Anchored Accumulation Calculus for Incompressible Navier–Stokes: Exact Viscous Propagation, Fourth-Order Forcing, and Numerical Verification

Rich C. Young (Guru#1)
https://youtube.com/@GuruOne

20 September 2026

Abstract

We introduce and numerically validate an Anchored Accumulation Calculus (AAC) realization of the periodic, constant-density, incompressible Newtonian Navier–Stokes equations. The central construction is operator-exact in the linear viscous part: after Fourier transformation and Leray projection, each solenoidal mode obeys a scalar stiff decay equation forced by the projected nonlinearity. AAC applies the exact exponential propagator to that viscous operator and accumulates the nonlinear forcing through a cubic Lagrange history integrated exactly against the exponential kernel. The fourth-order member, AAC4, therefore does not approximate the stiff linear propagator by an explicit polynomial; instead it evaluates the corresponding Duhamel accumulation mode by mode.

The paper first derives AAC from the variation-of-constants formula and proves four structural properties: exactness for the isolated linear viscous mode, preservation of the divergence-free Fourier subspace, fourth-order local consistency and conditional global fourth-order convergence for the smooth semidiscrete problem, and reduction to classical Adams–Bashforth 4 as the viscosity tends to zero. A separate proposition establishes the nonlinear energy orthogonality used by the benchmark, while a linear stability analysis shows why the method removes the explicit viscous stability barrier that limits classical RK4.

The implementation uses a Fourier pseudospectral discretization on the periodic cube, a Leray projection, quadratic nonlinear evaluation through the vorticity identity, coordinatewise two-thirds dealiasing, a real-to-complex half-spectrum FFT path, and deterministic three-dimensional vortex-braid initial data. The numerical verification suite includes exact ABC decay, nonlinear Taylor–Green matching, temporal convergence, the AAC4–AB4 limit, nonlinear energy orthogonality, variable-step consistency, three-dimensional vortex stretching, resolution refinement, operator audits, a long-time stress calculation, hard-stress timestep refinement, cross-resolution/cross-method agreement, and resolution scaling.

In the final verification run, all 14 tests executed, with 21 PASS, 0 FAIL, and 1 INFO. The hard-stress self-convergence orders were 3.9692 and 3.9849. The main three-dimensional AAC4/RK4 final-state difference was 2.66×10^{-9}; the $N = 48$ cross-resolution value was 6.17×10^{-10}; and the long-time stress difference at $T = 1$ was 3.61×10^{-8}. The $N = 64$ refinement changed energy by 1.88×10^{-13} and enstrophy by 7.30×10^{-12}, while maximum vorticity remained the more resolution-sensitive diagnostic. The fixed-step $N = 64$ performance snapshot gave a 1.625× AAC4/RK4 speedup, while the long-time diagnostic calculation gave 3.202×. These results support AAC4 as an accurate and computationally efficient stiff realization of the tested periodic Navier–Stokes discretization. They do not constitute a proof of global regularity, a proof of finite-time blowup, or a resolution-independent statement about a physical singularity.

Keywords: Anchored Accumulation Calculus; Navier–Stokes; Fourier pseudospectral method; exponential propagation; exponential integrator; stiffness; Leray projection; Adams–Bashforth; vortex stretching; direct numerical simulation

1 Introduction

The three-dimensional incompressible Navier–Stokes equations combine a dissipative linear operator with a quadratic transport nonlinearity. In a periodic Fourier representation the viscous operator is diagonal, with mode k decaying at rate $\nu|k|^2$, while the nonlinear term couples modes through a convolution. This separation is analytically simple but numerically consequential: as the resolution grows, the largest viscous eigenvalue grows quadratically with the largest retained wave number, forcing a fully explicit method to reduce its timestep even when the resolved nonlinear dynamics would permit a larger step. The classical fourth-order Runge–Kutta method (RK4) is a reliable baseline for smooth nonstiff problems, but its scalar stability polynomial only contains a finite approximation to the viscous exponential. For the negative real axis the classical RK4 stability interval ends at approximately $z = -2.78529356340528$. Consequently, a Fourier mode satisfying $y_t = -\lambda y$ becomes numerically unstable under explicit RK4 when $\lambda\Delta t$ exceeds that value. Exponential time-differencing methods were developed precisely to address this class of stiff semidiscrete problems; the exponential-integrator literature provides the appropriate broader context for the present construction [8–10]. The contribution of this paper is not to assert that exponential treatment of a diagonal linear operator is conceptually unknown. Rather, it develops and validates a specific Anchored Accumulation Calculus realization: the viscous semigroup is taken exactly, while the nonlinear forcing is represented by an anchored history polynomial whose exponentially weighted accumulation is evaluated analytically for every Fourier mode. The resulting AAC4 method has one nonlinear evaluation per step after startup, a mode-local coefficient calculation, and an exact viscous propagator. The implementation is deliberately transparent and reproducible in pure Python/NumPy. The numerical study is designed as a falsification-oriented benchmark rather than as a collection of favorable examples. It includes exact solutions, asymptotic order tests, independent RK4 comparisons, a hard-stress three-dimensional vortex-stretching state, resolution refinement through N=64, a low-viscosity amplified long-time stress run to T=1, and a fixed-step scaling measurement. The benchmark code is supplied as the paper's reproducibility appendix; the exact source used for the reported run is separately archived as a Python file and identified by its SHA-256 hash. The paper intentionally separates numerical evidence from mathematical claims about Navier–Stokes regularity. The inverse-vorticity fits reported below are extrapolative diagnostics only. A finite numerical solution through the computed horizon is evidence about the computed trajectory, not a theorem about the corresponding continuum initial-value problem.

2 How AAC Calculus Works

AAC begins from a general semilinear evolution equation

$$u_t = Lu + F(u, t). \tag{1}$$

where L is the part that can be represented exactly over one time step and F is the remaining forcing. In a periodic incompressible Fourier representation of Navier–Stokes, L is simply the diagonal viscous operator. For one wave vector k and one solenoidal component in a fixed projected basis, write

$$\frac{d\hat{u}_k}{dt} = -\Lambda_k \hat{u}_k - N_k(t), \qquad \Lambda_k = \nu|k|^2. \tag{2}$$

where N_k is the Leray-projected transform of $(u \cdot \nabla)u$. The minus sign is kept explicit because the momentum equation contains $-P[(u \cdot \nabla)u]$. External forcing can be included by replacing

N_k with $N_k - F_k$. For a step from t_n to $t_{n+1} = t_n + h$, variation of constants gives the exact identity

$$\widehat{u}_k^{n+1} = e^{-\Lambda_k h}\widehat{u}_k^n - \int_0^h e^{-\Lambda_k(h-\tau)} N_k(t_n + \tau)\, d\tau. \tag{3}$$

AAC calls this an anchored accumulation because the current state $\widehat{u}_k^n$ is propagated from the anchor t_n by the exact semigroup, while the nonlinear history is accumulated against the same exact kernel over the interval. Nothing in this construction introduces a material-memory constitutive law: the underlying Newtonian PDE remains memoryless. The history is numerical state carried by the high-order realization of the Duhamel term. With normalized local coordinate $\xi = \tau/h$ and constant step size, the nodes are $\xi = 0, -1, -2, -3$. Let $L_j(\xi)$ be the corresponding Lagrange basis. Then

$$N_k(t_n + \tau) \approx \sum_{j=0}^{3} L_j(\xi) N_k^{n-j}. \tag{4}$$

The exact mode-wise weights are therefore

$$w_{j,k}(h) = h \int_0^1 e^{-q_k(1-\xi)} L_j(\xi)\, d\xi. \tag{5}$$

with $q_k = \Lambda_k h$. The AAC4 update is

$$\widehat{u}_k^{n+1} = \beta_k \widehat{u}_k^n - \sum_{j=0}^{3} w_{j,k} N_k^{n-j}. \tag{6}$$

$$\beta_k = e^{-q_k}. \tag{7}$$

This formula is the entire numerical mechanism in compressed form: exact exponential propagation plus exact integration of a cubic approximation to the nonlinear forcing. For implementation, write $L_j(\xi) = \sum_{r=0}^{3} c_{jr}\xi^r$ and define

$$I_r(q) = \int_0^1 e^{-q(1-\xi)} \xi^r\, d\xi. \tag{8}$$

Then $w_{j,k} = h \sum_r c_{jr} I_r(q_k)$. The moments can be evaluated stably from the auxiliary integrals

$$J_0(q) = \frac{1 - e^{-q}}{q}, \qquad J_m(q) = \frac{m J_{m-1}(q) - e^{-q}}{q}. \tag{9}$$

and the binomial relation

$$I_r(q) = \sum_{m=0}^{r} (-1)^m \binom{r}{m} J_m(q). \tag{10}$$

The code additionally uses small-q series expansions so that the nearly inviscid limit is not contaminated by cancellation in $1 - e^{-q}$. With variable step sizes, the same construction is retained but the interpolation nodes are rebuilt from the actual previous time locations. If the

current step is h_n and the previous step sizes are h_{n-1}, h_{n-2}, and h_{n-3}, the normalized nodes are

$$\xi_0 = 0, \qquad \xi_1 = -\frac{h_{n-1}}{h_n}, \qquad \xi_2 = -\frac{h_{n-1}+h_{n-2}}{h_n}, \qquad \xi_3 = -\frac{h_{n-1}+h_{n-2}+h_{n-3}}{h_n}. \tag{11}$$

with the obvious notation adjustment that the previous/current step ratios are supplied directly to the implementation. The cubic basis is rebuilt at the actual nodes, and the same exponential moments are integrated exactly. Thus variable stepping changes the interpolation geometry, not the underlying AAC principle. AAC1 and AAC2 are obtained by lowering the history degree. They are retained in the benchmark as controlled sub-order comparisons; AAC4 is the principal method.

3 Governing Equations and Fourier Formulation

Consider the constant-density incompressible Newtonian equations on the periodic cube $\Omega = [0, 2\pi)^3$,

$$\nabla\cdot u = 0, \qquad u_t + (u\cdot\nabla)u = -\frac{1}{\rho}\nabla p + \nu\Delta u + f. \tag{12}$$

The benchmark uses nondimensional variables and takes $\rho = 1$. The forcing is set to zero in the production tests. Applying the Leray projection P onto divergence-free vector fields removes the pressure gradient and gives

$$u_t = -P[(u\cdot\nabla)u] + \nu\Delta u. \tag{13}$$

For a Fourier coefficient $\hat{u}(k)$ with $k \neq 0$, the projection is

$$P_k = I - \frac{kk^{\mathsf{T}}}{|k|^2}. \tag{14}$$

The viscous part is diagonal:

$$\widehat{\nu\Delta u}_k = -\nu|k|^2\hat{u}_k. \tag{15}$$

The nonlinear identity

$$(u\cdot\nabla)u = (\nabla\times u)\times u + \frac{1}{2}\nabla|u|^2. \tag{16}$$

shows that after projection the nonlinear term can be evaluated as $P(\omega\times u)$, where $\omega = \nabla\times u$. The code uses this form because the pressure-like gradient term is discarded automatically by P. The spectral implementation uses collocation points and Fourier transforms. Quadratic aliasing is controlled by a coordinatewise two-thirds mask. This is the standard pseudospectral strategy associated with the classic transform formulation of periodic incompressible flow [2,3]. The final implementation stores the real field in physical space and the Fourier representation in NumPy's real-to-complex half-spectrum format, reducing storage and transform work without changing the mathematical Fourier truncation. The nonlinear evaluation therefore follows the sequence: inverse transform of $\hat{u}$ to u; spectral curl to obtain ω followed by inverse transform; pointwise construction of $\omega\times u$; forward transform; Leray projection; and dealiasing. The linear AAC step then consists only of mode-wise real scalar multiplications of the projected complex Fourier coefficients.

4 Structural Theorems and Proofs

The following results concern the semidiscrete Fourier system with exact arithmetic. They are statements about the time-discrete realization once the spatial Fourier truncation and nonlinear projection have been fixed. They do not assert global existence or regularity of the continuum three-dimensional Navier–Stokes equations.

Theorem 4.1 (Exact viscous propagation)**.** *Let $N_k(t) \equiv 0$ on one Fourier mode with $\Lambda_k = \nu|k|^2 \geq 0$. Then the AAC update satisfies*

$$\widehat{u}_k^{n+1} = e^{-\Lambda_k h}\widehat{u}_k^n \tag{17}$$

for every $h > 0$.

Proof. With $N_k \equiv 0$, the Duhamel term vanishes identically in the exact variation-of-constants formula. AAC uses the same exponential factor $\beta_k = e^{-\Lambda_k h}$, hence the numerical update equals the exact solution operator over the complete step. No truncation in h occurs in the linear propagator. □

Theorem 4.2 (Preservation of the divergence-free Fourier subspace)**.** *Assume $k \cdot \widehat{u}_k^n = 0$ for every retained mode. Suppose the nonlinear term is projected by P_k and the viscous propagation is scalar multiplication by $e^{-\nu|k|^2 h}$. Then*

$$k \cdot \widehat{u}_k^{n+1} = 0 \tag{18}$$

for every AAC step in exact arithmetic.

Proof. By definition, $k^{\mathsf{T}} P_k = 0$, so every projected nonlinear coefficient satisfies $k \cdot N_k^{n-j} = 0$. Any history combination of such vectors is also orthogonal to k. The viscous term is $\beta_k \widehat{u}_k^n$, hence

$$k \cdot (\beta_k \widehat{u}_k^n) = \beta_k (k \cdot \widehat{u}_k^n) = 0. \tag{19}$$

Their sum is therefore orthogonal to k. The zero mode is treated independently. In floating-point arithmetic the reported divergence is limited by transform roundoff and projection arithmetic. □

Theorem 4.3 (Adams–Bashforth 4 zero-viscosity limit)**.** *For constant h, the AAC4 weights satisfy*

$$(w_0, w_1, w_2, w_3) \longrightarrow \frac{h}{24}(55, -59, 37, -9) \qquad \text{as } \nu \to 0. \tag{20}$$

Consequently the AAC4 update becomes the classical Adams–Bashforth 4 formula for the nonlinear forcing.

Proof. As $\nu \to 0$, $q_k = \nu|k|^2 h \to 0$ and $\beta_k \to 1$. Moreover,

$$I_r(0) = \int_0^1 \xi^r \, d\xi = \frac{1}{r+1}. \tag{21}$$

Substitution of these moments into the cubic Lagrange integrals yields the standard Adams–Bashforth coefficients. Because the construction is modewise, the limit holds for every retained Fourier mode. □

Theorem 4.4 (Fourth-order temporal consistency)**.** *Consider the smooth semidiscrete Fourier system*

$$y'(t) = -\Lambda y(t) + F(t, y(t)), \qquad \Lambda \geq 0, \tag{22}$$

on a fixed time interval, and assume that $F(t, y(t))$ has four continuous time derivatives along the exact solution. AAC4 uses the exact exponential propagator for $-\Lambda$ and a cubic interpolation of F in the Duhamel integral. Then the one-step defect is $O(h^5)$. If the resulting history recurrence is stable under the fixed-ratio step sequence, then the global temporal error is $O(h^4)$ on the fixed interval.

