Dedalus: A flexible framework for numerical simulations with spectral methods

Keaton J. Burns

Massachusetts Institute of Technology Departments of Mathematics and Physics, Cambridge, Massachusetts 02139, USA
and Center for Computational Astrophysics, Flatiron Institute, New York, New York 10010, USA

Geoffrey M. Vasil

School of Mathematics and Statistics, University of Sydney, Sydney, New South Wales 2006, Australia

Jeffrey S. Oishi

Bates College Department of Physics and Astronomy, Lewiston, Maine 04240, USA

Daniel Lecoanet

Princeton Center for Theoretical Science and Princeton University Department of Astrophysical Sciences, Princeton, New Jersey 08544, USA

Benjamin P. Brown

University of Colorado Laboratory for Atmospheric and Space Physics and Department of Astrophysical and Planetary Sciences,
Boulder, Colorado 80309, USA

Numerical solutions of partial differential equations enable a broad range of scientific research. The Dedalus project is a flexible, open-source, parallelized computational framework for solving general partial differential equations using spectral methods. Dedalus translates plain-text strings describing partial differential equations into efficient solvers. This paper details the numerical method that enables this translation, describes the design and implementation of the codebase, and illustrates its capabilities with a variety of example problems. The numerical method is a first-order generalized tau formulation that discretizes equations into banded matrices. This method is implemented with an object-oriented design. Classes for spectral bases and domains manage the discretization and automatic parallel distribution of variables. Discretized fields and mathematical operators are symbolically manipulated with a basic computer algebra system. Initial value, boundary value, and eigenvalue problems are efficiently solved using high-performance linear algebra, transform, and parallel communication libraries. Custom analysis outputs can also be specified in plain text and stored in self-describing portable formats. The performance of the code is evaluated with a parallel scaling benchmark and a comparison to a finite-volume code. The features and flexibility of the codebase are illustrated by solving several examples: the nonlinear Schrödinger equation on a graph, a supersonic magnetohydrodynamic vortex, quasigeostrophic flow, Stokes flow in a cylindrical annulus, normal modes of a radiative atmosphere, and diamagnetic levitation.

I. INTRODUCTION

Partial differential equations (PDEs) describe continuum processes. The continuous independent variables typically represent space and time, but can also represent more abstract quantities such as momentum, energy, age of a population, or currency. The ability to equate the infinitesimal rates of change of different quantities produces endless possible applications. Important examples include wave propagation, heat transfer, fluid flow, quantum mechanical probability flux, chemical and nuclear reactions, biological phenomena, financial markets [1], or social/population dynamics [2,3]. Even

more intriguing are possible combinations of several of the above [4,5].

Apart from a small handful of closed-form solutions, the vast majority of PDEs require serious numerical and computational intervention. A wide variety of numerical algorithms solve PDEs through the general approach of discretizing its continuous variables and operators to produce a finite-sized algebraic system yielding an approximate solution. Finite element, finite volume, and finite difference methods are common schemes that discretize the domain of the PDE into cells or points and derive algebraic relations between the values at neighboring cells or points from the governing equations. These methods can accommodate complex geometries (such as the flow around an aircraft), but can be difficult to implement for complex equations and typically converge relatively slowly as additional cells or points are added.

In contrast, spectral methods discretize variables by expanding them in a finite set of basis functions and derive

---

equations for the coefficients of these functions. These methods are well-suited to many equation types and provide rapidly converging solutions (e.g., exponential for smooth functions) as additional modes are included. However, spectral methods are typically limited to simple geometries (such as boxes, cylinders, and spheres). Recent literature has developed sparse representations of equations that are substantially better conditioned and faster than traditional dense collocation techniques [6–12]. These features make spectral methods an attractive choice for scientists seeking to study a wide variety of physical processes with high precision.

While computing capacities have grown exponentially over the past few decades, the progression of software development has been more gradual. Many software packages have chosen one or a few closely related PDEs and focused on creating highly optimized implementations of algorithms that are well-suited to those choices. These solvers usually hardcode not only the PDE but also the dynamical variables, choice of input control parameters, integration scheme, and analysis output. While many world-class simulation codes have been developed this way, often scientific questions lead beyond what a dedicated code can do. This is not always because of a lack of computational power or efficiency, but often because continued progress requires an alternative model, dynamical variable reformulation, or more exotic forms of analysis.

Simulation packages with flexible model specification also address an underserved scientific niche. It is often straightforward to write serial codes to solve simple one-dimensional equations for particular scientific questions. It is also worthwhile to invest multiple person-years building codes that solve well-known equations. However, it can be difficult to justify spending significant time developing codes for novel models that are initially studied by only a few researchers. We believe this leaves many interesting questions unaddressed simply from a local cost-benefit analysis. Flexible toolkits can lower the barrier to entry for a large number of interesting scientific applications.

