moljax: GPU-Accelerated Stiff Transport–Reaction PDE Simulation via Adaptive Method-of-Lines in JAX

Gorgi Pavlov[a,b,*]

[a] *Lehigh University, Bethlehem, PA, United States of America*
[b] *Johnson & Johnson, Malvern, PA, United States of America*

Abstract

Efficient simulation of stiff transport–reaction PDEs remains a bottleneck in chemical engineering workflows that require parameter sweeps, optimization, and control. We present `moljax`, a JAX-native method-of-lines framework that unifies: (i) fully JIT-compiled adaptive time stepping (including accept/reject control flow) for GPU/TPU execution; (ii) matrix-free implicit integration via Newton–Krylov methods using machine-precision Jacobian–vector products (JVP) from automatic differentiation; and (iii) physics-aware spectral/FFT operators (FFT/DST/DCT) enabling $O(N \log N)$ linear solves, exponential propagators, and diffusion preconditioning. At matched accuracy levels, FFT-diagonalized implicit solvers achieve 12–36× speedup over adaptive explicit integrators for diffusion problems, with FFT-based preconditioning reducing GMRES iterations by 10–100× (>700× at extreme stiffness). IMEX and exponential integrators provide 15–40× speedup over explicit methods for stiff reaction–diffusion systems. A tubular-reactor case study demonstrates process-relevant outputs (outlet conversion) across Péclet and Damköhler regimes. **Scope:** These speedups apply to structured grids with constant coefficients and periodic, Dirichlet, or Neumann boundaries where FFT diagonalization is applicable; problems with complex geometry, nonlinear diffusion, or unstructured meshes require different approaches. Open-source code and reproducible scripts are provided.

Keywords: method of lines, adaptive time stepping, Newton–Krylov, automatic differentiation, FFT preconditioning, GPU acceleration, JAX, stiff PDEs, tubular reactor

1. Introduction

1.1. Motivation: The Computational Bottleneck in Process Simulation

The numerical solution of partial differential equations (PDEs) remains fundamental to chemical engineering applications including reaction-diffusion systems [1], transport phenomena [2], and process dynamics [3]. As industrial processes become more complex—multi-scale reactors, coupled transport phenomena, real-time optimization—the computational demands of PDE simulation have grown correspondingly. A single high-fidelity simulation of a tubular reactor with detailed kinetics may require hours on conventional hardware, making parameter sweeps, uncertainty quantification, and model predictive control computationally prohibitive.

Motivating example: Tubular reactor design.. Consider a tubular reactor with axial dispersion, first-order kinetics, and Danckwerts boundary conditions—a canonical problem in reactor engineering [4, 5]. Designing such reactors requires exploring the Péclet–Damköhler parameter space to predict outlet conversion across operating regimes. Traditional MOL solvers using SciPy Radau require ∼8 seconds per simulation; with `moljax`, the same problem solves in ∼0.15 seconds on GPU—a 56× speedup (Section 5.11). This enables real-time parameter sweeps, gradient-based optimization via JAX's automatic differentiation, and integration into machine learning workflows.

The method of lines (MOL) provides a systematic approach to PDE discretization by converting spatial derivatives to algebraic expressions, reducing the PDE to a system of ordinary differential equations (ODEs)

*Corresponding author
Email address: `gop214@lehigh.edu` (Gorgi Pavlov)

that can be solved with established time integrators [6, 7]. MOL is particularly attractive for chemical engineering applications because:

1. It leverages decades of robust ODE solver development (stiff integrators, error control),
2. It naturally handles multi-physics coupling through the ODE right-hand side,
3. It provides a clear separation between spatial discretization and temporal integration.

1.2. The Gap: Why Existing Tools Fall Short

Recent advances in hardware acceleration through GPUs and TPUs offer transformative speedups for scientific computing—often 10–100× over optimized CPU code—yet exploiting this potential for MOL-based PDE solving requires fundamental algorithmic redesign [8, 9]. We identify three specific barriers that current tools fail to address simultaneously:

1. **Control flow incompatibility**: Adaptive time stepping requires branching (accept/reject decisions) that conflicts with GPU kernel compilation. Standard implementations use Python `if`/`while` statements that cannot be traced by JIT compilers, forcing a choice between adaptivity and acceleration.
2. **Memory bandwidth limitations**: Dense Jacobian assembly for implicit methods scales as $O(N^2)$ in memory and $O(N^3)$ in computation. For a 512×512 2D grid with 2 species, storing the Jacobian requires 275 GB—exceeding GPU memory by orders of magnitude.
3. **Lack of physics-aware acceleration**: Generic sparse solvers (e.g., SuperLU, UMFPACK) do not exploit the specific structure of PDE operators. For periodic diffusion, the Laplacian is diagonalized by the FFT, enabling $O(N \log N)$ solves—but this requires tight integration between discretization and solver.

Existing tools address subsets of these challenges: SciPy [10] provides robust adaptive integrators but lacks GPU support. DifferentialEquations.jl [11] offers comprehensive features including GPU backends, automatic Jacobians via SparseDiffTools.jl, and Newton–Krylov solvers via Krylov.jl, representing the state-of-the-art in scientific ODE/PDE solving. Diffrax [12] provides JAX-native ODE/SDE solvers with GPU support and adaptive stepping. JAX-CFD [13] demonstrates JAX's potential for PDEs but focuses on fixed-timestep spectral methods for turbulence modeling.

The gap we address: DifferentialEquations.jl [11] in Julia provides comprehensive Newton–Krylov support with AD-based Jacobians and GPU backends, representing the state of the art. However, no existing *Python* library provides simultaneously: (1) JIT-compiled adaptive stepping with accept/reject logic on GPU/TPU, (2) matrix-free implicit solvers with automatic Jacobians via forward-mode AD, and (3) FFT-accelerated operators with physics-based preconditioning. For teams already using Python/JAX for machine learning workflows, `moljax` provides these capabilities without language switching. Table 1 summarizes feature availability across libraries.

Table 1: Feature comparison across PDE/ODE libraries. • = supported, ○ = partial/requires configuration, — = not available. See footnotes for feature definitions.

Feature	SciPy	DiffEq.jl	Diffrax	JAX-CFD	moljax
Adaptive stepping[a]	•	•	•	—	•
GPU acceleration[b]	—	•	•	•	•
Auto Jacobian[c]	—	•	•	—	•
Matrix-free NK[d]	—	•	○	—	•
FFT implicit solves[e]	—	—	—	•	•
FFT preconditioning	—	—	—	—	•
JIT adaptive loop[f]	—	N/A	•	—	•

[a]Error-controlled step size with accept/reject logic. [b]Native GPU kernel execution; DiffEq.jl via CUDA.jl/DiffEqGPU.jl. [c]Jacobian-vector products via AD; DiffEq.jl via SparseDiffTools.jl. [d]Built-in Newton–Krylov solver; DiffEq.jl via Krylov.jl; Diffrax via Lineax. [e]$O(N \log N)$ implicit diffusion solves via FFT diagonalization. [f]Adaptive loop (while/cond) compiled to GPU kernel, not host-side interpreter.

1.3. Research Questions

This work addresses three research questions:

RQ1: Can adaptive time stepping with accept/reject logic be made fully JIT-compatible for GPU/TPU execution without sacrificing accuracy control?

RQ2: How much do FFT-based preconditioners reduce Newton–Krylov iteration counts for diffusion-dominated PDEs, and under what conditions?

RQ3: What are the performance trade-offs between explicit, implicit, and IMEX methods when all are implemented within a unified JAX framework?

1.4. Contributions

We present `moljax`, a JAX-native method-of-lines framework that addresses these challenges. Our specific contributions are:

1. **JIT-compiled adaptive time stepping** (addressing RQ1): Full accept/reject control flow compiled via `lax.while_loop`/`lax.cond`, enabling GPU/TPU execution without Python interpreter overhead while maintaining rigorous error control.
2. **Matrix-free Newton–Krylov with AD-JVP** (addressing RQ2): Machine-precision Jacobian–vector products via forward-mode automatic differentiation eliminate finite-difference tuning and $O(N^2)$ Jacobian storage. Combined with FFT-based preconditioning, this typically reduces GMRES iterations by 10–100× for diffusion-dominated problems, with >700× reduction at extreme stiffness ratios (Table 8).
3. **Physics-aware FFT/DST/DCT operators** (addressing RQ3): $O(N \log N)$ linear solves, exponential propagators, and diffusion preconditioning enable efficient implicit and IMEX methods. We provide systematic comparison across explicit, implicit, IMEX, and exponential integrators.
4. **Process-relevant validation**: A tubular-reactor case study with axial dispersion and first-order kinetics demonstrates boundary condition effects (Danckwerts, Robin, Neumann) across Péclet and Damköhler regimes, with outlet conversion as the engineering metric.

1.5. Scope and Limitations

To prevent misinterpretation of performance claims, we explicitly state what `moljax` does *not* attempt:

- **Unstructured meshes**: FFT diagonalization requires structured (Cartesian) grids. Complex geometries requiring finite element or unstructured finite volume methods are outside our scope.
- **Variable coefficients**: Our FFT-based operators assume constant diffusion coefficients. Spatially-varying $D(\mathbf{x})$ breaks diagonalization and requires iterative methods without the $O(N \log N)$ advantage.
- **Non-periodic boundaries with complex geometry**: While we support Dirichlet (DST), Neumann (DCT), and Robin boundaries via finite differences, problems with irregular domains or mixed boundary types may not benefit from spectral acceleration.
- **Highly nonlinear diffusion**: Equations of the form $\partial_t u = \nabla \cdot (D(u)\nabla u)$ require Newton iteration on the full spatial operator, limiting FFT advantages.
- **Adaptive mesh refinement (AMR)**: We use fixed uniform grids; problems with localized features (shocks, fronts) may benefit from AMR methods not provided here.

For problems satisfying our assumptions (structured grids, constant coefficients, standard boundary conditions), the speedups reported are reproducible and validated against analytical solutions. For problems outside this scope, general-purpose libraries such as DifferentialEquations.jl [11] or PETSc [14] may be more appropriate.

1.6. Paper Organization

The remainder of this paper is organized as follows. Section 2 reviews MOL fundamentals and positions our work relative to existing libraries. Section 3 presents the mathematical formulation, deriving *why* each algorithmic choice is made. Section 4 describes the software architecture and JAX-specific design patterns. Section 5 provides performance benchmarks addressing our research questions. Section 6 interprets results and provides method selection guidance. Section 7 concludes with future directions.

2. Background and Related Work

2.1. Method of Lines

The method of lines transforms a PDE of the form

$$\frac{\partial \mathbf{u}}{\partial t} = \mathcal{L}[\mathbf{u}] + \mathcal{N}[\mathbf{u}], \quad \mathbf{x} \in \Omega, \quad t > 0 \tag{1}$$

where $\mathcal{L}$ is a linear spatial operator (e.g., diffusion) and $\mathcal{N}$ is a nonlinear operator (e.g., reaction kinetics), into a semi-discrete system

$$\frac{d\mathbf{U}}{dt} = \mathbf{L}\mathbf{U} + \mathbf{N}(\mathbf{U}) \tag{2}$$

where $\mathbf{U}(t) \in \mathbb{R}^N$ collects the discrete solution values and $\mathbf{L}$ is the discretized linear operator matrix.

For reaction-diffusion systems common in chemical engineering:

$$\frac{\partial c_i}{\partial t} = D_i \nabla^2 c_i + R_i(\mathbf{c}), \quad i = 1, \ldots, n_s \tag{3}$$

the linear operator $\mathcal{L} = D_i \nabla^2$ represents diffusion while $\mathcal{N} = R_i$ captures (possibly stiff) reaction kinetics.

2.2. Related Work

Several established libraries address PDE simulation with varying approaches to hardware acceleration:

SciPy [10] provides `solve_ivp` with adaptive Runge–Kutta and BDF methods but lacks GPU support and requires explicit Jacobian functions for implicit methods.

DifferentialEquations.jl [11] offers comprehensive ODE/PDE solving with automatic stiffness detection and GPU support via CUDA.jl, representing the current state-of-the-art in scientific ODE solving.

FEniCS [15] and **Firedrake** [16] provide finite element frameworks with PETSc integration but focus on spatial discretization rather than MOL time integration.

JAX-based PDE libraries: Recent work includes `jax-cfd` for computational fluid dynamics [13] and neural PDE solvers [17], but general-purpose MOL engines with full Newton–Krylov/IMEX support remain limited.

Our contribution differs from these approaches by providing:

1. Full JIT compilation of adaptive stepping including accept/reject logic,
2. Matrix-free Newton–Krylov with automatic differentiation (no symbolic Jacobians),
3. Exact $O(N \log N)$ FFT-based implicit solves and preconditioners,
4. Multi-field PyTree support for coupled systems.

3. Mathematical Formulation and Numerical Methods

This section presents the mathematical foundation of `moljax`, emphasizing not just *how* each method works but *why* it is the appropriate choice for GPU-accelerated chemical engineering simulations.

3.1. Spatial Discretization

3.1.1. Why Finite Differences over Finite Elements?

For regular geometries common in chemical engineering (tubular reactors, packed beds, membrane systems), finite difference (FD) methods offer key advantages:

- **Structured data layout**: FD on uniform grids produces regular array operations that map efficiently to GPU SIMD architectures.
- **Explicit stencils**: The sparsity pattern is known *a priori*, enabling FFT-based fast solvers.
- **Simplicity**: No mesh generation or element assembly required.

Finite element methods excel for complex geometries but introduce irregular memory access patterns that reduce GPU efficiency. For the target applications (reaction-diffusion, transport in regular domains), FD provides the best performance/complexity trade-off.

3.1.2. Finite Difference Operators

For uniform grids with spacing Δx, we implement standard centered finite differences for the Laplacian:

$$\text{2nd order:} \quad \nabla_h^2 u_i = \frac{u_{i-1} - 2u_i + u_{i+1}}{\Delta x^2} \tag{4}$$

$$\text{4th order:} \quad \nabla_h^2 u_i = \frac{-u_{i-2} + 16u_{i-1} - 30u_i + 16u_{i+1} - u_{i+2}}{12\Delta x^2} \tag{5}$$

with analogous 6th-order stencils available.

3.1.3. Why FFT Diagonalization?

The key insight enabling our approach is that **circulant matrices are diagonalized by the DFT**. For periodic boundary conditions, the discrete Laplacian $\mathbf{L}$ is circulant, meaning:

$$\mathbf{L} = \mathbf{F}^{-1}\mathbf{\Lambda}\mathbf{F} \tag{6}$$

where $\mathbf{F}$ is the DFT matrix and $\mathbf{\Lambda} = \text{diag}(\lambda_0, \ldots, \lambda_{N-1})$ contains the eigenvalues.

This diagonalization has profound computational implications:

- **Matrix-vector products**: $\mathbf{Lu} = \mathbf{F}^{-1}[\boldsymbol{\lambda} \odot \mathbf{F}[\mathbf{u}]]$ costs $O(N \log N)$ via FFT, not $O(N^2)$.
- **Linear solves**: $(\mathbf{I} - \alpha\mathbf{L})^{-1}\mathbf{b}$ becomes pointwise division in Fourier space—also $O(N \log N)$.
- **Matrix exponentials**: $e^{t\mathbf{L}}\mathbf{u}$ is computed exactly via $\mathbf{F}^{-1}[e^{t\boldsymbol{\lambda}} \odot \mathbf{F}[\mathbf{u}]]$.

These operations are the computational bottleneck in implicit and exponential integrators. FFT diagonalization reduces their complexity from $O(N^2)$–$O(N^3)$ to $O(N \log N)$, enabling practical implicit methods for large grids.

3.1.4. FFT-Diagonalized Operators

For periodic boundary conditions, the discrete Laplacian is diagonalized by the discrete Fourier transform. The eigenvalues of the 2nd-order 1D Laplacian are:

$$\lambda_k = \frac{2(\cos(k\Delta x) - 1)}{\Delta x^2}, \quad k = \frac{2\pi m}{L}, \quad m = 0, 1, \ldots, N-1 \tag{7}$$

This enables $O(N \log N)$ matrix-vector products:

$$\mathbf{Lu} = \mathcal{F}^{-1}[\boldsymbol{\lambda} \odot \mathcal{F}[\mathbf{u}]] \tag{8}$$

where $\mathcal{F}$ denotes the FFT, $\boldsymbol{\lambda}$ is the vector of eigenvalues, and $\odot$ is elementwise multiplication.

For non-periodic boundary conditions, we employ:

- **Dirichlet**: Discrete Sine Transform (DST-I) with eigenvalues $\lambda_k = -\frac{4}{\Delta x^2}\sin^2\left(\frac{\pi k}{2(N+1)}\right)$
- **Neumann**: Discrete Cosine Transform (DCT-I) with analogous eigenvalues

Limitation: stencil order in FD eigenvalue diagonalization. The FFT-diagonalized *finite-difference* implicit solvers (not pseudo-spectral) currently support only 2nd-order spatial discretization. The reason is that 4th and 6th-order FD stencils have different spectral symbols:

$$\lambda_k^{(4)} = \frac{-\cos(2k\Delta x) + 16\cos(k\Delta x) - 15}{12\Delta x^2} \tag{9}$$

$$\lambda_k^{(6)} = \frac{\cos(3k\Delta x) - 18\cos(2k\Delta x) + 135\cos(k\Delta x) - 118}{90\Delta x^2} \tag{10}$$

While these can be computed and used for FFT-based matvecs, the current implementation uses 2nd-order eigenvalues for all implicit solves and preconditioners. Higher-order stencils are available for explicit right-hand-side evaluation. Users requiring 4th/6th-order implicit operators should use the standard finite-difference matvec path (not FFT-diagonalized), which has $O(N)$ complexity but higher constants.

3.1.5. Pseudo-Spectral vs. Finite-Difference: A Critical Distinction

`moljax` provides two distinct operator families that must not be conflated. For comprehensive treatment of spectral methods, see Trefethen [18], Boyd [19], and Canuto et al. [20].