Proof. The exact step is

$$y(t+h) = e^{-\Lambda h} y(t) + \int_0^h e^{-\Lambda(h-\tau)} F(t+\tau, y(t+\tau))\, d\tau. \tag{23}$$

Let $P_3(\tau)$ denote the cubic interpolation polynomial through the four history nodes. Standard interpolation theory gives

$$F(t+\tau, y(t+\tau)) - P_3(\tau) = \frac{F^{(4)}(\zeta_\tau)}{4!}\, \omega_4(\tau), \tag{24}$$

where ω_4 is the product of the four node differences. On a fixed-ratio multistep stencil, $|\omega_4(\tau)| \leq Ch^4$ for $0 \leq \tau \leq h$. Since $e^{-\Lambda(h-\tau)} \leq 1$ for $\Lambda \geq 0$,

$$\left| \int_0^h e^{-\Lambda(h-\tau)} [F - P_3]\, d\tau \right| \leq Ch^5, \tag{25}$$

Thus the exact semigroup removes the stiff linear truncation error and leaves an $O(h^5)$ local defect from cubic forcing interpolation. Under the stated bounded-stability hypothesis, the accumulated global temporal error is $O(h^4)$. □

The theorem is a semidiscrete temporal result. It does not imply fourth-order convergence of the full Navier–Stokes PDE unless the spatial approximation error is simultaneously controlled and the exact solution has the required smoothness.

Proposition 4.5 (Explicit RK4 viscous stability barrier)**.** *For $y' = -\lambda y$, $\lambda > 0$, classical RK4 advances*

$$y_{n+1} = R(-\lambda h) y_n, \qquad R(z) = 1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \frac{z^4}{24}. \tag{26}$$

On the negative real axis, $|R(-r)| \leq 1$ only through

$$r \approx 2.78529356340528. \tag{27}$$

Therefore RK4 requires $\lambda h \leq 2.78529356340528$ for nonamplifying viscous modes, whereas AAC uses $e^{-\lambda h}$ exactly and is contractive for every $h > 0$ on this isolated linear mode.

Proof. The RK4 stability function is the fourth-degree Taylor polynomial of e^z. Its negative-real stability interval terminates at the positive root of $|R(-r)| = 1$, yielding the stated value. AAC instead gives the exact factor e^{-r}, whose modulus is strictly less than one for every $r > 0$. □

This proposition concerns only the isolated linear viscous equation. It does not mean that the complete nonlinear AAC4 method is unconditionally stable: nonlinear advection, history interpolation, resolution, dealiasing, and the chosen step controller impose additional restrictions.

Proposition 4.6 (Nonlinear energy orthogonality)**.** *For a sufficiently smooth periodic divergence-free velocity field,*

$$\langle u, P[(u \cdot \nabla)u] \rangle = 0. \tag{28}$$

Proof. Since u is divergence-free, P is self-adjoint on periodic L^2 and $Pu = u$. Therefore

$$\begin{aligned} \langle u, P[(u \cdot \nabla)u] \rangle &= \langle u, (u \cdot \nabla)u \rangle && (29)\\ &= \frac{1}{2} \int_\Omega u \cdot \nabla |u|^2 \, dx && (30)\\ &= \frac{1}{2} \int_\Omega \nabla \cdot (u|u|^2) \, dx && (31)\\ &= 0, && (32) \end{aligned}$$

by periodic cancellation. Consequently,

$$\frac{dE}{dt} = -\nu \|\nabla u\|_{L^2}^2, \qquad E = \frac{1}{2}\|u\|_{L^2}^2, \tag{33}$$

for periodic divergence-free fields. In the dealiased Fourier implementation the corresponding discrete spectral inner product preserves this cancellation up to floating-point and transform roundoff. □

5 Spatial Discretization and Implementation

The computational domain is $\Omega = [0, 2\pi)^3$ with N^3 equispaced collocation points. Fourier differentiation is exact on the truncated basis. The code stores the retained wave-number arrays k_x, k_y, k_z and k^2, together with the Leray coefficients k_i/k^2 and the dealiasing mask. The coordinatewise two-thirds mask sets coefficients to zero whenever any component exceeds N/3 in magnitude. For quadratic products this is the classical Orszag-type dealiasing strategy [2,3]. The implementation does not claim radial isotropy of the filter; the mask follows the Cartesian tensor grid actually used by the FFT representation. The real-to-complex half-spectrum path is a performance optimization, not a change of discretization. Hermitian symmetry is handled by the real FFT transform, and spectral sums include the corresponding multiplicity weights for interior positive k_z modes. This is why the optimized v5 implementation can be compared directly against the earlier full-spectrum results. The nonlinear form $P(\omega \times u)$ is advantageous because it avoids calculating the pressure-like gradient contribution. The Leray projection is applied after transforming the pointwise cross product. Because every history term is already projected and dealiased, the AAC history accumulation itself requires no additional projection. The same invariant is used to avoid redundant projection work in the RK4 stages. The main computational cost is therefore Fourier transformation of nonlinear fields. After AAC4 startup, one nonlinear evaluation is required per step, whereas classical RK4 requires four. The exponential viscous update is a set of mode-local multiplications. The overall Navier–Stokes complexity is FFT-dominated, approximately $O(N^3 \log N)$ per nonlinear evaluation; the AAC history accumulation itself is $O(K)$ per spectral mode for $K = 4$ history states.

6 Benchmark Design

The final benchmark is deliberately organized as a complete numerical campaign. All 14 tests run on every invocation; no mode or skip switch can silently omit a diagnostic. The test numbering is fixed so that the output can serve as a reproducibility manifest. Test 1 isolates

viscous stiffness with the exact high-wavenumber shear mode $u = (0, \sin(kx), 0)$, $k = 8$, $\nu = 1$, $N = 24$, and $T = 0.5$. The continuum solution is a pure exponential decay. This experiment separates numerical stability from any nonlinear or physical blowup mechanism. Test 2 uses an ABC/Beltrami field, an exact nonlinear eigenfunction whose nonlinear contribution is a gradient removed by the Leray projection. The exact solution is exponential viscous decay. This is a stringent machine-precision operator test. Test 3 uses the Taylor–Green vortex at $N = 24$, $\nu = 0.01$, $T = 0.25$, and $\Delta t = 0.005$. A fine RK4 calculation at $\Delta t = 3.125 \times 10^{-4}$ supplies an independent reference for the coarse methods. Test 4 measures temporal convergence at $N = 16$, $T = 0.05$, and $\Delta t \in \{10^{-2}, 5 \times 10^{-3}, 2.5 \times 10^{-3}\}$ against RK4 at $\Delta t = 6.25 \times 10^{-4}$. Test 5 directly evaluates the AAC4 coefficient limit against the Adams–Bashforth 4 coefficients as ν becomes negligible. Test 6 evaluates the projected nonlinear energy work before time stepping. Test 7 exercises variable-step AAC4 with a deliberately nonuniform adaptive sequence and compares it against a much finer fixed-step AAC4 reference. Test 8 uses deterministic three-dimensional vortex-braid data at $N = 24$, $\nu = 5 \times 10^{-4}$, and $T = 0.4$. Initial vortex-stretching RMS and maximum are reported. Test 9 repeats the stress calculation at $N = 16$, $N = 32$, and $N = 64$. Energy, enstrophy, maximum vorticity, divergence, spectral tails, and inverse-vorticity fits are tracked. Test 10 is an operator audit of divergence, dealiasing, nonlinear energy work, and vortex stretching. The extended stress experiments comprise four additional calculations. Test 11 uses amplitude 1.5, $\nu = 2.5 \times 10^{-4}$, $N = 24$, and $T = 1.0$. Test 12 performs hard-stress AAC4 timestep refinement. Test 13 compares AAC4 and RK4 at $N = 24$ and $N = 48$. Test 14 performs a fixed eight-step scaling snapshot at $N = 16, 32, 64$.

7 Numerical Results

All values in this section are taken from the final verification run. The runtime environment was Python 3.12.1, NumPy 1.26.4, Emscripten-3.1.58 wasm32, and one CPU. Wall-clock values therefore characterize the demonstrated runtime rather than a hardware-independent performance constant. Speedups are most meaningful within the same run because both methods use the same FFT backend and memory hierarchy.

7.1 Exact and structural tests

All reported divergence residuals were far below 10^{-30}. The Taylor–Green test produced an AAC4/RK4 same-step L^2 difference of 7.307×10^{-11}. AAC4 was approximately 3.00× faster than coarse-step RK4 in this run. The fine RK4 reference required 801 steps, illustrating the cost of obtaining a high-accuracy independent reference with an explicit method. The zero-viscosity coefficient audit found a worst AAC4-to-AB4 coefficient error of 1.723×10^{-14}. The projected nonlinear energy-work residual was 2.305×10^{-20} after normalization.

7.2 Core validation results

Test	Quantity	Result	Interpretation
1	RK4 first failed factor	1.05× explicit limit	Explicit viscous instability; AAC4 remained stable through 2×
2	ABC AAC4 relative L2	1.727e-15	Machine-precision exact decay
3	AAC4 vs RK4, same Δt	7.307e-11	Strong nonlinear cross-method agreement
4	AAC4 observed order	3.728	Asymptotic trend toward fourth order
5	Worst AAC4–AB4 coefficient error	1.723e-14	Zero-viscosity identity
6	Normalized nonlinear work	2.305e-20	Discrete energy orthogonality
7	Variable-step L2 vs fine fixed AAC4	3.311e-7	Consistent variable-step realization

7.3 Temporal convergence

The coarse temporal convergence study measured AAC1 order 1.000, AAC2 order 2.000, AAC4 order 3.728, and RK4 order 4.002 on its last observable refinement pair. The AAC4 value is slightly below four because the experiment is entering the very small-error regime; a harder, independent self-convergence test was therefore included in Test 12. That test gave $p_{12} = 3.9692$ and $p_{24} = 3.9849$, providing the paper's principal numerical fourth-order evidence under the three-dimensional stress configuration.

7.4 Three-dimensional vortex stretching and refinement

The deterministic vortex-braid initial state has initial stretching maximum 3.751489 and stretching RMS 0.968840. At $N = 24$, $\nu = 5 \times 10^{-4}$, and $T = 0.4$, AAC4 and RK4 produced the same displayed energy and enstrophy, maximum vorticity 3.936815, BKM-like integral 1.389404, and spectral tails 1.542×10^{-6} and 2.288×10^{-7}. Their final-state relative L^2 difference was 2.660×10^{-9}. The refinement table shows extraordinary convergence of the integral quantities

while maximum vorticity remains the most demanding pointwise observable. From $N = 32$ to $N = 64$, energy changes by 1.88×10^{-13} and enstrophy by 7.30×10^{-12}, whereas maximum vorticity changes by about 0.502

7.5 Three-dimensional stress refinement

N	E	Ω	$\|\omega\|_\infty$	tail90

7.6 Long-time stress and singularity diagnostics

The long-time stress calculation was run at $N = 24$, $\nu = 2.5 \times 10^{-4}$, amplitude 1.5, and $T = 1.0$. Both AAC4 and RK4 remained finite and produced closely matching bulk quantities. AAC4 required 0.8235 s versus 2.6368 s for RK4, a 3.202× speedup; their final-state L^2 difference was 3.609×10^{-8}. The final spectral tail at 90The vorticity extrapolation fits gave $T_* \approx 1.9204$, 1.9248,

and 2.0226 for late-time fractions 0.50, 0.65, and 0.80, with R^2 values 0.99678, 0.99189, and 0.98607. The cumulative quantity reported as BKM is the numerical integral $\int_0^T \|\omega(t)\|_\infty \, dt$. The

classical Beale–Kato–Majda theorem is a rigorous continuation criterion for the three-dimensional

Euler equations; related continuation criteria exist for Navier–Stokes. Here the quantity is used as a monitoring diagnostic and not as a substitute for a continuum regularity proof [7,13].

7.7 Extended stress tests

Test	Quantity	Result	Interpretation

7.8 Stiffness isolation

The exact shear test gives the cleanest stability separation. For $N = 24$, $k = 8$, $\nu = 1$, $\Lambda = 64$, and $T = 0.5$, the RK4 negative-axis boundary is $\Delta t \approx 0.0435202$. At 1.05 times this step, RK4 is already classified as numerically unstable while AAC4 remains stable. At twice the RK4 boundary the reported trajectory maximum for RK4 reaches 19.27557, while the exact continuum amplitude and AAC4 remain bounded and decaying. Since the test equation is exactly solvable, the interpretation is unambiguous: the observed RK4 growth is a numerical stability failure of the time discretization, not a physical Navier–Stokes blowup.

8 Computational Performance

The optimized v5 implementation changes the performance balance in two ways. First, the Fourier representation is stored in real-to-complex half-spectrum form, reducing transformed data volume. Second, the nonlinear field, vorticity field, and diagnostic quantities are reused so that avoidable inverse transforms and projections are not repeated. The viscous exponential and AAC history weights are cached by timestep and wave number. The most important algorithmic cost difference is nonlinear-stage count. AAC4 needs one nonlinear history evaluation per production step after startup. RK4 requires four stage evaluations. Each stage includes inverse transforms, pointwise products, a forward transform, projection, and dealiasing. This gives a direct mechanism for the observed speedup; the paper does not attribute the performance difference to an unexplained low-level implementation artifact. The fixed eight-step scaling snapshot reported speedups of $1.467\times$ at $N = 16$, $1.516\times$ at $N = 32$, and $1.625\times$ at $N = 64$. The long-time diagnostic test gave $3.202\times$ because the method amortizes the same one-versus-four nonlinear-evaluation distinction across a longer trajectory and the diagnostic implementation reuses physical fields aggressively. These numbers should not be universalized. Hardware, FFT backend, threading, memory layout, compiler, transform planning, and the comparator all affect wall-clock speed. The scientifically portable claims are therefore the algorithmic stage-count reduction and the observed speedups under the explicitly reported environment.

9 Relation to Existing Exponential and Spectral Time Integrators

AAC4 belongs mathematically to the broad family of exponential time-differencing and exponential multistep ideas: an exact semigroup handles the linear stiff part and a polynomial history approximates the nonlinear forcing [8–10]. Classical ETD literature provides many alternative constructions, including Runge–Kutta and multistep variants, while integrating-factor formulations represent a closely related transformation. This paper does not claim that exact treatment of a linear parabolic operator is new in isolation. The specific AAC presentation emphasizes an operator-realization viewpoint. The exponential propagator is treated as a primitive exact update, and the forcing contribution is an anchored accumulation whose coefficients are

themselves exact mode-wise integrals. In the zero-viscosity limit this construction lands on Adams–Bashforth 4, making the transition between the stiff and inviscid limits transparent. The variable-step extension keeps the same operator identity while rebuilding the interpolation geometry at the actual time nodes. This distinction matters for scientific positioning. The appropriate novelty claim is therefore the complete AAC construction, its specific coefficient realization, the software implementation, and the demonstrated benchmark behavior—not the generic existence of exponential integrators. A future comparative study should include at least one established ETD or integrating-factor fourth-order method as an additional baseline. The present paper deliberately uses classical RK4 as its primary traditional baseline because the principal question is whether the exact viscous propagator eliminates the explicit viscous stiffness barrier in an otherwise conventional spectral Navier–Stokes computation.

