \quad \text{(density normalization used by this script).} \tag{45}$$

Optionally (default on), the script enforces nonnegativity on diagnostic grids by writing $f = f_M + \gamma(f - f_M)$ with f_M the Maxwellian having the same f_{000} and choosing the largest $\gamma \in [0,1]$ such that the slice $f(v_x, 0, 0)$ and plane $f(v_x, v_y, 0)$ remain nonnegative on the diagnostic grids.

`prl_m2`: low-order anisotropy IC (exact in coefficient space). When `--fig1_ic prl_m2` is selected, the IC is set directly in coefficient space:

$$f[0,0,0] = \frac{1}{v_{\text{th}}^3}, \tag{46}$$

$$f[2,0,0] = 2a, \qquad f[0,2,0] = -a, \qquad f[0,0,2] = -a, \tag{47}$$

with $a = \texttt{--amp1}$ and all other coefficients zero.

10.2 Fig. 1: two-species IC (positivity-safe, strongly non-Maxwellian)

The two-species IC is designed to produce a clean temperature-exchange relaxation while keeping both species noticeably non-Maxwellian *and* nonnegative. Both species are constructed as

$$f(x, y, z) = Z\,\psi_0(x)\psi_0(y)\psi_0(z)\Big(1 + a_2r^2 + a_4r^4\Big), \qquad r^2 = x^2 + y^2 + z^2, \tag{48}$$

with Z chosen so that $f[0, 0, 0] = 1/v_{\mathrm{th}}^3$.

For $n_{\max} \geq 4$ the script uses fixed quartic strengths (see `build_ic_fig1_2sp`):

$$a_{4,\mathrm{hot}} = 0.14, \qquad a_{4,\mathrm{cool}} = 0.08, \tag{49}$$

and then tunes the quadratic coefficient:

- the *hot* species uses $a_2 \geq 0$ (bisection) so that its temperature matches $T_{\mathrm{eq}} + |$`--dT2`$|$;
- the *cool* species uses $a_2 \leq 0$ (bisection) to keep its temperature near T_{eq} while maintaining global positivity. For $a_4 > 0$, global positivity for all r is ensured by enforcing $a_2^2 \leq 4a_4$.

11 Figures produced by the script

11.1 Main panels

11.2 Test suite summary and plots

The `--run_tests` mode runs a sweep over $n_{\max}$, checks Maxwellian fixed points, cross-invariant rates, finite-difference consistency of the linearization (when feasible), and short-run performance. It also performs physics-style checks of monotonicity (nonlinear KL entropy and linear quadratic free energy) and generates convergence sweeps in (Q, `maxK`).

Interpreting the test outputs. The tests are designed around standard invariants and stability properties of the Landau operator:

- **Maxwellian fixed point**: $RHS(M) \approx 0$ is a strict algebraic property of the collision operator. In floating point, the residual should be near roundoff if the discretization/tables are correct.
- **Conservation**: for cross-collisions, Q_{ab} and Q_{ba} exchange momentum/energy but must conserve totals. The test plots $|d/dt\,(n_{\mathrm{tot}}, P_{\mathrm{tot}}, W_{\mathrm{tot}})|$ computed directly from the RHS.
- **H-theorem diagnostics**: nonlinear KL entropy to the instantaneous local Maxwellian should be non-increasing (up to truncation/time-discretization). Linearized dynamics should instead be monitored with quadratic free energy about the fixed Maxwellian background.
- **Linearized operator structure**: the linearized Landau operator about a Maxwellian is (formally) self-adjoint and dissipative in a Maxwellian-weighted inner product, once the nullspace (collision invariants) is removed; the test suite includes an approximate symmetry/dissipation check on a velocity grid [4, 3].
- **Maxwellians with varied densities**: multispecies equilibrium allows different densities; the test suite verifies $Q_{ab}(M_a, M_b) \approx 0$ when M_a, M_b differ only by density factors.
- **Galilean invariance vs truncation**: a drifted Maxwellian requires infinitely many Hermite modes. The test suite therefore checks *convergence* of the drifted-Maxwellian residual as $n_{\max}$ increases, rather than expecting exact zero at fixed truncation.
- **Linearization correctness**: two complementary checks are used: (i) finite-difference self-consistency (NumPy, small $n_{\max}$) and (ii) small-perturbation agreement between nonlinear and tangent-linear time traces (JAX).
- **Performance**: steady-state RHS timing (after compilation) and a short integration benchmark.

Table 1: Test sweep summary (JAX backend). Reported times are steady-state. Conservation rates are for cross-collision totals $d/dt\,(n, P, W)$ evaluated on a random near-Maxwellian state.

$n_{\max}$	$N = (n_{\max}+1)^3$	t_{self} [ms]	t_{cross} [ms]	t_{int} [s]	$\text{speedup}_{\text{self}}$	$\text{speedup}_{\text{cross}}$	$\|RHS(M)\|/\|M\|$ (cross)
2	27	0.084	0.151	0.025	228.8×	252.3×	4.23e-16
3	64	0.298	0.577	0.088	459.4×	474.6×	4.35e-16
4	125	0.578	1.136	0.171	–	–	5.98e-16
5	216	0.853	1.668	0.263	–	–	6.00e-16

