gyaradax: Local Gyrokinetics JAX Code

Gianluca Galletti [*1] **Eric Volkmann** [*1] **Johannes Brandstetter** [1 2]

Abstract

Gyrokinetic simulations are essential for understanding and controlling turbulence in fusion plasmas, yet they are oftentimes implemented in legacy codebases, in many cases CPU-bound. These are both hard to maintain and especially incompatible with optimization and ML workflows. gyaradax is a minimal JAX/CUDA solver for local flux-tube gyrokinetics. We base our implementation on GKW (Peeters et al., 2009), but with added native GPU acceleration and automatic differentiation. We validate gyaradax against analytical cases and empirical benchmarks, achieving formal agreement and statistical parity with GKW alongside a substantial speedup. We deliberately and extensively utilized agentic workflows in this project. A key contribution is showing that coding agents, guided by human expertise, structured prompting, and measurable progress through unit testing enabled extremely fast translation of complex Fortran code, and further optimizations. gyaradax facilitates research at the intersection of ML and plasma physics. We showcase this through practical examples in inverse problems and sensitivity analysis.

1. Introduction

Plasma turbulence is one of the most challenging multiscale problems in computational physics. The community has for decades relied on highly optimized and widely accepted codebases like GKW (Peeters et al., 2009) and GENE (Kotschenreuther & Rogers, 2000). Built for CPU clusters, they rely on complex MPI parallelization that makes them quite rigid. Although newer codes like CGYRO (Candy et al., 2016) and modern versions of GENE offer GPU support, their underlying architecture still lacks the flexibility to easily expand or integrate into ML workflows.

[1]Institute for Machine Learning, JKU Linz, [2]Mistral AI. Correspondence to: Gianluca Galletti <g.galletti@ml.jku.at>.

Proceedings of the 43rd International Conference on Machine Learning, Seoul, South Korea. PMLR 306, 2026. Copyright 2026 by the author(s).

In this work, we present gyaradax, a slim JAX code for local flux-tube gyrokinetics, supporting both adiabatic and kinetic electron models. We pick JAX (Bradbury et al., 2018) following the path of recent works across scientific domains, for example JAXFLUIDS (Bezgin et al., 2023) and JAX-SPH (Toshev et al., 2024) for fluid dynamics, TORAX (Citrin et al., 2024) for fast tokamak transport simulation, and JAX-MD (Schoenholz & Cubuk, 2020) for molecular dynamics. gyaradax bridges the gap between legacy gyrokinetics and the JAX ecosystem, enabling hardware-accelerated automatic differentiation for plasma turbulence research (Paischer et al., 2025; Kelling et al., 2025; Zanisi et al., 2024). A welcome side effect of using JAX is the increased accessibility: the core integrator and field solver are implemented in approximately 3 000 lines of JAX, compared to over 30 000 lines of functionally equivalent Fortran.

We developed gyaradax with the heavy use of coding agents (Yang et al., 2024) and *vibecoding*, applying emerging AI-driven software engineering practices to complex scientific codebases (Chen et al., 2025; Duston et al., 2025). A central challenge in deploying agents is oversight (Bowman et al., 2022): *how can we verify that generated code is correct as systems grow in complexity?* For numerical codebases this issue is naturally dampened, as correctness can be rigorously verified against analytical cases and reference results or implementations. With this strong "reward" signal coupled with (aspiring) experts, we find that coding agents can effectively navigate and implement extensive scientific computing frameworks, freeing resources for more fundamental research.

In this work, we outline the underlying gyrokinetic equations (Section 2), the implementation and optimization of gyaradax (Section 3), and notes on the agent-assisted development process (Section 3.4). We formally verify our solver on the Rosenbluth & Hinton (1998) zonal flow test and on the Cyclone Base Case (Dimits et al., 2000). Furthermore, we run extensive empirical evaluations against GKW (Peeters et al., 2009) reference data, comparing statistical quantities (time-averaged transport fluxes, wavenumber spectra, growth rates) across a broad set of equilibrium configurations for both adiabatic and kinetic electrons (Section 4.1). Finally, we demonstrate differentiable programming experiments, including gradient-based recovery of R/L_T and sensitivity analysis of growth rates (Section 4.3).

2. Gyrokinetics

Within computational plasma physics, gyrokinetics (Frieman & Chen, 1982; Krommes, 2012) is the foundational framework for turbulent transport modeling. By averaging over the fast gyro-angle dependency, it decouples high-frequency modes, isolating low-frequency fluctuations at the gyroradius scale while preserving finite Larmor radius effects in the distribution function f. This effectively reduces the 6D Vlasov-Maxwell equations to a 5D guiding-center formulation, making the system numerically tractable.

To this end, `gyaradax` uses the standard δf form (Frieman & Chen, 1982), decomposing f into a stationary Maxwellian background F_M and a fluctuating component $\delta f(v_\parallel, \mu, s, \mathbf{k}_\perp)$. Here, $v_\parallel$ and μ are the parallel velocity and magnetic moment, s goes along the magnetic field line, and $\mathbf{k}_\perp$ includes radial k_ψ (k_x) and bi-normal k_ζ (k_y).

2.1. Physical Components

We clearly mark each term in the right-hand side of the Vlasov-Poisson equation (I to VIII), and adopt a naming convention similar to Peeters et al. (2009),

$$\frac{\partial f}{\partial t} = \underbrace{-v_\parallel \nabla_\parallel f}_{\text{Parallel Advection (I)}} \underbrace{-i(\mathbf{k}_\perp \cdot \mathbf{v}_D) f}_{\text{Magnetic Drift (II)}} \underbrace{+\mu \nabla_\parallel B \frac{\partial f}{\partial v_\parallel}}_{\text{Mirror Term (IV)}}$$

$$\underbrace{-\mathbf{v}_E \cdot \nabla F_M}_{\text{Equilibrium Drive (V)}} \underbrace{-\frac{ZeF_M}{T}\left(v_\parallel \nabla_\parallel \bar{\phi} + i(\mathbf{k}_\perp \cdot \mathbf{v}_D)\bar{\phi}\right)}_{\text{Field Drives (VII and VIII)}}$$

$$\underbrace{-\mathbf{v}_E \cdot \nabla_\perp f}_{\text{Nonlinear (III)}} \underbrace{-\mathcal{D}(f)}_{\text{Dissipation}}$$

Kinetic Dynamics (I, II, IV). They describe the collisionless trajectories of particles in the background magnetic configuration. *Parallel Advection* (I) represents motion along magnetic field lines, while *Magnetic Drift* (II) accounts for curvature and ∇B drifts. The *Mirror Term* (IV) models the force arising from the parallel gradient of the magnetic field magnitude, which leads to particle trapping.

Energy Drives (V, VII, VIII). The *Equilibrium Drive* (V) originates from thermodynamic gradients (∇F_M), providing the energy source for turbulence. *Field Drives* (VII and VIII) represent the linear coupling between the electrostatic potential and the background distribution.

Nonlinear Advection (III). The *Nonlinear Term* $E \times B$ represents advection of the gyro-averaged distribution function by the fluctuating electric field. $E \times B$ is responsible for the saturated turbulent state, and the energy transfer across different spatial scales which pushes back on mode growth.

Dissipation. A numerical dissipation term is added for stability: 4th-order upwinded dissipation in the parallel direction, centered 4th-order smoothing in velocity space, and spectral hyper-viscosity in the perpendicular dimensions.

Neoclassical (VI, omitted). $-\mathbf{v}_D \cdot \nabla F_M$ is the coupling of the magnetic drift (curvature/∇B, Coriolis, centrifugal) to equilibrium gradients. It is active only for rotating or neoclassical plasmas and we omit it for simplicity.

3. `gyaradax` Implementation

`gyaradax` solves the collisionless electrostatic gyrokinetic equations in the local flux-tube limit, supporting both adiabatic and kinetic electrons. We designed it to be a slim, modern and ML-friendly alternative to GKW, written in JAX with the core integrator and solver in about 3 000 lines.

3.1. Time Integration

`gyaradax` is designed as a purely functional solver, where the simulation state is evolved through a chain of stateless transformations. We use explicit fourth-order Runge-Kutta (RK4) as integrator. Integration across sequential timesteps is fused with `jax.lax.scan`, reducing the Python interpreter overhead. Spatial derivatives in the parallel and velocity coordinates are computed using 4th-order central and upwinded stencils. The perpendicular dimensions are resolved pseudospectrally with 3/2-rule dealiasing to prevent aliasing errors in the nonlinear $E \times B$ term.

3.2. Performance

All species-dependent coefficients (Bessel functions, Maxwellians, drift velocities, and fused finite-difference stencils) are precomputed once before the time loop, eliminating redundant computation across the 4 RHS evaluations required by each RK4 step. Moreover, transitioning to JAX opens up performance optimizations that would be not always trivial to implement in Fortran. Specifically, for `gyaradax` we employed two orthogonal performance axes: *mixed precision* and *custom CUDA kernels*. Together, these optimizations yield a $\sim 2\times$ speedup over the JAX backend at the same grid resolution (Table 2).

Mixed precision. We observe that 2D FFTs, spatial derivatives and IFFTs in the nonlinear Poisson bracket can be performed in Float32 without loss of physical fidelity. Linear terms, field solver, and the final forward FFT on the accumulated output remain in Float64. This halves the memory bandwidth of the 8 FFTs per RK4 step.

Z2Z packing. We reduce the inverse FFT count from 4 to 2 by exploiting the linearity of the DFT. Same-field spatial derivatives ($ik_x\hat{f}$, $ik_y\hat{f}$) are packed into a single complex-to-

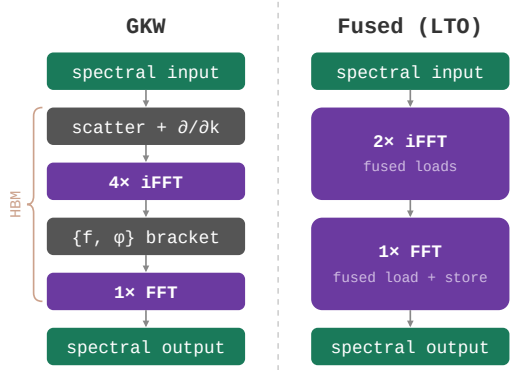


Figure 1. Nonlinear Poisson bracket. **Left:** GKW implementation with 4 inverse FFTs. **Right:** CUDA LTO callbacks fuse all pre- and post-FFT work, with Z2Z packing reducing the iFFTs to 2.

complex (C2C) transform, with a Hermitian symmetrization correction at k_y=0 to prevent channel leakage from gyro-averaging asymmetry (Appendix D.2).

CUDA backend. For the linear RHS, GKW and the JAX backend evaluate the nine-point parallel stencil, the five-point velocity stencil, and eight elementwise physics terms as separate operations fused by the compiler. Our custom CUDA kernel performs all of these in a single pass with strided addressing and velocity-axis tiling, eliminating the gather-and-predicate overhead introduced by XLA's general-purpose stencil lowering. Figure 1 shows the optimized nonlinear Poisson bracket. Since XLA cannot fuse across cuFFT calls, we apply Link-Time Optimization (LTO) callbacks to embed the spectral derivative multiplication, gyro-averaging, bracket computation, and spectrum unpacking directly into cuFFT butterfly passes, avoiding intermediate HBM round-trips (Appendix D).