The FENICS [13] and FIREDRAKE [14] packages both allow users to symbolically enter their equations in variational form and produce finite-element discretizations suitable for forward-modeling and optimization calculations. These are very powerful tools for solving wide ranges of PDEs in complicated geometries, however they remain less efficient than spectral methods for many PDEs in simple geometries. CHANNELFLOW [15] uses sparse Chebyshev methods to simulate the Navier-Stokes equations and allows users to find and analyze invariant solutions using dynamical systems techniques. However, the code is restricted to solving incompressible flow in a periodic channel geometry. The CHEBFUN [16] and APPROXFUN [17] packages are highly flexible toolkits for performing function approximation using spectral methods. They include a wide variety of features including sparse, well-conditioned, and adaptive methods for efficiently solving differential equations to machine precision. However, these packages are not optimized for the solution of multidimensional PDEs on parallel architectures.

The goal of the Dedalus project is to bridge this gap and provide a framework applying modern, sparse spectral techniques to highly parallelized simulations of custom PDEs. The codebase allows users to discretize domains using the direct

products of spectral series and symbolically specify systems of PDEs on those domains. The code then produces a sparse discretization of the equations and automatically parallelizes the solution of the resulting model. The Dedalus codebase is open-source, highly modular, and easy to use. While its development has been motivated by the study of turbulent flows in astrophysics and geophysics, Dedalus is capable of solving a much broader range of PDEs. To date, it has been used for applications and publications in applied mathematics [18–22], astrophysics [23–38], atmospheric science [39–44], biology [45–47], condensed matter physics [48,49], fluid dynamics [50–61], glaciology [62], limnology [63], numerical analysis [64–67], oceanography [68–72], planetary science [73,74], and plasma physics [75–82].

We begin this paper with a review of the fundamental theory of spectral methods and a description of the specific numerical method employed by Dedalus (Sec. II). We then provide an overview of the project and codebase using a simple example problem (Sec. III). Sections IV–X detail the implementations of the fundamental modules of the codebase, with a particular emphasis on its systems for symbolic equation entry and automatic distributed-memory parallelization. Although these sections describe essential details of the code, a careful reading is not necessary to begin using Dedalus. Finally, Sec. XI demonstrates the features and performance of the codebase with a parallel scaling analysis, a comparison to a finite volume code, and example simulations of nonlinear waves on graphs, compressible magnetohydrodynamic flows, quasigeostrophic flow in the ocean, Stokes flow in cylindrical geometry, atmospheric normal modes, and diamagnetic levitation.

II. SPARSE SPECTRAL METHODS

A. Fundamentals of spectral methods

1. Spectral representations of functions

A spectral method discritizes functions by expanding them over a set of basis functions. These methods find broad application in numerical analysis and give highly accurate and efficient algorithms for manipulating functions and solving differential equations. The classic reference Boyd *et al.* [83] covers the material in this section in great detail.

Consider a complete orthogonal basis $\{\phi_n(x)\}$ and the associated inner product $(\phi_n, \phi_m)_\phi \propto \delta_{n,m}$. The spectral representation of a function $f(x)$ comprises the coefficients $\{f_n^\phi\}$ appearing in the expansion of $f(x)$ as

$$f(x) = \sum_{n=0}^{\infty} f_n^\phi \phi_n(x), \tag{1}$$

with

$$f_n^\phi = \frac{(\phi_n, f)_\phi}{(\phi_n, \phi_n)_\phi} \equiv \langle \phi_n | f \rangle. \tag{2}$$

We will use bra-ket notation to denote such normalized bra-family inner products. Formally, an exact representation requires an infinite number of nonzero coefficients. Numerical spectral methods approximate functions (e.g., PDE solutions) using expansions that are truncated after N modes. The truncated coefficients $\tilde{f}_n^\phi$ are computed using quadrature rules of

the form

$$\tilde{f}_n^\phi = \sum_{i=0}^{N-1} w_i f(x_i) \qquad (3)$$

where the weights w_i and collocation points x_i depend on the underlying inner-product space. The quadrature scheme constitutes a discrete *spectral transform* for translating between the spectral coefficients and N samples of the function.

The error in the truncated approximation is often of the same order as the last retained coefficient. The spectral coefficients of smooth functions typically decay exponentially with n, resulting in highly accurate representations. The spectral coefficients of nonsmooth functions typically decay algebraically as $n^{-\alpha}$, where α depends on the order of differentiability of $f(x)$. Exact differentiation and integration on the underlying basis functions provides accurate calculus for general functions. For PDEs with highly differentiable solutions, spectral methods therefore give significantly more accurate results than fixed-order schemes.