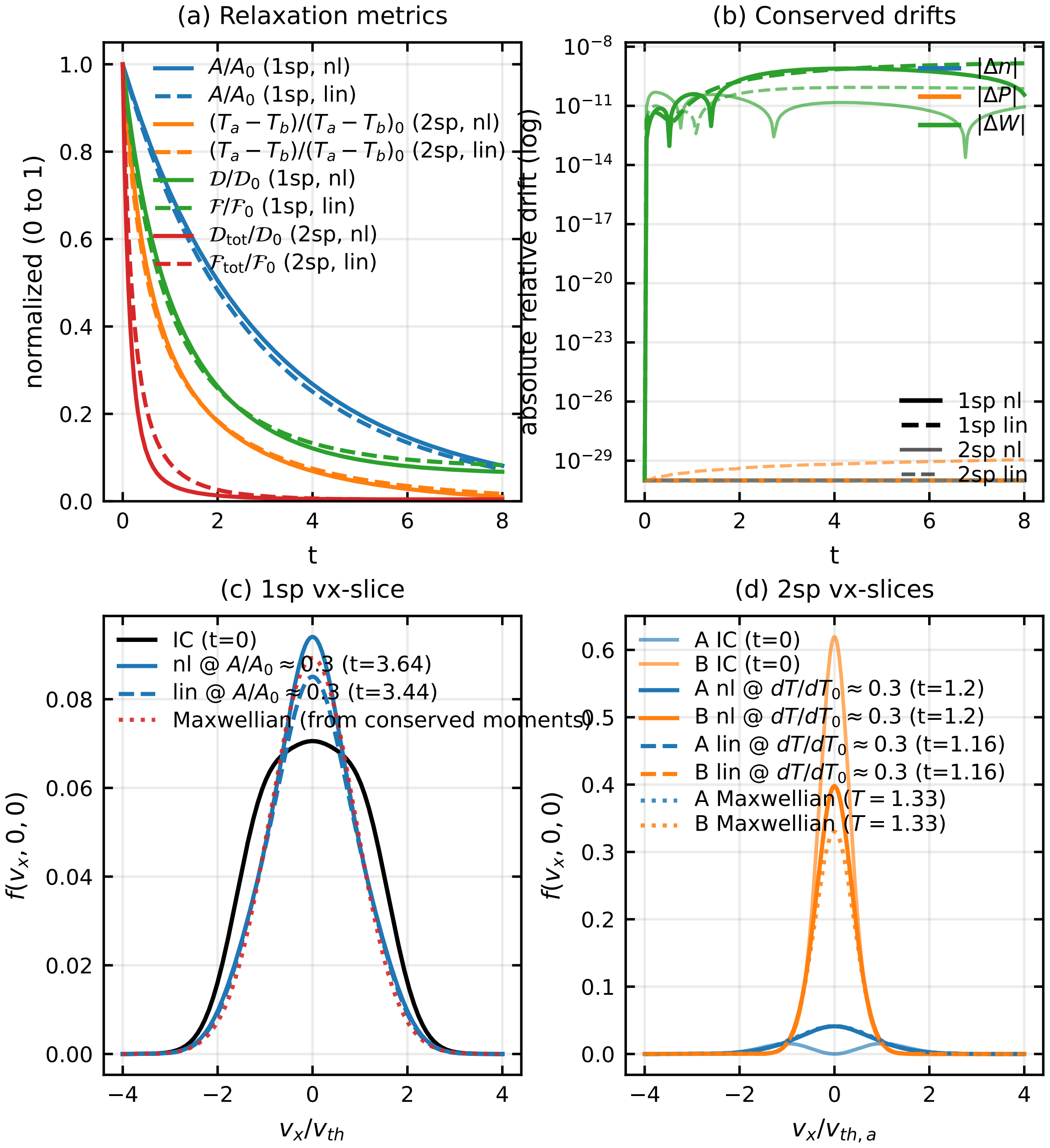


Figure 3: Fig. 1 panel produced by `landau_hermite_jax.py`.

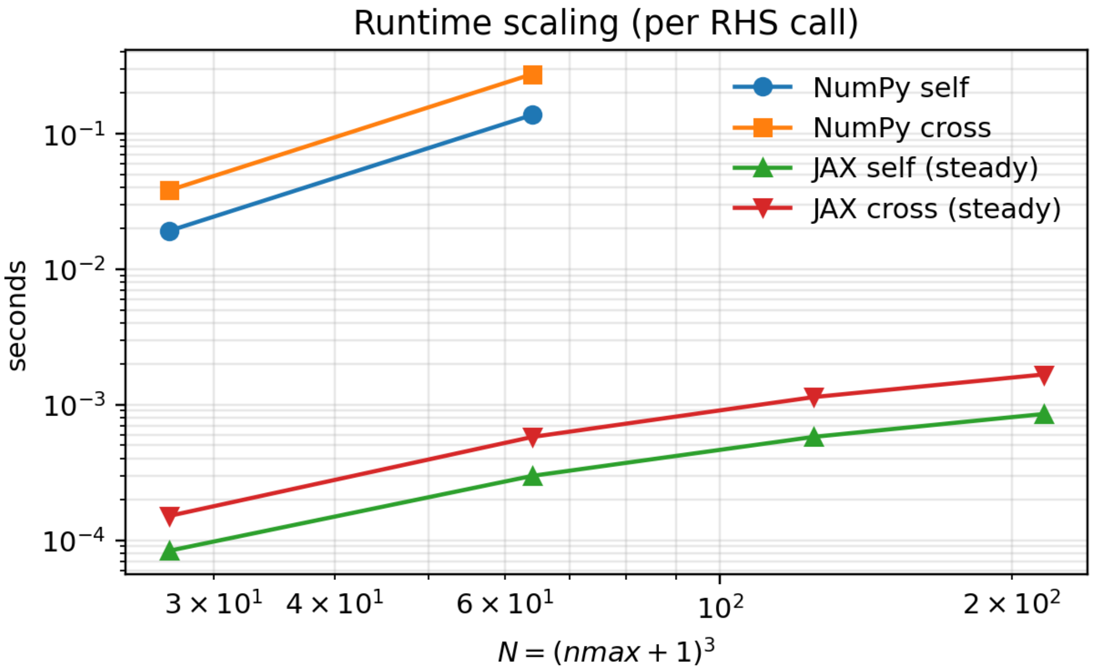


Figure 4: Per-RHS-call runtime scaling across $n_{\max}$ (from the latest test run).

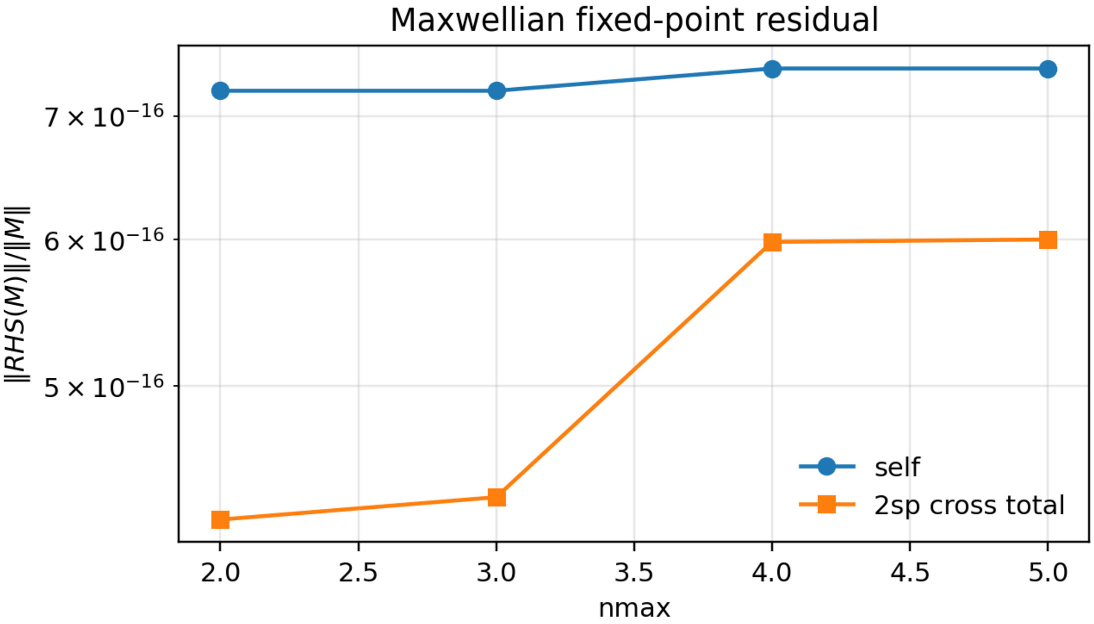


Figure 5: Maxwellian fixed-point residual $\|RHS(M)\|/\|M\|$ across $n_{\max}$.