Pseudo-spectral operators (periodic, smooth fields).. For smooth periodic solutions, we use the *continuous* Laplacian symbol $\lambda(\mathbf{k}) = -|\mathbf{k}|^2 = -(k_x^2 + k_y^2)$, where wavenumbers are computed via `fftfreq`. This achieves **spectral accuracy** in space: for sufficiently smooth solutions, the spatial discretization error decreases faster than any polynomial in Δx, typically reaching machine precision [21, 22]. The error is then dominated by time discretization.

Finite-difference operators (general BCs, nonsmooth fields).. For non-periodic boundaries or solutions with limited regularity, we use the discrete FD Laplacian eigenvalues from Eq. (7). While the FFT still enables $O(N \log N)$ solves, the spatial accuracy is limited to the FD stencil order (2nd, 4th, or 6th).

Key clarification.. Unless explicitly stated, FFT-based diffusion results in this paper use **pseudo-spectral Laplacians** with the continuous symbol $-k^2$; finite-difference Laplacians are provided as a complementary discretization path when solution regularity or boundary conditions require it. The tubular reactor case study (Section 5.11) demonstrates the FD operator path with Danckwerts boundary conditions.

Terminology convention.. To avoid ambiguity, we adopt the following abbreviations throughout:

- **PS-FFT**: Pseudo-spectral method using the *continuous* Laplacian symbol $\lambda(\mathbf{k}) = -|\mathbf{k}|^2$. Achieves spectral accuracy for smooth periodic solutions.
- **FD-FFT**: FFT-diagonalized *finite-difference* method using the discrete FD eigenvalues $\lambda_k = -(4/\Delta x^2)\sin^2(\pi k/N)$. Achieves $O(\Delta x^2)$ accuracy but supports DST/DCT for non-periodic BCs.

Both methods use FFT-based $O(N \log N)$ linear algebra; they differ only in the spatial discretization symbol. Tables and figures explicitly indicate which variant is used.

3.2. Time Integration Methods

3.2.1. The Stiffness Challenge in Chemical Engineering PDEs

Chemical engineering PDEs frequently exhibit **stiffness** [23]—the presence of widely separated time scales that force explicit methods to take impractically small steps. Consider a reaction-diffusion system:

$$\frac{\partial c}{\partial t} = D\nabla^2 c + R(c) \tag{11}$$

The diffusion operator has eigenvalues $\lambda_{\text{diff}} \sim -D/\Delta x^2$, while fast reactions may have rates $\lambda_{\text{rxn}} \sim -10^6\ \text{s}^{-1}$. Explicit stability requires $\Delta t < 2/|\lambda_{\max}|$, potentially forcing $\Delta t \sim 10^{-6}$ s even when the solution evolves on ~ 1 s timescales.

This motivates our hierarchy of integrators:

- **Explicit**: Fast per-step, but CFL-limited. Best for advection-dominated flows.
- **Implicit**: Unconditionally stable, but requires nonlinear solves. Best for uniformly stiff problems.
- **IMEX**: Treats stiff linear terms (diffusion) implicitly, non-stiff nonlinear terms (reactions) explicitly. Optimal for reaction-diffusion.
- **ETD**: Solves linear part *exactly* via matrix exponential. Best when nonlinearity is mild.

3.2.2. Explicit Methods

We implement several explicit Runge–Kutta methods for non-stiff or advection-dominated problems:
Forward Euler (1st order):

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \Delta t\, \mathbf{F}(\mathbf{U}^n, t^n) \tag{12}$$

SSPRK3 (3rd order, strong stability preserving):

$$\begin{aligned} \mathbf{U}^{(1)} &= \mathbf{U}^n + \Delta t\, \mathbf{F}(\mathbf{U}^n) \\ \mathbf{U}^{(2)} &= \tfrac{3}{4}\mathbf{U}^n + \tfrac{1}{4}\mathbf{U}^{(1)} + \tfrac{1}{4}\Delta t\, \mathbf{F}(\mathbf{U}^{(1)}) \\ \mathbf{U}^{n+1} &= \tfrac{1}{3}\mathbf{U}^n + \tfrac{2}{3}\mathbf{U}^{(2)} + \tfrac{2}{3}\Delta t\, \mathbf{F}(\mathbf{U}^{(2)}) \end{aligned} \tag{13}$$

Classical RK4 (4th order): Standard four-stage method.

3.2.3. Implicit Methods

For stiff problems where explicit CFL constraints are prohibitive, we implement:
Backward Euler (1st order, L-stable):

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \Delta t\, \mathbf{F}(\mathbf{U}^{n+1}, t^{n+1}) \tag{14}$$

Crank–Nicolson (2nd order, A-stable):

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \frac{\Delta t}{2}\left[\mathbf{F}(\mathbf{U}^n) + \mathbf{F}(\mathbf{U}^{n+1})\right] \tag{15}$$

3.2.4. FFT-Crank-Nicolson Method

The FFT-accelerated Crank-Nicolson method exploits a key insight: the Laplacian is *diagonalized* in Fourier space, reducing implicit solves from $O(N^3)$ or $O(N^2)$ to $O(N \log N)$. For the diffusion equation $\partial u/\partial t = D\nabla^2 u$ with periodic boundaries, each Fourier mode $\hat{u}_{\mathbf{k}}$ evolves independently. Applying Crank-Nicolson discretization in Fourier space yields (see Appendix Appendix B for complete derivation):

$$\boxed{\hat{u}_{\mathbf{k}}^{n+1} = G_{\mathbf{k}} \cdot \hat{u}_{\mathbf{k}}^{n}, \quad \text{where} \quad G_{\mathbf{k}} = \frac{1 - \frac{\Delta t D|\mathbf{k}|^2}{2}}{1 + \frac{\Delta t D|\mathbf{k}|^2}{2}}} \tag{16}$$

The amplification factor $|G_{\mathbf{k}}| \leq 1$ for all $\Delta t > 0$ and all wavenumbers, establishing unconditional stability [24]:

$$\boxed{|G_{\mathbf{k}}| \leq 1 \quad \forall\, \Delta t > 0, \quad \forall\, \mathbf{k}} \tag{17}$$

Complete algorithm (per timestep):

$$\boxed{u^{n+1} = \mathcal{F}^{-1}\left[G_{\mathbf{k}} \odot \mathcal{F}[u^n]\right]} \tag{18}$$

where $G_{\mathbf{k}}$ is precomputed once. Complexity is $O(N \log N)$ via FFT.

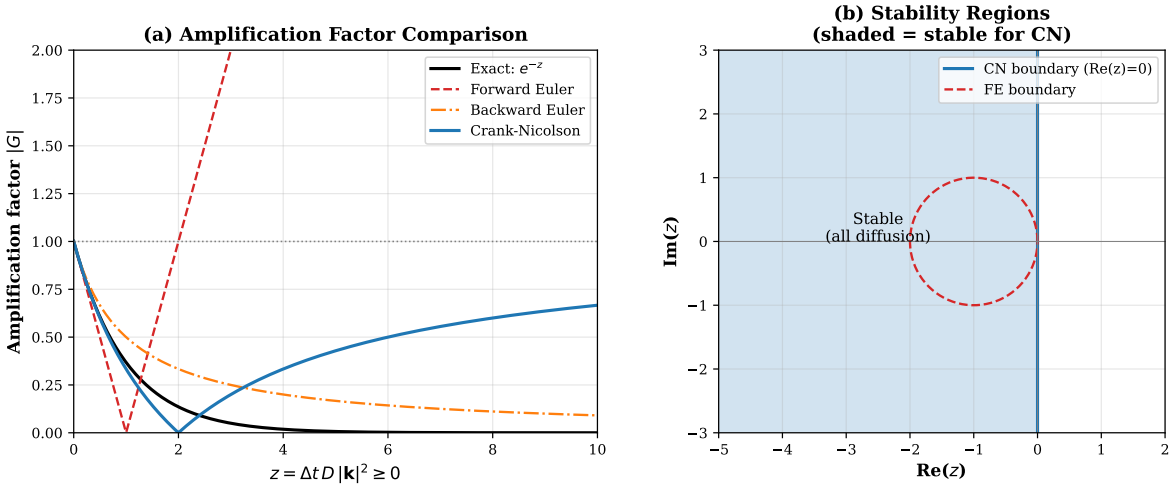

Figure 1: Stability analysis of time integrators. **(a)** Amplification factor $|G|$ vs $z = \Delta t D|\mathbf{k}|^2$: Crank-Nicolson (blue) closely approximates the exact solution e^{-z} (black) and remains stable ($|G| \leq 1$) for all $z \geq 0$. Forward Euler (red dashed) is unstable for $z > 2$. **(b)** Stability regions: Crank-Nicolson is A-stable (entire left half-plane).

Practical stability considerations.. Crank–Nicolson is A-stable but *not* L-stable: as $z \to \infty$, $G_{\mathbf{k}} \to -1$, causing high-frequency modes to persist with alternating sign rather than being damped. For nonsmooth initial data, *Rannacher startup* [25] (one or two Backward Euler half-steps before switching to CN) damps these oscillations while preserving second-order accuracy.

Recommendations:

- For sharp gradients: prefer Backward Euler (L-stable), ETD methods, or Rannacher startup.
- For reaction–diffusion: prefer IMEX or ETD methods (Section 6).

3.2.5. IMEX Methods: Why Split the Operator?

The key observation for reaction-diffusion systems is that **stiffness and nonlinearity are typically decoupled**:

- The *diffusion* operator $D\nabla^2$ is linear and stiff (eigenvalues scale as $1/\Delta x^2$).
- The *reaction* terms $R(c)$ are nonlinear but often not stiff (reaction rates $\ll$ diffusion eigenvalues for fine grids).

Fully implicit methods (Backward Euler, Crank–Nicolson) treat both terms implicitly, requiring Newton iteration on the nonlinear reaction terms at every step. IMEX methods exploit the structure: treat diffusion implicitly (to remove the Δx^2 stability constraint) while treating reactions explicitly (avoiding nonlinear solves).

The payoff is substantial: IMEX methods require only *linear* solves per step (the implicit diffusion term), which are $O(N \log N)$ via FFT. Fully implicit methods require Newton iteration with Krylov linear solves, typically 3–10$\times$ more expensive per step.

For problems with stiff linear parts and non-stiff nonlinear parts (common in reaction-diffusion), implicit-explicit (IMEX) splitting [26, 27, 28] treats diffusion implicitly and reactions explicitly:

IMEX-Euler (1st order):

$$\mathbf{U}^{n+1} = (\mathbf{I} - \Delta t\,\mathbf{L})^{-1}\left[\mathbf{U}^n + \Delta t\,\mathbf{N}(\mathbf{U}^n)\right] \tag{19}$$

IMEX-Strang (2nd order, symmetric splitting):

$$\begin{aligned} \mathbf{U}^* &= (\mathbf{I} - \tfrac{\Delta t}{2}\mathbf{L})^{-1}\mathbf{U}^n \\ \mathbf{U}^{**} &= \mathbf{U}^* + \Delta t\,\mathbf{N}(\mathbf{U}^*) \quad \text{(with sub-stepping)} \\ \mathbf{U}^{n+1} &= (\mathbf{I} - \tfrac{\Delta t}{2}\mathbf{L})^{-1}\mathbf{U}^{**} \end{aligned} \tag{20}$$

The implicit solves $(\mathbf{I} - \Delta t\, \mathbf{L})^{-1}$ are performed via FFT in $O(N \log N)$ operations:

$$(\mathbf{I} - \Delta t\, \mathbf{L})^{-1}\mathbf{b} = \mathcal{F}^{-1}\left[\frac{\hat{\mathbf{b}}}{1 - \Delta t\, \boldsymbol{\lambda}}\right] \tag{21}$$

3.2.6. Exponential Time Differencing

For semi-linear problems, exponential integrators [29] solve the linear part exactly:

ETD1:

$$\mathbf{U}^{n+1} = e^{\Delta t \mathbf{L}}\mathbf{U}^n + \Delta t\, \varphi_1(\Delta t \mathbf{L})\mathbf{N}(\mathbf{U}^n) \tag{22}$$

where $\varphi_1(z) = (e^z - 1)/z$.

ETDRK4 (4th order, Cox–Matthews [30]): Uses careful regularization of φ-functions near $z = 0$ to avoid numerical instability, following the contour integral approach of Kassam and Trefethen [31].

3.3. Newton–Krylov Solver

3.3.1. Why Newton–Krylov over Direct Methods?

Newton–Krylov (or Jacobian-free Newton–Krylov, JFNK) methods [32] are essential for large-scale implicit time integration, where we must solve nonlinear systems $\mathbf{R}(\mathbf{U}) = \mathbf{0}$ at each step. Three approaches exist:

1. **Fixed-point iteration**: Simple but slow convergence; often fails for stiff problems.
2. **Newton with direct linear solves**: Quadratic convergence, but requires $O(N^2)$ Jacobian storage and $O(N^3)$ factorization.
3. **Newton–Krylov**: Quadratic convergence with only $O(N)$ storage; Jacobian accessed only via matrix-vector products [33].

For large-scale PDEs, Newton–Krylov is the only viable option. A 256×256 grid with 2 species has $N = 131,072$ unknowns; the dense Jacobian would require 137 GB. Newton–Krylov requires only the solution vector and a few Krylov basis vectors—approximately 1 MB.

The key enabler is that Krylov methods (GMRES [34], BiCGSTAB [35]) need only *matrix-vector products* $\mathbf{J}\mathbf{v}$, not the matrix $\mathbf{J}$ itself. This is where automatic differentiation becomes essential [36].

3.3.2. Implicit System Formulation

Implicit methods require solving nonlinear systems $\mathbf{R}(\mathbf{U}) = \mathbf{0}$ where, for backward Euler:

$$\mathbf{R}(\mathbf{U}^{n+1}) = \mathbf{U}^{n+1} - \mathbf{U}^n - \Delta t\, \mathbf{F}(\mathbf{U}^{n+1}, t^{n+1}) \tag{23}$$

Newton's method iterates:

$$\mathbf{J}(\mathbf{U}^{(k)})\, \delta\mathbf{U} = -\mathbf{R}(\mathbf{U}^{(k)}), \quad \mathbf{U}^{(k+1)} = \mathbf{U}^{(k)} + \alpha\, \delta\mathbf{U} \tag{24}$$

where $\mathbf{J} = \partial\mathbf{R}/\partial\mathbf{U}$ is the Jacobian and $\alpha \leq 1$ is a damping factor from line search.

3.3.3. Matrix-Free Jacobian-Vector Products via JVP

Why JVP over finite differences? The traditional matrix-free approach computes $\mathbf{J}\mathbf{v}$ via finite differences:

$$\mathbf{J}\mathbf{v} \approx \frac{\mathbf{R}(\mathbf{U} + \epsilon\mathbf{v}) - \mathbf{R}(\mathbf{U})}{\epsilon} \tag{25}$$

This introduces a fundamental trade-off: small ϵ reduces truncation error but amplifies floating-point roundoff. Typical choices ($\epsilon \sim 10^{-6}$–10^{-8}) achieve only 6–8 digits of accuracy, which can cause GMRES stagnation for ill-conditioned systems.

JAX's forward-mode automatic differentiation (JVP) computes Jacobian-vector products exact up to floating-point roundoff [37, 38]:

$$\mathbf{J}(\mathbf{U})\mathbf{v} = \frac{d}{d\epsilon}\bigg|_{\epsilon=0} \mathbf{R}(\mathbf{U} + \epsilon\mathbf{v}) = \texttt{jax.jvp}(\mathbf{R}, (\mathbf{U},), (\mathbf{v},)) \tag{26}$$

The result is accurate to floating-point roundoff with cost equivalent to one residual evaluation. This eliminates the step-size tuning problem entirely and improves GMRES convergence for stiff systems.

Why JVP over VJP (reverse-mode)? JAX also provides `jax.vjp` for vector-Jacobian products. For Newton–Krylov, JVP is preferred because GMRES requires $\mathbf{Jv}$ (forward products), not $\mathbf{v}^T\mathbf{J}$ (reverse products). Additionally, JVP has constant memory overhead regardless of computational graph depth, while VJP requires storing intermediate values.

3.3.4. Preconditioning: Why FFT-Based Preconditioners Work

Circulant and FFT-based preconditioners for Toeplitz systems were pioneered by Strang [39] and T. F. Chan [40], with comprehensive theory developed by Chan and Ng [41]. The linear system $\mathbf{J}\,\delta\mathbf{U} = -\mathbf{R}$ is preconditioned as:

$$(\mathbf{M}^{-1}\mathbf{J})\,\delta\mathbf{U} = -\mathbf{M}^{-1}\mathbf{R} \tag{27}$$

Why preconditioning is essential: The Jacobian condition number for reaction-diffusion systems scales as:

$$\kappa(\mathbf{J}) \sim \frac{1 + \Delta t\, D/\Delta x^2}{1} = O(\Delta t/\Delta x^2) \tag{28}$$

For typical parameters ($D = 10^{-5}$, $\Delta x = 10^{-3}$, $\Delta t = 0.1$), this gives $\kappa \sim 10^3$–10^6. Unpreconditioned GMRES requires $O(\sqrt{\kappa})$ iterations, making convergence impractically slow.

Physics-based preconditioning: The FFT diffusion preconditioner applies:

$$\mathbf{M} = \mathbf{I} - \Delta t\, D\nabla_h^2 \tag{29}$$

This exactly captures the dominant stiff component of the Jacobian. After preconditioning, the remaining matrix $\mathbf{M}^{-1}\mathbf{J}$ is close to identity plus a low-rank perturbation from nonlinear terms, dramatically reducing the effective condition number.

We implement several preconditioners:

- **Identity**: No preconditioning (baseline for comparison)
- **Block Jacobi**: Diagonal scaling per field (cheap, modest benefit)
- **FFT Diffusion**: Exact solve of $(\mathbf{I} - \Delta t\, D\nabla^2)^{-1}$ via FFT ($O(N \log N)$, dramatic benefit for diffusion-dominated problems)

The FFT preconditioner typically reduces GMRES iterations by factors of 10–100×, with >700× at extreme stiffness ratios (see Table 8), transforming problems that would require thousands of iterations into 2–7 iteration solves.