2. Common spectral series

Trigonometric polynomials are the archetypal spectral bases: sine series, cosine series, and complex exponential Fourier series. These bases provide exponentially converging approximations to smooth functions on periodic intervals. The fast Fourier transform (FFT) can compute the series coefficients in $\mathcal{O}(N \log_2 N)$ time, enabling computations requiring both the coefficients and grid values to be performed efficiently.

The classical orthogonal polynomials also frequently appear as spectral bases. Most common are the Chebyshev polynomials $\{T_n(x)\}$, which provide exponentially converging approximations to smooth functions on the interval $[-1, 1]$. A simple change of variables relates the Chebyshev polynomials to cosine functions,

$$T_n(x) = \cos(n \cos^{-1}(x)). \qquad (4)$$

Geometrically, $T_n(x)$ is the projection of $\cos(nx)$ from the cylinder to the plane; see Fig. 1. The relation to cosines enables transforming between Chebyshev coefficients and values on collocation points using the fast discrete cosine transform (DCT). The fast transform often makes Chebyshev series preferable to other polynomials on finite intervals.

3. Solving differential equations with spectral methods

Spectral methods solve PDEs by creating algebraic equations for the coefficients of the truncated solution. Different approaches for constructing and solving these systems each come with advantages and disadvantages. In examining a few approaches, we consider a simple linear PDE of the form $Lu(x) = f(x)$, where L is a x-differential operator.

The *collocation* approach is perhaps the most common polynomial spectral method. In this case, the differential equation is enforced at the interior collocation points. The solution is written in terms of the values at these points:

$$Lu(x_i) = f(x_i), \quad i = 0, \dots, N-1. \qquad (5)$$

The boundary conditions typically replace the DE at the collocation endpoints. The collocation method works well in a
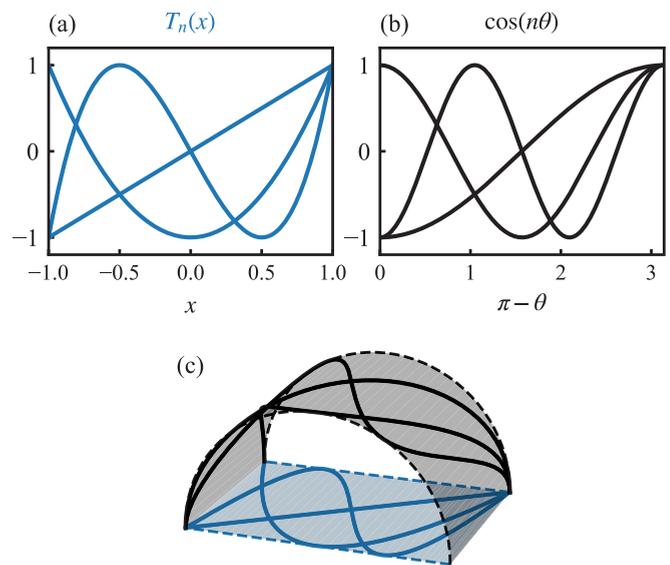

FIG. 1. Chebyshev polynomials (a) can be viewed as cosine functions (b) drawn on a cylinder and projected onto the bisecting plane (c).

broad range of applications. The primary advantages are that many boundary conditions are easily enforced and the solution occurs on the grid. The primary disadvantages are that the method produces dense matrices (L) and more complicated boundary conditions require more care to implement [84].

An alternative is the *Galerkin* method, where the solution is written in terms of "trial" functions $\{\phi_n\}$ that satisfy the boundary conditions. The differential equation is then projected against a set of "test" functions $\{\psi_n\}$:

$$\sum_n \langle \psi_i | L \phi_n \rangle u_n^\phi = \sum_n \langle \psi_i | \phi_n \rangle f_n^\phi, \quad i = 0, \dots, N-1. \qquad (6)$$

For periodic boundary conditions, the Galerkin method using Fourier series produces diagonal derivative operators, allowing constant-coefficient problems to be solved trivially. Galerkin bases can be constructed from Chebyshev polynomials for simple boundary conditions, with the caveat that the series coefficients must be converted back to Chebyshev coefficients to apply fast transforms.