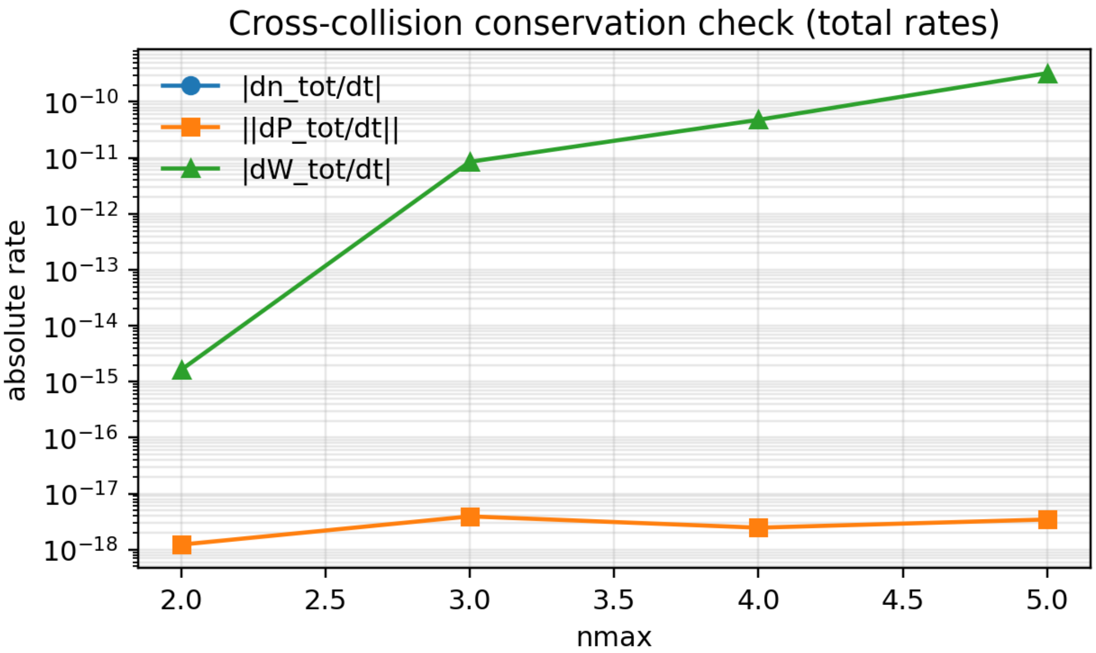


Figure 6: Cross-collision conservation-rate check: total invariant rates computed from the RHS on a random near-Maxwellian state.

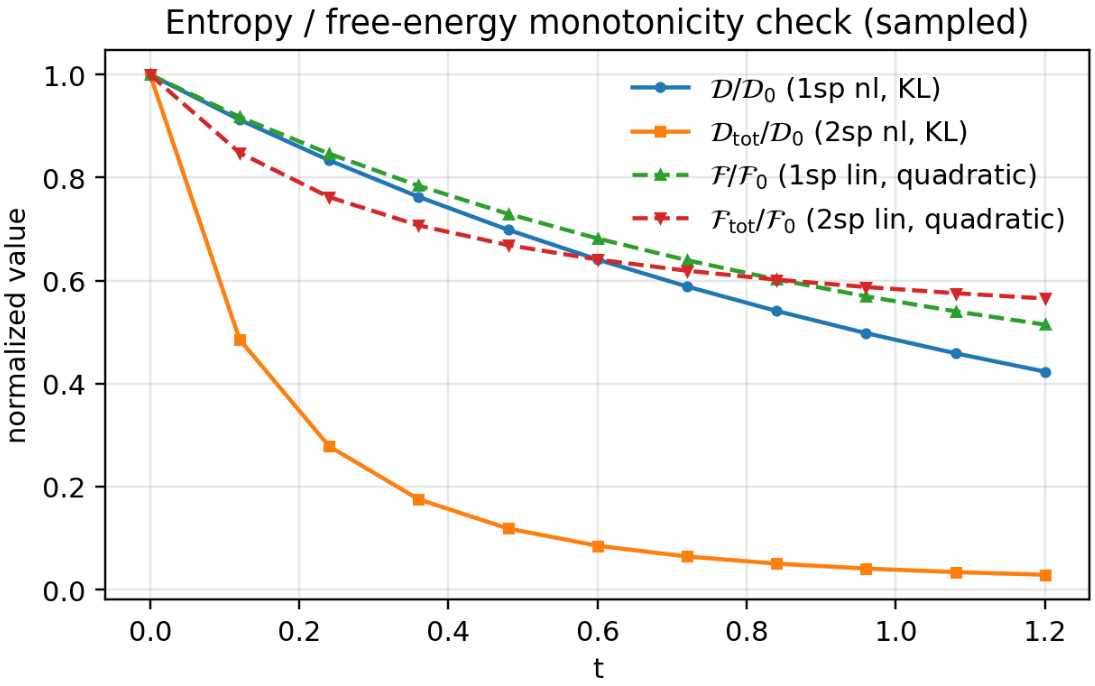


Figure 7: Physics-style monotonicity check: nonlinear KL entropy to instantaneous local Maxwellians (solid) and linearized quadratic free energy about fixed Maxwellians (dashed).