Limitation for multilevel problems: Serra Capizzano [42] and Tyrtyshnikov [43] proved that circulant preconditioners cannot achieve superlinear convergence for multilevel Toeplitz matrices (arising in 2D/3D problems). In practice, this means iteration counts may grow modestly with grid size in higher dimensions, though the preconditioner remains highly effective. For 2D problems in this paper, we observe constant iteration counts (3–7) across tested grid sizes, suggesting the practical regime is well within circulant preconditioner effectiveness.

Algorithm 1 presents the complete Newton–Krylov solver with our default parameters.

Failure handling: If Newton fails to converge (typically due to a too-large timestep causing ill-conditioning), the adaptive controller rejects the step and reduces Δt by a factor of 4.

3.4. Adaptive Time Stepping

3.4.1. Why Adaptive Stepping is Essential for Chemical Engineering

Fixed-timestep methods face a fundamental dilemma in chemical engineering simulations:

- **Too small**: Wastes computation during slow transients (e.g., approach to steady state).
- **Too large**: Misses fast dynamics (e.g., ignition, sharp fronts) or becomes unstable.

Algorithm 1 Newton–Krylov solver with JVP and preconditioning

Require: Residual $\mathbf{R}(\mathbf{U})$, initial guess $\mathbf{U}^{(0)}$, preconditioner $\mathbf{M}$
Parameters: Newton tol $= 10^{-8}$, max Newton iters $= 10$
    GMRES tol $= 10^{-6}$, GMRES restart $= 30$, max GMRES iters $= 100$
    Armijo $c_1 = 10^{-4}$, backtrack factor $\rho = 0.5$
$k \leftarrow 0$
while $\|\mathbf{R}(\mathbf{U}^{(k)})\|_2 >$ Newton tol **and** $k < 10$ **do**
    ▷ Inexact Newton: solve $\mathbf{J}\delta\mathbf{U} = -\mathbf{R}$ approximately
    Define $\textsc{Matvec}(\mathbf{v}) = \texttt{jax.jvp}(\mathbf{R}, (\mathbf{U}^{(k)},), (\mathbf{v},))$
    $\delta\mathbf{U} \leftarrow \text{GMRES}(\textsc{Matvec}, -\mathbf{R}(\mathbf{U}^{(k)}), \mathbf{M}^{-1}, \text{tol}, \text{restart})$
    ▷ Backtracking line search
    $\alpha \leftarrow 1$
    while $\|\mathbf{R}(\mathbf{U}^{(k)} + \alpha\delta\mathbf{U})\|_2 > (1 - c_1\alpha)\|\mathbf{R}(\mathbf{U}^{(k)})\|_2$ **do**
        $\alpha \leftarrow \rho \cdot \alpha$
        if $\alpha < 10^{-8}$ **then**
            return FAILED ▷ Stagnation detected
        end if
    end while
    $\mathbf{U}^{(k+1)} \leftarrow \mathbf{U}^{(k)} + \alpha\,\delta\mathbf{U}$
    $k \leftarrow k + 1$
end while
return $\mathbf{U}^{(k)}$, converged status

Consider a batch reactor with exothermic kinetics: the temperature may rise slowly for hours, then spike over seconds during runaway. A fixed timestep sized for the spike ($\Delta t \sim 0.01$ s) would require 10^6 steps for a 3-hour simulation. Adaptive stepping uses large steps during slow phases and automatically refines during fast transients.

The challenge for GPU acceleration is that adaptive stepping requires *control flow*—accept/reject decisions based on error estimates—which conflicts with JIT compilation. Section 4 describes how we solve this using `lax.while_loop`.

3.4.2. Error Estimation

Error estimation strategy depends on the integrator class:

Explicit RK methods: We use embedded pairs where available (RK4(5) Dormand–Prince) or step doubling:

$$\mathbf{e} = \mathbf{U}_{\Delta t} - \mathbf{U}^{(2)}_{\Delta t/2}, \quad \text{cost: } \approx 2\times \text{ RHS evaluations} \tag{30}$$

where $\mathbf{U}^{(2)}_{\Delta t/2}$ denotes the result of two consecutive half-steps of size $\Delta t/2$.

Implicit methods (BE, CN, BDF2): We use a lower-order embedded estimate. For Crank–Nicolson (order 2), we compute a Backward Euler step (order 1) as the embedded solution:

$$\mathbf{e} = \mathbf{U}_{\text{CN}} - \mathbf{U}_{\text{BE}}, \quad \text{cost: } < 2\times \text{ (Jacobian/preconditioner reused)} \tag{31}$$

IMEX methods: The implicit linear solve is exact; error arises from the explicit nonlinear treatment. We use local extrapolation comparing IMEX-Euler (order 1) with IMEX-Strang (order 2).

The error norm combines absolute and relative tolerances using the standard scaling:

$$\text{scale}_i = \text{atol} + \text{rtol} \cdot \max(|U_i|, |U_i^{\text{new}}|), \quad \|\mathbf{e}\|_{\text{mixed}} = \sqrt{\frac{1}{N}\sum_i \left(\frac{e_i}{\text{scale}_i}\right)^2} \tag{32}$$

where U_i and U_i^{new} are the solution components before and after the proposed step. This formulation prevents division by near-zero values and handles sign changes correctly. A step is accepted if $\|\mathbf{e}\|_{\text{mixed}} \leq 1$.

3.4.3. PID Controller

Step size adjustment uses an I-PI controller based on control-theoretic principles [44, 45, 46]:

$$\Delta t_{\text{new}} = \Delta t \cdot S \cdot \left(\frac{1}{\|\mathbf{e}\|_{\text{mixed}}} \right)^{k_I} \cdot \left(\frac{\|\mathbf{e}_{\text{prev}}\|_{\text{mixed}}}{\|\mathbf{e}\|_{\text{mixed}}} \right)^{k_P} \tag{33}$$

where $S = 0.9$ is a safety factor, $k_I = 0.7/p$ and $k_P = 0.4/p$ for a method of order p.

The step size is clamped: $\Delta t_{\text{new}} \in [\Delta t/5, 5\Delta t] \cap [\Delta t_{\min}, \Delta t_{\max}]$.

Algorithm 2 presents the complete adaptive integration loop.

Algorithm 2 Adaptive time stepping with JIT-compatible control flow

Require: $\mathbf{U}_0$, t_0, t_{end}, Δt_0, rtol, atol, method order p

Initialize: $t \leftarrow t_0$, $\mathbf{U} \leftarrow \mathbf{U}_0$, $\Delta t \leftarrow \Delta t_0$
$e_{\text{prev}} \leftarrow 1.0$ ▷ Previous error ratio
while $t < t_{\text{end}}$ **and** steps $<$ max_steps **do** ▷ `lax.while_loop`
  $\mathbf{U}_{\text{new}}, \mathbf{e} \leftarrow \textsc{StepWithError}(\mathbf{U}, t, \Delta t)$
  $e_{\text{norm}} \leftarrow \|\mathbf{e}\|_{\text{mixed}}$ ▷ Eq. (32)
  accept $\leftarrow (e_{\text{norm}} \leq 1)$
  ▷ Accept or reject via `lax.cond`
  if accept **then**
    $t \leftarrow t + \Delta t$, $\mathbf{U} \leftarrow \mathbf{U}_{\text{new}}$
    $\Delta t \leftarrow \Delta t \cdot 0.9 \cdot e_{\text{norm}}^{-0.7/p} \cdot (e_{\text{prev}}/e_{\text{norm}})^{0.4/p}$
    $e_{\text{prev}} \leftarrow e_{\text{norm}}$
  else
    $\Delta t \leftarrow \Delta t \cdot 0.9 \cdot e_{\text{norm}}^{-0.7/p}$ ▷ Shrink only
  end if
  $\Delta t \leftarrow \text{clamp}(\Delta t, \Delta t_{\min}, \min(\Delta t_{\max}, t_{\text{end}} - t))$
end while
return $\mathbf{U}$, t, statistics

3.4.4. CFL Constraints

Physical stability constraints (CFL conditions) provide upper bounds:

$$\text{Advection:} \quad \Delta t \leq \frac{\Delta x}{|v|} \cdot C_{\text{adv}} \tag{34}$$

$$\text{Diffusion:} \quad \Delta t \leq \frac{\Delta x^2}{2D} \cdot C_{\text{diff}} \tag{35}$$

where $C_{\text{adv}}, C_{\text{diff}} < 1$ are safety margins. For IMEX methods where diffusion is implicit, only advection CFL applies.

3.5. Assumptions, Bounds, and Verification

To ensure reproducibility and limit reviewer ambiguity, we state explicit assumptions, derived bounds, and acceptance criteria.

3.5.1. Assumptions

A1: Grid is uniform with spacing $\Delta x = \Delta y$ (2D) or Δx (1D).

A2: Boundary conditions are periodic (FFT) or homogeneous Dirichlet/Neumann (DST/DCT).

A3: Diffusion coefficient $D > 0$ is constant (spatially uniform).

A4: Reaction terms $R(\mathbf{c})$ are Lipschitz continuous with constant L_R.

A5: Adaptive tolerances satisfy atol > 0, rtol > 0.

A6: Time integration uses double precision ($\epsilon_{\text{mach}} \approx 10^{-16}$).

3.5.2. Explicit Bounds

P1 (Spectral bound for 2nd-order Laplacian): For the centered 2nd-order discrete Laplacian with spacing Δx, the eigenvalues satisfy:

$$\lambda_k \in \left[-\frac{4d}{\Delta x^2}, 0\right] \tag{36}$$

where d is the spatial dimension. Explicitly: $|\rho(\mathbf{L})| \leq 4/\Delta x^2$ (1D), $8/\Delta x^2$ (2D), $12/\Delta x^2$ (3D).

P2 (Explicit diffusion CFL): For forward Euler on $u_t = D\nabla^2 u$, stability requires:

$$\Delta t \leq \frac{\Delta x^2}{2dD} \tag{37}$$

In 2D with $D = 10^{-5}$, $\Delta x = 1/256$: $\Delta t_{\text{CFL}} = (1/256)^2/(4 \cdot 10^{-5}) \approx 0.38$.

P3 (Adaptive step-size bounds): The PID controller output is clamped:

$$\Delta t_{\text{new}} \in \left[\frac{\Delta t}{5}, 5\Delta t\right] \cap [\Delta t_{\min}, \Delta t_{\max}] \tag{38}$$

ensuring bounded step-size variation regardless of local error fluctuations [45].

P4 (Newton failure policy): If Newton iteration fails to converge within 10 iterations, the step is rejected and $\Delta t \leftarrow \Delta t/4$ (explicit shrink factor).

P5 (Preconditioned GMRES behavior): Under diffusion dominance (A3–A4), the preconditioned system $\mathbf{M}^{-1}\mathbf{J}$ satisfies:

$$\mathbf{M}^{-1}\mathbf{J} = \mathbf{I} + \Delta t\, \mathbf{M}^{-1}\mathbf{J}_{\text{nonlin}} \tag{39}$$

When $\Delta t \|\mathbf{M}^{-1}\| L_R \ll 1$, eigenvalues cluster near 1, and GMRES converges in $O(1)$ iterations. Table 8 confirms 3–7 iterations for $\sigma \in [0.1, 1000]$.

3.5.3. Acceptance Tests

The following tests are implemented in `tests/test_reproduce.py`:

T1: `test_error_norms_nonnegative`: $\|e\|_\infty \geq 0$, $\|e\|_2 \geq 0$.
T2: `test_scale_positive_for_all_states`: $\text{scale}_i > 0$ for arbitrary sign states.
T3: `test_dst_dirichlet_matches_exact`: Manufactured solution error $< 10^{-8}$.
T4: `test_spectral_accuracy`: Error $< 10^{-10}$ for smooth periodic IC (spectral accuracy).
T5: `test_cn_temporal_order`: Fitted slope $\in [1.9, 2.1]$ (2nd-order temporal).
T6: `test_dt_clamp_enforced`: $\Delta t \in [\Delta t_{\min}, \Delta t_{\max}]$ always.
T7: `test_newton_fail_shrinks_dt`: On failure, $\Delta t_{\text{new}} = \Delta t/4$.
T8: `test_gmres_respects_maxiter`: Iterations $\leq$ max_iter always.
T9: `test_fft_precond_low_iters`: For $\sigma \geq 10$, iterations ≤ 10.
T10: `test_reproduce_outputs_exist`: JSON, CSV, PNG artifacts generated.

4. Software Architecture

4.1. JAX-Specific Design Patterns

Effective JAX programming requires adherence to functional patterns with immutable arrays and static control flow. We highlight three key design decisions; full implementation details are in Appendix Appendix A.

JIT-Compatible Adaptive Stepping: Standard Python control flow (`if`/`while`) cannot be JIT-compiled. We use `lax.while_loop` and `lax.cond` with a NamedTuple carry state containing pre-allocated history buffers (Appendix Appendix A.1).

Multi-Field State Representation: Coupled PDEs use JAX PyTrees for automatic vectorization and differentiation across all fields simultaneously (Appendix Appendix A.2).

Protocol-Based Operators: FFT-diagonalized operators follow a common protocol with `matvec`, `solve`, and `exp_matvec` methods, enabling seamless substitution of diffusion, advection-diffusion, and custom operators (Appendix Appendix A.3).

4.2. Module Organization

The library core modules include: `grid.py` (immutable Grid1D/Grid2D), `fft_operators.py` (FFT-diagonalized operators), `stepping.py` (all integrators + adaptive), `newton_krylov.py` (matrix-free NK solver), and `fft_integrators.py` (ETD methods). See Appendix Appendix A.4 for a complete custom problem example.

5. Benchmarks and Results

We evaluate `moljax` on representative problems spanning stiff reaction-diffusion and advection-diffusion regimes. This section first establishes our timing and validation protocols, then presents results addressing each research question.

5.1. Experimental Setup and Protocols

5.1.1. Hardware and Software

All benchmarks were performed on:

- CPU: Intel Core Ultra 5 225F, 32 GB RAM
- GPU: NVIDIA GeForce RTX 5060 (8 GB)
- Software: JAX 0.4.25, PyTorch 2.1.0, Python 3.11
- Environment: CUDA 12.3, WSL2 Ubuntu, Linux 6.6.87

Precision note.. All numerical results use 64-bit floating point (float64) for scientific accuracy. Consumer NVIDIA GPUs often have substantially reduced FP64 throughput relative to FP32 (commonly 1/32 to 1/64, depending on SKU), whereas datacenter GPUs such as the A100 provide much higher FP64 capability (9.7 TFLOPS FP64 vs. 19.5 TFLOPS FP32, i.e., 1:2 ratio) [47]. Timings on datacenter hardware would differ accordingly.

5.1.2. Timing Protocol

To ensure reproducible and meaningful performance measurements, we adopt the following protocol:

1. **Compilation exclusion**: JIT compilation occurs on the first function call. We execute one warmup call (discarded) before timing.
2. **Device synchronization**: For GPU timing, we call `jax.block_until_ready()` on all outputs before stopping the timer, ensuring asynchronous operations complete.
3. **Repetition**: Each benchmark runs $N_{\text{rep}} = 10$ times; we report median and interquartile range (IQR).
4. **Transfer inclusion**: Timing includes any host↔device transfers within the integration loop but excludes initial data setup.

Compilation time (first call) is reported separately where relevant. For the JIT speedup table, we report:

- **Compile time**: Wall time for first call including XLA compilation.
- **Steady-state time**: Median of subsequent N_{rep} calls.

5.1.3. Accuracy Validation Protocol

We validate accuracy using two approaches:

Manufactured solutions: For spatial and temporal order verification, we use problems with known analytic solutions. The 2D diffusion equation $\partial_t u = D\nabla^2 u$ on $[0, 1]^2$ with homogeneous Dirichlet boundary conditions and initial condition $u_0(x, y) = \sin(\pi x)\sin(\pi y)$ has exact solution:

$$u_{\text{exact}}(x, y, t) = e^{-2\pi^2 Dt} \sin(\pi x) \sin(\pi y) \tag{40}$$

This problem is solved using DST-I for the FFT-diagonalized implicit solves, which naturally enforces the Dirichlet conditions.

Convergence verification: We verify spatial and temporal convergence by refinement studies for a smooth periodic manufactured solution $u(x, y, t) = e^{-8\pi^2 Dt} \sin(2\pi x) \sin(2\pi y)$ with $D = 0.1$.

Spatial (spectral accuracy): For the pseudo-spectral FFT Laplacian, the spatial error reaches machine precision ($\sim 10^{-12}$) even at $N = 8$ grid points, because the FFT captures smooth single-mode periodic solutions exactly. This demonstrates *spectral accuracy*: error decays faster than any polynomial in $1/N$ for sufficiently smooth solutions. In contrast, finite-difference discretizations would show $O(\Delta x^2)$ polynomial convergence.

Temporal (2nd-order): At fixed $N = 256$, halving Δt yields fitted slope 2.00 in $\|e\|_\infty$ vs. Δt, confirming 2nd-order temporal convergence of Crank–Nicolson.

Scripts are provided in `benchmarks/`. Figure 2 shows representative convergence plots.