The *tau* method generalizes the Galerkin method by solving the perturbed equation

$$Lu(x) + \tau P(x) = f(x), \qquad (7)$$

where $P(x)$ is specified. The parameter τ adjusts to accommodate the boundary conditions, which are enforced simultaneously. The tau method provides a conceptually straightforward way of applying general boundary conditions without needing a specialized basis. The classical tau method [85,86] uses the same test and trial functions and assumes $P(x) = \phi_{n-1}(x)$, making it equivalent to dropping the last row of the discrete L matrix and replacing it with the boundary condition. This classical formulation with Chebyshev series results in dense matrices, but (as described below) the tau method can be modified to produce sparse and banded matrices for many equations.
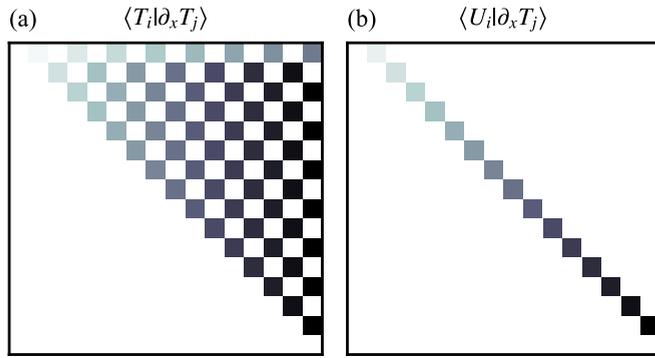
(a) $\langle T_i | \partial_x T_j \rangle$ (b) $\langle U_i | \partial_x T_j \rangle$

FIG. 2. Derivative matrices using (a) Chebyshev-T and (b) Chebyshev-U polynomials as test functions. Using different families of test and trial functions allows general differential operators to be represented with sparse matrices.

B. A general sparse tau method

This section describes the spectral method employed in Dedalus. We use a tau method with different trial and test bases, which produces sparse matrices for general equations. Formulating problems as first-order systems and using Dirichlet preconditioning (basis recombination) renders matrices fully banded.

The method is fundamentally one-dimensional, but generalizes trivially to D dimensions with all but one separable dimension (e.g., Fourier-Galerkin). Multidimensional problems then reduce to an uncoupled set of 1D problems that are solved individually.

Other approaches based on the ultraspherical method [9] or integral formulations [7] similarly result in sparse and banded matrices for many equations. Our particular approach is designed to accommodate general systems of equations and boundary conditions automatically.

1. Sparse differential operators

Traditional polynomial spectral formulations often result in dense derivative matrices. In particular, the derivatives of Chebyshev polynomials are dense when expanded back in Chebyshev polynomials:

$$\partial_x T_j(x) = \sum_{i=0}^{j-1} T_i(x) \frac{2j((j-i) \bmod 2)}{1 + \delta_{i,0}}. \tag{8}$$

Figure 2 shows the matrix version of Eq. (8), referred to as the "T-to-T" form. However, the derivatives of Chebyshev polynomials (of the first kind) are proportional to the Chebyshev polynomials of the second kind, $U_n(x)$:

$$\partial_x T_n(x) = n U_{n-1}(x). \tag{9}$$

Defined trigonometrically,

$$U_n(x) = \frac{\sin((n+1)\cos^{-1}(x))}{\sin(\cos^{-1}(x))}. \tag{10}$$

Using test functions $\psi_n = U_n$ and trial functions $\phi_n = T_n$ produces a single-band derivative matrix, called the "T-to-U" form (right side in Fig. 2). We convert nondifferential terms

from T-to-U via the sparse conversion relation

$$2T_n(x) = U_n(x) - U_{n-2}(x). \tag{11}$$

Together, these relations render first-order differential equations sparse. Higher-order equations can be handled by utilizing ultraspherical polynomials for higher derivatives [9], but our approach is to simply reduce all equations to first-order systems. The sparse-τ method extends to other orthogonal polynomial series. Appendix 1 lists the full set of derivative and conversion relations implemented in Dedalus.

2. Banded boundary conditions

Choosing the tau polynomial $P(x) = U_{N-1}(x)$ in a T-to-U method allows dropping the last matrix row and finding $u(x)$ without finding τ. The system then consists of a banded interior matrix, bordered with a dense boundary-condition row. Applying a right preconditioner renders the boundary row sparse and the system fully banded. For Dirichlet boundary conditions, using the adjoint relation of Eq. (11) gives the nonorthogonal polynomials,

$$D_n(x) = \begin{cases} T_n(x), & n = 0, 1 \\ T_n(x) - T_{n-2}(x), & n \geqslant 2 \end{cases}, \tag{12}$$

where

$$D_n(\pm 1) = \begin{cases} (\pm 1)^n, & n = 0, 1 \\ 0, & n \geqslant 2 \end{cases}. \tag{13}$$

In this basis, Dirichlet boundary conditions only involve the first two expansion coefficients. This technique is known as "Dirichlet preconditioning" or "basis recombination."