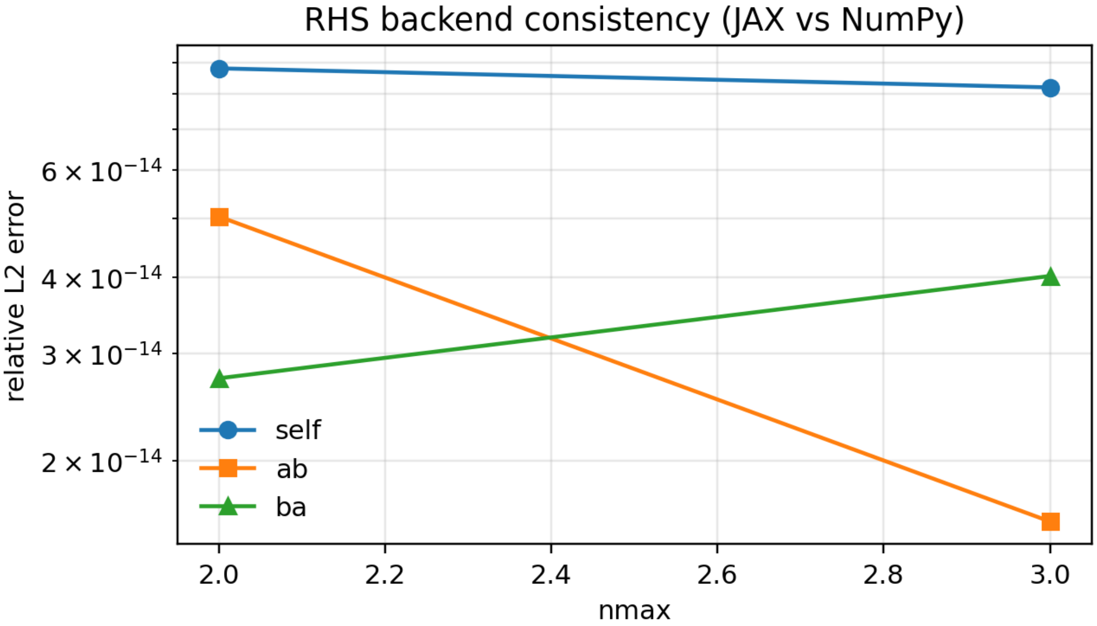


Figure 8: Backend consistency between NumPy and JAX for small $n_{\max}$ where NumPy comparisons are enabled.

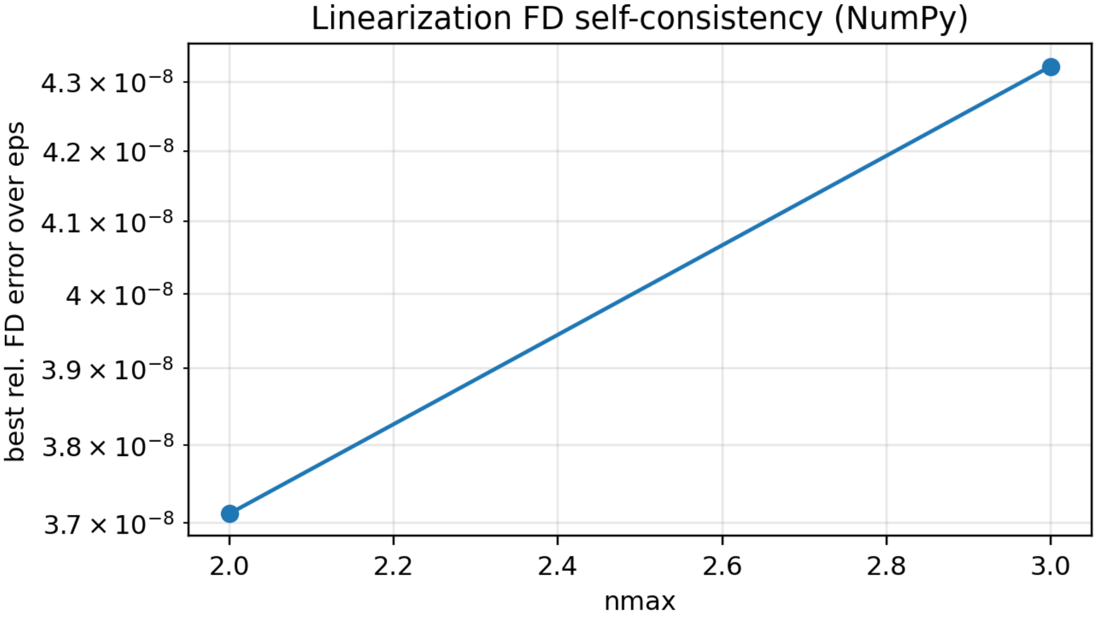


Figure 9: Finite-difference self-consistency check of the linearization (NumPy, where enabled).

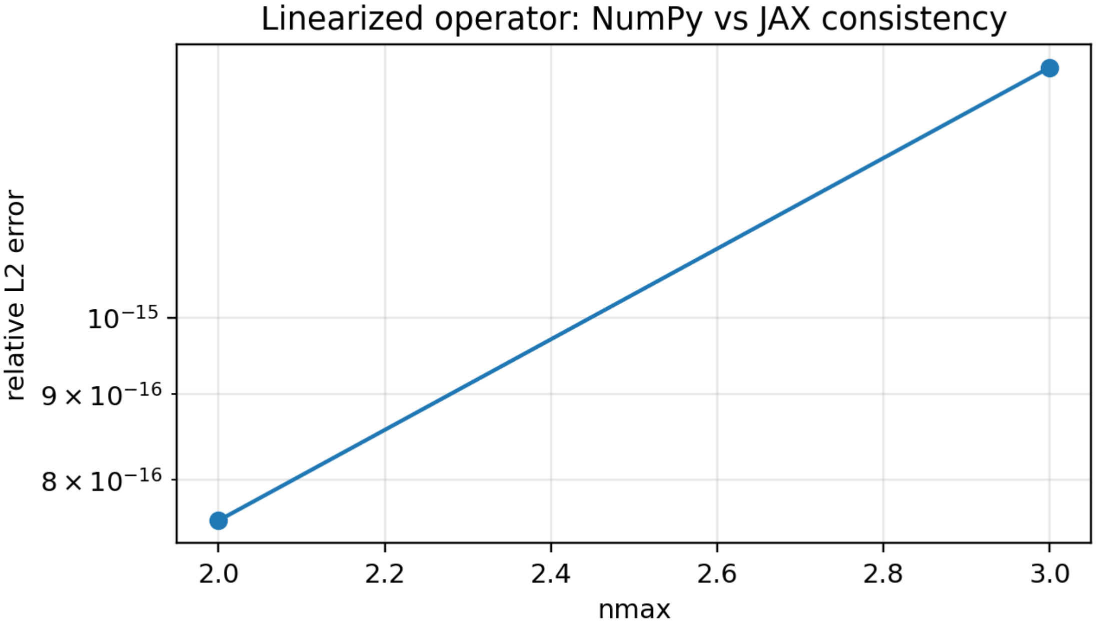


Figure 10: Linearized operator consistency between NumPy (explicit linearization) and JAX (JVP-based linearization), where enabled.