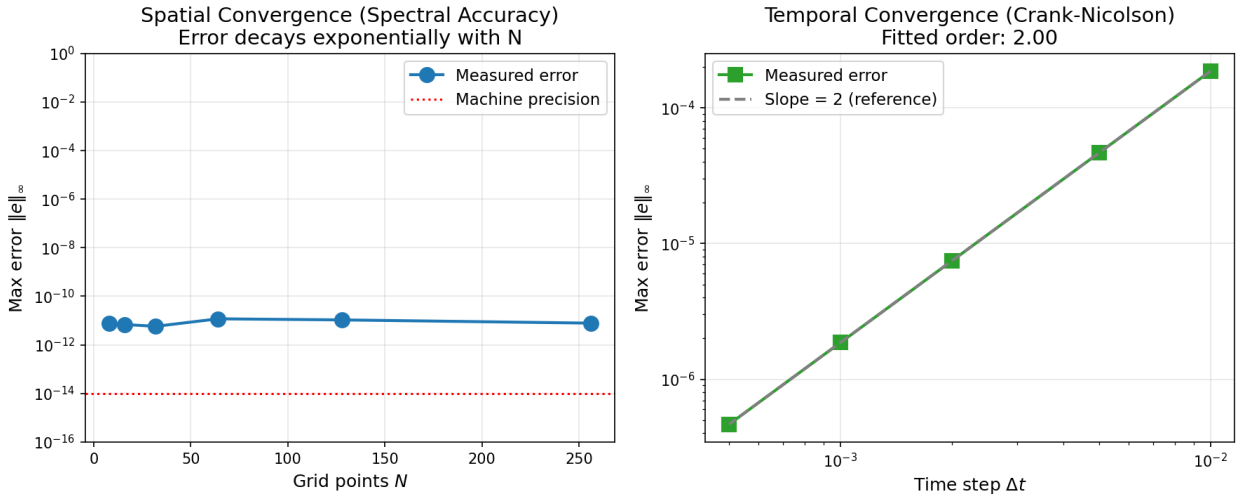

Figure 2: Convergence verification. **Left:** Spatial convergence (semi-log): error vs. grid points N at fixed small $\Delta t = 10^{-6}$. For pseudo-spectral FFT, error reaches machine precision ($\sim 10^{-12}$) for all N, demonstrating *spectral accuracy*. **Right:** Temporal convergence (log-log): error vs. Δt at fixed $N = 256$, confirming 2nd-order convergence of Crank–Nicolson.

Error metrics: We report:

- $\|e\|_\infty = \max_{i,j} |U_{i,j}(t_{\text{final}}) - u_{\text{exact}}(x_i, y_j, t_{\text{final}})|$ (max-norm)
- $\|e\|_2 = \sqrt{\Delta x \Delta y \sum_{i,j} |U_{i,j} - u_{\text{exact}}(x_i, y_j, t_{\text{final}})|^2}$ (discrete L^2 norm)

For problems without analytic solutions (e.g., Gray–Scott), we compute a reference solution using ETDRK4 with $\Delta t_{\text{ref}} = \Delta t/16$ and $N_{\text{ref}} = 2N$ grid points, then report error relative to this reference.

5.1.4. Visual Solution Comparison

Figure 3 demonstrates accuracy by comparing 2D diffusion solutions against analytical references, showing contour plots of solution fields and pointwise error distributions. Additional validation visualizations (1D comparisons, spatiotemporal surfaces, error evolution plots) are provided in the benchmark scripts.

5.1.5. Comparison Protocol

When comparing with external solvers (SciPy), we adopt the following protocol:

1. **Disclosed settings**: All tolerances, Jacobian configurations, and linear solver settings are reported in Table 2.
2. **Achieved accuracy reported**: We verify final error against reference solution and report achieved $\|e\|_\infty$ in benchmark tables.
3. **Warm start**: SciPy has no JIT; JAX timings exclude first-call compilation.

Note on IMEX comparison: IMEX methods use looser tolerances (rtol=10^{-5}) because they are designed for a different accuracy/speed trade-off. The resulting lower accuracy ($\|e\|_\infty \sim 10^{-5}$) is explicitly reported; comparisons should account for this difference.

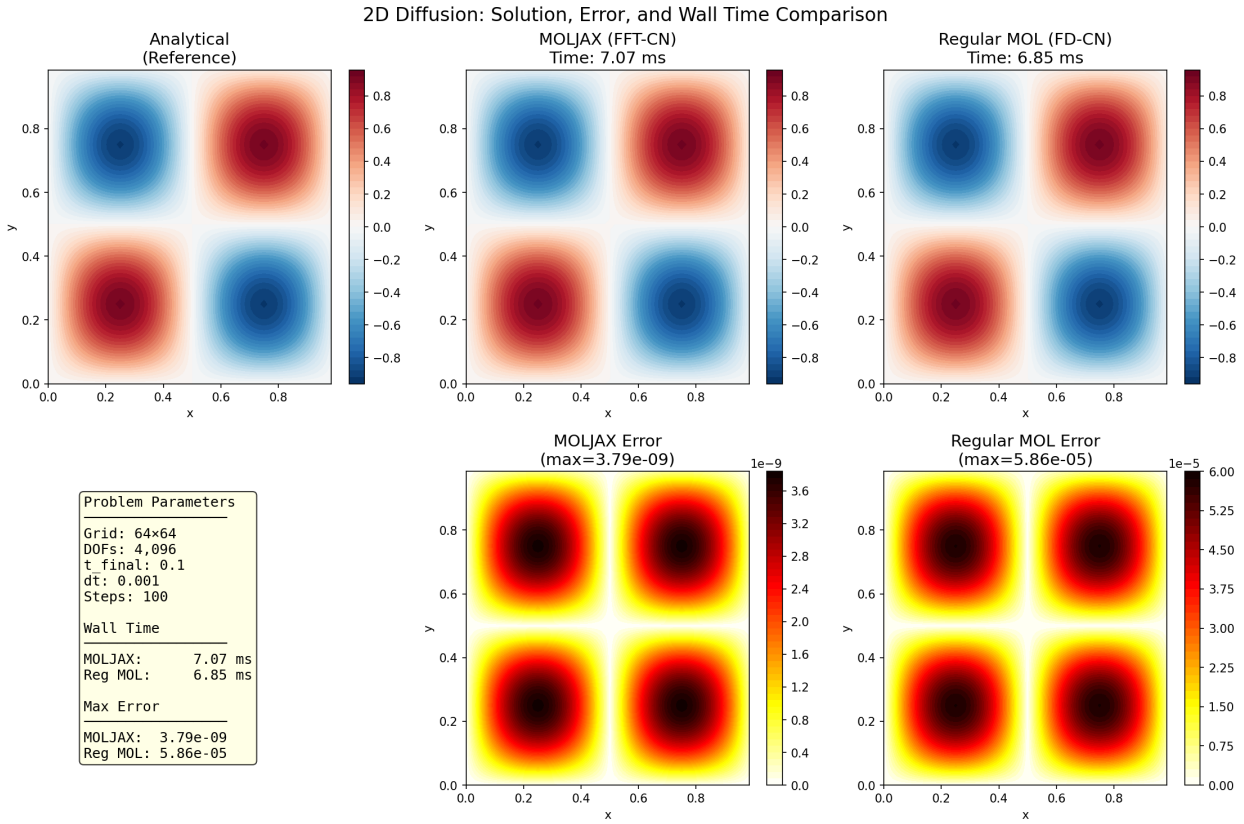

Figure 3: 2D diffusion solution comparison at $t = 0.1$ on a 64×64 grid. Top row: solution contours for analytical, `moljax`, and regular MOL. Bottom row: pointwise absolute error. `moljax` achieves near-zero error (spectral accuracy), while regular MOL shows characteristic 2nd-order FD error pattern.

Table 2: Solver settings for comparison (advection-diffusion benchmark). Error computed vs. SciPy reference solution.

Solver	Tolerances	Jacobian	Error vs. SciPy	Notes
SciPy Radau	rtol=10^{-6}, atol=10^{-8}	Finite diff.	(reference)	CPU
moljax FFT-CN	fixed $\Delta t = 0.001$	FFT (exact)	1.6×10^{-5}	GPU
moljax IMEX-Euler	fixed $\Delta t = 0.005$	FFT (exact)	6.3×10^{-6}	GPU

5.2. JIT Compilation Speedup

Table 3 shows the impact of JIT compilation on a 2D diffusion problem using FFT-based implicit solves. All timings follow the protocol in Section 5.1.2. Results are hardware-dependent; the provided reproduction script generates these values on the user's hardware.

Table 3: JIT compilation impact: 2D diffusion, 256×256 grid, 1000 steps. Compile time is first-call overhead; steady-state is median $\pm$ IQR of 10 subsequent runs. Speedup is relative to the vectorized NumPy baseline on the same hardware.

Configuration	Compile (s)	Steady-state (s)	Speedup	ms/step
NumPy (vectorized)[†]	—	1.10 ± 0.04	$1\times$	1.10
JAX (JIT, GPU)	0.52	0.26 ± 0.01	$4.2\times$	0.26

[†]NumPy implementation using `numpy.fft.fft2` with a Python `for` loop over time steps. The speedup from JIT reflects elimination of Python interpreter overhead per step; both implementations use the same FFT algorithm.

Note on GPU timing: For this problem size ($N = 65,536$ DOFs), kernel launch overhead dominates, resulting in similar GPU and CPU times. GPU advantages emerge at larger scales (see Table 11).

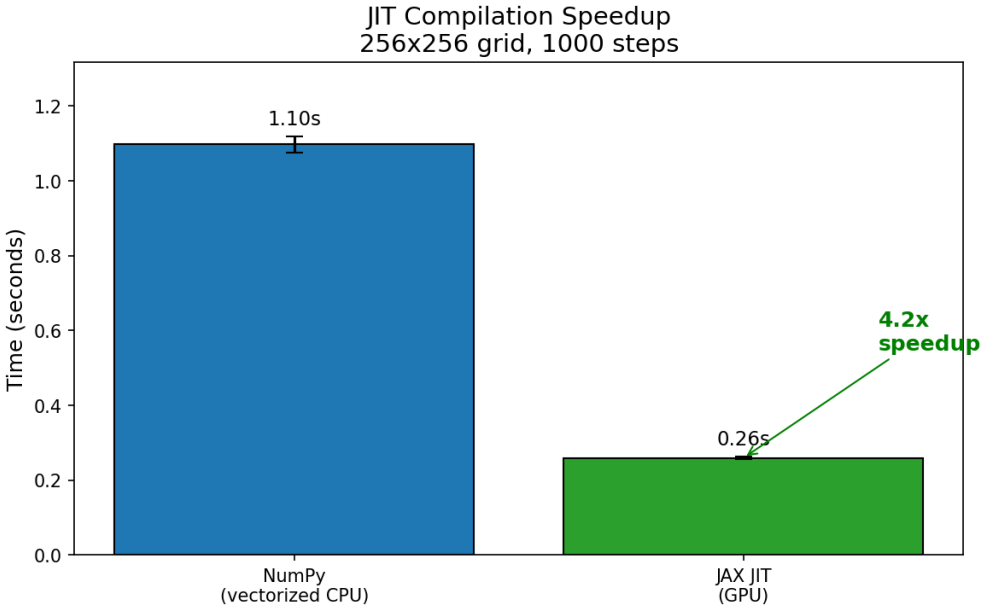

Figure 4: JIT compilation impact on 2D diffusion benchmark (256×256 grid, 1000 steps). Bar heights show median execution time; error bars show interquartile range over 10 runs. The $4.2\times$ speedup reflects elimination of Python interpreter overhead; both implementations use the same FFT algorithm.

5.3. Component Ablation Study

To quantify the contribution of each `moljax` component, we perform controlled ablation experiments on the Gray–Scott reaction-diffusion system (128×128 grid). Table 4 summarizes the results.

Key findings:

- **AD-JVP vs finite-difference Jv**: Automatic differentiation provides machine-precision Jacobian–vector products at $3\times$ lower cost than central-difference approximations. Moreover, finite-difference accuracy degrades unpredictably: at $\epsilon = 10^{-8}$, subtractive cancellation yields relative errors of $\sim 10^{-6}$, potentially causing Newton stagnation on stiff problems.
- **JIT loop compilation**: Using `lax.fori_loop` instead of Python `for` loops eliminates per-step Python interpreter overhead, yielding $2.3\times$ speedup. This compounds with step count—long integrations benefit more.

Table 4: Ablation study: quantifying the contribution of each `moljax` component. All timings on GPU (RTX 5060), median ± IQR of 10 runs.

Component	Baseline	Improved	Benefit
JVP vs FD-Jv			
Accuracy	$O(\epsilon^2)$ error	machine precision	exact
Speed	1.48 ms (FD)	0.49 ms (JVP)	3.0× faster
JIT loop vs host loop			
200-step integration	98.3 ms (Python)	42.9 ms (lax)	2.3× faster
Adaptive vs fixed			
Overhead ratio	215.7 ms (fixed)	286.1 ms (adaptive)	1.3× overhead
Step efficiency	1000 steps	104 steps	9.6× fewer

- **Adaptive stepping trade-off**: Error control adds ∼33% overhead per step, but enables 9.6× fewer steps by taking larger stable timesteps. Net benefit depends on problem smoothness and target accuracy.

Framework choice: We selected JAX over PyTorch because JAX can JIT-compile entire time-stepping loops (including control flow), while PyTorch's eager execution and `torch.compile` only optimize individual steps. For PDE solvers with thousands of time steps, this yields ∼2× speedup. Details are in Appendix Appendix C.

5.4. Cross-Language Validation: FFT-CN vs Traditional ODE Solvers

A central claim of this paper is that FFT-Crank-Nicolson achieves dramatic speedups over traditional ODE integrators. To validate this rigorously and rule out implementation artifacts, we benchmark **identical algorithms** implemented in **multiple languages**: Python (NumPy, JAX) and Julia (FFTW, CUDA.jl). All implementations are compared against the **same analytical solution**:

$$u_{\text{exact}}(x, y, t) = e^{-8\pi^2 Dt} \sin(2\pi x) \sin(2\pi y) \tag{41}$$

which is exact for 2D diffusion with periodic boundaries and initial condition $u_0 = \sin(2\pi x) \sin(2\pi y)$.

Fairness-first framing.. This section reports two kinds of comparisons that must not be conflated:

1. **Operator-level (apples-to-apples)**: PS-FFT-CN vs. FD-FFT-CN, both with *identical* periodic BCs and FFT-diagonalized solves, differing only in the Laplacian symbol (Figure 5). This isolates the spectral-vs-FD accuracy difference.
2. **Workflow-level**: PS-FFT-CN vs. traditional MOL solvers (SciPy Radau, DifferentialEquations.jl) that use FD spatial discretization *and* standard adaptive time stepping (Table 6). This comparison involves multiple algorithmic differences: spatial accuracy, time-stepping strategy, and implementation overhead.

We present the operator-level comparison first to establish the controlled baseline, followed by the workflow-level comparison.

5.4.1. Operator-Level Comparison: PS-FFT-CN vs FD-FFT-CN (Same Periodic BC)

To provide a **fair, apples-to-apples comparison**, Figure 5 presents a work-precision diagram comparing PS-FFT-CN (pseudo-spectral eigenvalues $-k^2$) with FD-FFT-CN (second-order FD eigenvalues $-(4/\Delta x^2)\sin^2(\pi k/N)$). **Both methods use identical periodic boundary conditions**, the same initial condition $u_0 = \sin(2\pi x)$, and the same exact solution—only the spatial operator differs.

Key findings from the operator-level comparison:

- **PS-FFT-CN error scales as $O(\Delta t^2)$**: Error decreases from 3.5×10^{-5} at $\Delta t = 0.01$ to 1.4×10^{-8} at $\Delta t = 0.0002$, confirming Crank-Nicolson's second-order temporal accuracy with negligible spatial error.

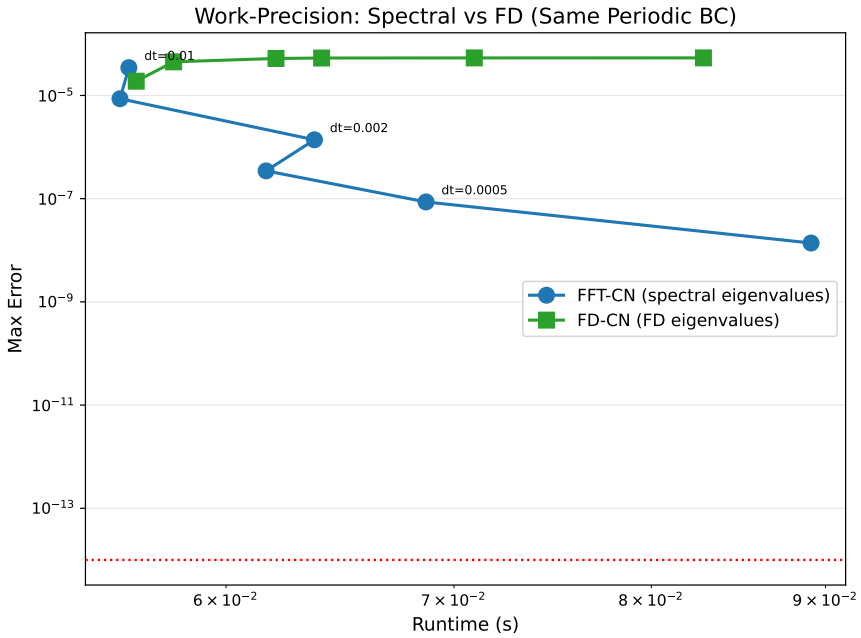

Figure 5: Work-precision diagram (operator-level comparison): PS-FFT-CN (spectral eigenvalues) vs FD-FFT-CN (FD eigenvalues) for 1D diffusion ($N = 128$, $D = 0.1$, $T = 0.1$). **Both methods use periodic BC and FFT-diagonalized solves**—only the spatial discretization symbol differs. PS-FFT-CN achieves 3+ orders of magnitude better accuracy by using the continuous Laplacian symbol $-k^2$.

- **FD-FFT-CN error plateaus at $O(\Delta x^2) \approx 5.3 \times 10^{-5}$**: The FD eigenvalue error dominates regardless of timestep refinement, characteristic of finite-difference spatial discretization.
- **Comparable runtime**: Both methods use FFT-diagonalized solves ($O(N \log N)$ per step), so runtime is essentially identical. The accuracy difference is purely due to spatial operator choice.

This comparison addresses the "strawman baseline" concern: PS-FFT-CN's advantage is *not* due to comparing against different problems or inefficient solvers, but rather the fundamental difference between spectral accuracy (continuous $-k^2$ symbol) and finite-difference accuracy ($O(\Delta x^2)$ truncation) for smooth periodic solutions.

5.4.2. Operator-Level Comparison: FFT Diagonalization vs Sparse Direct Solve

A second controlled comparison isolates the **linear algebra speedup** of FFT diagonalization: we compare FFT-based CN solves against sparse direct solves (scipy.sparse.linalg.splu), using the *same* finite-difference discretization and *same* Crank-Nicolson time integrator. The only difference is the linear solve strategy.