10 Discussion

The benchmark supports five conclusions. First, the linear viscous operator is integrated exactly in the sense of Theorem 1. This is not a fitted approximation and not a step-size truncation of the exponential. Second, AAC4 behaves as a genuine fourth-order method for the nonlinear forcing in the tested smooth semidiscrete regime. The strongest evidence is the hard-stress self-convergence sequence 3.9692 and 3.9849, reinforced by the independent cross-method experiments. Third, the method removes the explicit RK4 stability boundary for the isolated viscous mode. This is a structural property of the algorithm and is confirmed directly by the exact-shear experiment. Fourth, the three-dimensional stress calculations demonstrate that the method is not being validated only on linear, two-dimensional, or non-stretching states. The vortex-braid field has substantial vortex stretching, and the AAC4/RK4 discrepancy remains small through N=48 and in the T=1 long-time stress run. Fifth, the implementation is materially faster than the tested explicit RK4 baseline. The fixed-step scaling result reaches $1.625\times$ at $N = 64$, while the long-time diagnostic run reaches $3.202\times$. At the same time, the benchmark identifies exactly where stronger evidence is still required. The maximum vorticity is substantially more resolution-sensitive than energy and enstrophy. The inverse-vorticity fits vary with fitting window and extrapolate beyond the computed interval. No finite-time singularity is numerically observed in the reported run, and no global regularity theorem follows from the computation. The paper therefore treats the AAC result as a numerical-analysis contribution: exact treatment of a stiff diagonal operator, high-order forcing realization, divergence preservation, strong cross-method agreement, and favorable CPU cost. Those claims are directly supported by the derivation and tests reported here.

11 Limitations and Reproducibility

The present study is periodic and uses Fourier collocation on a cubic domain. Wall-bounded flows, complex geometries, variable viscosity, turbulence models, and coupled multiphysics problems are not covered. The nonlinear baseline is classical explicit RK4; no direct head-to-head ETD4 or IMEX benchmark is included in this paper. Such comparisons are an important next step before making broad claims about the optimality of AAC4 among stiff integrators. All production tests are deterministic. The final benchmark run used Python 3.12.1, NumPy 1.26.4, Emscripten-3.1.58 wasm32, and one CPU. The full campaign reported 14/14 tests executed, 21 PASS, 0 FAIL, and 1 INFO. The source file contains 2242 lines in the archived execution version

and has SHA-256 `45872e5373461922dde9edab3c912f66559ac8e1f1300cd421c6601d623b7b95`.

The benchmark source is provided in the appendices as the machine-readable Python implementation `aac_ns_benchmark_publication.py`. The published source listing is functionally identical to the executed benchmark, with only non-numerical presentation labels and comments normalized for publication; the exact execution source is archived separately as `aac_ns_benchmark.py`. The LaTeX source supplied with this paper includes the publication-normalized listing directly as the code appendix. For the numerical tables, the browser wall-clock time is reported because the final campaign was executed in Pyodide. Performance should be replicated on a native Python/NumPy environment before using the timings for procurement, cluster sizing, or production solver planning. The source hash and deterministic parameter set are the more portable reproducibility identifiers.

12 Conclusions

This paper has developed and validated a fourth-order Anchored Accumulation Calculus realization of periodic incompressible Navier–Stokes. AAC4 starts from the exact variation-of-constants identity, propagates each Fourier mode through the Newtonian viscous exponential exactly, and accumulates the nonlinear forcing through an analytically integrated cubic history polynomial. The resulting realization preserves the divergence-free subspace, reduces to Adams–Bashforth 4 in the zero-viscosity limit, and has fourth-order temporal consistency for the smooth semidiscrete problem. The benchmark confirms those properties numerically and adds independent evidence from exact decay, nonlinear matching, three-dimensional vortex stretching, resolution refinement, stiffness isolation, long-time stress, and performance scaling. The most important computational observation is that the viscous stiffness barrier of explicit RK4 disappears from the isolated linear mode because AAC4 uses the exact exponential propagator. The most important numerical observation is that this structural modification does not destroy high-order agreement with RK4 on the smooth nonlinear flows tested. The most important practical observation is that the optimized implementation obtains meaningful CPU speedups while retaining the same pseudospectral spatial discretization. No claim about the unsolved global regularity problem is made.

The long-time stress run remains a numerical stress experiment; inverse-vorticity fits remain extrapolations; and the resolution sensitivity of maximum vorticity is explicitly reported. The appropriate next stage is broader comparative testing against established ETD/IMEX methods, native multi-core benchmarks, and higher-resolution production studies. Within those limits, the results establish AAC4 as a well-defined, reproducible, and computationally competitive realization for stiff periodic Navier–Stokes dynamics.

13 References

1 R. Temam, Navier–Stokes Equations: Theory and Numerical Analysis, AMS Chelsea Publishing, Providence, RI, 2001. Originally published by North-Holland in 1977.

2 S. A. Orszag, “Numerical Simulation of Incompressible Flows Within Simple Boundaries. I. Galerkin (Spectral) Representations,” Studies in Applied Mathematics, 50(4), 293–327, 1971. DOI: 10.1002/sapm1971504293.

3 G. S. Patterson Jr. and S. A. Orszag, “Spectral Calculations of Isotropic Turbulence: Efficient Removal of Aliasing Interactions,” Physics of Fluids, 14(11), 2538–2541, 1971. DOI: 10.1063/1.1693365.

4 C. Canuto, M. Y. Hussaini, A. Quarteroni, and T. A. Zang, Spectral Methods: Fundamentals in Single Domains, Springer, Berlin, 2006. DOI: 10.1007/978-3-540-30726-6.

5 G. I. Taylor and A. E. Green, "Mechanism of the Production of Small Eddies from Large Ones," Proceedings of the Royal Society of London A, 158(895), 499–521, 1937. DOI: 10.1098/rspa.1937.0036.

6 R. S. Rogallo, Numerical Experiments in Homogeneous Turbulence, NASA Technical Memorandum 81315, NASA Ames Research Center, 1981.

7 J. T. Beale, T. Kato, and A. Majda, "Remarks on the Breakdown of Smooth Solutions for the 3-D Euler Equations," Communications in Mathematical Physics, 94(1), 61–66, 1984. DOI: 10.1007/BF01212349.

8 S. M. Cox and P. C. Matthews, "Exponential Time Differencing for Stiff Systems," Journal of Computational Physics, 176, 430–455, 2002. DOI: 10.1006/jcph.2002.6995.

9 A.-K. Kassam and L. N. Trefethen, "Fourth-Order Time-Stepping for Stiff PDEs," SIAM Journal on Scientific Computing, 26(4), 1214–1233, 2005. DOI: 10.1137/S1064827502410633.

10 M. Hochbruck and A. Ostermann, "Exponential Integrators," Acta Numerica, 19, 209–286, 2010. DOI: 10.1017/S0962492910000048.

11 J. C. Butcher, Numerical Methods for Ordinary Differential Equations, 2nd ed., Wiley, 2008. DOI: 10.1002/9780470753767.

12 E. Hairer, S. P. Nørsett, and G. Wanner, Solving Ordinary Differential Equations I: Nonstiff Problems, 2nd revised ed., Springer, Berlin, 1993.

13 Y. Giga, "Solutions for Semilinear Parabolic Equations in Lp and Regularity of Weak Solutions of the Navier–Stokes System," Journal of Differential Equations, 62(2), 186–212, 1986. DOI: 10.1016/0022-0396(86)90096-3.

14 A. J. Chorin, "Numerical Solution of the Navier–Stokes Equations," Mathematics of Computation, 22(104), 745–762, 1968. DOI: 10.1090/S0025-5718-1968-0242392-2.

15 L. N. Trefethen, Spectral Methods in MATLAB, SIAM, Philadelphia, 2000. DOI: 10.1137/1.9780898719598.

16 C. Canuto, M. Y. Hussaini, A. Quarteroni, and T. A. Zang, Spectral Methods in Fluid Dynamics, Springer, Berlin, 1988; later revised editions and extensions are listed by Springer.

17 A. J. Chorin, "A Numerical Method for Solving Incompressible Viscous Flow Problems," Journal of Computational Physics, 2(1), 12–26, 1967. DOI: 10.1016/0021-9991(67)90037-X.

18 R. C. Young, AAC–Navier–Stokes paper benchmark source, 'aac_ns_benchmark.py', version used for the present manuscript, 2026. SHA-256: `45872e5373461922dde9edab3c912f66559ac8e1f13` Companion source archive supplied with this paper.

A Coefficient Derivation and Implementation Notes

This appendix records the coefficient construction used by the reference implementation.

For constant h, let q=$\nu|k|^2$h and define

$$M_r(q) = hI_r(q), \tag{34}$$

$$I_r(q) = \int_0^1 e^{-q(1-\xi)}\xi^r\, d\xi. \tag{35}$$

The four cubic weights in the implementation are

$$w_0 = \frac{M_3 + 6M_2 + 11M_1 + 6M_0}{6}, \tag{36}$$

$$w_1 = -\frac{M_3 + 5M_2 + 6M_1}{2}, \tag{37}$$

$$w_2 = \frac{M_3 + 4M_2 + 3M_1}{2}, \tag{38}$$

$$w_3 = -\frac{M_3 + 3M_2 + 2M_1}{6}. \tag{39}$$

At $q = 0$ these become $55h/24$, $-59h/24$, $37h/24$, and $-9h/24$. For small q the implementation evaluates the moments through a convergent local series, avoiding cancellation in $1 - e^{-q}$. For larger q it uses the recurrence

$$J_0 = \frac{1 - e^{-q}}{q}, \qquad J_m = \frac{mJ_{m-1} - e^{-q}}{q}, \tag{40}$$

followed by

$$I_r = \sum_m (-1)^m \binom{r}{m} J_m. \tag{41}$$

For variable steps the code constructs the four actual normalized history nodes and computes the cubic Lagrange coefficients numerically from those nodes before evaluating the same exponential moments. The resulting update is therefore the exact exponential accumulation of a cubic history interpolant on the actual timestep geometry.

B Complete Python Benchmark Source

The complete Python benchmark source is supplied as the companion file `aac_ns_benchmark_publication.py` and is incorporated into the LaTeX manuscript through a source-code listing in the code appendix. The publication listing is functionally identical to the executed benchmark, with only non-numerical presentation labels and comments normalized for publication. The exact execution source is separately archived as `aac_ns_benchmark.py`.

45872e5373461922dde9edab3c912f66559ac8e1f1300cd421c6601d623b7b95

The final browser run was executed with Python 3.12.1, NumPy 1.26.4, Emscripten-3.1.58 wasm32, one CPU, deterministic initial data, and the complete 14-test campaign. The output reported 14/14 tests attempted, 21 PASS, 0 FAIL, and 1 INFO.