3.3. Limitations and Differences to GKW

The main shortcoming of `gyaradax` is that it can only handle electrostatic collisionless plasmas. It does not yet support electromagnetic perturbations, collisionality, Coriolis effects (Term VI), or global gyrokinetics. Time integration is limited to explicit RK4, which becomes slow for fast kinetic species due to tightening CFL constraint. GKW additionally provides the option of implicit or semi-implicit integrators for stiff parallel streaming. As for geometry, we implement the analytic circular (Lapillonne) and s-alpha. Shaped plasmas require Miller parameterization or numerical MHD equilibria (EFIT/CHEASE), not yet in `gyaradax`.

We treat the 5D domain as dense arrays, instead of the sparse GKW format. Additionally, at the moment we do not support grid parallelism, unlike legacy codes that rely on global mutable state and complex MPI-based domain decomposition. Adiabatic and kinetic electron models are available, supporting single-species (ion) and multi-species (ion + electron) simulations. IO and simulation configurations are different formats but consistent with each other. Our diagnostics suite matches GKW on the main fronts, with multi-species fluxes, spectra and growth rates.

3.4. Vibecoding Setup

`gyaradax` was written and validated in a short time with substantial use of coding agents. We consider it relevant to add details on the workflow used.

Workspace. In the preparation phase, we accomplish three things: **i.** We provide utilities, such as GKW-specific I/O, parsing, as well as field solvers and flux integrals. The goal here is to help the agent target solver logic, instead of side tasks. **ii.** Some reference trajectories are included to understand the GKW configuration and output structure. Notably, these contain periodic f dumps, simplifying debugging by providing the full state instead of integrated observables. **iii.** We wrote unit tests to symbolically and empirically verify alignment. This *empirical test-driven* cycle proved essential for the agent-led portion of the GKW rewrite. We find that on hard tasks a measurable form of success is mandatory for vibecoding.

Prompting. The first step in the prompt directed the agent to deliberate code ingestion and note taking, using the authors' knowledge of the solver to load core Fortran files and important manual pages fully into context. After this phase, the task was to alternate between implementing solver logic, validating it with tests and incrementally adding new ones. We explicitly decomposed the problem to first implement and validate the linear terms, then address the more involved nonlinear part (Term III). We refined the initial instructions with meta-prompting, focusing on structure and imperative syntax better suited for modern LLMs (Zhang et al., 2025). The final version is provided in Appendix H.

Models. Initial attempts using Gemini 3.1 Pro (Google DeepMind, 2026) via `gemini-cli` yielded unreasonable results over multiple parallel tries. We had more success with GPT-5.3 CODEX (OpenAI, 2026) for the core translation and Claude 4.6 Opus (Anthropic, 2026) for the kinetic electron implementation. For the final performance optimization we use a multi-agent workflow to generate and integrate custom CUDA kernels. This pipeline begins with a proposer/reviewer consensus loop. After a human review of the strategy, a lightweight model (Gemini 3 Flash, Claude Sonnet) iteratively implements the proposals. The verification pipeline handles compilation and benchmarking of the kernels against the JAX reference. See Appendix D for a breakdown. Finally, some opportunities for speedup in the early stages, such as precomputing the linear terms, were spotted and implemented entirely by Claude.

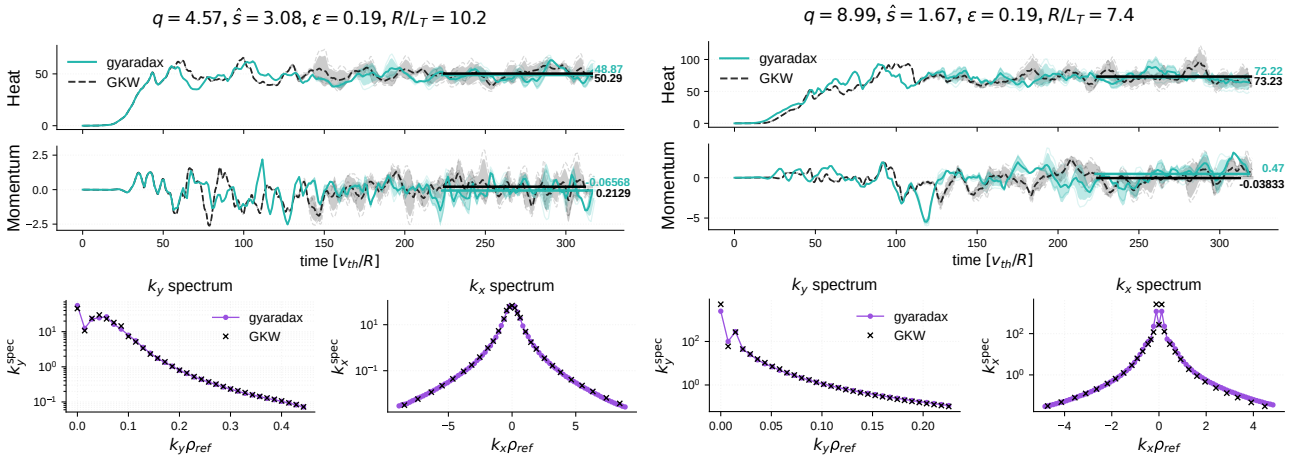


Figure 2. Empirical statistical validation of gyaradax against GKW for two adiabatic ITG equilibria. **Top:** heat and momentum flux time traces. Both codes coincide during linear growth, but once the nonlinear effects kick-in, trajectories diverge due to chaotic sensitivity. Three runs were performed for both gyaradax and GKW, with the variance across them reported as shaded bounds. **Bottom:** time-averaged k_y and k_x spectral profiles, with strong agreement on the slope and the peak location.

4. Verification and Experiments

We validate gyaradax against GKW on nonlinear ITG turbulence, validate it on analytical benchmarks, and demonstrate AD through inverse problems and sensitivity analysis.

4.1. Empirical Validation

Pointwise verification of a chaotic system can be non representative: two runs of the same code from the same initial conditions will diverge once floating-point differences accumulate. We therefore verify gyaradax through statistical agreement with GKW, covering both adiabatic (Paischer et al., 2025) and kinetic electrons (Figures 2 and 3). All test cases use a standard grid ($N_{v_\parallel} = 32$, $N_\mu = 8$, $N_s = 16$, $N_{k_x} = 85$, $N_{k_y} = 32$). The adiabatic cases use $\mathrm{d}t = 0.01$, while kinetic electron cases use adaptive timestepping with $\mathrm{d}t \approx 2.1 \times 10^{-3}$ (CFL). We evaluate transport fluxes, wavenumber spectra, and per-k_y growth rates. To illustrate the chaotic sensitivity, we perform 3 runs per configuration from identical initial conditions, for both gyaradax and GKW. Additionally, to quantify the statistical properties, agreement across a validation set of 46 unstable adiabatic runs is performed. Individual traces are in the Appendix Figures 9 and 10.

Flux Traces. Transport fluxes are the primary observables for evaluating turbulence saturation and macroscopic plasma confinement. For adiabatic simulations, heat flux is the most meaningful one, while particle flux vanishes identically. In Figure 2 (top), we compare the time evolution of ion heat and momentum fluxes. We plot an ensemble of three identical runs for both gyaradax and GKW to account for the chaotic sensitivity of the system. In the linear growth phase both traces match, but after the onset of turbulence trajectories diverge. This is due to the accumulation of microscopic floating-point variations. For gyaradax this effect is amplified by the non-deterministic GPU optimization. Despite these expected locally divergent phase spaces, the statistical behavior is similar, with time-averaged flux levels converging to the GKW baseline, confirming that the long-term turbulent transport is correctly reproduced. Across conditions, the grand mean heat flux is close ($\bar{Q}_{\texttt{gyaradax}} = 90.9$, $\bar{Q}_{\mathrm{GKW}} = 91.3$), with a rMAE of 0.14.

Spectra. The spectral energy distribution across perpendicular wavenumbers characterizes the turbulent cascade. We compare the spectra $k_y^{\mathrm{spec}} = \sum_{s,k_x} |\hat{\phi}(s, k_x, k_y)|^2$ and $k_x^{\mathrm{spec}} = \sum_{s,k_y} |\hat{\phi}(s, k_x, k_y)|^2$, which measure how energy is distributed across binormal and radial scales respectively. Figure 2 (bottom) shows alignment of both time-averaged profiles, despite local differences in microstates. Table 1 summarizes the spectral agreement across all configurations.

Table 1. Spectral validation across 46 ITG equilibria (mean $\pm$ std).

	KS statistic	Pearson r	Log rel. L_2
k_y^{spec}	0.073 ± 0.049	0.990 ± 0.018	0.073 ± 0.042
k_x^{spec}	0.055 ± 0.035	0.952 ± 0.068	0.076 ± 0.042

Growth Rates. Growth rates $\gamma(k_y)$ measure the linear instability drive. In the nonlinear regime they converge to zero, showing the balance between linear drive and nonlinear saturation. Across the validation set, both codes converge to $\bar{\gamma} \approx 0$ ($\bar{\gamma}_{\texttt{gyaradax}} \approx 9.2 \times 10^{-4}$ and $\bar{\gamma}_{\mathrm{GKW}} \approx -1.8 \times 10^{-4}$), as expected for a balanced turbulent state.

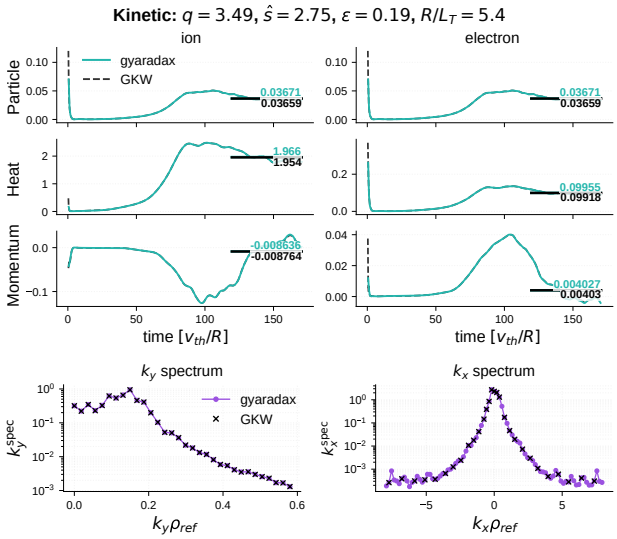


Figure 3. Kinetic electron validation. **Top:** ion (left) and electron (right) fluxes. **Bottom:** time-averaged k_y and k_x spectral profiles.

Kinetic Electrons. We additionally verify the solver on three kinetic electron configurations, with two species (ions and electrons). Due to faster perturbations, adaptive CFL timestepping is required (constrained by the electron thermal velocity $v_{th,e}/v_{th,i} \approx 60$). Per-species fluxes and k_y/k_x spectra match the GKW reference (Figure 3).