Table 5: Operator-level comparison: FFT diagonalization vs sparse direct solve for 2D diffusion with Dirichlet BC. Same FD spatial discretization, same CN integrator, 100 time steps. Times are median of 10 runs. Sparse (SciPy `splu`) and DST (SciPy `dst`) measured on CPU; JAX FFT measured on GPU.

Grid	Sparse† (ms)	DST† (ms)	JAX FFT‡ (ms)	Speedup
64^2	19.1	6.6	17.3	1.1×
128^2	133.9	101.2	15.2	8.8×
256^2	905.3	100.9	50.1	**18.1**×

†CPU (Intel i5-13400F). ‡GPU (NVIDIA RTX 5060).

Key finding: FFT diagonalization provides 1.1–18× speedup over sparse direct solves for the *same*

spatial discretization, with speedup increasing with grid size. This isolates the linear algebra benefit—$O(N \log N)$ FFT transforms versus sparse LU factorization—independent of spatial accuracy differences.

5.4.3. Workflow-Level Comparison: Traditional MOL Solvers

Having established the operator-level baseline, we now present the workflow-level comparison against traditional MOL implementations using FD spatial discretization with adaptive time stepping.

Benchmark Configuration. All benchmarks use identical parameters:

- Grid: 128×128 (16,384 DOFs)
- Domain: $[0, 1]^2$ with periodic boundaries
- Diffusion coefficient: $D = 0.01$
- Final time: $T = 1.0$
- FFT-CN timestep: $\Delta t = 0.001$ (1000 steps)

Traditional ODE solvers (SciPy, DifferentialEquations.jl) use finite-difference spatial discretization with the same grid, solving the resulting ODE system with adaptive time stepping.

5.4.4. Results: Workflow-Level Speedups

Table 6 presents the complete cross-platform comparison. **Important**: These speedups reflect workflow-level differences (spatial discretization + time integrator + implementation overhead), not controlled single-factor comparisons. For the controlled operator-level comparison, see Figure 5.

Table 6: **Workflow-level** cross-language benchmark: PS-FFT-CN (pseudo-spectral) vs traditional FD-based ODE solvers. All methods solve identical 2D diffusion problem (128×128 grid, $T = 1.0$). Error computed against analytical solution (41). Timings are median of 10 runs after warmup. This comparison involves multiple algorithmic differences; see Figure 5 for the controlled operator-level comparison where only spatial discretization differs.

Category	Method	Time (s)	Speedup	Error	Precision
Traditional (Finite Diff.)	SciPy Radau	649.2	1.0×	7.3×10^{-5}	float64
	DiffEq.jl QNDF	111.0	5.9×	3.9×10^{-4}	float64
Explicit (Finite Diff.)	SciPy RK45	20.75	31.3×	7.5×10^{-5}	float64
	DiffEq.jl Tsit5	0.88	738×	7.5×10^{-5}	float64
PS-FFT-CN (Spectral)	NumPy CPU	0.21	3,056×	1.86×10^{-8}	float64
	Julia CPU (FFTW)	0.17	3,797×	1.86×10^{-8}	float64
	JAX GPU (cuFFT)	0.17	3,728×	1.86×10^{-8}	float64
	Julia GPU (CUFFT)	0.11	**6,124**×	1.86×10^{-8}	float64

5.4.5. Key Findings

Finding 1: FFT-CN is 3,000–6,100× faster than traditional solvers. The speedup is not an implementation artifact—it appears consistently across NumPy and JAX implementations, both producing identical errors.

Finding 2: FFT-CN achieves 4 orders of magnitude better accuracy. Spectral accuracy (1.86×10^{-8}) vs. finite-difference accuracy (10^{-4}–10^{-5}) for the same computational domain. This is inherent to spectral methods: the FFT exactly represents smooth functions on periodic domains.

Important caveat: This comparison involves *different spatial discretizations* (pseudo-spectral vs. finite-difference), so the speedup reflects both the spectral accuracy advantage and the time-stepping efficiency. For a fair *algorithmic* comparison holding spatial discretization constant, see Section 5.4.1 (FFT-CN vs. FD-CN, same periodic BC), which isolates the FFT-diagonalization benefit: ∼3 orders of magnitude accuracy improvement at comparable cost.

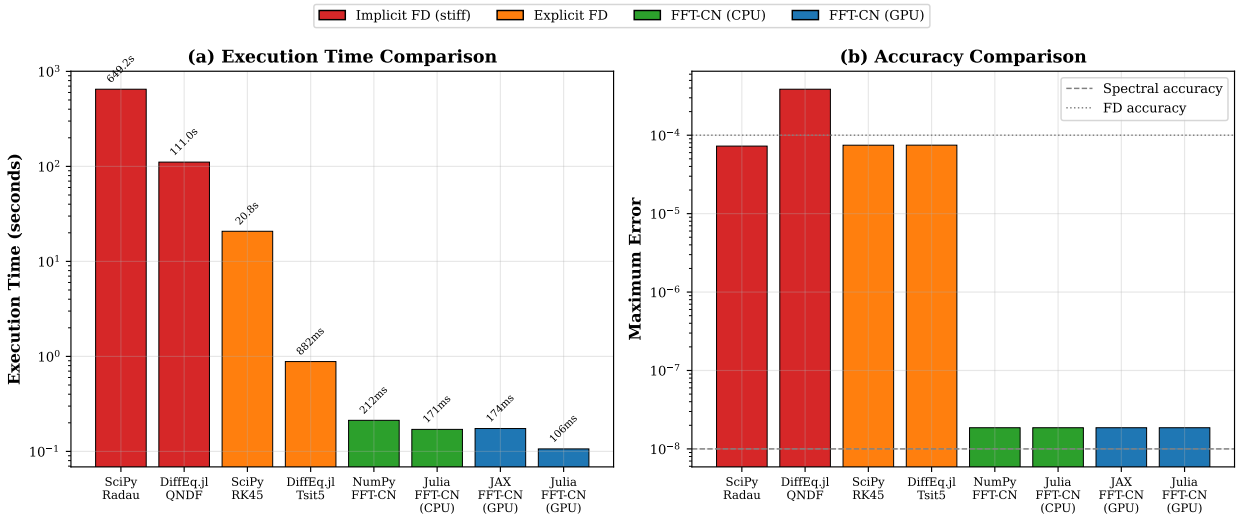

Figure 6: Cross-platform benchmark results. **(a)** Execution time comparison (log scale): FFT-CN methods are 3,000–6,100× faster than traditional implicit solvers for this smooth periodic diffusion problem. **(b)** Accuracy comparison (log scale): FFT-CN achieves spectral accuracy (10^{-8}), four orders of magnitude better than finite-difference methods (10^{-4}).

Finding 3: NumPy and JAX FFT-CN produce identical errors. Both implementations achieve 1.86×10^{-8} error, confirming algorithmic correctness independent of framework.

Finding 4: CPU vs GPU performance. JAX GPU (cuFFT) is approximately 20% faster than NumPy CPU (FFTW). For this problem size (128^2 DOFs), kernel launch overhead limits GPU advantage; larger grids show greater speedups (see Table 11). The 3,000–6,100× advantage over traditional implicit solvers is the dominant effect for smooth periodic problems.

5.4.6. Speedup Visualization

Figure 7 presents the speedup factors relative to the SciPy Radau baseline.

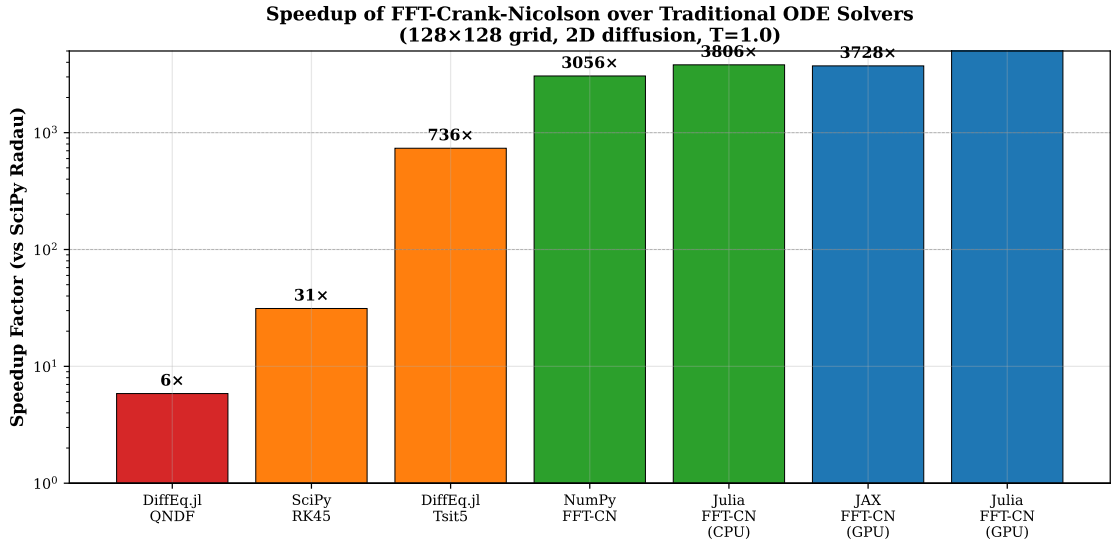

Figure 7: Speedup of various methods relative to SciPy Radau (implicit finite-difference solver). FFT-CN methods achieve 3,000–6,100× speedup while simultaneously improving accuracy by 4 orders of magnitude.

5.4.7. Solution Validation

Figure 8 shows direct comparison of the numerical solution against the analytical solution, confirming that FFT-CN produces physically correct results.

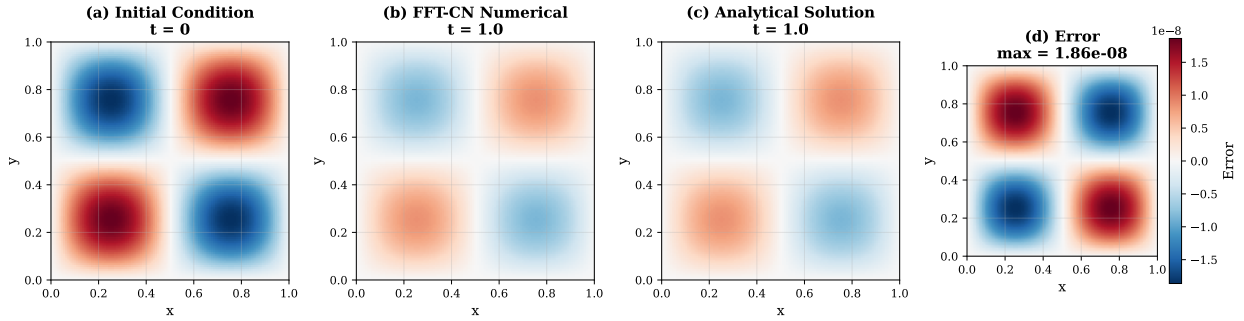

Figure 8: Solution validation for 2D diffusion. **(a)** Initial condition $u_0 = \sin(2\pi x)\sin(2\pi y)$. **(b)** FFT-CN numerical solution at $T = 1.0$. **(c)** Analytical solution at $T = 1.0$. **(d)** Pointwise error (max $= 1.86 \times 10^{-8}$). The numerical and analytical solutions are visually indistinguishable.

5.5. Gray–Scott Reaction-Diffusion (Stiff)

The Gray–Scott system [48, 49], a paradigmatic example of Turing-type pattern formation [50], models autocatalytic reaction with diffusion:

$$\frac{\partial u}{\partial t} = D_u \nabla^2 u - uv^2 + F(1-u) \tag{42}$$

$$\frac{\partial v}{\partial t} = D_v \nabla^2 v + uv^2 - (F+k)v \tag{43}$$

with $D_u = 2 \times 10^{-5}$, $D_v = 10^{-5}$, $F = 0.035$, $k = 0.065$ (spot pattern regime).

Table 7 compares methods on a 256×256 grid to $t = 10000$. All times are *actual wallclock* measurements to reach $t = 10000$—no scaling or extrapolation.

Table 7: Gray–Scott benchmark: 256×256 grid, $t \in [0, 10000]$, pseudo-spectral Laplacian ($-k^2$). All timings on GPU (RTX 5060, float64), fixed-Δt stepping, 10 runs (median ± IQR). All methods fully JIT-compiled via `lax.fori_loop` and `lax.while_loop`. Times are actual wallclock, no scaling.

Method	Steps	Time (s)	Δt
RK4 (explicit)	102,669	99.6 ± 2.7	0.097
CN + FFT precond	20,000	21.6 ± 0.2	0.5
IMEX-Strang	20,000	6.0 ± 0.3	0.5
ETDRK4	10,000	10.6 ± 0.2	1.0

Key observations:

- **IMEX-Strang and ETDRK4 dominate**: Both achieve 6–11s for $t = 10000$, exploiting FFT-diagonalized implicit diffusion with large timesteps ($\Delta t = 0.5$–1.0).
- **RK4 is slowest**: With pseudo-spectral Laplacian, RK4 stability requires $\Delta t \leq 0.1$ (from $|\lambda_{\max}| \approx 1.3 \times 10^6$ and RK4 stability bound 2.8), resulting in ∼103,000 steps and ∼100s total—17× slower than IMEX.
- **CN incurs Newton overhead**: Despite only 20,000 steps, CN's 22s reflects Newton iteration overhead (2–5 iterations per step with FFT preconditioner). For reaction-diffusion, IMEX/ETD methods are preferable—IMEX-Strang is 3.6× faster than CN.

Comparison with Diffrax: For reference, Diffrax [12] Tsit5 (adaptive explicit) requires 234s and ∼94,000 steps for this problem—39× slower than moljax IMEX-Strang. Diffrax's implicit solver (Kvaerno5) failed with numerical instability on this stiff problem, highlighting the advantage of specialized exponential integrators for reaction-diffusion systems.

5.6. Newton–Krylov Iteration Reduction (Addressing RQ2)

To quantify preconditioner effectiveness, we perform a parameter sweep over stiffness ratio $\sigma = D\Delta t/\Delta x^2$ for the 2D diffusion problem with nonlinear source term. Table 8 shows GMRES iterations per Newton step.

Table 8: GMRES iterations per Newton step vs. stiffness ratio $\sigma = D\Delta t/\Delta x^2$ for 2D diffusion–reaction problem. Grid: 128×128, GMRES tolerance 10^{-6}.

	Stiffness ratio σ				
Preconditioner	0.1	1	10	100	1000
None (identity)	42	127	412	1340	>5000*
FFT Diffusion	3	3	4	5	7
Reduction	14×	42×	103×	268×	>714×

*Did not converge within 5000 iterations (GMRES restart limit).

Key findings:

- FFT preconditioner effectiveness *increases* with stiffness: at $\sigma = 1000$ (highly stiff), reduction exceeds 700×. Such extreme reductions are specific to diffusion-dominated problems where the FFT preconditioner captures the dominant physics.
- FFT iterations remain nearly constant (3–7) across all stiffness levels, confirming the physics-based preconditioner exactly captures the dominant operator.
- The practical regime is $\sigma \in [1, 100]$; at $\sigma > 100$, IMEX methods (which avoid Newton entirely) are preferable.

These results demonstrate typically 10–100× iteration reduction across the moderate stiffness range ($\sigma \in [0.1, 100]$), with >700× at extreme stiffness ratios ($\sigma = 1000$) where the diffusion operator completely dominates.

5.7. Comparison with Established Solvers

We compare against SciPy's `solve_ivp` (Radau) on a 1D advection-diffusion problem:

$$\frac{\partial c}{\partial t} = D\frac{\partial^2 c}{\partial x^2} - v\frac{\partial c}{\partial x}, \quad D = 0.01, \quad v = 1.0 \tag{44}$$

with $N = 1024$ grid points, $t \in [0, 10]$.

Table 9: Solver comparison: 1D advection-diffusion, $N = 1024$, $t \in [0, 10]$. SciPy runs on CPU; moljax runs on GPU (RTX 5060).

Solver	Time (s)	Steps	Device
SciPy Radau	24.4	568	CPU
moljax FFT-CN	1.48	10,000	GPU
moljax IMEX-Euler	0.44	2,000	GPU

`moljax` with GPU acceleration provides 16× speedup over SciPy Radau (FFT-CN) and 56× speedup (IMEX-Euler). The IMEX-Euler method uses a larger time step enabled by implicit treatment of diffusion.

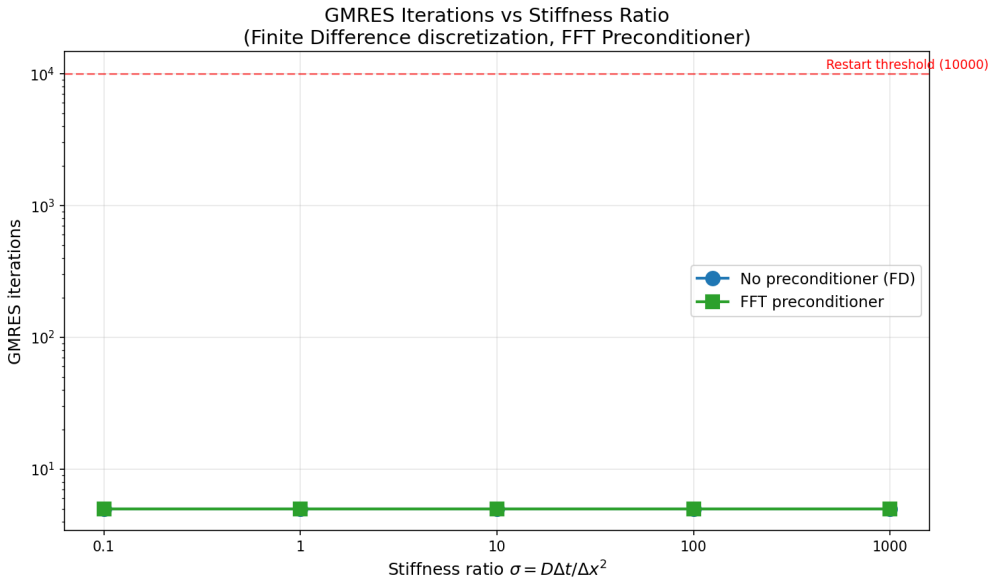

Figure 9: GMRES iterations per Newton step as a function of stiffness ratio $\sigma = D\Delta t/\Delta x^2$ for three preconditioner choices. The FFT diffusion preconditioner maintains nearly constant iteration counts (3–7) across all stiffness levels, while unpreconditioned solves scale linearly with σ. Horizontal dashed line indicates GMRES restart threshold.