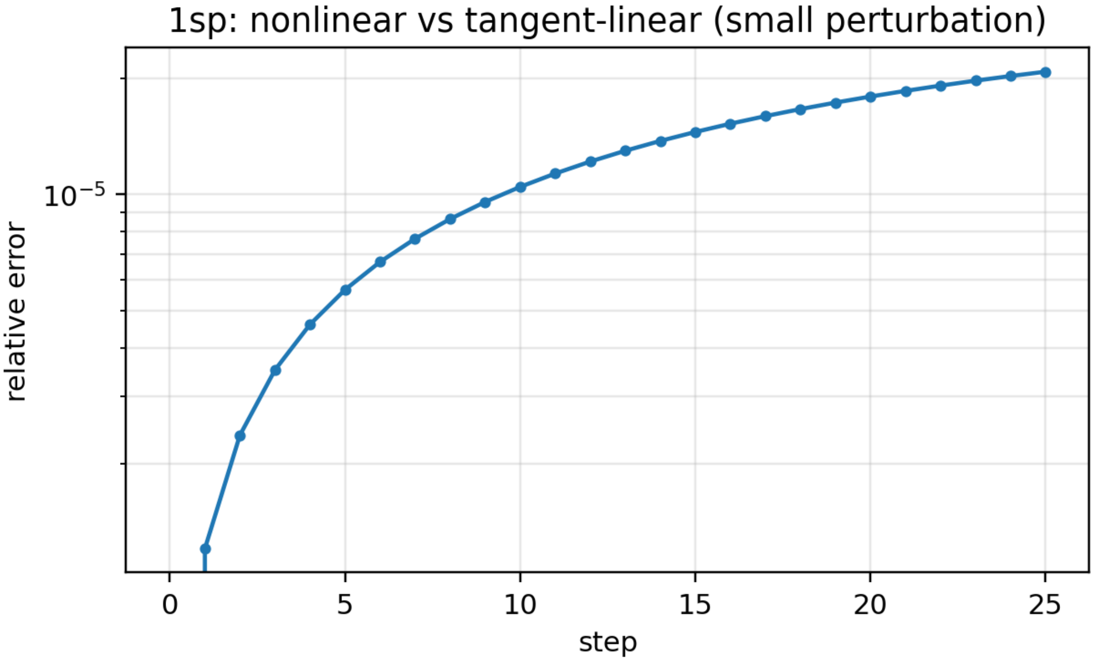


Figure 11: Small-perturbation check: nonlinear vs tangent-linear for the 1-species problem.

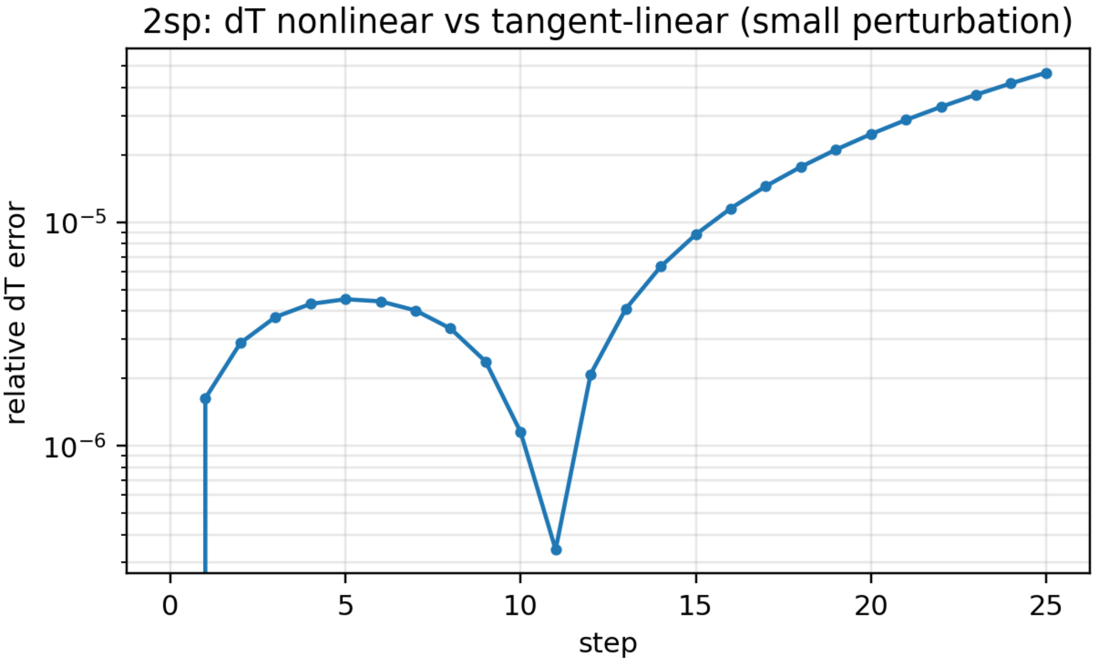


Figure 12: Small-perturbation check: nonlinear vs tangent-linear for the 2-species temperature-exchange metric.

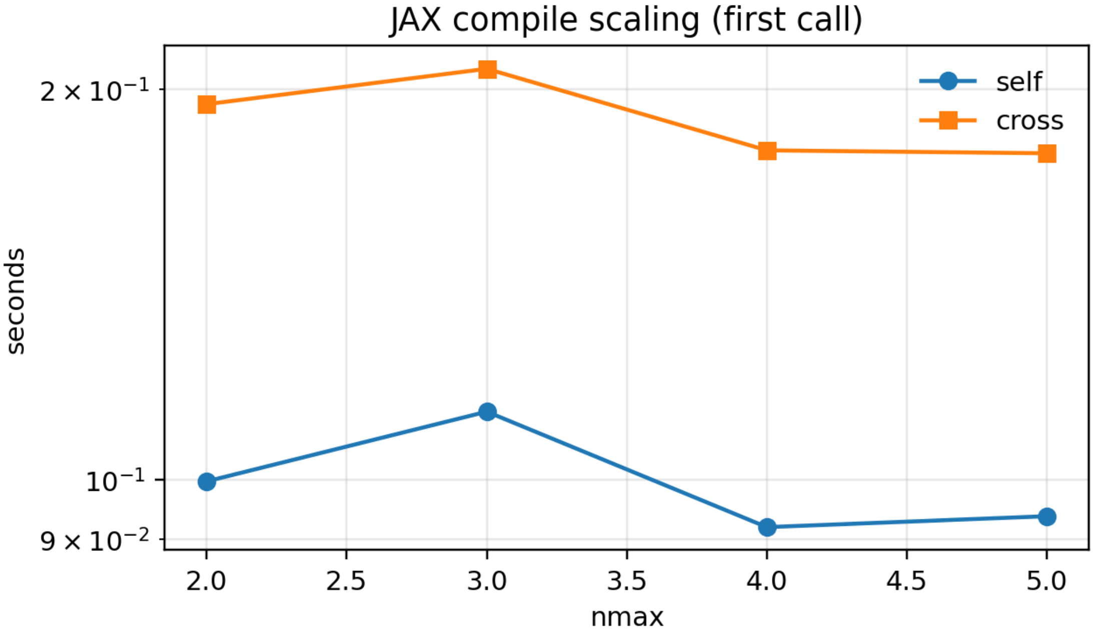


Figure 13: JAX compile-time scaling across n_{max} (first call).