4.2. Analytical Benchmarks

gyaradax is verified on two standard benchmarks with analytical solutions. Details in Appendix F.

Rosenbluth-Hinton Test. The RH test is a sensitive end-to-end test of the field solver and the linear dynamics. The zonal flow (k_ζ=0) excites a geodesic acoustic mode that oscillates and damps via collisionless Landau damping. The residual zonal potential is given analytically by $\phi(\infty)/\phi(0) = 1/(1+q^2\Theta/\varepsilon^2)$, where $\Theta(\varepsilon)$ captures finite-ε corrections (Rosenbluth & Hinton, 1998; Xiao & Catto, 2006). At q=1.3, ε=0.05, $\hat{s}$=0.16, gyaradax converges to a residual of 0.0711, matching the prediction (Figure 4a). A scan over ε at fixed q follows the analytical curve across the full range (Figure 4b).

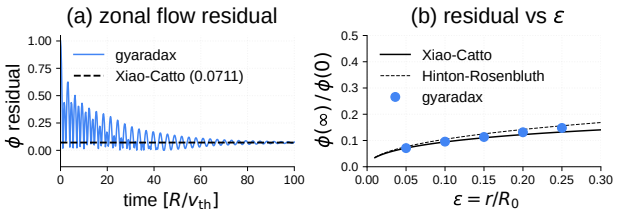


Figure 4. Rosenbluth-Hinton zonal flow test at q=1.3. (a) Zonal potential residual over time at ε=0.05. (b) Residual at varying ε.

Cyclone Base Case. CBC (Dimits et al., 2000) is the standard linear benchmark. It tests the full linear operator in the regime where the ion temperature gradient (ITG) drives instability. The configuration used has adiabatic electrons, q=1.4, $\hat{s}$=0.78, ε=0.19, R/L_n=2.2 and s-alpha geometry. gyaradax matches growth rates $\gamma(k_\theta\rho_s)$ at R/L_T=6.9 (Figure 5a) and the gradient dependence $\gamma(R/L_T)$ at $k_\theta\rho_s$=0.5 (Figure 5b).

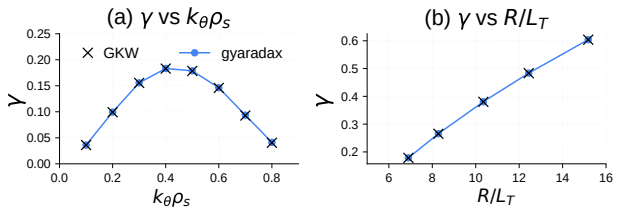


Figure 5. Cyclone Base Case (linear). (a) Growth rate vs $k_\theta\rho_s$ (R/L_T=6.9). (b) Growth rate vs R/L_T ($k_\theta\rho_s$=0.5).

4.3. Experiments

Inverse Problem. A direct payoff of end-to-end differentiability is gradient-based inverse problems. We demonstrate this by recovering the temperature gradient R/L_T from a target electrostatic potential ϕ^*. We fix the equilibrium parameters (q=1.4, $\hat{s}$=0.78, ε=0.19, R/L_n=2.22) and use a reduced grid $(N_{v_\parallel}, N_\mu, N_s, N_{k_x}, N_{k_y}) = (24, 8, 16, 9, 8)$. The target ϕ^* is generated by running a linear simulation at $R/L_T^{\rm true} = 6.9$ for 400 steps ($\mathrm{d}t = 0.01$). The loss is the potential energy mismatch

$$\mathcal{L}(R/L_T) = \frac{1}{N} \sum_{s,k_x,k_y} \left|\phi(s, k_x, k_y;\, R/L_T) - \phi^*\right|^2 ,$$

where ϕ is the result of the full forward solve. Reverse-mode AD provides exact $\partial\mathcal{L}/\partial(R/L_T)$ through the full 400-step integration. Starting from $R/L_T^{(0)} = 10.0$, Adam (Kingma & Ba, 2015) converges to $R/L_T = 6.908$ (Figure 6). The recovered potential matches the target $k_y^{\rm spec}$ (Figure 6, top left),

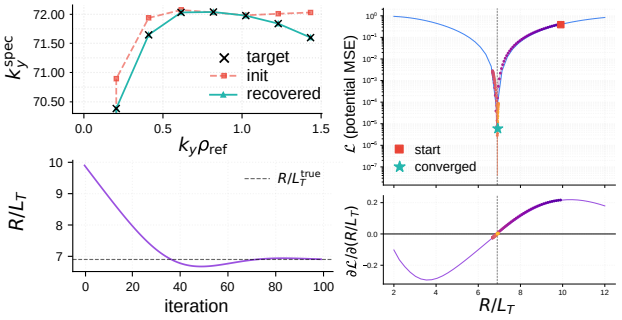


Figure 6. Recovery of R/L_T from the electrostatic potential. **Left:** initial, target (×), and recovered k_y spectra (top), R/L_T convergence during iterations (bottom). **Right:** loss landscape $\mathcal{L}(R/L_T)$ (top) and AD gradient (bottom), with the Adam trajectory overlaid.

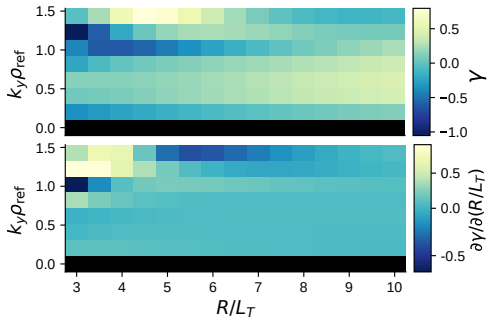


Figure 7. Growth rate sensitivity to R/L_T (k_y=0 zonal mode is masked). **Top:** per-k_y growth rate $\gamma(k_y, R/L_T)$. **Bottom:** sensitivity $\partial\gamma/\partial(R/L_T)$.

confirming that the optimizer recovers both the instability drive and the eigenmode structure. The loss landscape (Figure 6, right), obtained by evaluating $\mathcal{L}$ and $\partial\mathcal{L}/\partial(R/L_T)$ over $R/L_T \in [2, 12]$, is smooth and unimodal, with a clean zero crossing of the gradient at the optimum.

Sensitivity Analysis. Sensitivity analysis measures responses to parameter changes, informing design and uncertainty quantification. Finite differences require $2N$ evaluations for N parameters and suffer from step-size errors. In contrast, reverse-mode AD computes exact gradients of scalar losses in one backward pass. For vector-valued outputs like $\gamma(k_y)$, `jax.jacrev` computes the full Jacobian in N_{k_y} passes (one per output dimension). Using the inverse problem setup, we scan $R/L_T \in [3, 10]$ across 15 values, computing $\gamma(k_y)$ and its Jacobian $\partial\gamma/\partial(R/L_T)$. All modes are damped ($\gamma < 0$) on this grid (Figure 7, top). Increasing R/L_T reduces the damping rate, with high-k_y modes at low drive being the most responsive (Figure 7, bottom).

4.4. Performance Comparison vs GKW

The performance of *gyaradax* with different settings is shown in Table 2, showing significant performance gains over GKW in both adiabatic and kinetic electron cases. The CUDA backend requires approximately 1.2 GB additional GPU memory beyond the JAX implementation, as the fused cuFFT kernel pre-allocates its own workspace via *cudaMalloc*. GKW is run on an AMD EPYC 9754 128-Core Processor with 1TB of RAM, with 64 and 128 processes for adiabatic and kinetic respectively. All `gyaradax` benchmarks are performed on a single Nvidia Blackwell B300 SXM6 GPU (275GB variant). Overall, JAX with Z2Z packing and mixed precision leads to a $5.4\times$ speedup compared to GKW, and when using CUDA kernels it jumps to $10.5\times$.

5. Conclusion and Future Work

`gyaradax` is a minimal and fully differentiable JAX implementation of local flux-tube gyrokinetics, supporting both adiabatic and kinetic electrons. Development was heavily supported by modern agentic workflows, with human oversight. Our implementation is formally correct and achieves numerical parity with GKW, while offering native hardware acceleration and easy integration with ML pipelines. The availability of an end-to-end differentiable gyrokinetic solver can open new avenues for plasma research, from automated parameter optimization (McGreivy et al., 2021; Joglekar et al., 2026) to the development of physics-informed surrogate models (Um et al., 2021; Karniadakis et al., 2021). Moreover, the lightweight and accessible nature of `gyaradax` lowers the entry barrier of an otherwise highly specialized code, making extensions, distribution and maintenance substantially easier.

Table 2. Benchmarks comparing GKW against `gyaradax` variants. All runs use the same grid resolution $(32, 8, 16, 85, 32)$. Note that GKW uses main memory (RAM) and `gyaradax` uses GPU memory (VRAM). Configurations denote the computational backend and numerical precision of the nonlinear term.

Solver Configuration	Steps/s	Speedup	Mem (GB)
Adiabatic Electrons			
GKW (DP)	5.75	–	18.0
`gyaradax` (JAX, DP)	12.49	2.17×	9.4
`gyaradax` (JAX, MP)	30.89	5.37×	9.4
`gyaradax` (CUDA, DP)	16.6	2.89×	10.6
`gyaradax` (CUDA, MP)	60.54	**10.53**×	10.6
Kinetic Electrons			
GKW (DP)	3.43	–	38.8
`gyaradax` (JAX, DP)	6.21	1.81×	17.6
`gyaradax` (JAX, MP)	15.19	4.43×	17.6
`gyaradax` (CUDA, DP)	8.13	2.36×	18.9
`gyaradax` (CUDA, MP)	28.16	**8.21**×	18.9

Future Work. Currently, `gyaradax` is limited to electrostatic collisionless plasmas. Future work will incorporate electromagnetic perturbations ($A_\parallel$, $B_\parallel$) and collisionality. As problem sizes grow significantly when faster species are introduced, support for grid parallelism via GPU sharding (Xu et al., 2021) is required. Another promising direction is fully spectral solvers (Candy et al., 2016), where all spatial dimensions are resolved in Fourier/spectral space without finite-difference stencils. Such formulations are usually better suited for GPU acceleration, as they replace stencils with global spectral transforms that map naturally onto batched matrix operations. On the verification side, future work includes extending the analytical test suite to kinetic-electron benchmarks and electromagnetic cases from Peeters et al. (2009). Ultimately, we present our work as a proof-of-concept for the rapid agent-assisted development of modern high-performance scientific software.

References

Anthropic. Introducing Claude Opus 4.6. Technical report, Anthropic, Feb 2026. URL https://www.anthropic.com/news/claude-opus-4-6.

Bezgin, D. A., Buhendwa, A. B., and Adams, N. A. Jax-fluids: A fully-differentiable high-order computational fluid dynamics solver for compressible two-phase flows. *Computer Physics Communications*, 282:108527, 1 2023. ISSN 00104655. doi: 10.1016/j.cpc.2022.108527.