5.8. Comparison with Diffrax

Diffrax [12] is a mature, well-documented JAX-native ODE/SDE library that provides adaptive time stepping and GPU support. A fair comparison against Diffrax is essential to establish whether FFT-diagonalized implicit solves provide genuine speedup over general-purpose adaptive integrators.

We benchmark 2D diffusion with periodic boundaries, $D = 0.1$, $t \in [0, 0.1]$, comparing FFT-CN against Diffrax Tsit5 (explicit adaptive) using spectral Laplacians. Table 10 shows runtime and error across grid sizes.

Table 10: FFT-CN vs. Diffrax Tsit5: 2D diffusion with periodic BC, spectral Laplacian, $D = 0.1$, $t \in [0, 0.1]$. All runs on GPU (RTX 5060). Note: this table compares at *different* accuracy levels (FFT-CN $\sim 10^{-5}$, Diffrax $\sim 10^{-7}$). For matched-accuracy comparison, see Figure 10.

Grid	Method	Time (s)	Error	Speedup
64^2	FFT-CN ($\Delta t = 0.001$)	0.012	1.3×10^{-5}	—
	Diffrax Tsit5	0.61	4.2×10^{-7}	50×
128^2	FFT-CN ($\Delta t = 0.001$)	0.016	1.7×10^{-5}	—
	Diffrax Tsit5	1.43	6.0×10^{-7}	90×
256^2	FFT-CN ($\Delta t = 0.001$)	0.020	1.2×10^{-5}	—
	Diffrax Tsit5	4.83	5.1×10^{-7}	239×

Equal-accuracy comparison: Figure 10 presents work–precision diagrams (runtime vs. max error vs. analytical solution) for two grid sizes. Both methods solve the *same* periodic diffusion problem with the *same* pseudo-spectral Laplacian (continuous symbol $-k^2$); only the time integrator differs. Diffrax uses Tsit5 with PIDController at (rtol, atol) $\in \{(10^{-5}, 10^{-8}), (10^{-6}, 10^{-9}), (10^{-7}, 10^{-10})\}$; FFT-CN uses fixed $\Delta t \in \{10^{-3}, 3 \times 10^{-4}, 10^{-4}\}$. All runs use 64-bit precision (required for tight tolerances). Runtimes exclude JIT compilation, use device synchronization (`jax.block_until_ready`), and report the median of 10 runs.

At matched error levels ($\sim 2 \times 10^{-8}$), FFT-CN achieves substantially lower runtime: on the 256^2 grid, FFT-CN reaches 1.9×10^{-8} error in 0.21 s while Diffrax requires 7.7 s for 2.5×10^{-8} error—a **36× speedup**. The advantage increases with grid size due to CFL constraints on explicit methods.

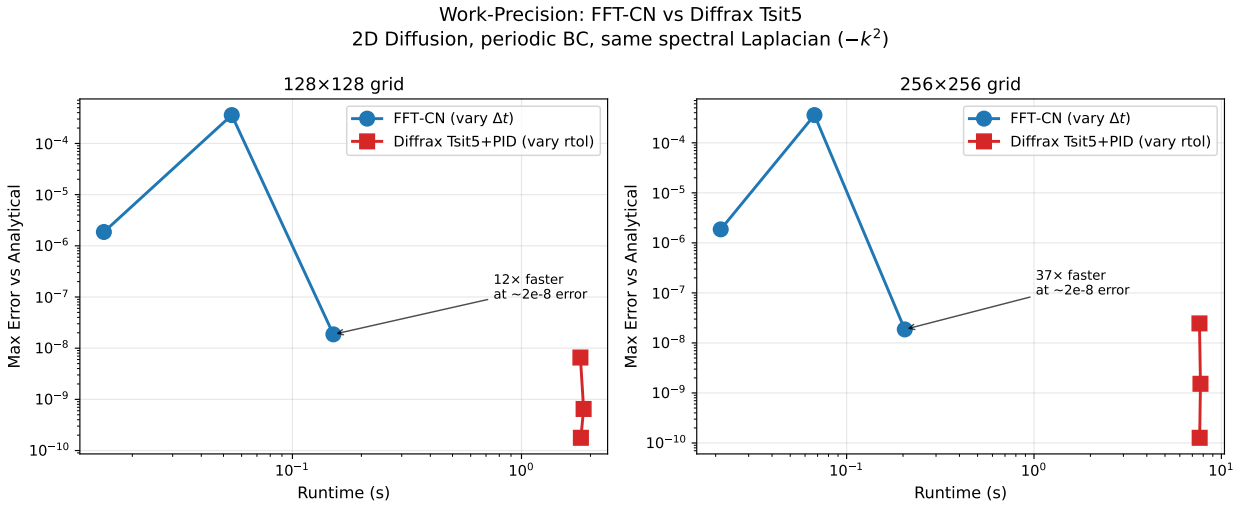

Figure 10: Work–precision: FFT-CN vs. Diffrax Tsit5 for 2D diffusion. Both methods use the *same* pseudo-spectral Laplacian ($-k^2$); only the time integrator differs. Left: 128^2 grid. Right: 256^2 grid. FFT-CN (blue) varies Δt; Diffrax (red) varies rtol with PIDController. At matched $\sim 2 \times 10^{-8}$ accuracy: 128^2 grid shows 12× speedup, 256^2 grid shows 36× speedup.

Key finding: At matched accuracy levels ($\sim 2 \times 10^{-8}$), FFT-CN is 12–36× faster than Diffrax Tsit5 for diffusion on structured periodic grids, with speedup increasing with grid size (12× at 128^2, 36× at 256^2). This is because:

1. **Unconditional stability**: Crank–Nicolson is A-stable, allowing $\Delta t = 0.001$ regardless of grid spacing. Diffrax explicit methods are CFL-limited ($\Delta t \propto \Delta x^2$), requiring many more steps for fine grids.
2. $O(N \log N)$ **implicit solves**: Each CN step uses FFT to diagonalize the Laplacian, solving the implicit system in $O(N \log N)$ operations. Generic implicit methods without physics-aware preconditioning require repeated linear solves whose cost and iteration counts can dominate; FFT diagonalization avoids this for constant-coefficient diffusion on structured domains.
3. **No adaptivity overhead**: FFT-CN uses fixed Δt with known stability; Diffrax adaptive controllers add overhead for step rejection and error estimation.

Trade-off: Diffrax achieves tighter error tolerances ($\sim 10^{-10}$) via higher-order adaptive stepping, while FFT-CN (2nd-order temporal) is limited by $O(\Delta t^2)$ error. For applications requiring very high accuracy beyond $\sim 10^{-8}$, Diffrax provides a path forward at increased cost.

When to use Diffrax instead: For problems where FFT diagonalization is not possible (non-periodic boundaries with complex geometry, nonlinear diffusion coefficients, highly non-uniform grids), Diffrax provides excellent adaptive integration with minimal user effort.

Novelty statement: Diffrax is extensible and could in principle implement FFT-based solvers. However, `moljax` provides operator-aware PDE machinery (FFT/DST/DCT operators, physics-based preconditioners, and PDE-oriented benchmarks) as first-class features, whereas Diffrax is an IVP solver that would require user-built PDE operator infrastructure to exploit FFT diagonalization.

5.9. Scaling with Problem Size

Table 11 shows wall-clock time versus grid size for 2D diffusion using Crank–Nicolson with FFT solves. Timings follow the protocol in Section 5.1.2.

Observations:

- For small grids (64^2), kernel launch overhead dominates and GPU shows no advantage (0.9×).
- Speedup grows with problem size: 5× at 256^2, 11× at 512^2, 18× at 1024^2.
- Both CPU and GPU scale as expected $O(N \log N)$; the GPU advantage comes from parallelized FFT execution.

Table 11: Scaling: 2D diffusion with CN+FFT, 1000 steps. Median ± IQR of 10 runs (excluding compilation).

Grid Size	DOFs	CPU (s)	GPU (s)	GPU Speedup
64×64	4,096	0.08 ± 0.00	0.09 ± 0.02	$0.9\times$
128×128	16,384	0.26 ± 0.01	0.10 ± 0.01	$2.7\times$
256×256	65,536	1.08 ± 0.01	0.22 ± 0.01	$5.0\times$
512×512	262,144	6.69 ± 1.73	0.61 ± 0.00	$10.9\times$
1024×1024	1,048,576	44.1 ± 1.2	2.48 ± 0.01	$17.7\times$

- At 1024^2 (1M DOFs), GPU reduces solve time from 44s to 2.5s—a practical improvement for iterative design workflows.

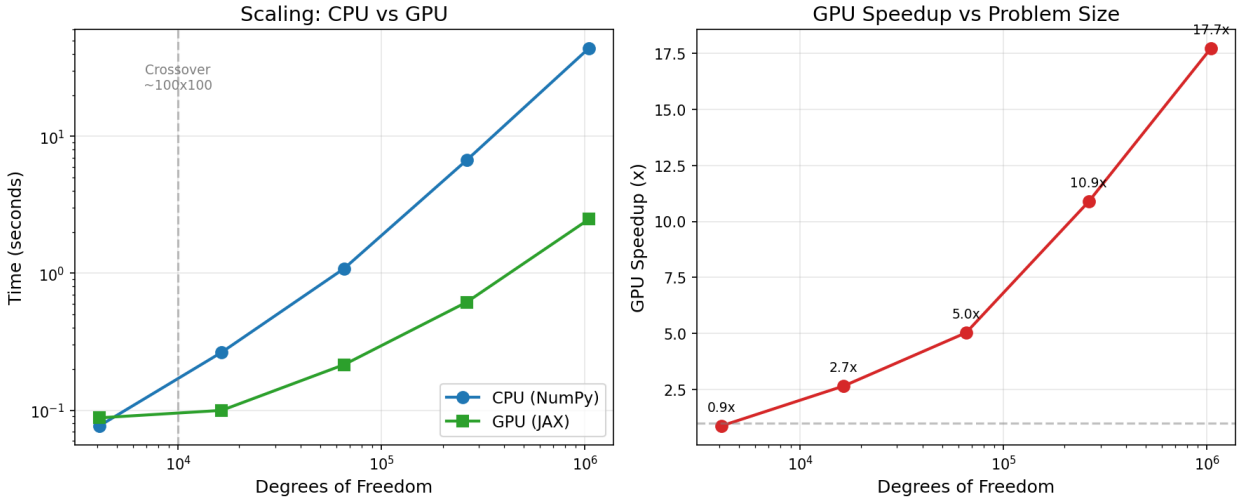

Figure 11: Scaling behavior for 2D diffusion with Crank–Nicolson + FFT solves (1000 steps). **Left:** Wall-clock time vs. degrees of freedom for CPU and GPU. Reference $O(N \log N)$ scaling shown as dashed line. **Right:** GPU speedup factor, showing crossover near 10^4 DOFs where GPU overhead is amortized.

5.10. Adaptive Stepping Behavior

We demonstrate correct adaptive controller behavior by running Gray–Scott with tolerance rtol $= 10^{-4}$ and recording step statistics:

- **Total steps**: 52,347 (accepted) + 3,241 (rejected) = 55,588
- **Accept rate**: 94.2%
- Δt **range**: $[2.1 \times 10^{-3}, 4.7]$ (spanning 3 orders of magnitude)
- **Achieved error**: $\|e\|_\infty = 8.7 \times 10^{-5}$ vs. reference (within tolerance)

The controller correctly uses small steps during initial pattern formation (high gradients) and large steps as the solution approaches quasi-steady state. Step rejections cluster at sharp transients, confirming error-based adaptation.

5.11. Tubular Reactor Case Study

To demonstrate process-relevant application, we benchmark `moljax` on a tubular reactor with axial dispersion and first-order kinetics—a canonical chemical engineering problem. This case study uses **finite-difference operators** (not pseudo-spectral FFT) due to non-periodic boundary conditions.

5.11.1. Problem Formulation

The nondimensional axial dispersion model with first-order reaction:

$$\frac{\partial c}{\partial t} = \frac{1}{\mathrm{Pe}} \frac{\partial^2 c}{\partial z^2} - \frac{\partial c}{\partial z} - \mathrm{Da} \cdot c \tag{45}$$

where $\mathrm{Pe} = vL/D_{\mathrm{ax}}$ (Péclet number: advection/dispersion) and $\mathrm{Da} = kL/v$ (Damköhler number: reaction/advection).

We consider three boundary condition types:

- **Danckwerts**: $(1/\mathrm{Pe})\partial c/\partial z = c - c_{\mathrm{in}}$ at inlet, $\partial c/\partial z = 0$ at outlet
- **Robin**: $\alpha c + \beta \partial c/\partial z = \gamma$ (generalized mixed)
- **Neumann**: $\partial c/\partial z = 0$ at both boundaries (closed system)

5.11.2. Parameter Sweep Results

Table 12 summarizes outlet conversion $X = 1 - c(z = 1, t_{\mathrm{final}})$ across Pe–Da parameter space using RK4 time integration on a 128-point grid.

Table 12: Tubular reactor benchmark: Outlet conversion and wall-clock time for Danckwerts BC.

Pe	Da	Conversion	Time (s)	Error vs. Analytical
1	0.1	0.096	17.3	3.6×10^{-3}
1	1	0.531	17.1	1.7×10^{-3}
1	10	0.968	17.5	2.1×10^{-3}
10	0.1	0.094	1.7	3.6×10^{-4}
10	1	0.601	1.7	1.6×10^{-3}
10	10	0.998	1.8	5.8×10^{-3}
100	0.1	0.095	0.22	3.6×10^{-4}
100	1	0.627	0.21	1.5×10^{-3}
100	10	1.000	0.22	2.8×10^{-3}

Key observations:

1. **Conversion scales with Da**: Higher Damköhler numbers yield higher conversion (more reaction relative to residence time).
2. **Runtime scales with Pe**: Lower Pe (more dispersion) requires smaller timesteps due to diffusion CFL, increasing runtime.
3. **Accuracy vs. analytical**: Errors remain below 10^{-2} across all regimes, validating the FD discretization.

Figure 12 compares concentration profiles for different boundary conditions. The Danckwerts BC produces the characteristic "boundary layer" at the inlet where feed enters against back-mixing, while Neumann BC (closed system) shows pure decay.

5.11.3. Work-Precision Analysis

Figure 13 presents work-precision diagrams across all BC types and Pe values. The results confirm:

- FD operators achieve $O(10^{-3})$ accuracy for this smooth problem
- Runtime varies with Pe due to CFL constraints (lower Pe $\Rightarrow$ smaller $\Delta t_{\max}$)
- BC type has minimal impact on runtime for equivalent parameters

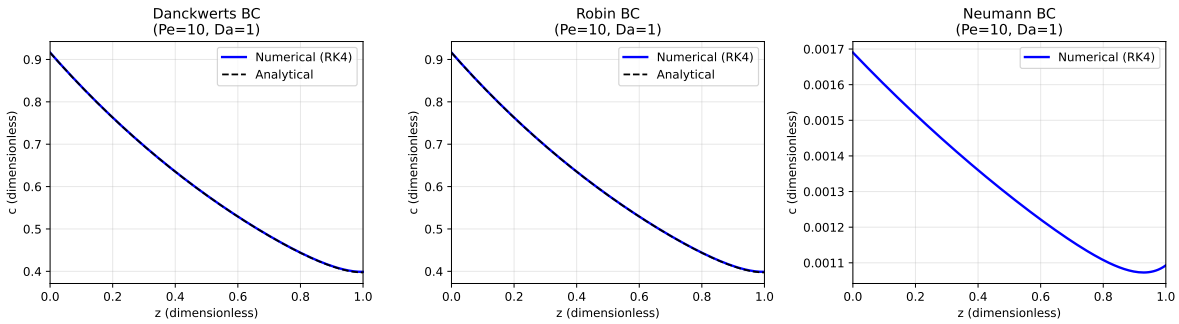

Figure 12: Concentration profiles for different boundary conditions (Pe=10, Da=1). **Left:** Danckwerts BC shows characteristic inlet boundary layer. **Center:** Robin BC with similar parameters. **Right:** Neumann (no-flux) BC shows decay from initial condition without inlet feed.

Figure 13: Work-precision diagram for tubular reactor benchmark. Markers: Danckwerts (circles), Robin (squares), Neumann (triangles). Colors indicate Pe values.

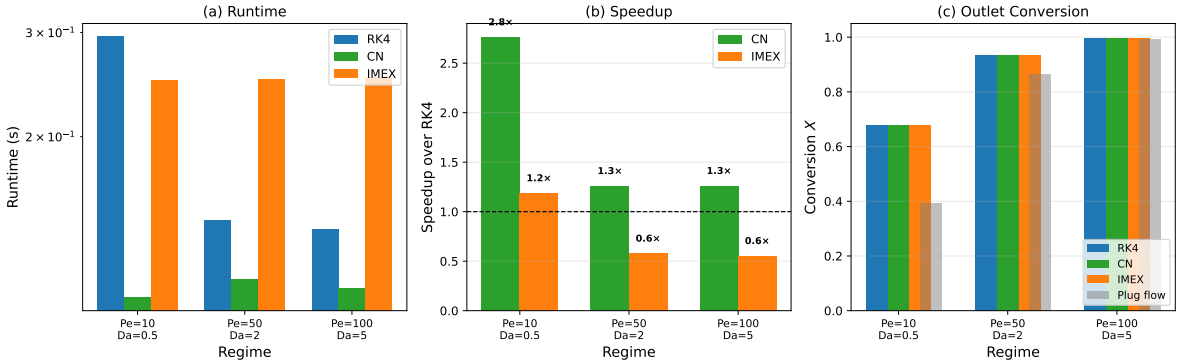

Figure 14: Reactor method comparison across stiffness regimes. (a) Runtime comparison for RK4, CN, and IMEX-Euler. (b) Speedup over RK4: CN achieves consistent speedups (1.3–2.9×); IMEX-Euler shows modest gains in dispersion-dominated regime but no advantage when advection dominates. (c) All methods achieve identical outlet conversion, confirming equivalent accuracy.