In summary, for first-order systems with Dirichlet boundary conditions, choosing $\phi_n = D_n$, $\psi_n = U_n$, and $P = U_{N-1}$ produces fully banded matrices. The resulting matrices are efficiently sovled using sparse/banded algorithms. With a first-order system, it is possible to reformulate any boundary condition (e.g., Neumann or global integral conditions) in terms of a Dirichlet condition on the first-order variables. This formulation extends to other orthogonal polynomial bases. Appendix 2 lists the full set of Dirichlet recombinations implemented in Dedalus.

3. Nonconstant coefficients

Many physical problems require multiplication by spatially nonconstant coefficients (NCCs) that vary slowly compared to the unknown solution. Olver *et al.* [9], in the context of Chebyshev polynomials, observed that multiplication by such NCCs corresponds to band-limited spectral operators.

Multiplication by a general NCC $g(x)$ acts linearly on $u(x)$ via

$$\sum_j \langle \psi_i | g \phi_j \rangle u_j^\phi = \sum_j G_{i,j}^{\psi\phi} u_j^\phi. \tag{14}$$

Given an expansion of the NCC in some basis as $g(x) = \sum_{n=0}^{\infty} g_n^\xi \xi_n(x)$, the NCC matrix is

$$G_{i,j}^{\psi\phi} = \sum_{n=0}^{\infty} g_n^\xi \langle \psi_i | \xi_n \phi_j \rangle. \tag{15}$$

For all orthogonal polynomials, $\langle \psi_i | \xi_n \phi_j \rangle = 0$ if $|i - j| > n$. We truncate NCC expansions by dropping all terms where
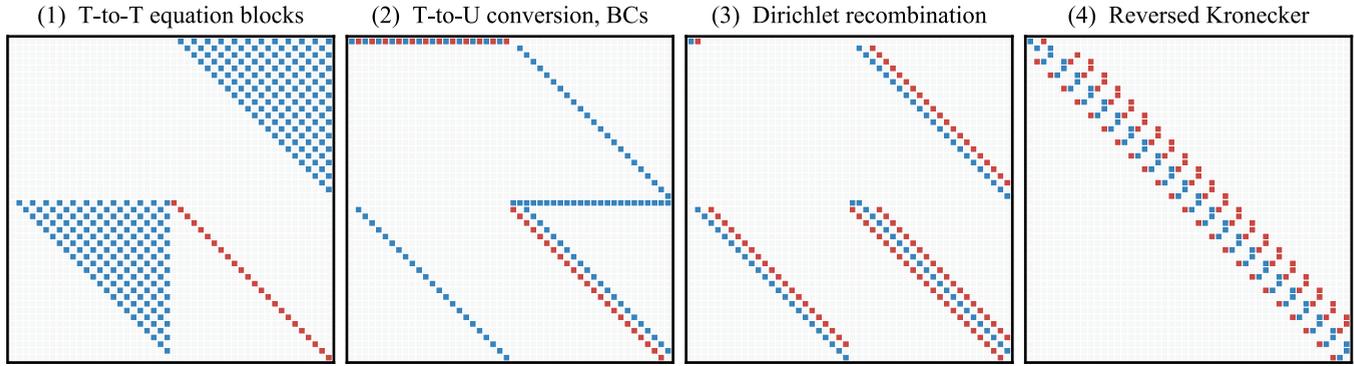
(1) T-to-T equation blocks	(2) T-to-U conversion, BCs	(3) Dirichlet recombination	(4) Reversed Kronecker

FIG. 3. Conceptual stages in matrix construction for the Chebyshev discretization of Poisson's equation with Dirichlet and Neumann boundary conditions (matrix entries colored by sign). (1) Original (T-to-T) matrices with block columns corresponding to u and u_x, and block rows corresponding to the LHS of $\partial_x u_x = 0$ and $\partial_x u - u_x = f(x)$. The T-to-T differentiation matrices are dense upper triangular. (2) After T-to-U conversion and the addition of boundary conditions via the tau method. The derivatives are sparse and the identity-block bandwidths increase slightly. The boundary conditions involve all coefficients of u and u_x. (3) After Dirichlet recombination. Boundary rows are sparse and the equation-block bandwidths increase slightly. (4) After reversing the Kronecker product to group by modes rather than variables. This final matrix is highly sparse and completely banded.

g_n^ξ is smaller than some threshold amplitude. The overall bandwidth of $G^{\psi\phi}$ is therefore N', the number of terms that are retained in the expansion of the NCC. For smooth NCCs, $N' \ll N$ and the multiplication matrix has low bandwidth. Appendix 3 lists the full set of multiplication matrices implemented in Dedalus.