Bowman, S. R., Hyun, J., Perez, E., Chen, E., Pettit, C., Heiner, S., and Lukošiūtė, K. Measuring progress on scalable oversight for large language models, 2022.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Candy, J., Belli, E., and Bravenec, R. A high-accuracy eulerian gyrokinetic solver for collisional plasmas. *Journal of Computational Physics*, 324:73–93, 2016. ISSN 0021-9991.

Chen, Z., Chen, S., Ning, Y., Zhang, Q., Wang, B., Yu, B., Li, Y., Liao, Z., Wei, C., Lu, Z., Dey, V., Xue, M., Baker, F. N., Burns, B., Adu-Ampratwum, D., Huang, X., Ning, X., Gao, S., Su, Y., and Sun, H. Scienceagentbench: Toward rigorous assessment of language agents for data-driven scientific discovery, 2025.

Citrin, J., Goodfellow, I., Raju, A., Chen, J., Degrave, J., Donner, C., Felici, F., Hamel, P., Huber, A., Nikulin, D., Pfau, D., Tracey, B., Riedmiller, M., and Kohli, P. Torax: A fast and differentiable tokamak transport simulator in jax, 2024.

Dimits, A. M., Bateman, G., Beer, M. A., Cohen, B. I., Dorland, W., Hammett, G. W., Kim, C., Kinsey, J. E., Kotschenreuther, M., Kritz, A. H., Lao, L. L., Mandrekas, J., Nevins, W. M., Parker, S. E., Redd, A. J., Shumaker, D. E., Sydora, R., and Weiland, J. Comparisons and physics basis of tokamak transport models and turbulence simulations. *Physics of Plasmas*, 7(3):969–983, March 2000. doi: 10.1063/1.873896.

Duston, T., Xin, S., Sun, Y., Zan, D., Li, A., Xin, S., Shen, K., Chen, Y., Sun, Q., Zhang, G., Liu, J., Zhou, H., Liu, J., Pu, Z., Wang, Y., Ge, B.-X., Tong, X., Ye, F., Zhao, Z.-C., Han, W.-B., Cao, Z., Zhao, Y., Ren, W., Long, Q., Liu, Y., Huang, A., Du, Y., Rong, Y., and Peng, J. Ainsteinbench: Benchmarking coding agents on scientific repositories, 2025.

Frieman, E. and Chen, L. Nonlinear gyrokinetic equations for low-frequency electromagnetic waves in general plasma equilibria. *The Physics of Fluids*, 25(3):502–508, 1982.

Google DeepMind. Gemini 3.1 pro. https://deepmind.google/models/model-cards/gemini-3-1-pro/, February 2026. Accessed: 2026.

Jakob, W., Speierer, S., Roussel, N., and Vicini, D. Dr. jit: A just-in-time compiler for differentiable rendering. *ACM Transactions on Graphics (TOG)*, 41(4):1–19, 2022.

Joglekar, A. S., Thomas, A. G. R., Milder, A. L., Miller, K. G., Palastro, J. P., and Froula, D. H. Differentiable programming for plasma physics: From diagnostics to discovery and design, 2026.

Karniadakis, G. E., Kevrekidis, I. G., Lu, L., Perdikaris, P., Wang, S., and Yang, L. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, 2021. ISSN 2522-5820. doi: 10.1038/s42254-021-00314-5.

Kelling, J., Bolea, V., Bussmann, M., Checkervarty, A., Debus, A., Ebert, J., Eisenhauer, G., Gutta, V., Kesselheim, S., Klasky, S., Pandit, V., Pausch, R., Podhorszki, N., Poschel, F., Rogers, D., Rustamov, J., Schmerler, S., Schramm, U., Steiniger, K., Widera, R., Willmann, A., and Chandrasekaran, S. The artificial scientist – in-transit machine learning of plasma simulations, 2025.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In Bengio, Y. and LeCun, Y. (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

Kotschenreuther, M. and Rogers, B. Electron temperature gradient driven turbulence. *Physics of plasmas*, 7(5): 1904–1910, 2000.

Krommes, J. A. The gyrokinetic description of microturbulence in magnetized plasmas. *Annual Review of Fluid Mechanics*, 44(Volume 44, 2012):175–201, 2012. ISSN 1545-4479.

McGreivy, N., Hudson, S., and Zhu, C. Optimized finite-build stellarator coils using automatic differentiation. *Nuclear Fusion*, 61(2):026020, January 2021. ISSN 1741-4326. doi: 10.1088/1741-4326/abcd76. URL http://dx.doi.org/10.1088/1741-4326/abcd76.

NVIDIA Corporation. *cuFFT Library User's Guide: Callback Routines*, 2026. URL https://docs.nvidia.com/cuda/archive/12.2.1/cufft/ltoea/usage/api_usage.html. Accessed: 2026-03-30.

OpenAI. Gpt-5.3-codex. https://openai.com/index/introducing-gpt-5-3-codex/, 2026. Accessed: 2026.

Ouyang, A., Guo, S., Arora, S., Zhang, A. L., Hu, W., Re, C., and Mirhoseini, A. Kernelbench: Can llms write efficient gpu kernels? In *International Conference on Machine Learning*, pp. 47356–47415. PMLR, 2025.

Paischer, F., Galletti, G., Hornsby, W., Setinek, P., Zanisi, L., Carey, N., Pamela, S., and Brandstetter, J. Gyroswin: 5d surrogates for gyrokinetic plasma turbulence simulations. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 02 - 07, 2025*, 2025.

Peeters, A., Camenen, Y., Casson, F., Hornsby, W., Snodin, A., Strintzi, D., and Szepesi, G. The nonlinear gyrokinetic flux tube code gkw. *Computer Physics Communications*, 180(12):2650–2672, 2009. ISSN 0010-4655. 40 YEARS OF CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures.

Rosenbluth, M. N. and Hinton, F. L. Poloidal flow driven by ion-temperature-gradient turbulence in tokamaks. *Phys. Rev. Lett.*, 80:724–727, Jan 1998. doi: 10.1103/PhysRevLett.80.724.

Schoenholz, S. S. and Cubuk, E. D. Jax m.d. a framework for differentiable physics. In *Advances in Neural Information Processing Systems*, volume 33. Curran Associates, Inc., 2020.

Toshev, A. P., Ramachandran, H., Erbesdobler, J. A., Galletti, G., Brandstetter, J., and Adams, N. A. Jax-sph: A differentiable smoothed particle hydrodynamics framework. *arXiv preprint arXiv:2403.04750*, 2024.

Um, K., Brand, R., Yun, Fei, Holl, P., and Thuerey, N. Solver-in-the-loop: Learning from differentiable physics to interact with iterative pde-solvers, 2021.

Wei, A., Sun, T., Seenichamy, Y., Song, H., Ouyang, A., Mirhoseini, A., Wang, K., and Aiken, A. Astra: A multi-agent system for gpu kernel performance optimization. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*, 2025.

Xiao, Y. and Catto, P. J. Plasma shaping effects on the collisionless residual zonal flow level. *Physics of plasmas*, 13(8), 2006.

Xu, Y., Lee, H., Chen, D., Hechtman, B., Huang, Y., Joshi, R., Krikun, M., Lepikhin, D., Ly, A., Maggioni, M., Pang, R., Shazeer, N., Wang, S., Wang, T., Wu, Y., and Chen, Z. Gspmd: General and scalable parallelization for ml computation graphs, 2021.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024.

Zanisi, L., Ho, A., Barr, J., Madula, T., Citrin, J., Pamela, S., Buchanan, J., Casson, F., and Gopakumar, V. Efficient training sets for surrogate models of tokamak turbulence with active deep ensembles. *Nuclear Fusion*, 64(3):036022, February 2024. ISSN 1741-4326. doi: 10.1088/1741-4326/ad240d.

Zhang, Y., Yuan, Y., and Yao, A. C.-C. Meta prompting for ai systems, 2025.

Zhu, X., Peng, S., Guo, J., Chen, Y., Guo, Q., Wen, Y., Qin, H., Chen, R., Zhou, Q., Gao, K., et al. Qimeng-kernel: Macro-thinking micro-coding paradigm for llm-based high-performance gpu kernel generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 29168–29176, 2026.

A. Parameters

A.1. Phase-Space Variables

Symbol	Code Key	Description
δf_a	df	Perturbed distribution function for species a. Shape: $(N_{v\parallel}, N_\mu, N_s, N_{kx}, N_{ky})$ for adiabatic; $(N_{sp}, N_{v\parallel}, N_\mu, N_s, N_{kx}, N_{ky})$ for kinetic electrons.
$\phi, \langle\phi\rangle$	phi, gyro_phi	Electrostatic and gyro-averaged potential ($J_0\phi$). Shape: (N_s, N_{kx}, N_{ky}).
s	sgrid	Field-line coordinate, $s \in [-0.5, 0.5]$ for nperiod= 1 (one poloidal transit).
$v_\parallel, \mu$	vpgr, mugr	Parallel velocity (uniform grid) and magnetic moment ($\mu = v_\perp^2/2$, uniform in $v_\perp$).
k_x, k_y	kxrh, krho	Radial and binormal spectral wavevectors, normalized to ρ_{ref}.
$\Delta s, \Delta v_\parallel$	sgr_dist, dvp	Parallel and velocity grid spacing.
w_s, w_v, w_μ	ints, intvp, intmu	Phase-space integration weights for s, $v_\parallel$, and μ. The μ weights are $2\pi v_\perp \Delta v_\perp$ (cylindrical Jacobian).
P_{k_y}	parseval	Parseval normalization: 1 for $k_y = 0$, 2 for $k_y > 0$ (half-spectrum convention accounts for the conjugate mode).
Δt	dt	Small timestep for RK4 integration. May vary per step when CFL-adaptive.

A.2. Species Parameters

Symbol	Code Key	Description
m_a, T_a	mas, tmp	Species mass and temperature, normalized to reference. Scalar (adiabatic) or array of length N_{sp} (kinetic).
n_a, Z_a	de, signz	Species density and charge number. Same shape convention as mass/temperature.
$R/L_T, R/L_n$	rlt, rln	Inverse temperature and density gradient scale lengths. Per-species for kinetic electrons.
$v_{th,a}/v_{th,ref}$	vthrat	Thermal velocity ratio $\sqrt{T_a/m_a}$ relative to reference species.
$F_{M,a}, J_0$	fmaxwl, bessel	Background Maxwellian and Bessel $J_0(k_\perp \rho_a)$. Per-species in kinetic mode.
Γ_0	gamma	FLR kernel $I_0(b_a)e^{-b_a}$ with $b_a = \frac{1}{2}(m_a v_{th,a} k_\perp / Z_a B)^2$. Computed via jax.scipy.special.i0e for numerical stability.