For environments without '___file___', the runtime cannot self-compute the source hash. In such environments the hash above is an externally computed checksum of the exact source submitted for the benchmark.

```
#!/usr/bin/env python3
"""--
AACNAVIERSTOKES NUMERICAL VERIFICATION v5
================================================

Purpose
-------
A reproducible CPU-only pseudospectral study of the constant-density,
```

```
incompressible, Newtonian -NavierStokes equations on a periodic cube:

    div u = 0
    u_t + (u·grad)u = -(1/rho) grad p + nu Laplacian(u) + f

The continuum PDE is unchanged. The comparison is between:

TRADITIONAL REALIZATION
-----------------------
Explicit classical RK4 applied to the projected spectral ODE.

REAL-AAC REALIZATION
--------------------
Exact exponential propagation of each linear viscous Fourier mode, with AAC4
cubic-history Duhamel forcing. The constant-step formula tends to AB4 as
nu -> 0; the variable-step version rebuilds the cubic Lagrange history using
actual time nodes.

IMPLEMENTATION NOTES
-----------
1. Real-to-complex half-spectrum FFT/IFFT path cuts spectral storage and transform work while retaining
     the same pseudospectral discretization.
2. In-place Leray projection paths reduce temporary allocation and redundant projection/mask work.
3. Cached Fourier multipliers and precomputed viscous multipliers remove repeated arithmetic.
4. A genuinely 3-D vortex-stretching benchmark is retained using a deterministic
   divergence-free vector-potential construction.
5. The stress suite reports initial/final vortex-stretching strength.
6. A stronger matched stress test can use the vortex-braid state.
7. All verification tests execute on every invocation; browser performance is obtained
   from a lower-allocation FFT/hot-loop implementation.
8. Diagnostics distinguish numerical instability from any physical singularity
   candidate; no numerical run is interpreted as a regularity proof.
9. An extended stress campaign can increase initial amplitude independently of the
   baseline case, directly stressing vortex stretching and effective Reynolds number.

IMPORTANT PHYSICAL DISTINCTION
------------------------------
AAC does not introduce fictitious constitutive memory into a Newtonian liquid.
It realizes the existing linear viscous memoryless operator exactly in time.
Actual material memory would require a different constitutive PDE.

Dependencies
------------
Python 3.10+ and NumPy only.
No plotting. No SciPy. No external data.
"""

from __future__ import annotations

import argparse
import gc
import hashlib
import math
import os
import platform
import sys
import time
from dataclasses import dataclass
from typing import Optional, Sequence, Tuple

import numpy as np

Array = np.ndarray


# ============================================================================
# REPORTING
# ============================================================================

@dataclass
class Tolerances:
    divergence_rms: float = 1.0e-12
    exact_l2: float = 1.0e-11
```

```
    cross_method_l2: float = 1.0e-7
    nonlinear_work_abs: float = 1.0e-10
    adaptive_cross_method: float = 2.0e-5
    tail_safe: float = 0.15
    finite_threshold: float = 1.0e12


TOL = Tolerances()


class Report:
    def __init__(self) -> None:
        self.passed = 0
        self.failed = 0
        self.info = 0

    def pass_(self, name: str, detail: str = "") -> None:
        self.passed += 1
        print(f"PASS {name}: {detail}")

    def fail(self, name: str, detail: str = "") -> None:
        self.failed += 1
        print(f"FAIL {name}: {detail}")

    def info_(self, name: str, detail: str = "") -> None:
        self.info += 1
        print(f"INFO {name}: {detail}")

    def summary(self) -> None:
        print("\n" + "=" * 108)
        print(
            f"SELF-AUDIT SUMMARY: PASS={self.passed} "
            f"FAIL={self.failed} INFO={self.info}"
        )
        print("=" * 108)


REPORT = Report()


# ============================================================================
# GRID / FOURIER DATA
# ============================================================================

@dataclass
class Grid:
    N: int
    L: float = 2.0 * math.pi

    def __post_init__(self) -> None:
        if self.N < 8:
            raise ValueError("N must be >= 8")
        if self.L <= 0.0:
            raise ValueError("L must be positive")

        self.dx = self.L / self.N
        k1 = np.fft.fftfreq(self.N, d=self.dx) * 2.0 * math.pi
        kz1 = np.fft.rfftfreq(self.N, d=self.dx) * 2.0 * math.pi

        self.kx, self.ky, self.kz = np.meshgrid(
            k1, k1, kz1, indexing="ij"
        )
        self.k2 = (
            self.kx * self.kx
            + self.ky * self.ky
            + self.kz * self.kz
        )
        self.k = np.sqrt(self.k2)

        kcut = (self.N // 3) * (2.0 * math.pi / self.L)
        self.kcut = float(kcut)

        self.dealias = (
```

```
            (np.abs(self.kx) <= kcut + 1.0e-12)
            & (np.abs(self.ky) <= kcut + 1.0e-12)
            & (np.abs(self.kz) <= kcut + 1.0e-12)
        )
        self.nonzero = self.k2 > 0.0
        self.k2_safe = np.where(self.nonzero, self.k2, 1.0)

        # Cached complex derivative multipliers.
        self.ikx = 1j * self.kx
        self.iky = 1j * self.ky
        self.ikz = 1j * self.kz

        # Spectral multiplicity for NumPy's real-to-complex half-spectrum:
        # kz=0 and kz=Nyquist are single modes; interior kz modes represent
        # both +/-kz partners. The dealias cutoff lies below Nyquist.
        self.rfft_weight = np.ones(self.kz.shape[-1], dtype=np.float64)
        if self.N > 2:
            self.rfft_weight[1:-1] = 2.0

        # Precompute projection coefficients used by the hot nonlinear path.
        invk2 = 1.0 / self.k2_safe
        self.pkx = self.kx * invk2
        self.pky = self.ky * invk2
        self.pkz = self.kz * invk2

        self.tail80_mask = self.k >= 0.80 * self.kcut
        self.tail90_mask = self.k >= 0.90 * self.kcut

        # Precompute viscous scalar multiplier; this avoids forming nu*k2 in
        # every RK4 stage and makes the exact AAC propagator cheaper to apply.
        self.k2max = float(np.max(self.k2[self.dealias]))


def fft_vec(u: Array) -> Array:
    """Vectorized real-to-complex 3-component FFT over spatial axes."""
    return np.fft.rfftn(u, axes=(1, 2, 3))


def ifft_vec(uh: Array) -> Array:
    """Vectorized half-spectrum inverse FFT back to real physical fields."""
    n = uh.shape[1]
    return np.fft.irfftn(uh, s=(n, n, n), axes=(1, 2, 3))


def project_inplace(uh: Array, g: Grid) -> Array:
    """In-place Leray projection using cached k/k^2 factors."""
    dot = (
        g.kx * uh[0]
        + g.ky * uh[1]
        + g.kz * uh[2]
    )
    uh[0] -= g.pkx * dot
    uh[1] -= g.pky * dot
    uh[2] -= g.pkz * dot
    return uh


def project_copy(uh: Array, g: Grid) -> Array:
    return project_inplace(uh.copy(), g)


def curl_hat(uh: Array, g: Grid) -> Array:
    """Spectral curl using cached derivative multipliers."""
    out = np.empty_like(uh)
    out[0] = g.iky * uh[2] - g.ikz * uh[1]
    out[1] = g.ikz * uh[0] - g.ikx * uh[2]
    out[2] = g.ikx * uh[1] - g.iky * uh[0]
    return out


def nonlinear_projected_details(
    uh: Array,
    g: Grid,
```

```
) -> Tuple[Array, Array, Array]:
    """Return projected nonlinearity plus the physical u and omega fields.

    The physical fields are intentionally returned because the diagnostic
    integrator can reuse them for CFL, max-velocity, and max-vorticity checks,
    avoiding several additional inverse FFTs per timestep.
    """
    u = ifft_vec(uh)
    omega = ifft_vec(curl_hat(uh, g))

    cross = np.empty_like(u)
    cross[0] = omega[1] * u[2] - omega[2] * u[1]
    cross[1] = omega[2] * u[0] - omega[0] * u[2]
    cross[2] = omega[0] * u[1] - omega[1] * u[0]

    nh = fft_vec(cross)
    project_inplace(nh, g)
    nh *= g.dealias
    return nh, u, omega


def nonlinear_projected(uh: Array, g: Grid) -> Array:
    nh, _, _ = nonlinear_projected_details(uh, g)
    return nh


def nonlinear_projected_with_omega(
    uh: Array,
    g: Grid,
) -> Tuple[Array, Array]:
    nh, _, omega = nonlinear_projected_details(uh, g)
    return nh, omega


# ============================================================================
# INITIAL CONDITIONS
# ============================================================================

def initial_abc(
    g: Grid,
    A: float = 1.0,
    B: float = 1.0,
    C: float = 1.0,
) -> Array:
    x = np.arange(g.N) * g.dx
    X, Y, Z = np.meshgrid(x, x, x, indexing="ij")
    u = np.empty((3, g.N, g.N, g.N), dtype=np.float64)
    u[0] = A * np.sin(Z) + C * np.cos(Y)
    u[1] = B * np.sin(X) + A * np.cos(Z)
    u[2] = C * np.sin(Y) + B * np.cos(X)
    return u


def initial_high_k_shear(
    g: Grid,
    k_mode: Optional[int] = None,
) -> Array:
    if k_mode is None:
        k_mode = g.N // 3
    if not (1 <= k_mode <= g.N // 3):
        raise ValueError("k_mode must satisfy 1 <= k_mode <= N//3")

    x = np.arange(g.N) * g.dx
    X = x[:, None, None]
    u = np.zeros((3, g.N, g.N, g.N), dtype=np.float64)
    u[1] = np.sin(k_mode * X)
    return u


def initial_taylor_green(
    g: Grid,
    amplitude: float = 1.0,
) -> Array:
```

```
    x = np.arange(g.N) * g.dx
    X, Y, Z = np.meshgrid(x, x, x, indexing="ij")
    u = np.zeros((3, g.N, g.N, g.N), dtype=np.float64)
    u[0] = amplitude * np.sin(X) * np.cos(Y) * np.cos(Z)
    u[1] = -amplitude * np.cos(X) * np.sin(Y) * np.cos(Z)
    return u


def initial_vortex_pair(
    g: Grid,
    amplitude: float = 2.0,
    width: float = 0.35,
) -> Array:
    x = np.arange(g.N) * g.dx
    X, Y, Z = np.meshgrid(x, x, x, indexing="ij")

    def wrap(a: Array) -> Array:
        return ((a + math.pi) % (2.0 * math.pi)) - math.pi

    d1 = wrap(X - 1.0) ** 2 + wrap(Y) ** 2
    d2 = wrap(X + 1.0) ** 2 + wrap(Y) ** 2
    g1 = np.exp(-d1 / (width * width))
    g2 = np.exp(-d2 / (width * width))
    wz = amplitude * (g1 - g2) * (1.0 + 0.15 * np.cos(Z))

    omega = np.zeros((3, g.N, g.N, g.N), dtype=np.float64)
    omega[2] = wz

    omegah = project_copy(fft_vec(omega), g)
    uh = curl_hat(omegah, g)
    uh[0] /= g.k2_safe
    uh[1] /= g.k2_safe
    uh[2] /= g.k2_safe
    uh[:, ~g.nonzero] = 0.0
    project_inplace(uh, g)
    uh *= g.dealias
    return ifft_vec(uh)


def normalize_rms(u: Array, target: float = 1.0) -> Array:
    speed2 = np.sum(u * u, axis=0)
    rms_speed = math.sqrt(float(np.mean(speed2)))
    if rms_speed <= 0.0:
        return u
    return u * (target / rms_speed)


def initial_vortex_braid(
    g: Grid,
    rms_target: float = 1.0,
) -> Array:
    """Deterministic 3-D divergence-free vortex-stretching state.

    The field is generated as u=curl(A), with several non-collinear modes in
    the vector potential. This gives an intrinsically 3-D vortex geometry and
    avoids imposing the planar symmetry of a simple shear test.
    """
    x = np.arange(g.N) * g.dx
    X, Y, Z = np.meshgrid(x, x, x, indexing="ij")

    A = np.empty((3, g.N, g.N, g.N), dtype=np.float64)

    A[0] = (
        np.sin(Y)
        + 0.35 * np.sin(2.0 * X + Z)
        + 0.20 * np.cos(Y + 2.0 * Z)
        + 0.12 * np.sin(2.0 * Y - Z)
    )
    A[1] = (
        np.sin(Z)
        + 0.32 * np.sin(2.0 * Y + X)
        + 0.24 * np.cos(Z + 2.0 * X)
        + 0.10 * np.sin(2.0 * Z - X)
```

```
    )
    A[2] = (
        np.sin(X)
        + 0.38 * np.sin(2.0 * Z + Y)
        + 0.18 * np.cos(X + 2.0 * Y)
        + 0.11 * np.sin(2.0 * X - Y)
    )

    Ah = fft_vec(A)
    uh = curl_hat(Ah, g)
    uh[:, ~g.nonzero] = 0.0
    project_inplace(uh, g)
    uh *= g.dealias
    u = ifft_vec(uh)
    return normalize_rms(u, rms_target)


def make_initial_state(
    g: Grid,
    kind: str,
) -> Array:
    if kind == "taylor-green":
        return initial_taylor_green(g)
    if kind == "vortex-braid":
        return initial_vortex_braid(g)
    if kind == "vortex-pair":
        return initial_vortex_pair(g)
    raise ValueError(f"unknown initial state: {kind}")


def make_initial_state_hat(
    g: Grid,
    kind: str,
    amplitude: float = 1.0,
) -> Array:
    """Deterministic normalized divergence-free initial state with amplitude control."""
    if amplitude <= 0.0:
        raise ValueError("amplitude must be positive")
    return normalize_state_hat(
        make_initial_state(g, kind),
        g,
    ) * float(amplitude)


# ============================================================================
# AAC VISCOUS COEFFICIENTS
# ============================================================================

def _small_q_moment_series(
    q: Array,
    power: int,
    terms: int = 18,
) -> Array:
    out = np.zeros_like(q, dtype=np.float64)
    term = np.ones_like(q, dtype=np.float64) * (
        math.factorial(power) / math.factorial(power + 1)
    )
    out += term

    for m in range(1, terms):
        term *= -q / (power + m + 1.0)
        out += term
    return out


def viscous_moments(
    q: Array,
    max_power: int = 3,
) -> Tuple[Array, ...]:
    """I_r(q) =_0^1 exp(-q(1-x)) x^r dx, stable for small q."""
    q = np.asarray(q, dtype=np.float64)
    out = [np.empty_like(q, dtype=np.float64) for _ in range(max_power + 1)]
```

```
    small = np.abs(q) < 1.0e-4
    if np.any(small):
        qs = q[small]
        for r in range(max_power + 1):
            out[r][small] = _small_q_moment_series(qs, r)

    large = ~small
    if np.any(large):
        ql = q[large]
        J = [np.empty_like(ql, dtype=np.float64) for _ in range(max_power + 1)]
        J[0] = -np.expm1(-ql) / ql
        e = np.exp(-ql)
        for k in range(1, max_power + 1):
            J[k] = (k * J[k - 1] - e) / ql
        for r in range(max_power + 1):
            val = np.zeros_like(ql)