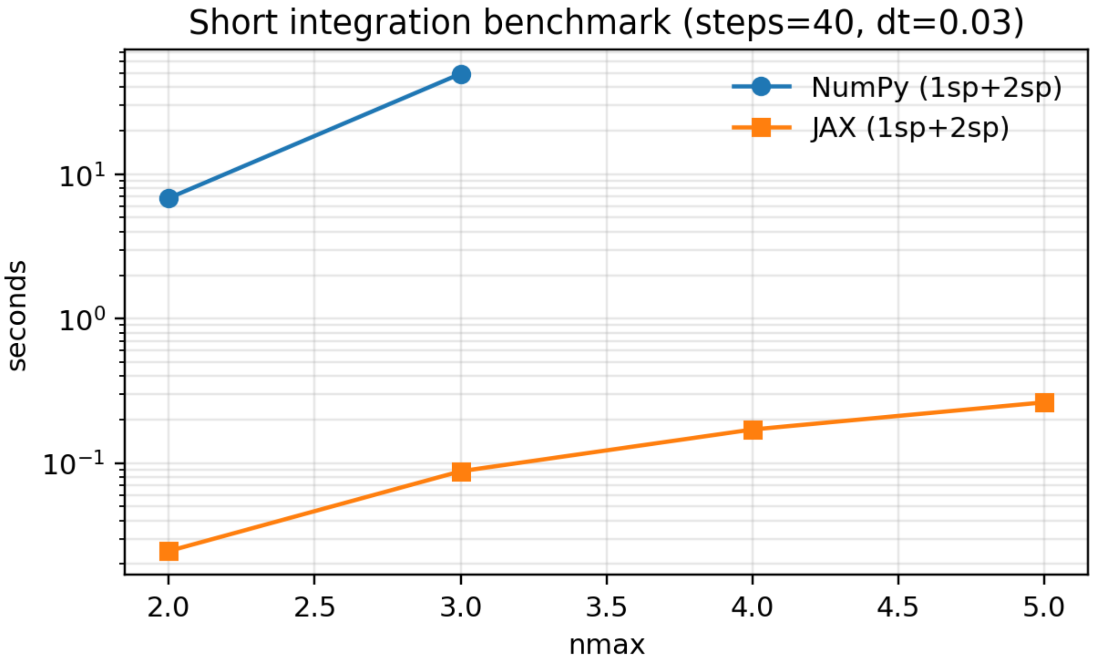


Figure 14: Short integration benchmark (NumPy vs JAX where enabled).

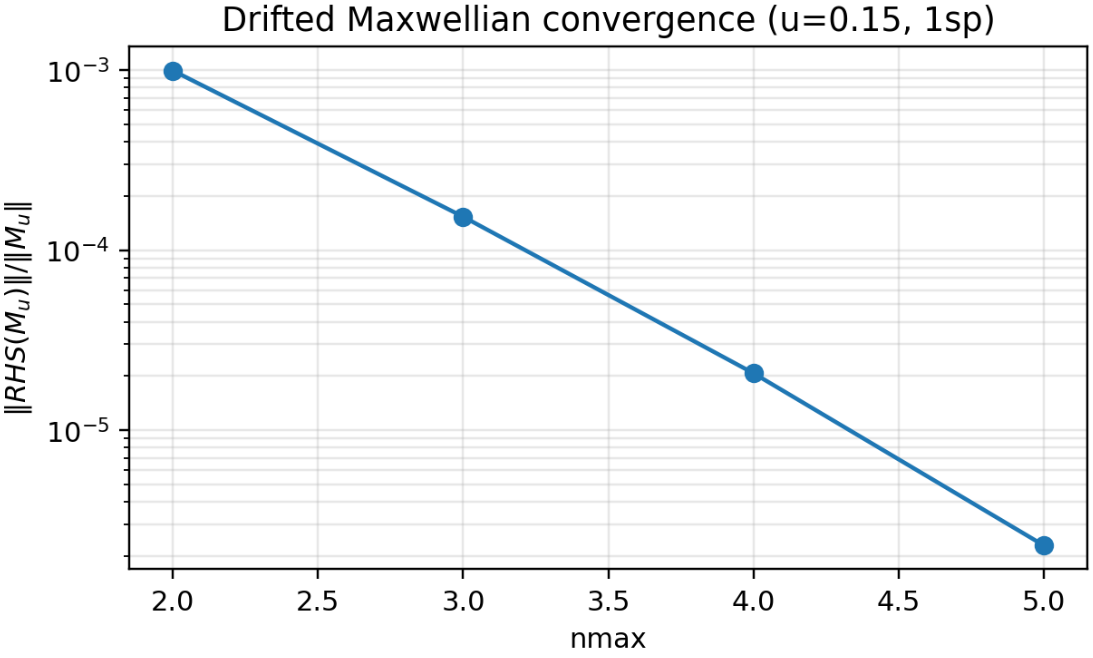


Figure 15: Drifted Maxwellian residual versus truncation: at fixed small drift u, the residual decreases as $n_{\max}$ increases. This is a practical proxy for Galilean invariance under spectral truncation.

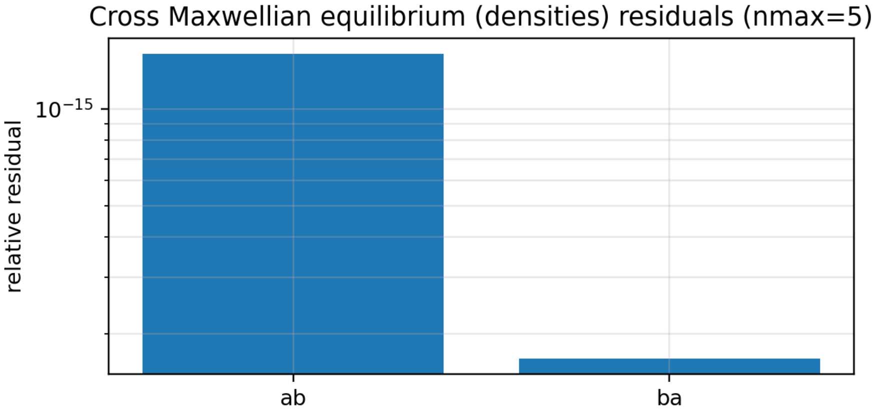


Figure 16: Two-species equilibrium check with different densities: $Q_{ab}(M_a, M_b) \approx 0$ and $Q_{ba}(M_b, M_a) \approx 0$ for Maxwellians with shared temperature/velocity but different density prefactors.