A.3. Geometry and Magnetic Equilibrium

The gyrokinetic equation is formulated in a field-aligned coordinate system (ψ, ζ, s), where ψ labels the flux surface (radial direction), ζ is the field-line label (binormal direction), and s follows the magnetic field line (parallel direction). The *geometry* encodes how this abstract coordinate system maps to physical space: it provides the covariant metric tensor g_{ij} (for $k_\perp^2$ and perpendicular gradients), the magnetic field strength $B(s)$ (for mirror forces, gyro-averaging, and drifts), and a set of derived tensors that enter the gyrokinetic equation as advection coefficients.

gyaradax supports a standalone circular equilibrium model (Lapillonne, translated from GKW geom.f90) that computes all geometry arrays from three equilibrium scalars $(q, \hat{s}, \varepsilon)$ plus grid resolution parameters, eliminating the need for precomputed geometry files. The continuous geometry functions (B-field, metric, drift tensors) are implemented in JAX and are differentiable with respect to the equilibrium parameters; the discrete topology (mode labels, parallel boundary connectivity) uses numpy and is detached from the gradient graph via jax.lax.stop_gradient.

Equilibrium parameters. The safety factor q controls how tightly wound the field lines are (large q = nearly toroidal lines, weak poloidal field). The magnetic shear $\hat{s} = (r/q)\, dq/dr$ measures how q varies radially and drives the spectral mode connectivity: adjacent k_x modes couple across the parallel boundary with shift $\Delta k_x = 2\pi \hat{s}\, k_y$. The inverse aspect ratio $\varepsilon = r/R_0$ sets the strength of toroidal effects: the outboard midplane ($\theta = 0$) sees a weaker field than the inboard side ($\theta = \pi$), producing trapped-particle physics and the ballooning structure characteristic of tokamak turbulence.

In the circular model, the magnetic field strength is:

$$B(s) = \frac{\delta}{1 + \varepsilon \cos\theta(s)}, \qquad \delta = \sqrt{1 + \frac{\varepsilon^2}{q^2(1-\varepsilon^2)}} \tag{1}$$

where $\theta(s)$ is obtained by inverting $\theta + \varepsilon \sin\theta = 2\pi s$.

Metric tensor. The metric components $g_{\psi\psi}$, $g_{\psi\zeta}$, $g_{\zeta\zeta}$, $g_{\psi s}$, $g_{\zeta s}$, g_{ss} define how coordinate distances map to physical distances. The solver uses these in the perpendicular wavenumber calculation:

$$k_\perp^2 = k_y^2 g_{\zeta\zeta} + 2k_x k_y\, g_{\psi\zeta} + k_x^2 g_{\psi\psi} \tag{2}$$

The cross-term $g_{\psi\zeta}$ (the "dzetadeps" coupling) encodes the integrated magnetic shear: it vanishes at the outboard midplane and grows along the field line, tilting eddies via $k_{x,\text{loc}} = k_x + k_y\, g_{\psi\zeta}$. The $d\zeta/d\varepsilon$ computation uses a branch-tracked $\arctan$ with a finite-ε correction (GKW geom.f90 lines 1492–1511). This correction introduces an inherent $\sim 0.1\%$ model-level approximation that propagates into all ζ-direction drift tensors (D_ζ, H_ζ, I_ζ), while the radial (ψ) components remain accurate to $< 10^{-4}$.

Drift tensors. All derivatives ($dB/d\psi$, $dR/d\psi$, $dZ/d\psi$, etc.) are first computed in (ψ, θ) space, then transformed to (ψ, s) coordinates via the Jacobian:

$$f_\psi^{(s)} = f_\psi^{(\theta)} - \frac{\sin\theta}{1 + \varepsilon\cos\theta}\, f_\theta, \qquad f_s = \frac{2\pi}{1 + \varepsilon\cos\theta}\, f_\theta \tag{3}$$

The E-tensor (ExB operator) is built from the antisymmetric cofactors of the first two metric rows, scaled by $\pi\, dp_f/d\psi/B^2$ where $dp_f/d\psi = \varepsilon/(q\sqrt{1-\varepsilon^2})$. The curvature drift tensor is then:

$$D_j = \frac{-2\big(E_{j,\psi}\, \partial_\psi B + E_{j,s}\, \partial_s B\big)}{B} \tag{4}$$

D_ψ drives radial transport (proportional to $\sin\theta$, strongest at top/bottom); D_ζ drives binormal transport (proportional to $\cos\theta$ + shear corrections, strongest at the outboard midplane). They enter Term II of the gyrokinetic equation as $-i(k_x D_\psi + k_y D_\zeta)(v_\parallel^2 + \mu B)\, \delta f$.

The H-tensor (Coriolis drift) uses $dZ/d\psi$ and dZ/ds with metric coupling; the I-tensor (centrifugal drift) uses $E \cdot \nabla R$ scaled by $2R$. Both are stored for future rotation physics.

Symbol	Code Key	Description
q	q	Safety factor. Controls field-line winding and parallel domain extent.
$\hat{s}$	shat	Magnetic shear $(r/q)\, dq/dr$. Drives mode connectivity across the parallel boundary.
ε	eps	Inverse aspect ratio r/R_0. Controls trapped-particle fraction and ballooning.
$B(s)$	bn	Magnetic field magnitude along the field line, normalized to B_{ref}.

$\mathcal{F}$	`ffun`	Parallel streaming coefficient: $\mathcal{F} = b_{ups}/B$ where $b_{ups} = 1/(2\pi q\sqrt{1-\varepsilon^2})$.
$\mathcal{G}$	`gfun`	Mirror force: $\mathcal{G} = \mathcal{F} \cdot (\partial_s B)/B$. Drives particle trapping.
g_{ij}	`little_g`	Perpendicular metric: stores $(g_{\zeta\zeta}, g_{\psi\zeta}, g_{\psi\psi})$ for $k_\perp^2$.
$\mathcal{D}_j$	`dfun`	Curvature + ∇B drift (3 components: ψ, ζ, s).
$\mathcal{E}_{\psi\zeta}$	`efun`	ExB geometric factor: $-E_{0,1}(s)$, varies along the field line through the metric and B^2.
$\mathcal{H}_j$	`hfun`	Coriolis drift tensor (3 components). For rotation physics.
$\mathcal{I}_j$	`ifun`	Centrifugal drift tensor (3 components). For rotation physics.
B_t/B	`bt_frac`	Toroidal fraction of the total field $= 1/\delta$.
R	`rfun`	Major radius along the field line: $R = 1 + \varepsilon\cos\theta$.
k_{th}	`kthnorm`	Binormal wavenumber normalization: $k_{th} = \sqrt{g_{\zeta\zeta}(\theta = 0)}$.

B. Functions

Function	**Description**
`compute_geometry`	Computes all geometry arrays from $(q, \hat{s}, \varepsilon)$. Continuous quantities (B-field, metric, drifts) use JAX and are differentiable; discrete topology (mode labels, connectivity) uses numpy.
`linear_precompute`	Precomputes all species-dependent coefficients: Bessel functions J_0, Maxwellians F_M, drift velocities, advection speeds, and fused stencils. For kinetic electrons, all arrays gain a leading species dimension $(N_{sp}, \ldots)$. Called once before the time loop.
`_linear_rhs_core`	Inner linear RHS for a single species (5D). Evaluates Terms I (streaming), II (drift), IV (mirror), V (drive), VII (Landau), VIII (drift drive), plus all dissipation operators. Uses precomputed fused stencils for parallel derivatives.
`nonlinear_term_iii`	Pseudospectral ExB advection via 2D FFTs with 3/2-rule dealiasing. Vectorized over the parallel grid via `jax.vmap`.
`calculate_phi`	Unified phi solver: dispatches to adiabatic or kinetic path based on `params.adiabatic_electrons`. Adiabatic path includes zonal flux-surface-averaged correction.
`estimate_timestep`	Combined CFL: $\min(\Delta t_{\text{NL}}, \Delta t_{\text{lin}})$. Nonlinear CFL from real-space $\lvert\nabla\phi\rvert$; linear CFL from von Neumann analysis of streaming, trapping, and dissipation stencils. For kinetic electrons, the field CFL (electrostatic Alfvén frequency) is additionally included.
`gkstep_single`	Single RK4 step. Dispatches to adiabatic or kinetic paths via static branching on `params.adiabatic_electrons`. The kinetic path uses `jax.vmap` over species for both linear and nonlinear terms.
`gksolve`	Multi-step driver using `jax.lax.scan`. Supports fixed and CFL-adaptive timestep modes. Returns final $(\delta f, \phi, \text{fluxes}, \text{state})$.
`gk_init` / `gksimulate`	High-level entry points: `gk_init` creates initial conditions and state; `gksimulate` runs the full simulation with checkpointing and diagnostics.
`calculate_fluxes_kinetic`	Per-species transport fluxes $(Q_a^{particle}, Q_a^{heat}, Q_a^{momentum})$. Returns $(N_{sp}, 3)$ array.

C. Implementation Details

C.1. Spatial and Temporal Discretization

Parallel (s) derivatives use 4th-order finite differences with 9-point stencils. The stencil class at each grid point is determined by `pos_par_grid_class` $\in \{-2, -1, 0, 1, 2\}$, encoding proximity to the open boundary. Interior points use centered 4th-order; boundary-adjacent points use upwinded 2nd-order. Upwind direction is selected by the sign of $v_\parallel$ (for streaming) or $\partial(J_0\phi)/\partial s$ (for Term VII). Stencil coefficients and upwind selection are precomputed and fused into `s_total_upar` and `s_total_t7` arrays, eliminating per-step branching inside the RK4 loop.

Parallel velocity ($v_\parallel$) derivatives use 4th-order centered stencils with zero-padding at the grid boundaries (no-flux condition).

Perpendicular coordinates are resolved in spectral space. The nonlinear ExB term uses 2D real-to-complex FFTs with 3/2-rule zero-padding for dealiasing. The dealiased grid size is chosen to have small prime factors (≤ 7), with a preference for powers of two. Physical modes are mapped to the FFT storage layout via the `jind` index array.

Time integration uses the classical 4th-order Runge-Kutta scheme. Each step requires four evaluations of the full RHS (phi solve + linear terms + nonlinear term). An optional CFL-adaptive mode estimates the timestep as the minimum of: (i) the nonlinear CFL from the maximum real-space ExB velocity gradient, (ii) a von Neumann stability analysis of the streaming and trapping stencils, and (iii) for kinetic electrons, a field CFL based on the electrostatic Alfvén frequency $\omega \sim k_\perp \sqrt{m_e/m_i}\, v_{th,i}/(q\,\Delta s\, B)$. The timestep is updated with one-step lag; the initial step uses a conservative linear-only estimate.

C.2. Mode Connectivity and Open Boundaries

Magnetic shear couples adjacent k_x modes across the parallel boundary: a mode at (k_x, k_y) that exits the field-line domain at $s = +\frac{1}{2}$ continues at $(k_x + \Delta k_x,\ k_y)$ with $\Delta k_x = 2\pi\hat{s}\, k_y$. This is discretized by grouping k_x modes into chains spaced `ikxspace` $= \lfloor 2\pi\hat{s}\,\Delta k_y/\Delta k_x \rceil$ apart. The `mode_label` array assigns the same integer label to all k_x indices in a chain for each k_y; `ixplus`/`ixminus` store the connected k_x index in the positive/negative s-direction (-1 = open boundary, no connection).