            for k in range(r + 1):
                val += ((-1.0) ** k) * math.comb(r, k) * J[k]
            out[r][large] = val

    return tuple(out)


def lagrange_polynomial_coefficients(nodes: Sequence[float]) -> Array:
    n = len(nodes)
    C = np.zeros((n, n), dtype=np.float64)
    for j in range(n):
        poly = np.array([1.0], dtype=np.float64)
        denom = 1.0
        xj = float(nodes[j])
        for m in range(n):
            if m == j:
                continue
            xm = float(nodes[m])
            poly = np.polynomial.polynomial.polymul(
                poly,
                np.array([-xm, 1.0], dtype=np.float64),
            )
            denom *= xj - xm
        C[j, : len(poly)] = poly / denom
    return C


def aac4_weights_constant(
    g: Grid,
    nu: float,
    dt: float,
) -> Tuple[Array, Array, Array, Array]:
    a = nu * g.k2
    q = a * dt
    I0, I1, I2, I3 = viscous_moments(q, 3)
    M0, M1, M2, M3 = dt * I0, dt * I1, dt * I2, dt * I3

    w0 = (M3 + 6.0 * M2 + 11.0 * M1 + 6.0 * M0) / 6.0
    w1 = -(M3 + 5.0 * M2 + 6.0 * M1) / 2.0
    w2 = (M3 + 4.0 * M2 + 3.0 * M1) / 2.0
    w3 = -(M3 + 3.0 * M2 + 2.0 * M1) / 6.0
    return w0, w1, w2, w3


def aac4_weights_variable(
    g: Grid,
    nu: float,
    h: float,
    previous_steps: Sequence[float],
) -> Tuple[Array, Array, Array, Array]:
    if len(previous_steps) < 3:
        raise ValueError("three previous step sizes are required")
    if h <= 0.0:
        raise ValueError("h must be positive")

    h1, h2, h3 = map(float, previous_steps[:3])
    xnodes = (
```

```
        0.0,
        -h1 / h,
        -(h1 + h2) / h,
        -(h1 + h2 + h3) / h,
    )

    C = lagrange_polynomial_coefficients(xnodes)
    q = nu * g.k2 * h
    I0, I1, I2, I3 = viscous_moments(q, 3)
    I = (I0, I1, I2, I3)

    weights = []
    for j in range(4):
        w = np.zeros_like(q, dtype=np.float64)
        for r in range(4):
            w += C[j, r] * I[r]
        weights.append(h * w)
    return tuple(weights) # type: ignore[return-value]


def aac4_zero_viscosity_reference(
    dt: float,
) -> Tuple[float, float, float, float]:
    return (
        55.0 * dt / 24.0,
        -59.0 * dt / 24.0,
        37.0 * dt / 24.0,
        -9.0 * dt / 24.0,
    )


class AACOperators:
    """Cached constant-step exponential propagation coefficients."""

    def __init__(self, g: Grid, nu: float, dt: float) -> None:
        if nu < 0.0:
            raise ValueError("nu must be >= 0")
        if dt <= 0.0:
            raise ValueError("dt must be > 0")

        self.g = g
        self.nu = float(nu)
        self.dt = float(dt)

        a = self.nu * g.k2
        self.a = a
        q = a * self.dt
        self.beta = np.exp(-q)

        self.alpha = np.empty_like(a, dtype=np.float64)
        nz = g.nonzero & (a > 0.0)
        self.alpha[nz] = -np.expm1(-q[nz]) / a[nz]
        self.alpha[~nz] = self.dt

        self.w0, self.w1, self.w2, self.w3 = aac4_weights_constant(
            g, nu, dt
        )


# RK4 uses this lightweight cached linear multiplier so nu*k^2 is formed only
# once per run rather than once per stage.
class LinearOperators:
    def __init__(self, g: Grid, nu: float) -> None:
        self.g = g
        self.nu = float(nu)
        self.a = nu * g.k2


# ============================================================================
# TIME STEPPERS
# ============================================================================

def rhs_ns(
```

```
    uh: Array,
    g: Grid,
    nu: float,
    viscous_a: Optional[Array] = None,
) -> Array:
    """Projected spectral RHS with cached linear multiplier when supplied.

    The state entering this routine is divergence-free. The nonlinear term is
    explicitly Leray-projected/dealiased; the viscous term preserves that
    subspace algebraically, so an additional projection at every RHS stage is
    redundant and is deliberately omitted from the hot path.
    """
    nh = nonlinear_projected(uh, g)
    a = viscous_a if viscous_a is not None else (nu * g.k2)
    return -nh - a * uh


def step_rk4(
    uh: Array,
    g: Grid,
    nu: float,
    dt: float,
    viscous_a: Optional[Array] = None,
) -> Array:
    a = viscous_a if viscous_a is not None else (nu * g.k2)
    k1 = rhs_ns(uh, g, nu, a)
    k2 = rhs_ns(uh + 0.5 * dt * k1, g, nu, a)
    k3 = rhs_ns(uh + 0.5 * dt * k2, g, nu, a)
    k4 = rhs_ns(uh + dt * k3, g, nu, a)
    out = uh + (dt / 6.0) * (k1 + 2.0 * k2 + 2.0 * k3 + k4)
    project_inplace(out, g)
    out *= g.dealias
    return out


def step_aac1(uh: Array, nh: Array, op: AACOperators) -> Array:
    # Both uh and nh lie in the projected/dealiased subspace. The scalar
    # exponential/History multipliers preserve that subspace, so another
    # projection here is redundant.
    return op.beta * uh - op.alpha * nh


def step_aac4(
    uh: Array,
    nh0: Array,
    history: Tuple[Array, Array, Array],
    weights: Tuple[Array, Array, Array, Array],
    beta: Array,
    g: Grid,
) -> Array:
    hn1, hn2, hn3 = history
    w0, w1, w2, w3 = weights
    return beta * uh - w0 * nh0 - w1 * hn1 - w2 * hn2 - w3 * hn3


# ============================================================================
# DIAGNOSTICS
# ============================================================================

def _weighted_spectral_sum(ah: Array, g: Grid) -> float:
    return float(
        np.sum(np.abs(ah) ** 2 * g.rfft_weight[None, None, :])
    )


def kinetic_energy(uh: Array, g: Grid) -> float:
    return 0.5 * _weighted_spectral_sum(uh, g) / (g.N ** 3) * g.dx ** 3


def enstrophy(uh: Array, g: Grid) -> float:
    omegah = curl_hat(uh, g)
    return 0.5 * _weighted_spectral_sum(omegah, g) / (g.N ** 3) * g.dx ** 3
```

```
def vorticity_physical(uh: Array, g: Grid) -> Array:
    return ifft_vec(curl_hat(uh, g))


def max_vorticity(uh: Array, g: Grid) -> float:
    omega = vorticity_physical(uh, g)
    return float(np.max(np.sqrt(np.sum(omega * omega, axis=0))))


def rms_divergence(uh: Array, g: Grid) -> float:
    divh = g.ikx * uh[0] + g.iky * uh[1] + g.ikz * uh[2]
    return float(np.sqrt(np.mean(np.fft.irfftn(divh, s=(g.N, g.N, g.N), axes=(0, 1, 2)) ** 2)))


def spectral_tail_ratio(
    uh: Array,
    g: Grid,
    frac: float = 0.80,
) -> float:
    if frac == 0.80:
        mask = g.tail80_mask
    elif frac == 0.90:
        mask = g.tail90_mask
    else:
        mask = g.k >= frac * g.kcut
    w = g.rfft_weight[None, None, :]
    total = float(np.sum(np.abs(uh) ** 2 * w))
    if total <= 0.0:
        return 0.0
    return float(np.sum(np.abs(uh) ** 2 * w * mask[None, :, :]) / total)


def spectral_tail_ratio_90(uh: Array, g: Grid) -> float:
    return spectral_tail_ratio(uh, g, 0.90)


def nonlinear_energy_work(uh: Array, g: Grid) -> float:
    nh = nonlinear_projected(uh, g)
    return float(
        np.real(np.vdot(uh, nh)) * g.dx ** 3 / (g.N ** 3)
    )


def cfl_number(uh: Array, g: Grid, dt: float) -> float:
    u = ifft_vec(uh)
    speed = np.sqrt(np.sum(u * u, axis=0))
    umax = max(float(np.max(speed)), 1.0e-14)
    return float(umax * dt / g.dx)


def cfl_dt(uh: Array, g: Grid, cfl: float = 0.30) -> float:
    u = ifft_vec(uh)
    speed = np.sqrt(np.sum(u * u, axis=0))
    umax = max(float(np.max(speed)), 1.0e-14)
    return cfl * g.dx / umax


def diffusive_number(g: Grid, nu: float, dt: float) -> float:
    return nu * g.k2max * dt


def rk4_stability_dt(g: Grid, nu: float, safety: float = 0.80) -> float:
    if nu <= 0.0:
        return math.inf
    return safety * 2.75 / (nu * g.k2max)


def rel_l2(a: Array, b: Array, g: Optional[Grid] = None) -> float:
    d = np.abs(a - b) ** 2
    bb = np.abs(b) ** 2
    if g is None:
        num = np.sqrt(np.sum(d))
```

```
        den = max(np.sqrt(np.sum(bb)), 1.0e-300)
    else:
        w = g.rfft_weight[None, None, :]
        num = np.sqrt(np.sum(d * w))
        den = max(np.sqrt(np.sum(bb * w)), 1.0e-300)
    return float(num / den)


def convergence_order(e1: float, e2: float, h1: float, h2: float) -> float:
    if e1 <= 0.0 or e2 <= 0.0 or h1 == h2:
        return float("nan")
    return math.log(e1 / e2) / math.log(h1 / h2)


def blowup_intercept(
    times: Array,
    omega: Array,
    tails: Array,
) -> Optional[float]:
    if len(times) < 8:
        return None

    lo = max(0, int(0.65 * len(times)))
    t = np.asarray(times[lo:])
    w = np.asarray(omega[lo:])
    tail = np.asarray(tails[lo:])

    good = (
        (tail < TOL.tail_safe)
        & np.isfinite(w)
        & (w > 0.0)
    )
    if np.count_nonzero(good) < 5:
        return None

    t = t[good]
    invw = 1.0 / w[good]
    A = np.vstack([np.ones_like(t), t]).T
    coef, *_ = np.linalg.lstsq(A, invw, rcond=None)
    a, b = float(coef[0]), float(coef[1])
    if b >= -1.0e-14:
        return None
    return -a / b


def stretching_norm(
    uh: Array,
    g: Grid,
) -> Tuple[float, float]:
    """Return absolute and RMS vortex-stretching norms."""
    omega = vorticity_physical(uh, g)

    grads = np.empty((3, 3, g.N, g.N, g.N), dtype=np.float64)
    for j, ik in enumerate((g.ikx, g.iky, g.ikz)):
        for i in range(3):
            grads[i, j] = np.fft.irfftn(ik * uh[i], s=(g.N, g.N, g.N), axes=(0, 1, 2))

    # S_i = sum_j omega_j * d_j u_i
    stretch = np.einsum("jxyz,jixyz->ixyz", omega, grads, optimize=True)
    point = np.sqrt(np.sum(stretch * stretch, axis=0))
    rms = float(np.sqrt(np.mean(point * point)))
    return float(np.max(point)), rms


@dataclass
class BlowupFit:
    fraction: float
    t_star: Optional[float]
    slope: float
    intercept: float
    r2: float
    n_points: int
```

```
def inverse_vorticity_fit(
    times: Array,
    omega: Array,
    tails: Array,
    fraction: float,
) -> BlowupFit:
    """Fit 1/||omega||_inf against time on a selectable late-time window."""
    if len(times) < 8:
        return BlowupFit(fraction, None, float("nan"), float("nan"), float("nan"), 0)

    lo = max(0, int(fraction * len(times)))
    t = np.asarray(times[lo:])
    w = np.asarray(omega[lo:])
    tail = np.asarray(tails[lo:])

    good = (
        (tail < TOL.tail_safe)
        & np.isfinite(t)
        & np.isfinite(w)
        & (w > 0.0)
    )

    if np.count_nonzero(good) < 5:
        return BlowupFit(fraction, None, float("nan"), float("nan"), float("nan"), int(np.count_nonzero(
            good)))

    t = t[good]
    y = 1.0 / w[good]
    A = np.vstack([np.ones_like(t), t]).T
    coef, *_ = np.linalg.lstsq(A, y, rcond=None)
    a, b = float(coef[0]), float(coef[1])
    yhat = A @ coef
    ss_res = float(np.sum((y - yhat) ** 2))
    ss_tot = float(np.sum((y - np.mean(y)) ** 2))
    r2 = 1.0 if ss_tot == 0.0 else 1.0 - ss_res / ss_tot
    tstar = None if b >= -1.0e-14 else -a / b
    return BlowupFit(fraction, tstar, b, a, r2, len(t))


def blowup_dashboard(
    times: Array,
    omega: Array,
    tails: Array,
) -> Tuple[BlowupFit, ...]:
    """Return several late-time inverse-vorticity fits for robustness checks."""
    return tuple(
        inverse_vorticity_fit(times, omega, tails, f)
        for f in (0.50, 0.65, 0.80)
    )


# ============================================================================
# INITIAL-STATE HELPER
# ============================================================================

def make_initial_state(g: Grid, kind: str) -> Array:
    if kind == "taylor-green":
        return initial_taylor_green(g)
    if kind == "vortex-braid":
        return initial_vortex_braid(g)
    if kind == "vortex-pair":
        return initial_vortex_pair(g)
    raise ValueError(f"unknown initial state: {kind}")


def normalize_state_hat(u: Array, g: Grid) -> Array:
    uh = fft_vec(u)
    project_inplace(uh, g)
    uh *= g.dealias
    return uh
```

```
# ============================================================================
# METADATA / STARTUP
# ============================================================================

def source_sha256() -> str:
    override = os.environ.get("AAC_SOURCE_SHA256", "").strip()
    if override:
        return f"{override} (external override)"
    source_path = globals().get("__file__")
    if not source_path:
        return "unavailable (runtime does not define __file__)"
    try:
        with open(source_path, "rb") as f:
            return hashlib.sha256(f.read()).hexdigest()
    except (OSError, TypeError, ValueError):
        return "unavailable (source file not readable)"


def print_metadata() -> None:
    print("\n=== REPRODUCIBILITY METADATA ===")
    print(f"Python : {sys.version.split()[0]}")
    print(f"NumPy : {np.__version__}")
    print(f"Platform : {platform.platform()}")
    print(f"Processor : {platform.processor()}")
    print(f"CPU count : {os.cpu_count()}")
    print(f"Executable : {sys.executable}")
    print(f"Source SHA256 : {source_sha256()}")
    print(
        "Source-hash note : hash the exact submitted .py externally "
        "when the browser runtime has no __file__"
    )
    print("NumPy RNG seed : deterministic / no stochastic ICs")
    print(
        f"Tolerances : div={TOL.divergence_rms:.1e}, "
        f"exact={TOL.exact_l2:.1e}, cross={TOL.cross_method_l2:.1e}, "
        f"tail={TOL.tail_safe:.2f}"
    )
    print("FFT path : vectorized 3-component real-to-complex half-spectrum transforms")
    print("AAC4 history : constant-step + actual-node variable-step cubic")


# ============================================================================
# INTEGRATION
# ============================================================================

def startup_rk4(uh: Array, g: Grid, nu: float, h: float) -> Array:
    if h <= 0.0:
        raise ValueError("h must be positive")
    if nu <= 0.0:
        nsub = 1
    else:
        qmax = diffusive_number(g, nu, h)
        nsub = max(1, int(math.ceil(qmax / 1.25)))
    hs = h / nsub