4. Solving systems of equations

For coupled systems of equations with S variables, $\sum_{j=1}^{S} L^{i,j} u_j(x) = f_i(x)$ for $i = 1, \ldots, S$. In block-operator form,

$$\mathcal{L} \cdot \mathcal{X} = \mathcal{F}, \qquad (16)$$

where $\mathcal{X}$ is the state vector of variables, u_j, and $\mathcal{L}$ is a matrix of the operators. We discretize the system by replacing each $L^{i,j}$ with its sparse-tau matrix representation described above. In block-banded form, using Kronecker products and the placement matrix $(E^{i,j})_{k,l} = \delta_{i,k}\delta_{k,l}$,

$$L_{\text{sys}} = \sum_{i,j=1}^{S} E^{i,j} \otimes L^{i,j}. \qquad (17)$$

The system matrix L_{sys} acts on the concatenation of the variable coefficients and has bandwidth $\mathcal{O}(SN)$.

If the operator matrices are interleaved rather than block-concatenated, the resulting system matrix will act on the interleaved variable coefficients and takes the form

$$L_{\text{sys}} = \sum_{i,j=1}^{S} L^{i,j} \otimes E^{i,j}. \qquad (18)$$

This matrix will have bandwidth $\mathcal{O}(S)$, making it practical to simultaneously solve coupled systems of equations with large N.

5. Summary and example

Dedalus uses a modified tau method that produces banded and well-conditioned matrices. Carefully chosen test-trial ba-

sis pairs render derivatives sparse. The first-order formulation makes all boundary conditions equivalent to Dirichlet conditions, which become banded under basis recombination. Truncated NCC expansions retain bandedness for smooth NCCs. Finally, matrices and coefficients are interleaved to keep systems of equations banded.

Figure 3 shows the matrices at various conceptual stages for a Chebyshev discretization of Poisson's equation in 1D with Dirichlet and Neumann boundary conditions:

$$\frac{\partial^2 u}{\partial x^2} = f(x), \quad u(-1) = 0, \quad \partial_x u(1) = 0. \qquad (19)$$

The combination of writing equations as first-order systems, T-to-U derivative mapping, Dirichlet preconditioning, and grouping modes before variables produces a banded pencil matrix.

We also note that coupled systems easily allow constraint equations (those without temporal derivatives) to be imposed alongside evolution equations, avoiding variable reformulations and/or splitting methods. For instance, the divergence condition in incompressible hydrodynamics can be imposed directly (determining the pressure). The momentum equation can then be integrated without splitting or derived pressure boundary conditions.

III. PROJECT OVERVIEW AND DESIGN

A. Codebase structure

Dedalus makes extensive use of object-oriented programming to provide a simple interface for the parallel solution of general systems of PDEs. The basic class structure reflects the mathematical objects that are encountered when posing and solving a PDE. As an illustrative example, we consider solving the Fisher-KPP equation, a reaction-diffusion equation that first arose in ecology:

$$\frac{\partial u}{\partial t} - D \nabla^2 u = R\, u(1 - u), \qquad (20)$$

with unknown variable $u(\mathbf{x}, t)$, diffusion coefficient D, and reaction rate R [87,88].

Properly posing the PDE first requires specifying its spatial domain. This is done by creating a `Basis` object discretizing each dimension over a specified interval and forming a `Domain` object as the direct product of these bases. Here we construct a 2D channel domain, periodic in x and with Neumann boundary conditions on the boundaries in y, as the direct product of a Fourier basis and a Chebyshev basis:

```python
import numpy as np
import dedalus.public as de

# Bases: names, modes, intervals, dealiasing
x_basis = de.Fourier('x', 128,
    interval=(0, 2*np.pi), dealias=3/2)
y_basis = de.Chebyshev('y', 64,
    interval=(0, 1), dealias=3/2)
# Domain: bases, datatype
domain = de.Domain([x_basis, y_basis], float)
```