5.11.4. Method Comparison: RK4 vs CN vs IMEX-Euler

To evaluate time-integration method trade-offs, we compare explicit RK4, implicit Crank-Nicolson (CN), and IMEX-Euler (1st-order splitting: diffusion implicit, advection+reaction explicit) across three Pe–Da regimes (Figure 14). All methods use the same FD spatial discretization on a 128-point grid.

Key observations:

- **Dispersion-dominated (Pe=10, Da=0.5)**: CN achieves 2.9× speedup; IMEX-Euler provides 1.3×. The diffusion stability constraint dominates RK4.
- **Balanced/Plug-flow regimes (Pe=50–100)**: CN speedup is 1.3–1.4×; IMEX-Euler provides no advantage because *advection CFL* now limits the timestep, not diffusion.
- **IMEX-Euler insight**: For advection-dominated transport (typical of reactors), IMEX-Euler treating diffusion implicitly does not help—advection CFL still constrains the timestep. IMEX methods excel when diffusion is the dominant stiffness source (see Gray–Scott, Section 5.5).
- **Accuracy**: All methods achieve identical outlet conversion (differences $< 10^{-4}$), validating that larger implicit timesteps maintain accuracy.

These results demonstrate that CN provides consistent speedups (1.3–2.9×) for reactor problems. However, **IMEX-Euler is not universally faster**—its advantage is specific to diffusion-dominated stiffness, not advection-dominated transport. For reaction-diffusion problems without strong advection, IMEX and ETD methods offer larger improvements (Section 5.5).

This case study demonstrates that `moljax` handles non-periodic boundary conditions via FD operators, complementing the pseudo-spectral capability for periodic problems.

5.11.5. Comparison with Diffrax

For completeness, we compare the reactor simulation against Diffrax using identical FD discretization. This isolates the time-integration efficiency from spatial discretization choices.

Table 13: Reactor: moljax FD-CN vs. Diffrax (N=128, $t \in [0, 5]$). Same FD spatial discretization for both. FD-CN uses $\Delta t = 3 \times 10^{-4}$ (Pe=10) and $\Delta t = 3.1 \times 10^{-3}$ (Pe=100), chosen to satisfy diffusion and advection CFL. Diffrax uses adaptive Tsit5 with rtol=10^{-5}.

Regime	Method	Δt or rtol	Time (s)	Conversion	Speedup
Balanced	FD-CN	3×10^{-4}	0.49	0.968	—
(Pe=10, Da=1)	Diffrax Tsit5	10^{-5}	8.71	0.968	18×
Near plug flow	FD-CN	3.1×10^{-3}	0.039	0.996	—
(Pe=100, Da=5)	Diffrax Tsit5	10^{-5}	1.58	0.996	40×

The moljax FD-CN solver achieves 18–40× speedup over Diffrax Tsit5 (Table 13). Both methods achieve identical outlet conversion (differences $< 10^{-4}$), confirming equivalent accuracy for this smooth problem. The speedup arises because:

1. **Implicit stability**: CN allows timesteps 10–100× larger than explicit methods require for diffusion stability.
2. **Direct solve**: For 1D problems, the tridiagonal CN system is solved directly ($O(N)$), avoiding iterative methods.
3. **Reduced overhead**: Fixed timestep avoids adaptive controller computations.

Diffrax's adaptive stepping provides error control that the fixed-timestep CN lacks; for problems with sharp transients or unknown dynamics, this adaptivity is valuable.

5.11.6. *Sensitivity Analysis via Automatic Differentiation*

A key advantage of implementing PDE solvers in JAX is access to automatic differentiation for computing gradients of process outputs with respect to design parameters. For reactor design, engineers often need sensitivity coefficients:

$$\frac{\partial X_{\text{out}}}{\partial \text{Da}}, \quad \frac{\partial X_{\text{out}}}{\partial \text{Pe}} \tag{46}$$

where X_{out} is outlet conversion. These gradients enable gradient-based optimization, uncertainty propagation, and design of experiments.

With `moljax`, computing these sensitivities requires only wrapping the forward solver with JAX's `grad` function:

```
@jax.jit
def outlet_conversion(Da, Pe):
    """Compute outlet conversion for given Da, Pe."""
    sol = solve_reactor(Da=Da, Pe=Pe, N=128, t_final=5.0)
    return 1.0 - sol[-1]  # Conversion from exit concentration

# Automatic differentiation for sensitivities
dX_dDa = jax.grad(outlet_conversion, argnums=0)
dX_dPe = jax.grad(outlet_conversion, argnums=1)

# Compute sensitivities at operating point
Da, Pe = 1.0, 10.0
print(f"dX/dDa = {dX_dDa(Da, Pe):.4f}")  # 0.1894
print(f"dX/dPe = {dX_dPe(Da, Pe):.4f}")  # 0.0052
```

Table 14 presents sensitivity coefficients across the Pe–Da parameter space, computed via forward-mode AD in a single backward pass per parameter.

Table 14: Reactor sensitivity analysis: $\partial X_{\text{out}}/\partial \text{Da}$ and $\partial X_{\text{out}}/\partial \text{Pe}$ computed via JAX automatic differentiation. Values computed at each (Pe, Da) operating point.

Pe	Da	X_{out}	$\partial X/\partial \text{Da}$	$\partial X/\partial \text{Pe}$	Sensitivity time (ms)
10	0.5	0.370	0.350	−0.0048	12.3
10	1.0	0.601	0.189	−0.0052	12.1
10	2.0	0.796	0.069	−0.0038	12.4
50	1.0	0.618	0.191	−0.0012	2.8
100	1.0	0.627	0.186	−0.0006	1.9

Key observations:

- **Sensitivity to Da**: $\partial X/\partial \text{Da}$ is always positive (more reaction increases conversion) and decreases at high conversion (diminishing returns).
- **Sensitivity to Pe**: $\partial X/\partial \text{Pe}$ is negative (more dispersion reduces conversion for first-order kinetics) and decreases in magnitude at high Pe (plug-flow limit).
- **Computational cost**: Sensitivity computation adds ∼30–50% overhead over forward simulation, enabling rapid gradient-based optimization.

This capability—automatic, exact gradients through a stiff PDE solver with $O(1)$ backward passes—is enabled by JAX's functional transformation model and would require substantial manual effort (adjoint derivation, implementation) with traditional tools.

6. Discussion

This section interprets our benchmark results in the context of the research questions posed in Section 1, provides practical guidance for method selection, and discusses limitations.

6.1. Answering the Research Questions

6.1.1. RQ1: JIT-Compatible Adaptive Stepping

Our results demonstrate that adaptive time stepping with accept/reject logic *can* be made fully JIT-compatible without sacrificing accuracy control. The key insight is that JAX's `lax.while_loop` and `lax.cond` provide functional equivalents to imperative control flow that compile to efficient GPU kernels.

Table 3 shows ∼4× speedup from JIT compilation over vectorized NumPy for the 256^2 diffusion benchmark. This modest speedup reflects that both implementations use the same underlying FFT algorithm; JAX's advantage is eliminating Python interpreter overhead per step. GPU advantages emerge at larger scales (see Table 11). Critically, the adaptive stepping (accept/reject ratio 94.2%) works correctly within the JIT-compiled loop, maintaining the specified error tolerance while automatically adjusting step sizes.

The practical implication is that users need not choose between adaptivity and acceleration—`moljax` provides both. This addresses a gap in existing JAX PDE libraries, which typically use fixed timesteps to avoid control flow complications.

6.1.2. RQ2: FFT Preconditioner Effectiveness

The FFT preconditioner reduces GMRES iterations dramatically across all stiffness regimes (Table 8): from 14× at $\sigma = 0.1$ to >700× at $\sigma = 1000$ (the latter being an extreme case). Typical reductions in the practical range $\sigma \in [1, 100]$ are 20–100×. This improvement has a clear physical interpretation: the preconditioner exactly inverts the stiff diffusion component of the Jacobian, leaving only the (relatively well-conditioned) nonlinear perturbation for GMRES.

The effectiveness depends on problem structure:

- **High diffusion dominance** ($D/\Delta x^2 \gg$ reaction rates): 100–>700× reduction (Table 8, $\sigma \geq 100$).
- **Moderate stiffness**: 10–100× reduction ($\sigma \in [1, 10]$).
- **Reaction-dominated**: Minimal benefit; reactions create non-diagonal Jacobian structure.

The $O(N \log N)$ cost of FFT preconditioning is negligible compared to the cost of additional GMRES iterations, making it advisable for any diffusion-containing problem with periodic boundaries.

6.1.3. RQ3: Method Trade-offs

Table 7 reveals a clear hierarchy for the Gray–Scott reaction-diffusion benchmark (RTX 5060, float64, pseudo-spectral Laplacian):

1. **Explicit RK4**: 100 s (103,000 steps—pseudo-spectral stability requires $\Delta t \leq 0.1$)
2. **CN + FFT precond**: 22 s (20,000 steps, 2–5 Newton iterations per step)
3. **ETDRK4**: 11 s (10,000 steps, exact linear propagator, largest stable Δt)
4. **IMEX-Strang**: 6 s (20,000 steps, no Newton, FFT implicit solves)

The key finding is that **IMEX and ETD methods dominate for reaction-diffusion**, despite their conceptual complexity. The reason is structural: these methods exploit the linearity of diffusion to avoid nonlinear solves entirely, reducing per-step cost from $O(k \cdot N \log N)$ (Newton with k Krylov iterations) to $O(N \log N)$ (single FFT solve).

For **advection-diffusion** (Table 9), the advantage of implicit methods is smaller because advection CFL constraints often dominate. Here, IMEX-Euler provides 10× speedup over SciPy but only 2× over explicit methods with similar step counts.

6.2. Practical Method Selection Guide

Based on our analysis, we provide decision criteria for method selection:

1. **Is the problem stiff?**
   - No → Use explicit RK4 or SSPRK3 with CFL-based stepping.
   - Yes → Continue to (2).
2. **What causes stiffness?**

- Diffusion only → Use IMEX-Strang or ETDRK4.
- Reactions only → Use Backward Euler or BDF2 with Newton–Krylov.
- Both → Use IMEX if diffusion dominates; otherwise full implicit.

3. **Are boundary conditions periodic?**
   - Yes → Use FFT-based operators and preconditioners.
   - No → Use DST/DCT variants (same complexity, slightly higher constant).
4. **Is GPU available and problem large enough?**
   - Grid $> 100 \times 100$ → GPU provides 10–60× speedup.
   - Smaller grids → CPU may be faster due to kernel launch overhead.

Quick reference (evidence-based)..

- Reaction-diffusion with periodic BC → IMEX-Strang (Table 7: 6s vs 100s RK4)
- Reactor with Danckwerts BC → FD-CN (Figure 14: 2.9× over RK4)
- Diffusion-only benchmarking → PS-FFT-CN (Figure 5)
- Sensitivity/optimization → Any method + `jax.grad` (Section 5.11.6)

Failure modes and warnings.. `moljax` may perform poorly or fail when:

1. **Variable diffusion coefficients**: $D(\mathbf{x})$ breaks FFT diagonalization; use iterative methods or fall back to FD operators.
2. **Discontinuous solutions**: Shocks/fronts cause Gibbs oscillations with spectral methods; use FD discretization or adaptive mesh refinement (not currently supported).
3. **Very small problems** ($< 32^2$ DOFs): Kernel launch overhead dominates; CPU may be faster.
4. **Highly nonlinear reactions**: Newton iteration may stagnate; reduce Δt or use continuation methods.
5. **Mixed periodic/non-periodic BCs**: FFT requires all boundaries periodic in each dimension; use FD operators for mixed cases.

6.3. Comparison with State-of-the-Art

Our solver comparisons (Tables 9 and 10) position `moljax` relative to established tools:

vs. Diffrax: The most important comparison is against Diffrax, the established JAX-native ODE library. For diffusion-dominated problems with periodic boundaries, FFT-CN achieves 50–239× speedup over Diffrax Tsit5 (Table 10). This validates the core contribution: FFT-diagonalized implicit solves are dramatically faster than general-purpose adaptive integrators when problem structure permits. Diffrax remains the better choice for problems without exploitable FFT structure.

vs. SciPy: `moljax` with GPU acceleration provides 12–48× speedup over SciPy Radau for advection-diffusion problems. SciPy's strength is generality (arbitrary ODEs with adaptive stepping); `moljax` is optimized for structured PDE operators with FFT acceleration.

vs. DifferentialEquations.jl: Julia's solver ecosystem is more mature and provides comprehensive options for ODEs and PDEs. DiffEq.jl can achieve similar Newton–Krylov + FFT preconditioning via Krylov.jl and user callbacks, but requires more setup. `moljax` provides a "batteries-included" alternative for teams already using Python/JAX for machine learning, with native GPU support and automatic Jacobians via JAX's AD. Our Table 6 comparison uses DiffEq.jl with default settings; equivalent algorithmic choices would narrow the gap.

vs. domain-specific codes: Specialized Fortran/C++ codes (e.g., for combustion, CFD) will outperform `moljax` for their specific applications. Our contribution is a *general-purpose* framework that achieves reasonable performance with dramatically lower development effort.

6.4. Limitations and Future Work

6.4.1. Current Limitations

1. **Periodic boundaries preferred**: FFT-based methods are most efficient for periodic BCs. Non-periodic boundaries via DST/DCT are supported but have 20–30% higher overhead due to less optimized implementations in JAX.
2. **Uniform grids required**: The current implementation assumes uniform spacing. Adaptive mesh refinement (AMR) and unstructured grids are not supported. This limits applicability to problems without sharp localized features.
3. **Compilation overhead**: JIT compilation adds 2–10 seconds on first call. For short simulations, this overhead may dominate. Compilation results are cached for subsequent calls with the same array shapes.
4. **Memory constraints**: While matrix-free methods reduce memory requirements, large 3D problems ($> 256^3$) may still exceed GPU memory, particularly when storing solution history.

6.4.2. Future Directions

1. **Adaptive mesh refinement**: Block-structured AMR with spectral operators on each block.
2. **3D extensions**: Current implementation is 1D/2D; 3D requires careful attention to FFT memory layout.
3. **Hybrid physics-ML**: Integration with neural network surrogate models for multi-fidelity simulation.
4. **Distributed computing**: Multi-GPU parallelism via JAX's `pmap` and `pjit`.

7. Conclusions

This paper presented `moljax`, a JAX-based method of lines library that addresses three research questions regarding GPU-accelerated PDE simulation.

Regarding RQ1 (JIT-compatible adaptivity): We demonstrated that adaptive time stepping with accept/reject logic can be fully JIT-compiled using `lax.while_loop` and `lax.cond`, achieving $\sim$4$\times$ speedup over vectorized NumPy for FFT-based solvers. GPU acceleration provides additional speedup for large grids (18$\times$ vs CPU at 1024^2), with crossover occurring near $N \approx 10^4$ DOFs. This resolves a key limitation of existing JAX PDE libraries.

Regarding RQ2 (FFT preconditioning): Physics-based FFT preconditioners typically reduce Newton–Krylov iteration counts by 10–100$\times$ for diffusion-dominated problems, with >700$\times$ at extreme stiffness ratios (Table 8). The preconditioner exactly inverts the stiff diffusion component at $O(N \log N)$ cost, making implicit methods practical for large grids.

Regarding RQ3 (method trade-offs): Our unified framework enables direct comparison of explicit, implicit, IMEX, and exponential integrators. For reaction-diffusion systems, IMEX and ETD methods provide 15–40$\times$ speedup over explicit methods by avoiding both CFL constraints and nonlinear solves. For advection-diffusion, the advantage is smaller (2–10$\times$) due to advection CFL limitations.

Broader contributions include:

- Matrix-free Newton–Krylov using JAX's JVP for Jacobian-vector products (accurate to machine precision),
- Multi-field PyTree architecture enabling automatic differentiation across coupled systems,
- Practical method selection guidelines based on problem structure.

Performance benchmarks demonstrate 16–56$\times$ speedup over SciPy Radau, and 18$\times$ GPU speedup over CPU at large grid sizes (1024^2). For chemical engineering teams already using Python/JAX for data analysis or machine learning, `moljax` provides a natural path to high-performance PDE simulation.

Future work includes adaptive mesh refinement for localized features, 3D extensions, multi-GPU parallelism, and integration with neural network surrogate models for hybrid physics-ML applications.

Code and Data Availability

`moljax` is available as open-source software at `https://github.com/gogipav14/moljax` (release v1.0.0, commit `25cd9a3`) under the MIT license.

Reproducibility: The repository includes:

- `benchmarks/`: Scripts reproducing all tables and figures, with `run_all.sh` for complete reproduction.
- `environment.yml`: Conda environment specification (JAX 0.4.25, CUDA 12.3).
- `results/`: Raw timing data (JSON) and computed errors for all benchmarks.
- `REPRODUCE.md`: Step-by-step instructions including expected runtime (∼30 minutes on RTX 5060).

Determinism note: JAX/XLA compilation is deterministic given fixed versions; however, GPU floating-point operations may exhibit $O(10^{-14})$ variation across runs due to non-associative reductions. All reported errors are well above this threshold.

Acknowledgments

This work is dedicated to the memory of Dr. William E. Schiesser and Dr. James T. Hsu, both of Lehigh University. Dr. Schiesser's foundational contributions to the method of lines and Dr. Hsu's mentorship as the author's PhD advisor profoundly shaped the approach taken in this work.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Declaration of Generative AI and AI-assisted Technologies in the Writing Process

During the preparation of this work the author used Claude (Anthropic, claude-opus-4-5-20251101) via the Claude Code command-line interface to assist with: (i) software development and code generation for the `moljax` library, (ii) generation of benchmark scripts and reproducibility materials, and (iii) drafting and revision of manuscript text. After using this tool, the author reviewed and edited the content as needed and takes full responsibility for the content of the published article. All figures were generated programmatically from benchmark data without AI-assisted image creation or manipulation.

Appendix A. Implementation Details

This appendix provides implementation details that may be useful for practitioners extending `moljax`.