    for _ in range(nsub):
        uh = step_rk4(uh, g, nu, hs)
    return uh


def choose_step(
    uh: Array,
    g: Grid,
    nu: float,
    dt_max: float,
    adaptive: bool,
    cfl_target: float,
    method: str,
) -> float:
    h = dt_max
    if adaptive:
        h = min(h, cfl_dt(uh, g, cfl_target))
        if method == "RK4":
            h = min(h, rk4_stability_dt(g, nu))
```

```
    return h


def evolve_fast(
    method: str,
    uh0: Array,
    g: Grid,
    nu: float,
    T: float,
    dt: float,
) -> Tuple[Array, int, float]:
    start = time.perf_counter()
    uh = uh0.copy()
    t = 0.0
    steps = 0
    hist: list[Array] = []
    prev_steps: list[float] = []

    linear = LinearOperators(g, nu) if method == "RK4" else None
    op = AACOperators(g, nu, dt) if method in ("AAC1", "AAC2", "AAC4") else None
    half = AACOperators(g, nu, 0.5 * dt) if method == "AAC2" else None
    op_cache = {float(dt): op} if op is not None else {}
    half_cache = {float(0.5 * dt): half} if half is not None else {}

    while t < T - 1.0e-15:
        h = min(dt, T - t)
        exact_const = abs(h - dt) <= 1.0e-15

        if exact_const:
            op_h = op
            half_h = half
        else:
            hkey = float(h)
            op_h = op_cache.get(hkey)
            if op_h is None and method in ("AAC1", "AAC2", "AAC4"):
                op_h = AACOperators(g, nu, h)
                op_cache[hkey] = op_h
            half_h = None
            if method == "AAC2":
                hk = float(0.5 * h)
                half_h = half_cache.get(hk)
                if half_h is None:
                    half_h = AACOperators(g, nu, 0.5 * h)
                    half_cache[hk] = half_h

        if method == "RK4":
            uh = step_rk4(uh, g, nu, h, linear.a if linear is not None else None)

        elif method == "AAC1":
            nh0 = nonlinear_projected(uh, g)
            uh = step_aac1(uh, nh0, op_h) # type: ignore[arg-type]

        elif method == "AAC2":
            nh0 = nonlinear_projected(uh, g)
            u_mid = half_h.beta * uh - half_h.alpha * nh0 # type: ignore[union-attr]
            nhm = nonlinear_projected(u_mid, g)
            uh = step_aac1(uh, nhm, op_h) # type: ignore[arg-type]

        elif method == "AAC4":
            nh0 = nonlinear_projected(uh, g)
            if len(hist) >= 3:
                if len(prev_steps) >= 3 and not exact_const:
                    weights = aac4_weights_variable(g, nu, h, prev_steps)
                else:
                    assert op_h is not None
                    weights = (op_h.w0, op_h.w1, op_h.w2, op_h.w3)
                uh = step_aac4(
                    uh,
                    nh0,
                    (hist[-1], hist[-2], hist[-3]),
                    weights,
                    op_h.beta, # type: ignore[union-attr]
                    g,
```

```
            )
        else:
            uh = startup_rk4(uh, g, nu, h)
        hist.append(nh0)
        hist = hist[-4:]

    else:
        raise ValueError(f"unknown method {method}")

    t += h
    steps += 1
    prev_steps.insert(0, h)
    prev_steps = prev_steps[:3]

return uh, steps, time.perf_counter() - start


@dataclass
class DiagnosticRun:
    method: str
    elapsed: float
    steps: int
    E: float
    Omega: float
    w_inf: float
    div_rms: float
    tail80: float
    tail90: float
    bkm: float
    max_cfl: float
    max_diffusive_number: float
    min_step: float
    max_step: float
    stretch_max_initial: float
    stretch_rms_initial: float
    stretch_max_final: float
    stretch_rms_final: float
    times: Array
    omega_hist: Array
    tail_hist: Array
    energy_hist: Array
    uh_final: Array
    stopped_early: bool


def integrate_diagnostic(
    method: str,
    uh0: Array,
    g: Grid,
    nu: float,
    T: float,
    dt: float,
    adaptive: bool = False,
    cfl_target: float = 0.30,
    record_every: int = 5,
    abort_amplitude: float = TOL.finite_threshold,
) -> DiagnosticRun:
    """Diagnostic integrator with hot-loop FFT reuse.

    One nonlinear evaluation already produces u and omega. Those same physical
    fields are reused for CFL, max-velocity, and max-vorticity diagnostics, so
    the diagnostic loop adds no extra inverse FFTs per timestep.
    """
    start = time.perf_counter()
    uh = uh0.copy()
    t = 0.0
    steps = 0
    stopped_early = False

    hist: list[Array] = []
    prev_steps: list[float] = []
    linear_a = nu * g.k2
    op_cache: dict[float, AACOperators] = {}
```

```
# Initial diagnostics are obtained from the first nonlinear evaluation.
nh0, u_cur, omega_cur = nonlinear_projected_details(uh, g)
speed2 = np.sum(u_cur * u_cur, axis=0)
umax_cur = max(float(np.sqrt(np.max(speed2))), 1.0e-14)
w_cur = float(np.sqrt(np.max(np.sum(omega_cur * omega_cur, axis=0))))

E0 = kinetic_energy(uh, g)
O0 = enstrophy(uh, g)
s0_max, s0_rms = stretching_norm(uh, g)

times = [0.0]
Es = [E0]
Os = [O0]
Ws = [w_cur]
Ts = [spectral_tail_ratio(uh, g)]

bkm = 0.0
w_prev = w_cur
t_prev = 0.0
max_cfl_seen = 0.0
max_q_seen = 0.0
min_step_seen = math.inf
max_step_seen = 0.0

while t < T - 1.0e-15:
    # The already-computed u/omega/nh0 belong to the current state.
    h = min(dt, T - t)
    if adaptive:
        h = min(h, cfl_target * g.dx / umax_cur)
        if method == "RK4" and nu > 0.0:
            h = min(h, 0.80 * 2.75 / (nu * g.k2max))

    if h < 1.0e-14:
        stopped_early = True
        break

    cfl_here = umax_cur * h / g.dx
    max_cfl_seen = max(max_cfl_seen, cfl_here)
    max_q_seen = max(max_q_seen, diffusive_number(g, nu, h))
    min_step_seen = min(min_step_seen, h)
    max_step_seen = max(max_step_seen, h)

    if method == "RK4":
        k1 = -nh0 - linear_a * uh
        k2 = rhs_ns(
            uh + 0.5 * h * k1, g, nu, linear_a
        )
        k3 = rhs_ns(
            uh + 0.5 * h * k2, g, nu, linear_a
        )
        k4 = rhs_ns(
            uh + h * k3, g, nu, linear_a
        )
        uh_new = uh + (h / 6.0) * (k1 + 2.0 * k2 + 2.0 * k3 + k4)
        project_inplace(uh_new, g)
        uh_new *= g.dealias

    elif method == "AAC1":
        op = op_cache.get(float(h))
        if op is None:
            op = AACOperators(g, nu, h)
            op_cache[float(h)] = op
        uh_new = step_aac1(uh, nh0, op)

    elif method == "AAC2":
        op = op_cache.get(float(h))
        if op is None:
            op = AACOperators(g, nu, h)
            op_cache[float(h)] = op
        half = op_cache.get(float(0.5 * h))
        if half is None:
            half = AACOperators(g, nu, 0.5 * h)
```

```
            op_cache[float(0.5 * h)] = half
        u_mid = half.beta * uh - half.alpha * nh0
        nhm = nonlinear_projected(u_mid, g)
        uh_new = step_aac1(uh, nhm, op)

    elif method == "AAC4":
        op = op_cache.get(float(h))
        if op is None:
            op = AACOperators(g, nu, h)
            op_cache[float(h)] = op
        if len(hist) >= 3:
            if len(prev_steps) >= 3 and any(abs(prev_steps[j] - h) > 1.0e-15 for j in range(3)):
                weights = aac4_weights_variable(g, nu, h, prev_steps)
            else:
                weights = (op.w0, op.w1, op.w2, op.w3)
            uh_new = step_aac4(
                uh,
                nh0,
                (hist[-1], hist[-2], hist[-3]),
                weights,
                op.beta,
                g,
            )
        else:
            uh_new = startup_rk4(uh, g, nu, h)
        hist.append(nh0)
        hist = hist[-4:]

    else:
        raise ValueError(f"unknown method {method}")

    t += h
    steps += 1
    prev_steps.insert(0, h)
    prev_steps = prev_steps[:3]

    if not np.all(np.isfinite(uh_new)):
        uh = uh_new
        stopped_early = True
        break

    # Advance state. Physical diagnostics for the new state are evaluated
    # once, at the top of the next iteration, rather than through separate
    # cfl/max-u/max-vorticity FFT calls.
    uh = uh_new

    # For the next state, evaluate one nonlinear operator and reuse its
    # physical u/omega fields for all diagnostics.
    nh0, u_cur, omega_cur = nonlinear_projected_details(uh, g)
    speed2 = np.sum(u_cur * u_cur, axis=0)
    umax_cur = max(float(np.sqrt(np.max(speed2))), 1.0e-14)
    w_cur = float(np.sqrt(np.max(np.sum(omega_cur * omega_cur, axis=0))))

    # BKM integral uses the exact state endpoints even though diagnostics
    # are evaluated in a deferred, one-FFT-shared manner.
    bkm += 0.5 * (w_prev + w_cur) * (t - t_prev)
    w_prev = w_cur
    t_prev = t

    if umax_cur > abort_amplitude or not np.isfinite(umax_cur) or not np.isfinite(w_cur):
        stopped_early = True
        break

    if steps % record_every == 0 or t >= T - 1.0e-15:
        times.append(t)
        Es.append(kinetic_energy(uh, g))
        Os.append(enstrophy(uh, g))
        Ws.append(w_cur)
        Ts.append(spectral_tail_ratio(uh, g))

# Ensure a final state diagnostic is available even if the loop exited at T.
if len(times) == 1 or abs(times[-1] - t) > 1.0e-15:
    times.append(t)
```

```
        Es.append(kinetic_energy(uh, g))
        Os.append(enstrophy(uh, g))
        Ws.append(w_cur)
        Ts.append(spectral_tail_ratio(uh, g))

    sf_max, sf_rms = stretching_norm(uh, g)

    return DiagnosticRun(
        method=method,
        elapsed=time.perf_counter() - start,
        steps=steps,
        E=Es[-1],
        Omega=Os[-1],
        w_inf=Ws[-1],
        div_rms=rms_divergence(uh, g),
        tail80=Ts[-1],
        tail90=spectral_tail_ratio_90(uh, g),
        bkm=bkm,
        max_cfl=max_cfl_seen,
        max_diffusive_number=max_q_seen,
        min_step=min_step_seen if np.isfinite(min_step_seen) else 0.0,
        max_step=max_step_seen,
        stretch_max_initial=s0_max,
        stretch_rms_initial=s0_rms,
        stretch_max_final=sf_max,
        stretch_rms_final=sf_rms,
        times=np.asarray(times),
        omega_hist=np.asarray(Ws),
        tail_hist=np.asarray(Ts),
        energy_hist=np.asarray(Es),
        uh_final=uh,
        stopped_early=stopped_early,
    )


# ============================================================================
# TESTS
# ============================================================================

def test_exact_shear_instability(
    N: int,
    nu: float,
    T: float,
    factors: Tuple[float, ...],
) -> None:
    print("\n=== TEST 1: EXACT SHEAR -RK4 VS REAL-AAC VISCOUS STIFFNESS ===")
    print("Continuum solution is exact exponential decay; growth is numerical.")

    g = Grid(N)
    k_mode = N // 3
    uh0 = normalize_state_hat(initial_high_k_shear(g, k_mode), g)
    lam = nu * (k_mode ** 2)
    rk_limit = 2.78529356340528 / lam
    exact_amp = math.exp(-lam * T)

    print(
        f"N={N}, nu={nu}, k={k_mode}, lambda={lam:.8e}, "
        f"RK4 boundary={rk_limit:.8e}"
    )
    print(
        "factor dt exact_amp(T) RK4 status AAC4 status "
        "RK4 max|u| AAC4 max|u|"
    )

    first_rk_blow = None
    first_aac_blow = None

    for factor in factors:
        dt = factor * rk_limit

        uh_rk = uh0.copy()
        rk_status = "STABLE"
        rk_max = 1.0
```

```
        t = 0.0
        while t < T - 1.0e-15:
            h = min(dt, T - t)
            uh_rk = step_rk4(uh_rk, g, nu, h)
            t += h
            m = float(np.max(np.abs(ifft_vec(uh_rk))))
            rk_max = max(rk_max, m)
            if not np.all(np.isfinite(uh_rk)) or m > 1.01:
                rk_status = "BLOWUP"
                break

        uh_aac = uh0.copy()
        hist: list[Array] = []
        prev_steps: list[float] = []
        aac_status = "STABLE"
        aac_max = 1.0
        t = 0.0
        while t < T - 1.0e-15:
            h = min(dt, T - t)
            op = AACOperators(g, nu, h)
            nh0 = nonlinear_projected(uh_aac, g)
            if len(hist) >= 3:
                weights = aac4_weights_variable(g, nu, h, prev_steps)
                uh_aac = step_aac4(
                    uh_aac, nh0,
                    (hist[-1], hist[-2], hist[-3]),
                    weights, op.beta, g,
                )
            else:
                uh_aac = startup_rk4(uh_aac, g, nu, h)
            hist.append(nh0)
            hist = hist[-4:]
            prev_steps.insert(0, h)
            prev_steps = prev_steps[:3]
            t += h
            m = float(np.max(np.abs(ifft_vec(uh_aac))))
            aac_max = max(aac_max, m)
            if not np.all(np.isfinite(uh_aac)) or m > 1.01:
                aac_status = "BLOWUP"
                break

        print(
            f"{factor:6.2f} {dt:12.5e} {exact_amp:14.6e} "
            f"{rk_status:>10s} {aac_status:>10s} "
            f"{rk_max:13.6e} {aac_max:13.6e}"
        )

        if rk_status == "BLOWUP" and first_rk_blow is None:
            first_rk_blow = factor
        if aac_status == "BLOWUP" and first_aac_blow is None:
            first_aac_blow = factor

    if first_rk_blow is not None and first_aac_blow is None:
        REPORT.pass_(
            "TRADITIONAL-VS-AAC-VISCOUS-STIFFNESS",
            f"RK4 first failed near {first_rk_blow:.2f}x explicit limit; AAC4 remained stable",
        )
    else:
        REPORT.fail(
            "TRADITIONAL-VS-AAC-VISCOUS-STIFFNESS",
            "expected explicit-stiffness separation not observed",
        )


def test_abc(N: int, nu: float, T: float, dt: float) -> None:
    print("\n=== TEST 2: ABC / BELTRAMI EXACT DECAY ===")
    g = Grid(N)
    uh0 = normalize_state_hat(initial_abc(g), g)
    exact = math.exp(-nu * T) * uh0
    print("method rel_L2_error runtime_s steps div_RMS")

    for method in ("AAC1", "AAC2", "AAC4", "RK4"):
        uh, steps, elapsed = evolve_fast(method, uh0, g, nu, T, dt)
```

```
        err = rel_l2(uh, exact, g)
        div = rms_divergence(uh, g)
        print(f"{method:>5s} {err:15.8e} {elapsed:9.5f} {steps:5d} {div:9.2e}")