Next, we define an initial value problem on this domain consisting of the PDE in first-order form (for temporal and Chebyshev derivatives) along with the boundary conditions. This is done by creating an `Problem` object representing the problem type (here an initial value problem, IVP). Problem parameters, string-substitutions (to simplify equation entry), and plain-text equations and boundary conditions are then added to the problem. Under the hood, Dedalus constructs `Field` objects to represent the variables and `Operator` objects that symbolically represent the mathematical expressions in the PDE.

```python
# Problem: domain and variables
prob = de.IVP(domain, variables=['u','uy'])
# Fixed parameters
prob.parameters['D'] = 0.1
prob.parameters['R'] = 1
# Parsing substitutions
prob.substitutions['Lu'] = "dx(dx(u)) + dy(uy)"
# First-order reduction
prob.add_equation("uy - dy(u) = 0")
# Fisher-KPP equation
prob.add_equation("dt(u) - D*Lu = R*u*(1-u)")
# Neumann boundary conditions
prob.add_bc("left(uy) = 0")
prob.add_bc("right(uy) = 0")
```

To finish posing the IVP, we need to specify a temporal integration scheme, the temporal integration limits, and the initial values of the variables. This is done through the `Solver` object built by each `Problem`. The initial data of the state fields can be easily accessed from the solver and set in grid ('g') or coefficient ('c') space.

```python
# Pick a timestepper
ts = de.timesteppers.RK222
# Build solver
solver = prob.build_solver(ts)
# Set integration limits
solver.stop_sim_time = 10
# Set initial conditions
u = solver.state['u']
u['g'] = np.random.rand(*u['g'].shape)
```

Finally, the problem is solved by iteratively applying the temporal integration scheme to advance the solution in time. In Dedalus, this main loop is directly written by the user, allowing for arbitrary data interactions as the integration pro-

gresses. Although not included in this example, Dedalus also provides extensive analysis tools for evaluating and saving quantities during the integration.

```python
dt = 0.01
# Main loop chceking stopping criteria
while solver.ok:
    # Step forward
    solver.step(dt)
    # Perform some analysis
    print(np.mean(u['g']), np.std(u['g']))
```

While this simple example covers the core user-facing classes, several other classes control the automatic MPI parallelization and efficient solution of the solvers. All together, the fundamental class hierarchy consists of the following:

(1) `Basis`: A one-dimensional spectral basis.

(2) `Domain`: The direct product of multiple bases, forming the spatial domain of dependence of a PDE.

(3) `Distributor`: Directs the parallel decomposition of a domain and spectral transformations of distributed data between different states.

(4) `Layout`: A distributed transformation state, e.g., grid space or coefficient space.

(5) `Field`: A scalar-valued field over a given domain. The fundamental data unit in Dedalus.

(6) `Operator`: Mathematical operations on sets of fields, composed to form mathematical expressions.

(7) `Handler`: Captures the outputs of multiple operators to store in memory or write to disk.

(8) `Evaluator`: Efficiently coordinates the simultaneous evaluation of the tasks from multiple handlers.

(9) `Problem`: User-defined PDEs (initial, boundary, and eigen-value problems).

(10) `Timestepper`: ODE integration schemes that are used to advance initial value problems.

(11) `Solver`: Coordinates the actual solution of a problem by evaluating the underlying operators and performing time integration, linear solves, or eigenvalue solves.

The following sections detail the functionality and implementation of these classes.

B. Dependencies

Dedalus is provided as an open-source python3 package. We choose to develop the code in PYTHON because it is an open-source, high-level language with a vast ecosystem of libraries for numerical analysis, system interaction, input/output, and data visualization. While numerical algorithms written directly in PYTHON sometimes suffer from poor performance, it is quite easy to wrap optimized C libraries into high-level interfaces with PYTHON . A typical high-resolution Dedalus simulation will spend a majority of its time in optimized C libraries.

The primary dependencies of Dedalus include the following:

(1) The NUMPY, SCIPY, and CYTHON packages for PYTHON [89,90].

(2) The FFTW C-library for fast Fourier transforms [91].

(3) An implementation of the MPI communication interface and its PYTHON wrapper mpi4py [92].

(4) The HDF5 C-library for reading and writing HDF5 files and its PYTHON wrapper h5py [93,94].

A wide range of standard-library PYTHON packages are used to build logging, configuration, and testing interfaces following standard practices.

Additionally, and perhaps counter-intuitively, we have found that creating algorithms to accommodate a broad range of equations and domains has resulted in a compact and maintainable codebase. Currently, the Dedalus package consists of roughly 10 000 lines of PYTHON. By producing sufficiently generalized algorithms, it is possible to compactly and robustly provide a great deal of functionality.

C. Documentation

Dedalus has been publicly available under the open-source GPL3 license since its creation, and is developed under distributed version control. The online documentation includes a series of tutorials and example problems demonstrating the code's capabilities and walking new users through the basics of constructing and running a simulation. Links to the source code repository and the documentation are available through the project website [95].