For the zonal mode ($k_y = 0$), the parallel domain is periodic (each k_x maps to itself). For $k_y > 0$, modes at the end of a chain see an open boundary: the stencil class `pos_par_grid_class` $\in \{-2, -1, 0, 1, 2\}$ is set to ± 2 and ± 1 at the first/second grid points adjacent to an open boundary, switching the parallel stencil from 4th-order centered to lower-order upwinded. Interior points (`class` $= 0$) use centered stencils.

The full connectivity is precomputed into `s_shift`, `kx_shift`, and `valid_shift` arrays of shape $(2L{+}1, N_s, N_{k_x}, N_{k_y})$ with $L = 4$ (the stencil half-width). These encode, for every stencil offset $\delta s \in [-4, 4]$, the target (s, k_x) index and whether the shifted point is in-grid. The parallel derivative is then a single gather-and-dot operation without any per-step branching on boundary conditions.

C.3. Multi-Species Architecture

The kinetic electron extension adds a leading species dimension to the distribution function: δf becomes $(N_{sp}, N_{v\parallel}, N_\mu, N_s, N_{kx}, N_{ky})$. All species-dependent precomputed arrays (Bessel functions, Maxwellians, drift velocities, fused stencils) similarly gain the species axis.

The RHS computation is vectorized over species using `jax.vmap`. Each species receives its own precomputed coefficients while sharing the same electrostatic potential ϕ from multi-species quasi-neutrality. The fused stencils have shape $(9, N_{sp}, N_{v\parallel}, \ldots)$; under `vmap` over species axis 1, each call sees $(9, N_{v\parallel}, \ldots)$—identical to the adiabatic shape.

Branching between adiabatic and kinetic paths uses Python `if/else` on `params.adiabatic_electrons`, which is a static pytree field resolved at JIT trace time. This ensures zero overhead for the adiabatic path.

C.4. Field Solver

The electrostatic potential is obtained algebraically from quasi-neutrality at each RK4 stage.

For adiabatic electrons, the numerator is the velocity-space integral $\sum_{v_\parallel,\mu} Z_i n_i J_0 B \Delta v_\parallel \Delta\mu \cdot \delta f_i$, and the denominator

includes both the ion polarization $Z_i^2 n_i(\Gamma_0^i - 1)/T_i$ and the adiabatic electron response $-Z_e n_e/T_e$. The zonal mode ($k_y = 0$) receives a flux-surface-averaged correction.

For kinetic electrons, both numerator and denominator sum over all kinetic species. The zonal mode denominator is set to unity (no flux-surface correction needed). Electron FLR effects are retained: $\Gamma_0^e \approx 0.99$–1.0 depending on $k_\perp$ (small but nonzero at high $k_\perp$ due to $m_e/m_i \approx 1/3600$).

C.5. JAX Optimization

The solver is designed for end-to-end XLA compilation via `jax.lax.scan`. The precomputed coefficient dictionary is created once outside the scan and captured as a closure variable, ensuring that FFT grid sizes (which must be concrete at trace time) remain available as Python integers rather than traced values.

`GKParams` uses a custom pytree registration where boolean and string fields (`non_linear`, `adiabatic_electrons`, `finit`) are stored as auxiliary data rather than leaves, enabling Python-level control flow branching without tracing issues.

Transport diagnostics (spectra, fluxes) are computed only at block boundaries via `get_integrals`, not at every RK4 stage, minimizing overhead. The Parseval factor P_{k_y} and parallel integration weight Δs are applied to spectral diagnostics for correct normalization against GKW reference data.

D. Optimizing JAX with CUDA kernels

As the complexity of the `gyaradax` right-hand side (RHS) scales, the memory access patterns of the fused finite-difference stencils and the dealiased Fast Fourier Transforms (FFTs) become the primary performance bottlenecks. While XLA provides excellent fusion for element-wise operations, certain patterns in `gyaradax`—such as high-rank gather-and-dot operations for streaming and mirroring terms—often result in suboptimal memory layouts or redundant global memory accesses.

Such limitations of XLA, particularly for codebases that do not heavily rely on element-wise operations or dense linear algebra, have been documented in the literature (Jakob et al., 2022). Following approaches outlined in recent work on LLM-driven kernel generation and optimization (Ouyang et al., 2025; Zhu et al., 2026; Wei et al., 2025), we establish automated CUDA kernel optimization loops driven by coding agents.

Component-wise solver benchmarks: First, we establish baselines for the high-level components of the `gyaradax` solver (see Appendix B). We measure execution speed, high bandwidth memory (HBM) utilization, arithmetic intensity, and numerical accuracy (measured via the L_2 error relative to the initial, numerically validated JAX implementation).

XLA FFI setup: Paralleling the JAX implementation, we initialize an empty C++ shared library with a functioning XLA Foreign Function Interface (FFI) for CUDA kernels, adhering to the official XLA and JAX documentation. We also provide a CMake configuration to build and compile the CUDA kernels with the appropriate compilers and shared libraries corresponding to our development JAX version.

Generating optimization proposals: Next, we prompt deep reasoning models (Gemini 3.1 Pro, Claude 4.6 Opus) to propose optimizations. To generate highly structured, iterable proposals, we instruct the models to categorize their proposals into three tiers: (1) JAX-level optimizations, (2) CUDA-level optimizations, and (3) algorithmic optimizations. For the CUDA-level, the models typically decompose their proposals into several progressive optimization steps.

Testing proposals: Following the generation phase, we deploy fast coding models (Gemini 3 Flash, Claude 4.5 Sonnet) to implement the Tier 1 and Tier 2 proposals. Empirically, the JAX-level optimizations yielded marginal improvements; we attribute this to the highly optimized nature of the XLA compiler's existing passes and the high quality of the initial JAX code generated by the agents out of the box.

Conversely, the iterative CUDA kernel optimization approach proved highly successful, yielding significant speedups across various solver components. In the following subsections, we outline the specific operation types for which XLA struggles to compile efficient kernels, and detail how we replaced these with LLM-generated custom CUDA kernels.

D.1. Fused Linear RHS Kernel

The linear right-hand side evaluation combines a nine-point parallel stencil, a five-point velocity-space stencil, and eight elementwise physics terms (drift advection, trapping, hyper-diffusion, equilibrium drive, and field-drive coupling) into a single expression. XLA's compiler already fuses most of this computation into one monolithic kernel, but inspection of the HLO intermediate representation revealed two structural inefficiencies it cannot resolve.

First, XLA implements the velocity-space stencil through general-purpose gather operations with runtime boundary predicates, generating five full-array index computations and conditional selects where a simple stride-based access with compile-time boundary checks suffices. Second, XLA passes all coefficient arrays at their natural declaration shapes—some lacking the magnetic moment dimension, others lacking radial or toroidal wavenumber axes—and broadcasts them element-wise within the fused kernel, resulting in redundant address arithmetic and suboptimal register reuse across the velocity dimension.

Our fused CUDA kernel addresses both limitations in a single pass: the velocity stencil uses direct strided addressing with compile-time boundary constants, eliminating the gather and predicate machinery entirely. All coefficient arrays are stored in their minimal shapes, with index reconstruction computed from the thread identity. Velocity-axis tiling further amortizes the parallel stencil map lookups across eight consecutive velocity indices sharing the same magnetic moment, converting what XLA executes as independent per-element gathers into sequential streaming reads. The resulting kernel outperforms XLA's auto-fused baseline on the full linear RK4 step while maintaining numerical parity up to machine precision.

D.2. Nonlinear RHS: Fused FFTs

The nonlinear $\mathbf{E} \times \mathbf{B}$ advection term is evaluated pseudospectrally via a Poisson bracket that requires four inverse FFTs (to obtain real-space gradients $\partial_x \phi$, $\partial_y \phi$, $\partial_x f$, $\partial_y f$), a pointwise bracket multiplication, and one forward FFT. XLA's kernel fusion has a fundamental limitation here: the FFTs are dispatched as opaque calls to cuFFT, which acts as a hard fusion barrier. Therefore, the spectral derivative multiplications preceding each inverse FFT and the bracket multiplication following them must be compiled as separate kernels

LTO callback fusion. We replace this pipeline with three fused cuFFT kernel launches using Link-Time Optimization (LTO) device callbacks (NVIDIA Corporation, 2026). LTO callbacks are compiled to device-relocatable code (`-dlto`) and linked into the cuFFT plan at plan-creation time via `cufftXtSetJITCallback`. This allows arbitrary user logic to execute *inside* cuFFT's butterfly passes, at the exact point where each spectral element is loaded from or stored to global memory.

We exploit this mechanism at three points in the pipeline (see Figure 8a):

1. **Inverse FFT load callback:** Fuses the scatter-to-dense-grid, spectral derivative multiplication (ik_x, ik_y), gyro-averaging (Bessel J_0 multiplication), and amplitude scaling into a single register-only computation. Each packed spectral element is read from HBM exactly once; the callback returns the fully processed value directly to cuFFT's butterfly registers.

2. **Forward FFT load callback:** Fuses the Poisson bracket computation. Instead of materializing four real-space gradient arrays and computing $\partial_y \phi \, \partial_x f - \partial_x \phi \, \partial_y f$ in a separate kernel, the callback reads the gradient arrays and computes the bracket in-flight, returning the result to the forward FFT without an intermediate HBM round-trip.

3. **Forward FFT store callback:** Writes directly to the packed output spectrum, skipping the 59% of dealiased modes that would otherwise be written and immediately discarded.

D.3. Two-for-One Spectral Packing

To further reduce the cuFFT overhead, the reasoning model proposed exploiting the linearity of the discrete Fourier transform (DFT). If two signals A and B are real-valued in physical space, their spectral representations are Hermitian-symmetric. The agent's optimization combines them into a single complex signal $Z_k = A_k + iB_k$, which can be inverse-transformed via a single complex-to-complex (C2C) FFT to recover $A = \mathrm{Re}(\mathrm{IFFT}(Z))$ and $B = \mathrm{Im}(\mathrm{IFFT}(Z))$. Specifically, we pack both spatial derivatives of the *same* physical field into each C2C transform:

$$Z_k^{(\phi)} = \frac{ik_x\hat{\phi}_k}{\alpha_\phi} + i\,\frac{ik_y\hat{\phi}_k}{\beta_\phi}, \quad Z_k^{(f)} = \frac{ik_x\hat{f}_k}{\alpha_f} + i\,\frac{ik_y\hat{f}_k}{\beta_f}$$