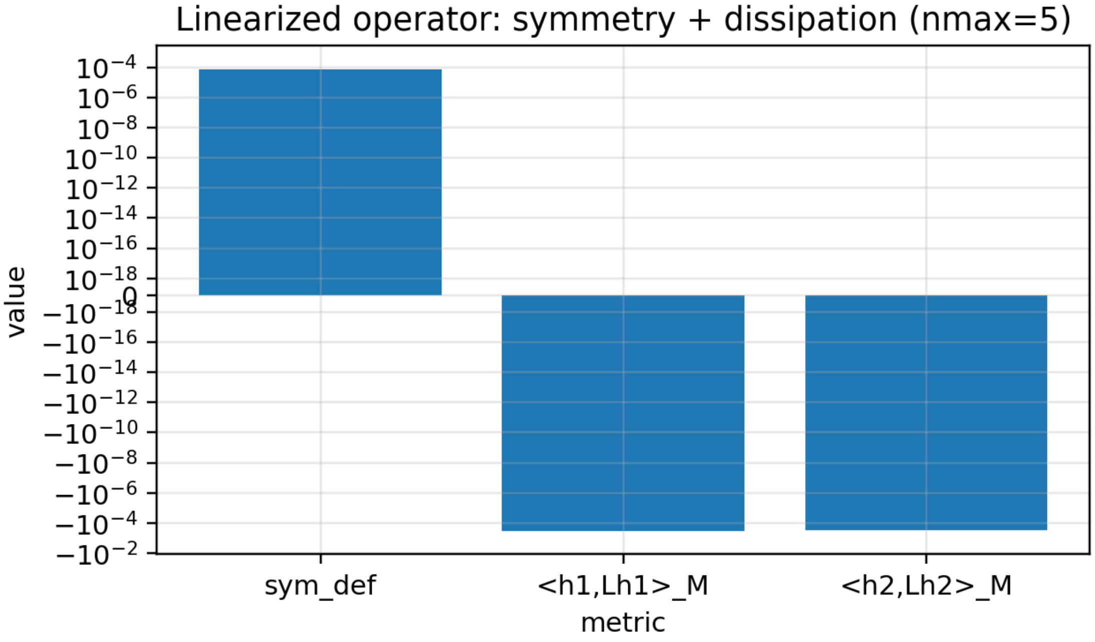


Figure 17: Linearized operator structure check (approximate, grid-based): symmetry defect in the weighted inner product and weighted dissipation values $\langle h, Lh \rangle_M$ for two random perturbations with collision-invariant components removed.

12 Interpreting results and limitations

12.1 Positivity

Hermite spectral truncations do *not* guarantee pointwise positivity of the reconstructed distribution function, even if the underlying physical solution is nonnegative. This is a well-known spectral/Gibbs phenomenon. To keep the plotted slices physically meaningful, the default Fig. 1 one-species IC is built from a manifestly nonnegative two-stream mixture and is optionally "damped" in high modes to remain nonnegative on diagnostic grids.

12.2 Monotonicity

For the nonlinear operator, $\mathcal{D}(t)$ (KL divergence to the instantaneous local Maxwellian) is the referee-proof diagnostic. For the linearized evolution, the correct monotone quantity is the quadratic free energy $\mathcal{F}(t; M)$ about the fixed Maxwellian background, not KL to a time-dependent Maxwellian.

12.3 Truncation and parameter choices

The separation accuracy is controlled by:

- $n_{\max}$: Hermite resolution (spectral truncation),
- Q: quadrature nodes in the SOE separation,
- `maxK`: maximum Hankel index in the Coulomb moment tables.

The test suite includes Q/`maxK` sweeps and Maxwellian fixed-point residuals to help select safe defaults.

Practical note on Q at larger $n_{\max}$. For moderate truncations ($n_{\max} \leq 5$), $Q = 8$ is typically sufficient to achieve Maxwellian fixed-point residuals near roundoff in double precision. As $n_{\max}$ increases, however, the quadrature error can become visible unless Q is increased. For this reason, the script contains a simple heuristic auto-selection for Q at larger $n_{\max}$ (enabled by default and disable-able via `--no_auto_Q`). The `--run_tests` suite also includes a Q-sweep plot to make this dependence explicit.

12.4 Other fast Landau algorithms in the literature (future directions)

The present script focuses on a Hermite basis and a separated (SOE) evaluation that is well matched to tensor-product contractions. There are also alternative fast strategies for the Landau operator:

- **FFT-based fast spectral methods:** represent f on a uniform velocity grid and evaluate the collision operator using Fourier techniques to reduce the cost of the nonlocal convolution-like structures (e.g. Pareschi–Russo–Toscani [14] and follow-ups). These methods can achieve $\mathcal{O}(N \log N)$-type costs in grid settings.
- **Hermite with local basis adaptation:** recenter/rescale the Hermite basis using local Maxwellian parameters (velocity shift and temperature) to reduce the number of modes needed to represent far-from-equilibrium states (e.g. the local-approximation Hermite strategy in [15]). This is a promising way to push to higher effective resolution without increasing global $n_{\max}$ as much.
- **Potential-based solvers:** compute Rosenbluth potentials via Poisson solves in velocity space, then apply drift/diffusion operators. In Fourier bases, Poisson solves are diagonal; in Hermite bases, they are structured but not diagonal, suggesting specialized solvers might help.

These directions are not implemented in this standalone script (which intentionally stays within one file and one numerical framework), but the test suite and modular structure are designed to make comparisons feasible.

13 References (selected)

References

[1] L. D. Landau, “Kinetic equation for the Coulomb effect,” *Physikalische Zeitschrift der Sowjetunion* (1936).

[2] M. N. Rosenbluth, W. M. MacDonald, and D. L. Judd, “Fokker–Planck equation for an inverse-square force,” *Physical Review* **107** (1957).

[3] P. Helander and D. J. Sigmar, *Collisional Transport in Magnetized Plasmas*, Cambridge Univ. Press (2002).

[4] C. Villani, “A review of mathematical topics in collisional kinetic theory,” in *Handbook of Mathematical Fluid Dynamics*, Vol. I (2002).

[5] C.-W. Shu and S. Osher, “Efficient implementation of essentially non-oscillatory shock-capturing schemes,” *Journal of Computational Physics* **77** (1988).

[6] S. Gottlieb, C.-W. Shu, and E. Tadmor, “Strong stability-preserving high-order time discretization methods,” *SIAM Review* **43** (2001).

[7] I. V. Oseledets, “Tensor-train decomposition,” *SIAM Journal on Scientific Computing* **33** (2011).

[8] W. Hackbusch, *Tensor Spaces and Numerical Tensor Calculus*, Springer (2012).

[9] U. Schollwöck, “The density-matrix renormalization group in the age of matrix product states,” *Annals of Physics* **326** (2011).

[10] R. Orús, “A practical introduction to tensor networks: Matrix product states and projected entangled pair states,” *Annals of Physics* **349** (2014).

[11] G. H. Golub and J. H. Welsch, “Calculation of Gauss quadrature rules,” *Mathematics of Computation* **23** (1969).

[12] J. P. Boyd, *Chebyshev and Fourier Spectral Methods*, 2nd ed., Dover (2001). (Background on spectral truncation/oscillations.)

[13] H. Grad, “On the kinetic theory of rarefied gases,” *Communications on Pure and Applied Mathematics* **2** (1949).

[14] L. Pareschi, G. Russo, and G. Toscani, “Fast spectral methods for the Fokker–Planck–Landau collision operator,” *Journal of Computational Physics* **165** (2000).

[15] R. Li, Y. Ren, and Y. Wang, “A Hermite spectral method for the spatially homogeneous Landau equation based on local approximation,” *Journal of Computational Physics* **434** (2021), 110235. (See also arXiv:2004.06484.)