        if err < TOL.exact_l2 and div < TOL.divergence_rms:
            REPORT.pass_(f"ABC-{method}", f"L2={err:.3e}, div={div:.3e}")
        else:
            REPORT.fail(f"ABC-{method}", f"L2={err:.3e}, div={div:.3e}")


def test_taylor_green(N: int, nu: float, T: float, dt: float) -> None:
    print("\n=== TEST 3: -TAYLORGREEN NONLINEAR MATCH ===")
    g = Grid(N)
    uh0 = normalize_state_hat(initial_taylor_green(g), g)
    ref_dt = dt / 16.0
    uh_ref, ref_steps, ref_time = evolve_fast("RK4", uh0, g, nu, T, ref_dt)

    runtimes = {}
    errors = {}
    states = {}
    steps_map = {}
    for method in ("AAC1", "AAC2", "AAC4", "RK4"):
        uh, steps, elapsed = evolve_fast(method, uh0, g, nu, T, dt)
        runtimes[method] = elapsed
        errors[method] = rel_l2(uh, uh_ref, g)
        states[method] = uh
        steps_map[method] = steps

    rk_time = runtimes["RK4"]
    print("method rel_L2_vs_ref runtime_s steps speedup")
    for method in ("AAC1", "AAC2", "AAC4", "RK4"):
        print(
            f"{method:>5s} {errors[method]:16.8e} {runtimes[method]:9.5f} "
            f"{steps_map[method]:5d} {rk_time / runtimes[method]:7.3f}x"
        )

    cross = rel_l2(states["AAC4"], states["RK4"], g)
    print(f"Reference RK4: dt={ref_dt:.6e}, steps={ref_steps}, runtime={ref_time:.5f}s")
    print(f"AAC4 vs RK4 same-dt L2 difference = {cross:.8e}")

    if cross < 10.0 * TOL.cross_method_l2:
        REPORT.pass_("TAYLOR-GREEN-CROSS-METHOD", f"L2={cross:.3e}")
    else:
        REPORT.info_("TAYLOR-GREEN-CROSS-METHOD", f"L2={cross:.3e}")


def test_temporal_convergence(
    N: int,
    nu: float,
    T: float,
    dt: float,
    levels: int,
    ref_divisor: int,
) -> None:
    print("\n=== TEST 4: TEMPORAL CONVERGENCE ===")
    g = Grid(N)
    uh0 = normalize_state_hat(initial_taylor_green(g), g)
    ref_dt = dt / ref_divisor
    uh_ref, _, _ = evolve_fast("RK4", uh0, g, nu, T, ref_dt)
    dts = [dt / (2 ** j) for j in range(levels)]
    methods = ("AAC1", "AAC2", "AAC4", "RK4")
    errors = {m: [] for m in methods}

    print(f"N={N}, T={T}, base_dt={dt}, reference_dt={ref_dt}")
    print("method dt rel_L2_error observed_p")

    for h in dts:
        for method in methods:
            uh, _, _ = evolve_fast(method, uh0, g, nu, T, h)
            err = rel_l2(uh, uh_ref, g)
            errors[method].append(err)
            if len(errors[method]) == 1:
                p_text = " ---"
            else:
```

```
            p = convergence_order(errors[method][-2], errors[method][-1], dts[len(errors[method]) - 2], h
                )
            p_text = f"{p:8.3f}"
        print(f"{method:>5s} {h:9.3e} {err:15.8e} {p_text}")

    targets = {
        "AAC1": (0.65, 1.35),
        "AAC2": (1.60, 2.40),
        "AAC4": (3.20, 4.80),
        "RK4": (3.20, 4.80),
    }
    for method in methods:
        pvals = []
        for j in range(1, len(errors[method])):
            e1, e2 = errors[method][j - 1], errors[method][j]
            if e1 > 1.0e-13 and e2 > 1.0e-13:
                pvals.append(convergence_order(e1, e2, dts[j - 1], dts[j]))
        if not pvals:
            REPORT.info_(f"ORDER-{method}", "reference/roundoff floor")
            continue
        p_last = pvals[-1]
        lo, hi = targets[method]
        if lo <= p_last <= hi:
            REPORT.pass_(f"ORDER-{method}", f"last observed p={p_last:.3f}")
        else:
            REPORT.info_(f"ORDER-{method}", f"last observed p={p_last:.3f}; inspect floor/stiffness")


def test_ab4_limit() -> None:
    print("\n=== TEST 5: AAC4 -> AB4 ZERO-VISCOSITY LIMIT ===")
    g = Grid(16)
    dt = 0.01
    nu = 1.0e-12
    weights = aac4_weights_constant(g, nu, dt)
    ref = aac4_zero_viscosity_reference(dt)
    errs = [float(np.max(np.abs(weights[j] - ref[j]))) for j in range(4)]
    print("weight max_abs_error_to_AB4")
    for j, e in enumerate(errs):
        print(f"w{j} {e:.8e}")
    worst = max(errs)
    if worst < 1.0e-9:
        REPORT.pass_("AAC4-AB4-LIMIT", f"worst coefficient error={worst:.3e}")
    else:
        REPORT.fail("AAC4-AB4-LIMIT", f"worst coefficient error={worst:.3e}")


def test_nonlinear_energy_identity(N: int) -> None:
    print("\n=== TEST 6: PROJECTED NONLINEAR ENERGY ORTHOGONALITY ===")
    g = Grid(N)
    uh = normalize_state_hat(initial_taylor_green(g), g)
    work = nonlinear_energy_work(uh, g)
    E = kinetic_energy(uh, g)
    ratio = abs(work) / max(E, 1.0)
    print(f"<u,P[(u·grad)u]> = {work:.8e}")
    print(f"normalized residual = {ratio:.8e}")
    if ratio < TOL.nonlinear_work_abs:
        REPORT.pass_("NONLINEAR-ENERGY-ORTHOGONALITY", f"normalized work residual={ratio:.3e}")
    else:
        REPORT.fail("NONLINEAR-ENERGY-ORTHOGONALITY", f"normalized work residual={ratio:.3e}")


def test_adaptive_aac4(N: int, nu: float, T: float, dt: float) -> None:
    print("\n=== TEST 7: VARIABLE-STEP AAC4 CONSISTENCY ===")
    g = Grid(N)
    uh0 = normalize_state_hat(initial_vortex_braid(g), g)
    adaptive = integrate_diagnostic(
        "AAC4", uh0, g, nu, T, dt,
        adaptive=True, cfl_target=0.15, record_every=2,
    )
    uh_fixed, fixed_steps, fixed_time = evolve_fast(
        "AAC4", uh0, g, nu, T, dt / 16.0
    )
    err = rel_l2(adaptive.uh_final, uh_fixed, g)
```

```
    ratio = adaptive.max_step / max(adaptive.min_step, 1.0e-300)
    print(
        f"adaptive: steps={adaptive.steps} runtime={adaptive.elapsed:.5f}s "
        f"max_CFL={adaptive.max_cfl:.4f} max_q={adaptive.max_diffusive_number:.4f} "
        f"h=[{adaptive.min_step:.3e},{adaptive.max_step:.3e}] ratio={ratio:.3f}"
    )
    print(
        f"fixed reference: steps={fixed_steps} runtime={fixed_time:.5f}s "
        f"dt={dt/16.0:.6e}"
    )
    print(f"variable-step AAC4 vs fine fixed AAC4 L2 = {err:.8e}")
    variable_ok = ratio > 1.05
    if err < TOL.adaptive_cross_method and variable_ok:
        REPORT.pass_("VARIABLE-STEP-AAC4", f"L2={err:.3e}, step-ratio={ratio:.3f}")
    elif err < TOL.adaptive_cross_method:
        REPORT.info_("VARIABLE-STEP-AAC4", f"L2={err:.3e}, but step ratio={ratio:.3f} is nearly constant")
    else:
        REPORT.info_("VARIABLE-STEP-AAC4", f"L2={err:.3e}; refine adaptive controller if needed")


def run_stress_pair(
    g: Grid,
    uh0: Array,
    nu: float,
    T: float,
    dt: float,
    cfl: float,
    record_every: int,
) -> Tuple[DiagnosticRun, DiagnosticRun]:
    runs = {}
    for method in ("AAC4", "RK4"):
        runs[method] = integrate_diagnostic(
            method, uh0, g, nu, T, dt,
            adaptive=True,
            cfl_target=cfl,
            record_every=record_every,
        )
    return runs["AAC4"], runs["RK4"]


def print_stress_run(label: str, r: DiagnosticRun) -> None:
    fits = blowup_dashboard(r.times, r.omega_hist, r.tail_hist)
    tstar = fits[-1].t_star
    print(
        f"{label:>5s}: steps={r.steps:5d} runtime={r.elapsed:9.4f}s "
        f"E={r.E:.8e} Omega={r.Omega:.8e} w_inf={r.w_inf:.8e} "
        f"BKM={r.bkm:.8e} tail80={r.tail80:.3e} tail90={r.tail90:.3e} "
        f"div={r.div_rms:.3e} maxCFL={r.max_cfl:.4f} "
        f"maxq={r.max_diffusive_number:.4f} "
        f"h=[{r.min_step:.3e},{r.max_step:.3e}] "
        f"stretch0={r.stretch_rms_initial:.6e} "
        f"stretchT={r.stretch_rms_final:.6e} T*80={tstar} "
        f"early_stop={r.stopped_early}"
    )
    fit_text = ", ".join(
        f"f={fit.fraction:.2f}:T*={fit.t_star},R2={fit.r2:.5f}"
        for fit in fits
    )
    print(f" inverse-vorticity fits: {fit_text}")


def test_high_re(
    N: int,
    nu: float,
    T: float,
    dt: float,
    init: str,
    cfl: float,
    amplitude: float = 1.0,
) -> None:
    print("\n=== TEST 8: 3-D VORTEX-STRETCHING HIGH-RE STRESS ===")
    g = Grid(N)
    uh0 = make_initial_state_hat(g, init, amplitude)
```

```
    stretch0_max, stretch0_rms = stretching_norm(uh0, g)
    print(
        f"initial state={init} amplitude={amplitude} N={N} nu={nu} T={T} dt_max={dt} "
        f"stretch_max={stretch0_max:.6e} stretch_rms={stretch0_rms:.6e}"
    )

    aac, rk = run_stress_pair(g, uh0, nu, T, dt, cfl, record_every=5)
    print_stress_run("AAC4", aac)
    print_stress_run("RK4", rk)

    cross = rel_l2(aac.uh_final, rk.uh_final, g)
    print(f"AAC4/RK4 final-state L2 difference = {cross:.8e}")

    if (
        cross < TOL.cross_method_l2
        and not aac.stopped_early
        and not rk.stopped_early
        and stretch0_rms > 1.0e-4
    ):
        REPORT.pass_(
            "3D-VORTEX-STRETCHING-CROSS-METHOD",
            f"final L2={cross:.3e}, initial stretch RMS={stretch0_rms:.3e}",
        )
    else:
        REPORT.info_(
            "3D-VORTEX-STRETCHING-CROSS-METHOD",
            f"final L2={cross:.3e}, initial stretch RMS={stretch0_rms:.3e}",
        )


def test_stress_refinement(
    N0: int,
    levels: int,
    nu: float,
    T: float,
    dt: float,
    init: str,
    cfl: float,
    amplitude: float = 1.0,
) -> None:
    print("\n=== TEST 9: 3-D STRESS REFINEMENT / SINGULARITY GATE ===")
    records = []

    for lev in range(levels):
        N = N0 * (2 ** lev)
        h = dt / (2 ** lev)
        g = Grid(N)
        uh0 = make_initial_state_hat(g, init, amplitude)
        r = integrate_diagnostic(
            "AAC4", uh0, g, nu, T, h,
            adaptive=True,
            cfl_target=cfl,
            record_every=5,
        )
        tstar = blowup_intercept(r.times, r.omega_hist, r.tail_hist)
        records.append((N, h, r, tstar))
        print_stress_run(f"N={N}", r)
        # Release per-resolution working arrays before the next refinement.
        del uh0, g
        gc.collect()

    if len(records) < 2:
        REPORT.info_("STRESS-REFINEMENT", "only one level")
        return

    rel_w = []
    rel_E = []
    rel_0 = []
    for j in range(1, len(records)):
        r0 = records[j - 1][2]
        r1 = records[j][2]
        rel_w.append(abs(r1.w_inf - r0.w_inf) / max(abs(r1.w_inf), 1.0e-300))
        rel_E.append(abs(r1.E - r0.E) / max(abs(r1.E), 1.0e-300))
```

```
        rel_O.append(abs(r1.Omega - r0.Omega) / max(abs(r1.Omega), 1.0e-300))

    print(f"Successive relative changes: w_inf={rel_w}")
    print(f"Successive relative changes: E={rel_E}")
    print(f"Successive relative changes: Omega={rel_O}")

    tail_ok = all(
        N < 16 or (r.tail80 < TOL.tail_safe)
        for N, _, r, _ in records
    )
    finite_ok = all(np.all(np.isfinite(r.uh_final)) for _, _, r, _ in records)
    div_ok = all(r.div_rms < TOL.divergence_rms for _, _, r, _ in records)
    stretch_ok = all(r.stretch_rms_initial > 1.0e-4 for _, _, r, _ in records)

    if tail_ok and finite_ok and div_ok and stretch_ok:
        REPORT.pass_(
            "STRESS-REFINEMENT-CONTROL",
            f"all {len(records)} levels finite, divergence/tail controlled, nonzero stretching",
        )
    else:
        REPORT.fail(
            "STRESS-REFINEMENT-CONTROL",
            "stress refinement control failed",
        )

    if all(x < 1.0e-2 for x in rel_w):
        REPORT.info_(
            "STRESS-SINGULARITY-GATE",
            "max-vorticity is numerically consistent across tested levels",
        )
    else:
        REPORT.info_(
            "STRESS-SINGULARITY-GATE",
            "max-vorticity has not tightly converged; more refinement required",
        )


def test_operator_audit(N: int) -> None:
    print("\n=== TEST 10: FOURIER OPERATOR / DIVERGENCE AUDIT ===")
    g = Grid(N)
    uh = normalize_state_hat(initial_vortex_braid(g), g)
    div = rms_divergence(uh, g)
    tail80 = spectral_tail_ratio(uh, g, 0.80)
    work = nonlinear_energy_work(uh, g)
    stretch_max, stretch_rms = stretching_norm(uh, g)

    print(f"divergence RMS = {div:.8e}")
    print(f"spectral tail ratio80 = {tail80:.8e}")
    print(f"nonlinear energy work = {work:.8e}")
    print(f"stretching max = {stretch_max:.8e}")
    print(f"stretching RMS = {stretch_rms:.8e}")

    if div < TOL.divergence_rms:
        REPORT.pass_("OPERATOR-DIVERGENCE", f"div={div:.3e}")
    else:
        REPORT.fail("OPERATOR-DIVERGENCE", f"div={div:.3e}")

    if tail80 < TOL.tail_safe:
        REPORT.pass_("OPERATOR-DEALIASING", f"tail80={tail80:.3e}")
    else:
        REPORT.fail("OPERATOR-DEALIASING", f"tail80={tail80:.3e}")

    if stretch_rms > 1.0e-4:
        REPORT.pass_("OPERATOR-VORTEX-STRETCHING", f"stretch RMS={stretch_rms:.3e}")
    else:
        REPORT.fail("OPERATOR-VORTEX-STRETCHING", f"stretch RMS={stretch_rms:.3e}")


# ============================================================================
# EXTENDED TESTS
# ============================================================================

def test_long_time_limit(
```

```
    N: int,
    nu: float,
    T: float,
    dt: float,
    init: str,
    cfl: float,
    amplitude: float,
) -> None:
    """Long-time paired AAC4/RK4 stress run with richer singularity diagnostics."""
    print("\n=== EXTENDED TEST 11: LONG-TIME 3-D STRETCHING ===")
    g = Grid(N)
    uh0 = make_initial_state_hat(g, init, amplitude)
    print(f"state={init} amplitude={amplitude} N={N} nu={nu} T={T} dt_max={dt} CFL={cfl}")
    aac, rk = run_stress_pair(g, uh0, nu, T, dt, cfl, record_every=10)
    print_stress_run("AAC4", aac)
    print_stress_run("RK4", rk)
    cross = rel_l2(aac.uh_final, rk.uh_final, g)
    speedup = rk.elapsed / max(aac.elapsed, 1e-300)
    print(f"AAC4/RK4 final L2 difference = {cross:.8e}")
    print(f"AAC4/RK4 runtime speedup = {speedup:.4f}x")

    aac_fit = blowup_dashboard(aac.times, aac.omega_hist, aac.tail_hist)
    stable_fits = [f for f in aac_fit if f.t_star is not None and f.r2 > 0.95]
    print(
        "stable inverse-vorticity extrapolations = "
        f"{len(stable_fits)}/{len(aac_fit)}"
    )

    if (
        cross < 10.0 * TOL.cross_method_l2
        and not aac.stopped_early