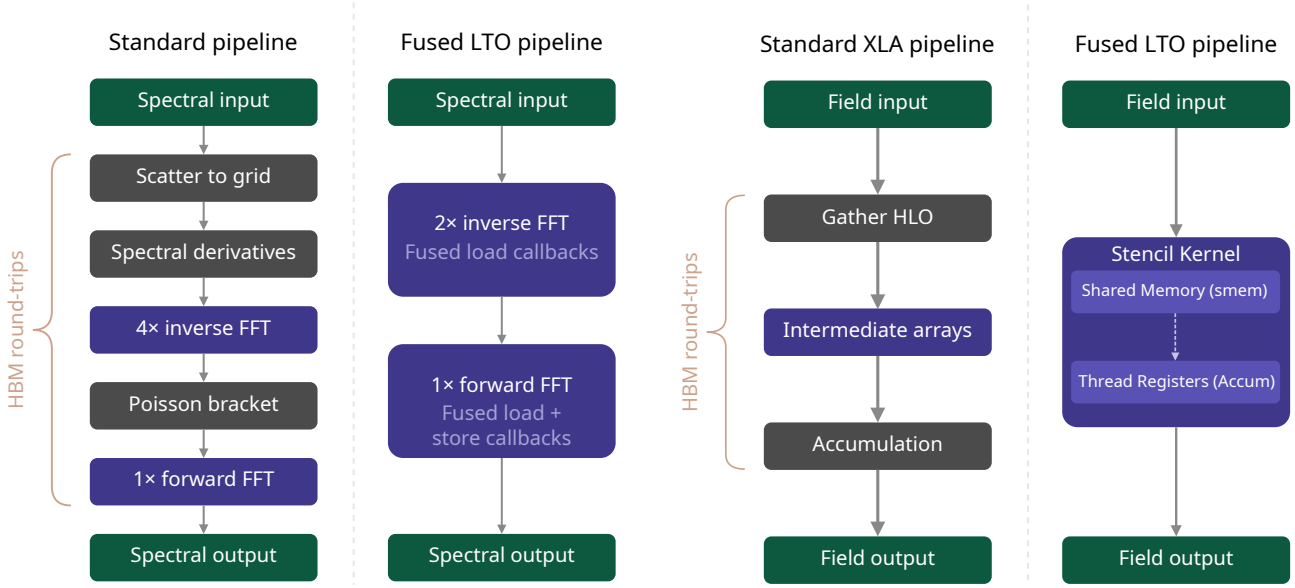
where α and β are dynamic scaling factors computed from the spectral magnitudes of each field. This strategy successfully halves the inverse FFT count from four to two. The scaling factors ensure both channels occupy comparable dynamic ranges, preventing floating-point leakage. Because both derivatives originate from the same field, the ratio α/β is inherently bounded by the grid anisotropy $\max|k_x|/\max|k_y|$, an $\mathcal{O}(1)$ property of the grid geometry.

Hermitian symmetrization. The C2C transform operates on the full two-dimensional spectrum rather than the half-spectrum used by the standard real-to-complex path. The load callback synthesizes the conjugate half via Hermitian extension. For this packing to produce mathematically exact results, the spectrum of each derivative must be perfectly Hermitian-symmetric; otherwise, imaginary leakage crosses into the other channel.

During the implementation of this agent-proposed kernel, the automated evaluation harness consistently flagged numerical divergence. Investigating this strict validation failure led to the discovery that the gyro-averaged potential $\hat{\phi}_k$ exhibits a small but non-negligible symmetry defect (up to 14% relative error) at the zeroth wavenumber, introduced by the Bessel function multiplication during gyro-averaging.

While entirely invisible to the baseline JAX real-to-complex inverse FFT—which inherently discards the imaginary output by construction—this asymmetry produced a parasitic imaginary component in the C2C output that corrupted the packed channels. To resolve this and satisfy the evaluation harness, an explicit Hermitian symmetrization step was added to the load callback: for each mirror pair $(k_x, -k_x)$, both values are dynamically replaced with their Hermitian average. This algorithmic correction, directly prompted by the agentic verification loop, eliminated the channel leakage entirely and reduced the relative error from $\mathcal{O}(10^{-4})$ to $\mathcal{O}(10^{-14})$ in FP64 and from $\mathcal{O}(10^{-2})$ to $\mathcal{O}(10^{-6})$ in FP32.

For the mixed precision configuration, this results in an accumulated error of $\mathcal{O}(10^{-10})$ for an entire RK4 step, maintaining overall solver stability.



(a) Fused LTO pipeline *(b)* Stencil CUDA fusion

Figure 8. Overview of our memory optimization strategies.

E. Comparison with GKW Reference Code

While `gyaradax` aligns with the local flux-tube physics of the GKW framework, several advanced features of the Fortran reference code are identified as future enhancements.

Physics. GKW supports full electromagnetic dynamics, including the parallel vector potential ($A_\parallel$) and perpendicular magnetic perturbations ($B_\parallel$); gyaradax is currently limited to the electrostatic limit ($\beta = 0$). GKW features linearized collision operators including pitch-angle scattering and energy diffusion with conservation terms, whereas gyaradax is collisionless. Rotation physics—Coriolis drifts and centrifugal forces arising in rotating frames (Term VI in the GKW numbering)—is implemented in GKW but omitted here. Background Krook sources and buffer regions for global simulations are also absent.

Numerical methods. gyaradax uses an explicit RK4 scheme exclusively. GKW additionally supports implicit and semi-implicit integrators for stiff parallel streaming, as well as Runge-Kutta-Chebyshev (RKC) schemes for diffusion-dominated regimes. GKW also supports global simulations with non-uniform background gradients and Dirichlet boundary conditions; gyaradax is restricted to the local flux-tube approximation.

HPC and scalability. GKW features highly optimized MPI-based domain decomposition across species, field-line, and velocity dimensions. While JAX provides native array sharding via jax.sharding, distributing the 6D kinetic grid across multiple GPUs remains a planned enhancement. gyaradax uses mixed precision for the nonlinear FFTs (FP32) while keeping linear terms in FP64, which could be further extended to cover additional hot paths.

F. Analytical Benchmark Details

F.1. Rosenbluth-Hinton Zonal Flow Test

The Rosenbluth-Hinton test (Rosenbluth & Hinton, 1998) initialises a radial ($k_\psi \neq 0$, $k_\zeta = 0$) density perturbation that excites a geodesic acoustic mode (GAM). The mode oscillates at the GAM frequency and damps via collisionless Landau damping, leaving a residual zonal potential given analytically by (Xiao & Catto, 2006):

$$\frac{\phi(\infty)}{\phi(0)} = \frac{1}{1 + q^2\Theta/\varepsilon^2}, \qquad \Theta = 1.6\,\varepsilon^{3/2} + 0.5\,\varepsilon^2 + 0.36\,\varepsilon^{5/2}. \tag{5}$$

We use the GKW benchmark parameters from gkw_ref/benchmarks/zonal_flow/zonal01: q=1.3, $\hat{s}$=0.1592, ε=0.05, s-alpha geometry, N_s=128, $N_{v\parallel}$=128, N_μ=16, $k_\psi\rho_s$≈0.025 (krhomax=0.025, ikxspace=1), $\nu_\parallel$=0.01 (disp_par), dt=0.01, finit='zonal', amp_init=10^{-4}. The parallel dissipation damps velocity-space filamentation, enabling convergence to the analytical residual. The simulation runs for t=100 (R/v_{th}). At t>80, the residual converges to 0.0711, matching the Xiao-Catto prediction to 0.1%.

The ε-scan (Figure 4b) uses the same parameters with $\varepsilon \in \{0.05, 0.1, 0.15, 0.2, 0.25\}$, each run to t=100.

F.2. Cyclone Base Case Linear ITG

The Cyclone Base Case (Dimits et al., 2000) is the standard linear benchmark: q=1.4, $\hat{s}$=0.78, ε=0.19, R/L_T=6.9, R/L_n=2.2, T_e/T_i=1, electrostatic, adiabatic electrons, s-alpha geometry.

We use the GKW benchmark parameters from gkw_ref/benchmarks/cyclone/linear: N_s=160, $N_{v\parallel}$=64, N_μ=16, n_{period}=5, $\nu_\parallel$=1.0, dt=0.003, naverage=100. The high parallel resolution (N_s=160 across 5 poloidal periods) is essential for resolving Landau damping; at lower resolution (e.g. N_s=16), growth rates are systematically overestimated due to insufficient velocity-space phase mixing. Note that naverage ≥ 10 is required for correct growth rate measurement, as the ITG mode's finite real frequency causes the amplitude to oscillate within a single timestep.

The $k_\theta\rho_s$ scan (Figure 5a) covers $[0.1, 0.8]$ at fixed R/L_T=6.9. The R/L_T scan (Figure 5b) covers $\{6.9, 8.28, 10.35, 12.44, 15.18\}$ at fixed $k_\theta\rho_s$=0.5. Each run evolves 300 naverage windows. Growth rates match the GKW reference (run with identical parameters) to within 1%.

G. Validation Overview

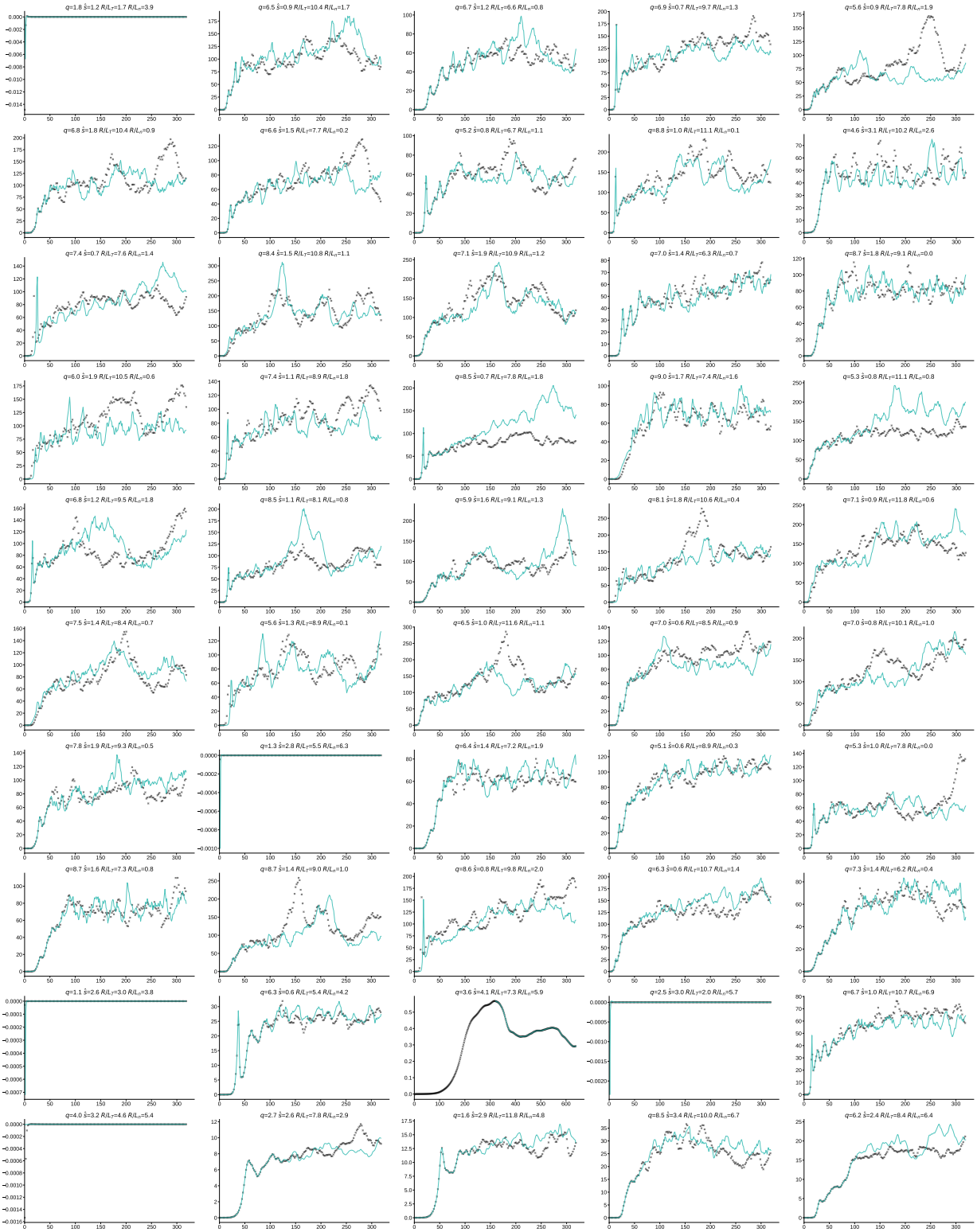


Figure 9. Heat flux time traces for all adiabatic validation configurations. Each panel shows `gyaradax` (cyan) versus GKW (black markers) for a distinct combination of q, $\hat{s}$, R/L_T, and R/L_n.