[16] G. Beylkin and L. Monzón, “Approximation by exponential sums revisited,” *Applied and Computational Harmonic Analysis* (2005).

[17] W. Hackbusch and B. N. Khoromskij, “Low-rank Kronecker-product approximation to multi-dimensional nonlocal operators. Part I. Separable approximation of multi-variate functions,” *Computing* **76** (2006).

[18] W. Hackbusch and B. N. Khoromskij, “Low-rank Kronecker-product approximation to multidimensional nonlocal operators. Part II. HKT representation of certain operators,” *Computing* **76** (2006).

[19] R. M. Parrish, E. G. Hohenstein, T. J. Martínez, and C. D. Sherrill, “Tensor hypercontraction. I. Least-squares tensor hypercontraction for the electron repulsion integral tensor,” *J. Chem. Phys.* **137** (2012).

[20] J. Bradbury et al., “JAX: composable transformations of Python+NumPy programs,” (2018). https://github.com/jax-ml/jax

[21] Google, “XLA: Accelerated Linear Algebra,” documentation/reference for the XLA optimizing compiler used by JAX/TF. https://www.tensorflow.org/xla

14 Appendix: full CLI help

For completeness, this is the exact output of `python landau_hermite_jax.py -h` captured at the time this companion was generated:

```
usage: landau_hermite_jax.py [-h] [--backend {jax,numpy}] [--nmax NMAX]
                             [--Q Q] [--maxK MAXK]
                             [--integrator {rk2,ssprk3,rk4}]
                             [--linearized {on,off}]
                             [--linearized_method {tangent,matrix}]
                             [--progress_chunks PROGRESS_CHUNKS] [--quiet]
                             [--use_tt] [--tt_tol TT_TOL] [--tt_rmax TT_RMAX]
                             [--dt_1sp DT_1SP] [--dt_2sp DT_2SP]
                             [--tmax_1sp TMAX_1SP] [--tmax_2sp TMAX_2SP]
                             [--steps_1sp STEPS_1SP] [--steps_2sp STEPS_2SP]
                             [--fig1_ic {prl_m2,twostream}]
                             [--no_enforce_nonneg_ic] [--amp1 AMP1]
                             [--dT2 DT2] [--outprefix_fig1 OUTPREFIX_FIG1]
                             [--outprefix_fig2 OUTPREFIX_FIG2]
                             [--outprefix OUTPREFIX] [--skip_fig1]
                             [--skip_fig2] [--fig2_strength FIG2_STRENGTH]
                             [--seed SEED] [--entropy_nx ENTROPY_NX]
                             [--entropy_xlim ENTROPY_XLIM] [--run_tests]
                             [--tests_outdir TESTS_OUTDIR]
                             [--tests_nmax_list TESTS_NMAX_LIST]
                             [--tests_reps_rhs TESTS_REPS_RHS]
                             [--tests_reps_bench TESTS_REPS_BENCH]
                             [--tests_steps TESTS_STEPS] [--tests_dt TESTS_DT]
                             [--tests_entropy_nx TESTS_ENTROPY_NX]
                             [--tests_entropy_xlim TESTS_ENTROPY_XLIM]
                             [--tests_Q_sweep TESTS_Q_SWEEP]
                             [--tests_maxK_sweep TESTS_MAXK_SWEEP]
                             [--tests_max_numpy_nmax TESTS_MAX_NUMPY_NMAX]

Standalone fast SOE→MPO/TT Landau-Hermite (JAX-first): Fig1 + Fig2 panels.

options:
  -h, --help            show this help message and exit
  --backend {jax,numpy}
  --nmax NMAX
  --Q Q
  --maxK MAXK
  --integrator {rk2,ssprk3,rk4}
  --linearized {on,off}
                        Include linearized (Maxwellian-background) overlays.
  --linearized_method {tangent,matrix}
                        How to compute linearized evolution (default: matrix-
                        free tangent linear).
  --progress_chunks PROGRESS_CHUNKS
                        If >0, run JAX time stepping in fixed-size chunks and
                        print progress per chunk.
  --quiet               Reduce prints.
  --use_tt              Use optional TT/MPO contraction in NumPy backend
                        (debug/scaling).
  --tt_tol TT_TOL
  --tt_rmax TT_RMAX
  --dt_1sp DT_1SP
  --dt_2sp DT_2SP
  --tmax_1sp TMAX_1SP
  --tmax_2sp TMAX_2SP
  --steps_1sp STEPS_1SP
  --steps_2sp STEPS_2SP
  --fig1_ic {prl_m2,twostream}
                        Fig1 IC family for the 1-species case.
  --no_enforce_nonneg_ic
                        Disable IC nonnegativity enforcement on diagnostic
                        grids (not recommended at low nmax).
  --amp1 AMP1           Fig1 1sp control: for prl_m2 it's the 2nd-moment
                        anisotropy amplitude; for twostream it's the stream
                        separation u in vx/vth.
  --dT2 DT2             Target temperature excess for the hot species in 2sp
                        ICs (positivity-safe).
  --outprefix_fig1 OUTPREFIX_FIG1
  --outprefix_fig2 OUTPREFIX_FIG2
  --outprefix OUTPREFIX
                        (compat) Alias for --outprefix_fig1; if
                        --outprefix_fig2 is default, set it to
                        OUTPREFIX+'_Fig2'.
  --skip_fig1
  --skip_fig2
  --fig2_strength FIG2_STRENGTH
                        Scales the positive polynomial distortion used for the
                        Fig2 initial condition (larger => farther from
                        Maxwellian).
  --seed SEED
  --entropy_nx ENTROPY_NX
```

```
--entropy_xlim ENTROPY_XLIM
--run_tests           Run internal correctness/performance tests and write
                      plots into tests_landau_hermite/.
--tests_outdir TESTS_OUTDIR
--tests_nmax_list TESTS_NMAX_LIST
--tests_reps_rhs TESTS_REPS_RHS
--tests_reps_bench TESTS_REPS_BENCH
--tests_steps TESTS_STEPS
--tests_dt TESTS_DT
--tests_entropy_nx TESTS_ENTROPY_NX
                      Entropy grid resolution used in --run_tests physics
                      checks (smaller is faster).
--tests_entropy_xlim TESTS_ENTROPY_XLIM
                      Entropy grid half-width used in --run_tests physics
                      checks.
--tests_Q_sweep TESTS_Q_SWEEP
                      Comma-separated Q values for convergence sweeps in
                      --run_tests.
--tests_maxK_sweep TESTS_MAXK_SWEEP
                      Comma-separated maxK values for convergence sweeps in
                      --run_tests.
--tests_max_numpy_nmax TESTS_MAX_NUMPY_NMAX
                      In --run_tests, only do NumPy-vs-JAX comparisons up to
                      this nmax (NumPy gets slow for larger nmax).
```