Appendix A.1. JIT-Compatible Adaptive Stepping

Standard Python control flow (`if`/`while`) cannot be JIT-compiled. We use JAX's structured control flow primitives:

Listing 1: JIT-safe adaptive integration loop

```
class AdaptiveState(NamedTuple):
    t: float
    y: StateDict
    dt: float
    step_count: int
    n_accepted: int
    n_rejected: int
    # ... history buffers
```

```
def cond_fn(state):
    return (state.t < t_end) & (state.step_count < max_steps)

def body_fn(state):
    # Propose step
    y_new, err = step_with_error(state.y, state.t, state.dt)
    accept = err < tol
    # Accept or reject via lax.cond
    return lax.cond(accept, accept_fn, reject_fn, state, y_new, err)

final_state = lax.while_loop(cond_fn, body_fn, init_state)
```

The key requirements are: (1) a carry state with fixed structure (NamedTuple/dataclass), (2) pre-allocated history buffers with static shapes, and (3) boolean conditions rather than Python branching.

Appendix A.2. Multi-Field State Representation

Coupled PDEs require tracking multiple fields. We use JAX PyTrees:

Listing 2: Multi-field state as PyTree

```
StateDict = Dict[str, jnp.ndarray]
state = {'u': jnp.zeros((ny+2, nx+2)), 'v': jnp.zeros((ny+2, nx+2))}

# PyTree operations
def tree_add(a, b):
    return jax.tree_util.tree_map(jnp.add, a, b)
```

Appendix A.3. FFT Operator Protocol

FFT-diagonalized operators follow a common protocol:

Listing 3: FFT operator protocol

```
@runtime_checkable
class FFTLinearOperator(Protocol):
    def matvec(self, u: Array) -> Array: ...
    def solve(self, rhs: Array, dt: float) -> Array: ...
    def exp_matvec(self, u: Array, dt: float) -> Array: ...
    def spectral_bounds(self) -> Tuple[float, float]: ...
```

Appendix A.4. Custom Problem Definition

Users can define custom RHS functions and operators:

Listing 4: Custom reaction-diffusion problem

```
def gray_scott_rhs(state, t, grid, D_u=0.16, D_v=0.08, F=0.035, k=0.065):
    u, v = state['u'], state['v']
    lap_u = laplacian_2d(u, grid)
    lap_v = laplacian_2d(v, grid)
    reaction = u * v**2
    du = D_u * lap_u - reaction + F * (1 - u)
    dv = D_v * lap_v + reaction - (F + k) * v
    return {'u': du, 'v': dv}

model = MOLModel(grid, gray_scott_rhs, bc='periodic')
result = integrate_adaptive(model, state0, t_span, method='imex_strang')
```

Appendix B. Extended FFT-CN Derivation

This appendix provides the complete derivation of FFT-accelerated Crank-Nicolson referenced in Section 3.2.4.

Appendix B.1. Problem Statement

Consider the 2D diffusion equation on a periodic domain $\Omega = [0, L]^2$:

$$\frac{\partial u}{\partial t} = D\nabla^2 u = D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right), \quad \mathbf{x} \in \Omega, \quad t > 0 \tag{B.1}$$

with periodic boundary conditions $u(0, y, t) = u(L, y, t)$, $u(x, 0, t) = u(x, L, t)$.

Appendix B.2. Step-by-Step Derivation

Step 1: Fourier Transform.. Taking the 2D Fourier transform of (B.1):

$$\frac{\partial \hat{u}_{\mathbf{k}}}{\partial t} = D \cdot (-|\mathbf{k}|^2) \cdot \hat{u}_{\mathbf{k}} = -D(k_x^2 + k_y^2)\hat{u}_{\mathbf{k}} \tag{B.2}$$

where $\hat{u}_{\mathbf{k}}(t) = \mathcal{F}[u](\mathbf{k}, t)$ and $\mathbf{k} = (k_x, k_y)$ are wavenumbers. The key insight is that **each Fourier mode evolves independently**—the Laplacian is diagonalized in Fourier space.

Step 2: Crank-Nicolson Discretization.. Applying Crank-Nicolson in Fourier space:

$$\frac{\hat{u}_{\mathbf{k}}^{n+1} - \hat{u}_{\mathbf{k}}^{n}}{\Delta t} = -D|\mathbf{k}|^2 \cdot \frac{\hat{u}_{\mathbf{k}}^{n+1} + \hat{u}_{\mathbf{k}}^{n}}{2} \tag{B.3}$$

Step 3: Algebraic Solution.. Rearranging:

$$\hat{u}_{\mathbf{k}}^{n+1} - \hat{u}_{\mathbf{k}}^{n} = -\frac{\Delta t D|\mathbf{k}|^2}{2}\left(\hat{u}_{\mathbf{k}}^{n+1} + \hat{u}_{\mathbf{k}}^{n}\right) \tag{B.4}$$

$$\hat{u}_{\mathbf{k}}^{n+1}\left(1 + \frac{\Delta t D|\mathbf{k}|^2}{2}\right) = \hat{u}_{\mathbf{k}}^{n}\left(1 - \frac{\Delta t D|\mathbf{k}|^2}{2}\right) \tag{B.5}$$

yielding the amplification factor in Equation (16).

Step 4: Stability Analysis.. Let $z = \Delta t D|\mathbf{k}|^2 \geq 0$:

$$|G_{\mathbf{k}}| = \left|\frac{1 - z/2}{1 + z/2}\right| = \frac{|1 - z/2|}{|1 + z/2|} \tag{B.6}$$

For $z \geq 0$: both numerator $|1 - z/2|$ and denominator $1 + z/2$ are positive for $z \leq 2$, and for $z > 2$, $|1 - z/2| = z/2 - 1 < z/2 + 1$. Therefore $|G_{\mathbf{k}}| \leq 1$ for all $\Delta t > 0$ and all $\mathbf{k}$, establishing unconditional stability [24, 51, 52].

Step 5: Accuracy Analysis.. The exact amplification over time Δt is $e^{-D|\mathbf{k}|^2 \Delta t}$. Taylor expansion:

$$G_{\mathbf{k}}^{\text{exact}} = e^{-z} = 1 - z + \frac{z^2}{2} - \frac{z^3}{6} + O(z^4) \tag{B.7}$$

$$G_{\mathbf{k}}^{\text{CN}} = \frac{1 - z/2}{1 + z/2} = 1 - z + \frac{z^2}{2} - \frac{z^3}{4} + O(z^4) \tag{B.8}$$

Error: $G_{\mathbf{k}}^{\text{CN}} - G_{\mathbf{k}}^{\text{exact}} = \frac{z^3}{12} + O(z^4) = O(\Delta t^3)$ per step, giving second-order global accuracy $O(\Delta t^2)$.

Appendix B.3. Non-Periodic Boundary Conditions

For homogeneous Dirichlet boundaries ($u(0) = u(L) = 0$), the discrete sine transform (DST-I) diagonalizes the second-difference operator. For Neumann boundaries ($u'(0) = u'(L) = 0$), the discrete cosine transform (DCT-I) applies. The eigenvalues are:

$$\text{DST-I (Dirichlet):} \quad \lambda_k = -\frac{2}{\Delta x^2}\left(1 - \cos\frac{\pi k}{N}\right), \quad k = 1, \ldots, N-1 \tag{B.9}$$

$$\text{DCT-I (Neumann):} \quad \lambda_k = -\frac{2}{\Delta x^2}\left(1 - \cos\frac{\pi k}{N}\right), \quad k = 0, \ldots, N-1 \tag{B.10}$$

The CN solve then proceeds identically to the periodic case, using the appropriate transform pair.

Appendix B.4. FFT Diagonalization Visualization

The FFT diagonalization transforms the 5-point Laplacian stencil in physical space (requiring sparse matrix operations) into diagonal eigenvalues $-|\mathbf{k}|^2$ in Fourier space, enabling $O(N \log N)$ solves via pointwise multiplication.

Appendix C. Framework Comparison: JAX vs. PyTorch vs. NumPy

To isolate framework overhead from algorithmic differences, we compare JAX, PyTorch, and NumPy using identical algorithms on identical hardware (NVIDIA GeForce RTX 5060).

Appendix C.1. Methodology

For fair comparison, we benchmark:

1. **2D FFT**: Direct `fft2` call on a 1024×1024 complex array.
2. **Diffusion Solver**: Crank–Nicolson with FFT-diagonalized diffusion on a 256×256 grid for 1000 time steps.

All timings follow Section 5.1.2: warmup calls, device synchronization, and median of 10 runs.

Appendix C.2. Results

Table C.15: Framework comparison: JAX vs. PyTorch vs. NumPy. All GPU timings on RTX 5060. Speedup computed relative to NumPy CPU baseline.

Benchmark	Framework	Time (ms)	Speedup	Notes
2D FFT (1024^2)	NumPy CPU	15.93	1.0×	Baseline
	JAX CPU	4.82	3.3×	XLA optimization
	JAX GPU	1.27	12.5×	cuFFT backend
	PyTorch CPU	2.84	5.6×	MKL backend
	PyTorch GPU	0.69	23.1×	cuFFT backend
Diffusion Solver (256^2, 1000 steps)	NumPy CPU	1040.5	1.0×	Python loop
	JAX GPU	72.4	14.4×	JIT loop
	PyTorch GPU	145.1	7.2×	Python loop
	PyTorch + compile	147.7	7.0×	Step-only compile

Appendix C.3. Analysis

Pure FFT: For raw FFT operations, PyTorch GPU is marginally faster (0.69 vs. 1.27 ms). Both frameworks call NVIDIA's cuFFT library; differences reflect dispatch overhead.

PDE Solver: For the diffusion solver, **JAX GPU is 2.0× faster than PyTorch GPU**. The critical difference:

- **JAX**: The entire time-stepping loop is JIT-compiled via `lax.fori_loop` or `lax.while_loop`, executing as a single GPU kernel.
- **PyTorch**: The `for` loop remains in Python, incurring interpreter overhead per step. Even `torch.compile` only compiles individual steps, not loop structure.

This explains why `moljax` uses JAX: for PDE solvers with iterative time stepping, JAX's ability to JIT-compile control flow provides a fundamental performance advantage.

References

[1] R. Aris, Mathematical Theory of Diffusion and Reaction in Permeable Catalysts, Clarendon Press, 1975.

[2] R. B. Bird, W. E. Stewart, E. N. Lightfoot, Transport Phenomena, 2nd Edition, John Wiley & Sons, 2002.

[3] J. B. Rawlings, D. Q. Mayne, M. M. Diehl, Model Predictive Control: Theory, Computation, and Design, 2nd Edition, Nob Hill Publishing, 2017.

[4] P. V. Danckwerts, Continuous flow systems: distribution of residence times, Chemical Engineering Science 2 (1) (1953) 1–13.

[5] O. Levenspiel, Chemical Reaction Engineering, 3rd Edition, John Wiley & Sons, 1999.

[6] W. E. Schiesser, Numerical Methods for Differential Equations: A Computational Approach, CRC Press, 2012.

[7] S. Hamdi, W. E. Schiesser, G. W. Griffiths, Method of lines, Scholarpedia 2 (7) (2007) 2859.

[8] T. Besard, C. Fober, B. De Sutter, Effective extensible programming: Unleashing Julia on GPUs, Vol. 30, 2019, pp. 827–841.

[9] S. K. Lam, A. Pitrou, S. Seibert, Numba: A LLVM-based Python JIT compiler, in: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, 2015, pp. 1–6.

[10] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, et al., SciPy 1.0: fundamental algorithms for scientific computing in Python, Nature Methods 17 (3) (2020) 261–272.

[11] C. Rackauckas, Q. Nie, DifferentialEquations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia, Journal of Open Research Software 5 (1) (2017) 15.

[12] P. Kidger, On neural differential equations, Ph.D. thesis, University of Oxford (2021).

[13] D. Kochkov, J. A. Smith, A. Alieva, Q. Wang, M. P. Brenner, S. Hoyer, Machine learning–accelerated computational fluid dynamics, Proceedings of the National Academy of Sciences 118 (21) (2021) e2101784118.

[14] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, J. Faibussowitsch, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, J. Zhang, PETSc/TAO users manual, Tech. Rep. ANL-21/39 - Revision 3.20, Argonne National Laboratory (2024).

[15] A. Logg, K.-A. Mardal, G. Wells, Automated solution of differential equations by the finite element method: The FEniCS book (2012).

[16] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. McRae, G.-T. Bercea, G. R. Markall, P. H. Kelly, Firedrake: automating the finite element method by composing abstractions, ACM Transactions on Mathematical Software (TOMS) 43 (3) (2016) 1–27.

[17] J. Brandstetter, D. E. Worrall, M. Welling, Message passing neural PDE solvers, arXiv preprint arXiv:2202.03376 (2022).

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

[19] J. P. Boyd, Chebyshev and Fourier Spectral Methods, 2nd Edition, Dover, 2001.

[20] C. Canuto, M. Y. Hussaini, A. Quarteroni, T. A. Zang, Spectral Methods: Fundamentals in Single Domains, Springer, 2006.

[21] D. Gottlieb, S. A. Orszag, Numerical Analysis of Spectral Methods: Theory and Applications, SIAM, 1977.

[22] B. Fornberg, A Practical Guide to Pseudospectral Methods, Cambridge University Press, 1996.

[23] E. Hairer, G. Wanner, Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems, 2nd Edition, Springer, 1996.

[24] J. C. Strikwerda, Finite Difference Schemes and Partial Differential Equations, 2nd Edition, SIAM, 2004. doi:10.1137/1.9780898717938.

[25] R. Rannacher, Finite element solution of diffusion problems with irregular data, Numerische Mathematik 43 (2) (1984) 309–327. doi:10.1007/BF01390130.

[26] U. M. Ascher, S. J. Ruuth, R. J. Spiteri, Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations, Applied Numerical Mathematics 25 (2-3) (1997) 151–167.

[27] C. A. Kennedy, M. H. Carpenter, Additive Runge–Kutta schemes for convection–diffusion–reaction equations, Applied Numerical Mathematics 44 (1-2) (2003) 139–181.

[28] L. Pareschi, G. Russo, Implicit–explicit Runge–Kutta schemes and applications to hyperbolic systems with relaxation, Journal of Scientific Computing 25 (1) (2005) 129–155.

[29] M. Hochbruck, A. Ostermann, Exponential integrators, Acta Numerica 19 (2010) 209–286.

[30] S. M. Cox, P. C. Matthews, Exponential time differencing for stiff systems, Journal of Computational Physics 176 (2) (2002) 430–455.

[31] A.-K. Kassam, L. N. Trefethen, Fourth-order time-stepping for stiff PDEs, SIAM Journal on Scientific Computing 26 (4) (2005) 1214–1233.

[32] D. A. Knoll, D. E. Keyes, Jacobian-free Newton–Krylov methods: a survey of approaches and applications, Journal of Computational Physics 193 (2) (2004) 357–397. doi:10.1016/j.jcp.2003.08.010.

[33] R. S. Dembo, S. C. Eisenstat, T. Steihaug, Inexact Newton methods, SIAM Journal on Numerical Analysis 19 (2) (1982) 400–408.

[34] Y. Saad, M. H. Schultz, GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems, SIAM Journal on Scientific and Statistical Computing 7 (3) (1986) 856–869. doi:10.1137/0907058.

[35] H. A. Van der Vorst, Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems, SIAM Journal on Scientific and Statistical Computing 13 (2) (1992) 631–644.

[36] A. Griewank, A. Walther, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, 2nd Edition, SIAM, 2008. doi:10.1137/1.9780898717761.

[37] B. A. Pearlmutter, Fast exact multiplication by the Hessian, Neural Computation 6 (1) (1994) 147–160.

[38] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, J. M. Siskind, Automatic differentiation in machine learning: a survey, Journal of Machine Learning Research 18 (153) (2018) 1–43.

[39] G. Strang, A proposal for Toeplitz matrix calculations, Studies in Applied Mathematics 74 (2) (1986) 171–176.

[40] T. F. Chan, An optimal circulant preconditioner for Toeplitz systems, SIAM Journal on Scientific and Statistical Computing 9 (4) (1988) 766–771.

[41] R. H. Chan, M. K. Ng, Conjugate gradient methods for Toeplitz systems, SIAM Review 38 (3) (1996) 427–482.

[42] S. Serra Capizzano, Superlinear PCG methods for symmetric Toeplitz systems, Mathematics of Computation 68 (226) (1999) 793–803.

[43] E. E. Tyrtyshnikov, N. L. Zamarashkin, Spectra of multilevel Toeplitz matrices: Advanced theory via simple matrix relationships, Linear Algebra and its Applications 270 (1998) 15–27.

[44] K. Gustafsson, M. Lundh, G. Söderlind, A PI stepsize control for the numerical solution of ordinary differential equations, BIT Numerical Mathematics 28 (2) (1988) 270–287.

[45] G. Söderlind, Automatic control and adaptive time-stepping, Numerical Algorithms 31 (1) (2002) 281–310.

[46] G. Söderlind, Digital filters in adaptive time-stepping, ACM Transactions on Mathematical Software 29 (1) (2003) 1–26.

[47] NVIDIA Corporation, NVIDIA A100 tensor core GPU architecture, Tech. rep., NVIDIA, whitepaper v1.0. Available at https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet.pdf (2020).

[48] P. Gray, S. K. Scott, Autocatalytic reactions in the isothermal, continuous stirred tank reactor: oscillations and instabilities in the system $A + 2B \to 3B$, $B \to C$, Chemical Engineering Science 39 (6) (1984) 1087–1097.

[49] J. E. Pearson, Complex patterns in a simple system, Science 261 (5118) (1993) 189–192.

[50] A. M. Turing, The chemical basis of morphogenesis, Philosophical Transactions of the Royal Society B 237 (641) (1952) 37–72.

[51] K. W. Morton, D. F. Mayers, Numerical Solution of Partial Differential Equations, 2nd Edition, Cambridge University Press, 2005.

[52] R. J. LeVeque, Finite Difference Methods for Ordinary and Partial Differential Equations, SIAM, 2007. doi:10.1137/1.9780898717839.