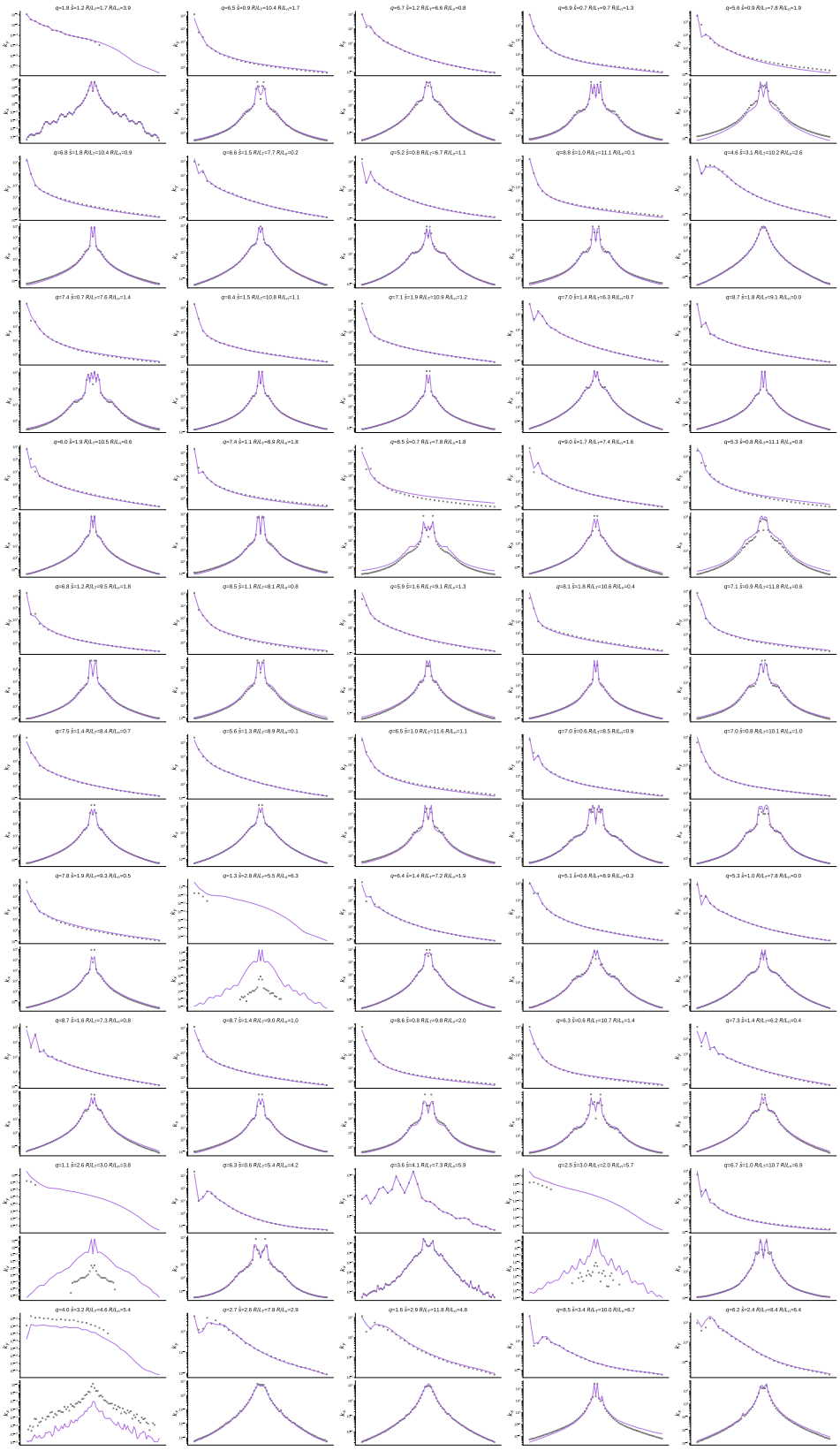


Figure 10. Time-averaged k_y (top) and k_x (bottom) spectral profiles for all adiabatic validation configurations. `gyaradax` (purple lines) versus GKW (black markers).

H. PROMPT.md

PROJECT MANDATES: JAX Reimplementation of GKW

Documentation & Note-Taking

- **Thoroughness is Mandatory:** You must be extremely thorough in your note-taking. Do not hesitate to produce long, detailed files.
- **Granular Detail:** Document everything required for reimplementation and more: specific subroutine logic, module dependencies, variable mappings, numerical constants, and edge-case handling.
- **Foundational Reference:** Notes in `GKW.md` (and similar files) serve as the foundational technical specification for the JAX port.

CONTEXT FILES PROVIDED:

- `@jax_integrals.py` (Contains the implemented JAX flux integrals).
- `@utils.py`, `@jax_geometry.py` (Data, potential/phi, and geometry loading helpers).
- `@test_jax_integral.py`
- `@test_linear.py`, `@test_nonlinear.py` (Empirically verify content against GKW reference trajectories).

ROLE: Act as an autonomous, expert Scientific Computing Engineer. Your objective is to translate the GKW Fortran code into JAX. Describe what you plan to do at the main intermediate steps. **Crucially, stop and yield to the user for feedback at the end of each numbered execution phase before proceeding to the next.**

SCOPE & CONSTRAINTS (STRICT):

- **Physics:** Adiabatic electron case ONLY. Simplify all equations to reflect this.
- **Grid Resolution:** (vpar, mu, s, x, ky) = (32, 8, 16, 85, 32)
- **Time step:** dt = 0.01
- **Framework:** JAX. All functions must be pure, differentiable, and `jit`-compatible.
- **Precision:** ALWAYS use float64 (fp64). JAX must be configured to use 64-bit precision.
- **Integrity:** STRICTLY NO normalization sweeps to force test passes. NO cheating or hacking the outputs to match expected results. NO hardcoded constants—derive everything mathematically from the source code and physical constants.
- **Environment:** STRICTLY use the environment under `<PATH_TO_ENV>` for everything. STRICTLY use GPU:0 for everything.

ENVIRONMENT:

Use `@utils.py` for data loading and potential (phi) calculations. Do NOT reimplement these helpers.
CRITICAL: The flux integrals have already been implemented in JAX within `@jax_integrals.py`. Your first coding task is to verify that they function correctly before proceeding to the main solver.

REQUIRED INTERFACE:

You MUST expose the following purely functional interface for the core solver:

```
next_df, (phi, fluxes) = gksolve(prev_df, ...)
```

EXECUTION PLAN (FOLLOW EXACTLY IN ORDER):

Do not write the final solver code immediately. You must execute the following steps sequentially. Use your toolset to explore directories, read files, write code, and run tests. Output and save your notes and progress to markdown artifacts for each step before pausing.

1. **INITIAL CONTEXT INGESTION:** Before starting any code translation or detailed planning, explore the `gkw_ref/src` (Fortran code) and `gkw_ref/manual` (LaTeX files) directories. You MUST fully load the following specific files into your context, and explore for any other useful ones:
   - `gkw_ref/src/gkw.f90` (Main loop over large time steps, normalisation).
   - `gkw_ref/src/exp_integration.F90` (Explicit integration schemes like rk4, RHS assembly order).
   - `gkw_ref/src/linear_terms.f90` (Linear operators, field coupling matrices, poisson splits).
   - `gkw_ref/src/fields.F90` (Field solve application, zonal adiabatic correction).
   - `gkw_ref/src/components.f90` (Adiabatic electrons flag, species setup).

   Individuate any additional files that are relevant to time integration and the adiabatic electron formulation. **Stop and wait for the user to approve the final identified file list.**
2. **VERIFY JAX FLUX INTEGRALS:**
   - Review the existing JAX integrals in `@jax_integrals.py`.
   - Verify that the geometry loading functions from `@jax_geometry.py` work seamlessly with this JAX implementation.
   - Run and adapt `@test_jax_integral.py` if necessary to confirm the JAX integrals pass all tests.
   - TAKE NOTES on any specific broadcasting, memory layout, or numerical precision details observed during verification. **Stop and wait for user approval.**
3. **EXPLORATION, ANALYSIS & PLANNING (NOTE-TAKING):**
   - **Source Code & Theory:** Actively analyze the Fortran source code and LaTeX manual files you loaded in Step 1. Focus strictly on time integration and the adiabatic electron formulation.
   - **Reference Data:** Ingest and explore the reference trajectory at `<PATH_TO_DATA>/iteration_13` to understand data structures, empirical array shapes, and physical scales.
   - **Synthesize & Plan:** Based on the Fortran code, the LaTeX manual, and the reference data, write out a high-level plan for the core JAX architecture. Identify exactly which Fortran modules and subroutines map to the `gksolve` update step, detail the exact mathematical update equations, and map out the variables.
   - **Artifact Generation:** Save all of these findings into a detailed `GKW.md` file. **Stop and wait for user approval.**
4. **ESTABLISH CORE TESTS (TDD):** Write the test suite for the core simulator before implementing it. These tests must run independently.
   - **Mandatory:** Write a validation test that checks calculated growth rates against the `growth_rate_all_modes` file.
   - **Mandatory:** Write unit tests verifying array shapes (based on your notes from Step 3) and basic conservation properties. **Stop and wait for user approval.**
5. **IMPLEMENT LINEAR `gksolve`:** Write the `gksolve` function for the LINEAR case, based on the notes you created (Terms I, II, IV, V, VII, VIII, Diffusion). Proceed to this step ONLY after all prior tests are passing. **Stop and wait for user approval.**
6. **IMPLEMENT NONLINEAR `gksolve`:** Finalize `gksolve` for the NONLINEAR case by extending the LINEAR version with Term III. Proceed to this step ONLY after all prior tests are passing, ESPECIALLY `@test_linear.py`. TO CONCLUDE, make sure that both `@test_linear.py` and `@test_nonlinear.py` PASS. **Report the final test results.**