        and not rk.stopped_early
        and aac.tail90 < 0.01
    ):
        REPORT.pass_(
            "LONG-TIME-3D-STRESS",
            f"cross={cross:.3e}, speedup={speedup:.3f}x, tail90={aac.tail90:.3e}",
        )
    else:
        REPORT.info_(
            "LONG-TIME-3D-STRESS",
            f"cross={cross:.3e}, speedup={speedup:.3f}x, tail90={aac.tail90:.3e}",
        )


def test_stress_timestep_limit(
    N: int,
    nu: float,
    T: float,
    dt: float,
    init: str,
    amplitude: float,
) -> None:
    """AAC4 self-convergence on a hard 3-D state, independent of RK4."""
    print("\n=== EXTENDED TEST 12: HARD-STRESS TIMESTEP REFINEMENT ===")
    g = Grid(N)
    uh0 = make_initial_state_hat(g, init, amplitude)
    dts = (dt, dt / 2.0, dt / 4.0, dt / 8.0)
    states = []
    runtimes = []

    for h in dts:
        t0 = time.perf_counter()
        uh, steps, _ = evolve_fast("AAC4", uh0, g, nu, T, h)
        runtimes.append(time.perf_counter() - t0)
        states.append(uh)
        print(f"dt={h:.6e} steps={steps:5d} runtime={runtimes[-1]:.5f}s")

    e12 = rel_l2(states[0], states[1], g)
    e24 = rel_l2(states[1], states[2], g)
    e48 = rel_l2(states[2], states[3], g)
    p1 = convergence_order(e12, e24, dts[0], dts[1])
    p2 = convergence_order(e24, e48, dts[1], dts[2])
```

```
    print(f"self-error dt/dt2 = {e12:.8e}")
    print(f"self-error dt2/dt4 = {e24:.8e}")
    print(f"self-error dt4/dt8 = {e48:.8e}")
    print(f"self-convergence p12={p1:.4f} p24={p2:.4f}")

    if (
        np.isfinite(p1)
        and np.isfinite(p2)
        and p2 > 3.0
        and p1 > 2.5
    ):
        REPORT.pass_(
            "HARD-STRESS-TIMESTEP-ORDER",
            f"p12={p1:.3f}, p24={p2:.3f}",
        )
    else:
        REPORT.info_(
            "HARD-STRESS-TIMESTEP-ORDER",
            f"p12={p1:.3f}, p24={p2:.3f}",
        )


def test_cross_resolution_pair(
    N: int,
    nu: float,
    T: float,
    dt: float,
    init: str,
    amplitude: float,
) -> None:
    """Cross-method comparison at two consecutive resolutions."""
    print("\n=== EXTENDED TEST 13: CROSS-RESOLUTION / CROSS-METHOD ===")
    results = []

    for level in range(2):
        n = N * (2 ** level)
        h = dt / (2 ** level)
        g = Grid(n)
        uh0 = make_initial_state_hat(g, init, amplitude)
        aac, rk = run_stress_pair(g, uh0, nu, T, h, 0.22, record_every=10)
        cross = rel_l2(aac.uh_final, rk.uh_final, g)
        results.append((n, aac, rk, cross))
        print(f"N={n}: AAC4 runtime={aac.elapsed:.4f}s RK4 runtime={rk.elapsed:.4f}s cross={cross:.8e}")
        print(f" AAC4 E={aac.E:.8e} Omega={aac.Omega:.8e} w_inf={aac.w_inf:.8e} tail90={aac.tail90:.3e}")

    w_change = abs(results[1][1].w_inf - results[0][1].w_inf) / max(abs(results[1][1].w_inf), 1e-300)
    e_change = abs(results[1][1].E - results[0][1].E) / max(abs(results[1][1].E), 1e-300)
    o_change = abs(results[1][1].Omega - results[0][1].Omega) / max(abs(results[1][1].Omega), 1e-300)
    print(f"N->{2*N} relative changes: w_inf={w_change:.6e} E={e_change:.6e} Omega={o_change:.6e}")

    if all(r[3] < 10.0 * TOL.cross_method_l2 for r in results):
        REPORT.pass_(
            "CROSS-RESOLUTION-CROSS-METHOD",
            f"method agreement retained through N={2*N}",
        )
    else:
        REPORT.info_(
            "CROSS-RESOLUTION-CROSS-METHOD",
            "cross-method difference increased at one tested resolution",
        )


def test_limit_scaling(
    N: int,
    nu: float,
    dt: float,
    init: str,
    amplitude: float,
    steps_per_level: int = 8,
) -> None:
    """Fixed-step cost scaling using a short, identical step count per level.
```

```
    This is a performance measurement, not a physical-time accuracy test. A
    fixed step count gives a much cleaner wall-clock scaling snapshot while
    preserving the exact same per-step algorithm and resolution progression.
    """
    print("\n=== EXTENDED TEST 14: RESOLUTION SCALING SNAPSHOT ===")
    print(f"fixed measurement steps per level = {steps_per_level}")
    for level in range(3):
        n = N * (2 ** level)
        h = dt / (2 ** level)
        run_T = h * steps_per_level
        g = Grid(n)
        uh0 = make_initial_state_hat(g, init, amplitude)

        t0 = time.perf_counter()
        _, steps_aac, _ = evolve_fast("AAC4", uh0, g, nu, run_T, h)
        t_aac = time.perf_counter() - t0

        t0 = time.perf_counter()
        _, steps_rk, _ = evolve_fast("RK4", uh0, g, nu, run_T, h)
        t_rk = time.perf_counter() - t0

        print(
            f"N={n:4d} dt={h:.3e} AAC4={t_aac:.4f}s({steps_aac:4d}) "
            f"RK4={t_rk:.4f}s({steps_rk:4d}) "
            f"AAC4/step={t_aac/max(steps_aac,1):.5e}s "
            f"RK4/step={t_rk/max(steps_rk,1):.5e}s "
            f"speedup={t_rk/max(t_aac,1e-300):.3f}x"
        )
        del g, uh0
        gc.collect()


# ============================================================================
# NUMERICAL VERIFICATION TEST RUNNER
# ============================================================================

def run_forced_test(
    test_number: int,
    name: str,
    fn,
) -> None:
    """Run every test and continue after ordinary test exceptions.

    The benchmark is intentionally fail-soft at the suite level: one broken
    test is reported as FAIL, but subsequent verification tests continue to run.
    This prevents an expensive extended stress campaign from silently skipping the
    later diagnostics because of one implementation/reporting defect.
    """
    print("\n" + "-" * 108)
    print(f"TEST {test_number:02d}: {name}")
    print("-" * 108)

    try:
        fn()
    except SystemExit:
        raise
    except Exception as exc:
        REPORT.fail(
            f"TEST-{test_number:02d}-{name}",
            f"unhandled exception: {type(exc).__name__}: {exc}",
        )


# ============================================================================
# MAIN -FORCE ALL TESTS
# ============================================================================

def main() -> None:
    ap = argparse.ArgumentParser(
        description=(
            "AAC vs conventional incompressible "
            "-NavierStokes numerical verification"
        )
```

```
)

# Core suite parameters.
ap.add_argument("--N", type=int, default=24)
ap.add_argument("--nu", type=float, default=0.01)
ap.add_argument("--T", type=float, default=0.25)
ap.add_argument("--dt", type=float, default=0.005)

ap.add_argument("--instability-N", type=int, default=24)
ap.add_argument("--instability-nu", type=float, default=1.0)
ap.add_argument("--instability-T", type=float, default=0.5)

ap.add_argument("--conv-levels", type=int, default=3)
ap.add_argument("--conv-reference-divisor", type=int, default=16)
ap.add_argument("--conv-N", type=int, default=16)
ap.add_argument("--conv-T", type=float, default=0.05)
ap.add_argument("--conv-dt", type=float, default=0.01)

ap.add_argument("--adaptive-N", type=int, default=16)
ap.add_argument("--adaptive-nu", type=float, default=0.01)
ap.add_argument("--adaptive-T", type=float, default=0.12)
ap.add_argument("--adaptive-dt", type=float, default=0.04)

ap.add_argument(
    "--stress-init",
    choices=("vortex-braid", "taylor-green", "vortex-pair"),
    default="vortex-braid",
)
ap.add_argument("--stress-N", type=int, default=24)
ap.add_argument("--stress-nu", type=float, default=0.0005)
ap.add_argument("--stress-T", type=float, default=0.40)
ap.add_argument("--stress-dt", type=float, default=0.004)
ap.add_argument("--stress-cfl", type=float, default=0.24)

ap.add_argument("--refine-N", type=int, default=16)
ap.add_argument("--refine-levels", type=int, default=3)
ap.add_argument("--refine-nu", type=float, default=0.0005)
ap.add_argument("--refine-T", type=float, default=0.20)
ap.add_argument("--refine-dt", type=float, default=0.004)
ap.add_argument("--refine-cfl", type=float, default=0.24)

# Extended stress-suite parameters.
ap.add_argument("--limit-N", type=int, default=24)
ap.add_argument("--limit-nu", type=float, default=0.00025)
ap.add_argument("--limit-T", type=float, default=1.0)
ap.add_argument("--limit-dt", type=float, default=0.003)
ap.add_argument(
    "--limit-init",
    choices=("vortex-braid", "taylor-green", "vortex-pair"),
    default="vortex-braid",
)
ap.add_argument("--limit-amplitude", type=float, default=1.50)
ap.add_argument("--limit-cfl", type=float, default=0.22)
ap.add_argument("--limit-timestep-T", type=float, default=0.30)
ap.add_argument("--limit-timestep-dt", type=float, default=0.004)
ap.add_argument("--limit-resolution-T", type=float, default=0.20)
ap.add_argument("--limit-resolution-dt", type=float, default=0.004)
ap.add_argument("--limit-scaling-dt", type=float, default=0.004)
ap.add_argument("--limit-scaling-N", type=int, default=16)
ap.add_argument("--limit-scaling-steps", type=int, default=8)

args = ap.parse_args()

print("=" * 108)
print("--AACNAVIERSTOKES NAVIER-STOKES NUMERICAL VERIFICATION v5")
print("COMPLETE NUMERICAL VERIFICATION SUITE -CORE + STRESS + EXTENDED TESTS")
print("Same Newtonian incompressible -NavierStokes continuum PDE")
print("Traditional realization: explicit RK4")
print("REAL-AAC realization: exact exponential viscous propagation + AAC4 forcing")
print("v5: half-spectrum FFT path + 3-D vortex-stretching + long-time stress campaign")
print("All verification tests execute on every run.")
print("=" * 108)
print_metadata()
```

```
# ----------------------------------------------------------------------
# TESTS -110: CORE + STRESS
# ----------------------------------------------------------------------
run_forced_test(
    1,
    "EXACT-SHEAR-STABILITY",
    lambda: test_exact_shear_instability(
        args.instability_N,
        args.instability_nu,
        args.instability_T,
        (0.50, 0.80, 0.95, 1.00, 1.05, 1.10, 1.25, 1.50, 2.00),
    ),
)

run_forced_test(
    2,
    "ABC-EXACT-DECAY",
    lambda: test_abc(
        args.N,
        args.nu,
        args.T,
        args.dt,
    ),
)

run_forced_test(
    3,
    "TAYLOR-GREEN-CROSS-METHOD",
    lambda: test_taylor_green(
        args.N,
        args.nu,
        args.T,
        args.dt,
    ),
)

run_forced_test(
    4,
    "TEMPORAL-CONVERGENCE",
    lambda: test_temporal_convergence(
        args.conv_N,
        args.nu,
        args.conv_T,
        args.conv_dt,
        args.conv_levels,
        args.conv_reference_divisor,
    ),
)

run_forced_test(
    5,
    "AAC4-AB4-LIMIT",
    test_ab4_limit,
)

run_forced_test(
    6,
    "NONLINEAR-ENERGY-IDENTITY",
    lambda: test_nonlinear_energy_identity(
        args.conv_N,
    ),
)

run_forced_test(
    7,
    "VARIABLE-STEP-AAC4",
    lambda: test_adaptive_aac4(
        args.adaptive_N,
        args.adaptive_nu,
        args.adaptive_T,
        args.adaptive_dt,
    ),
```

```
)

run_forced_test(
    8,
    "3D-VORTEX-STRETCHING-STRESS",
    lambda: test_high_re(
        args.stress_N,
        args.stress_nu,
        args.stress_T,
        args.stress_dt,
        args.stress_init,
        args.stress_cfl,
        1.0,
    ),
)

run_forced_test(
    9,
    "3D-STRESS-REFINEMENT",
    lambda: test_stress_refinement(
        args.refine_N,
        args.refine_levels,
        args.refine_nu,
        args.refine_T,
        args.refine_dt,
        args.stress_init,
        args.refine_cfl,
        1.0,
    ),
)

run_forced_test(
    10,
    "FOURIER-OPERATOR-AUDIT",
    lambda: test_operator_audit(
        args.N,
    ),
)

# ------------------------------------------------------------------------
# TESTS -1114: EXTENDED STRESS CAMPAIGN
# ------------------------------------------------------------------------
run_forced_test(
    11,
    "LONG-TIME-3D-EXTENDED STRESS",
    lambda: test_long_time_limit(
        args.limit_N,
        args.limit_nu,
        args.limit_T,
        args.limit_dt,
        args.limit_init,
        args.limit_cfl,
        args.limit_amplitude,
    ),
)

run_forced_test(
    12,
    "HARD-STRESS-TIMESTEP-REFINEMENT",
    lambda: test_stress_timestep_limit(
        args.limit_N,
        args.limit_nu,
        args.limit_timestep_T,
        args.limit_timestep_dt,
        args.limit_init,
        args.limit_amplitude,
    ),
)

run_forced_test(
    13,
    "CROSS-RESOLUTION-CROSS-METHOD",
    lambda: test_cross_resolution_pair(
```

```
            args.limit_N,
            args.limit_nu,
            args.limit_resolution_T,
            args.limit_resolution_dt,
            args.limit_init,
            args.limit_amplitude,
        ),
    )

    run_forced_test(
        14,
        "RESOLUTION-SCALING-SNAPSHOT",
        lambda: test_limit_scaling(
            args.limit_scaling_N,
            args.limit_nu,
            args.limit_scaling_dt,
            args.limit_init,
            args.limit_amplitude,
            args.limit_scaling_steps,
        ),
    )

    print("\n" + "=" * 108)
    print("COMPLETE VERIFICATION TEST CAMPAIGN COMPLETE")
    print("Tests attempted: 14 / 14")
    print("=" * 108)

    REPORT.summary()

    if REPORT.failed:
        print(
            "FINAL STATUS: NOT PUBLICATION NUMERICAL -"
            "one or more tests failed or threw an exception."
        )
        raise SystemExit(2)

    print(
        "FINAL STATUS: ALL TESTS PASSED."
    )
    print(
        "Passing numerical tests do not constitute a mathematical proof of "
        "global regularity or blowup."
    )


if __name__ == "__main__":
    main()
```

C Reproducibility Manifest

Author: Rich C. Young (Guru#1)

Website: https://youtube.com/@GuruOne

Benchmark source: `aac_ns_benchmark_publication.py` (publication listing; execution source archived separately)

SHA-256: `45872e5373461922dde9edab3c912f66559ac8e1f1300cd421c6601d623b7b95`

Source lines: 2242

Final forced campaign: 14 tests attempted; PASS=21, FAIL=0, INFO=1.