D. Community

The Dedalus collaboration uses open-source code development and strongly supports open scientific practices. The benefits of the philosophy include distributed contributions to the codebase, a low barrier-to-entry (especially for students), and detailed scientific reproducibility. We outline these ideas in detail in Oishi *et al.* [96].

Code development occurs through a public system of pull requests and reviews on the source code repository. Periodic releases are issued to the PYTHON package index (PyPI), and a variety of full-stack installation channels are supported, including single-machine and cluster install scripts and a conda-based build procedure.

The core developers maintain mailing lists for the growing Dedalus user and developer communities. The mailing list is publicly archived and searchable, allowing new users to find previous solutions to common problems with installation and model development. The Dedalus user list currently has over 150 members. A list of publications using the code is maintained online [97].

IV. SPECTRAL BASES

Dedalus currently represents multidimensional fields using the direct product of one-dimensional spectral bases. This direct product structure generally precludes domains including coordinate singularities, such as full disks or spheres. However, curvilinear domains without coordinate singularities, such as cylindrical annuli can still be represented using this direct product structure (for an example, see Sec. XI F). Basis implementations form the lowest level of the program's class hierarchy. The primary responsibilities of the basis classes are to define their collocation points and to provide an interface for transforming between the spectral coefficients of a function and the values of the function on their collocation points.

An instance of a basis class represents a series of its respective type truncated to a given number of modes N_c, and

remapped from *native* to *problem* coordinates with an affine map. A basis object is instantiated with a `name` string defining the coordinate name, a `base_grid_size` integer setting N_c, parameters fixing the affine coordinate map (a *problem* coordinate interval for bases on finite intervals, or stretching and offset parameters for bases on infinite intervals), and a `dealias` factor. Each basis class is defined with respect to a *native* coordinate interval, and contains a method for producing a collocation grid of N_g points on this interval, called a *native grid* of *scale* $s = N_g/N_c$. Conversions between the *native coordinates* x_n and *problem coordinates* x_p are done via an affine map of the form $x_p = ax_n + b$, which is applied to the native grid to produce the *basis grid* .

Each basis class defines methods for *forward transforming* (moving from grid values to spectral coefficients) and *backward transforming* (vice versa) data arrays along a single axis. Using objects to represent bases allows the transform methods to easily cache plans or matrices that are costly to construct. The basis classes also present a unified interface for implementing identical transforms using multiple libraries with different performance and build requirements. We now define the basis functions, grids, and transform methods for the currently implemented spectral bases.

A. Fourier basis

For periodic dimensions, we implement a `Fourier` basis consisting of complex exponential modes on the native interval $[0, 2\pi]$:

$$\phi_k^F(x) = \exp(ikx) \tag{21}$$

and a native grid consisting of evenly spaced points beginning at the left side of the interval:

$$x_i = \frac{2\pi i}{N_g}, \quad i = 0, \ldots, N_g - 1. \tag{22}$$

A function is represented as a symmetric sum over positive and negative wave numbers

$$f(x) = \sum_{-k_m}^{k_m} f_k \phi_k^F(x), \tag{23}$$

where $k_m = \text{floor}((N_c - 1)/2)$ is the maximum resolved wave number, excluding the Nyquist mode $k_N = N_c/2$ when N_c is even. When f is a real function, we store only the complex coefficients corresponding to $k \geqslant 0$; the $k < 0$ coefficients are determined by conjugate symmetry. We discard the Nyquist mode since it is only marginally resolved: for real functions, the Nyquist mode captures $\cos(k_N x)$, but not $\sin(k_N x)$, which vanishes on the grid when $N_g = N_c$.

The expansion coefficients are given explicitly by

$$f_k = \frac{1}{2\pi} \int_0^{2\pi} f(x)\phi_k^{F*}(x)dx \tag{24}$$

$$= \frac{1}{N_g} \sum_{i=0}^{N_g-1} f(x_i)\phi_k^{F*}(x_i) \tag{25}$$

and are computed with the fast Fourier transform (FFT). We implement FFTs from both the SCIPY and FFTW libraries, and rescale the results to match the above normalizations,

i.e., the coefficients directly represent mode amplitudes. The coefficients are stored in the traditional FFT output format, starting from $k = 0$ and increasing to k_m, then following with $-k_m$ and increasing to -1.

B. Sine/cosine basis

For periodic dimensions possessing definite symmetry with respect to the interval endpoints, we implement a SinCos basis consisting of either sine waves or cosine waves on the native interval $[0, \pi]$:

$$\phi_k^c(x) = \cos(kx), \tag{26}$$

$$\phi_k^s(x) = \sin(kx), \tag{27}$$

and a native grid consisting of evenly spaced interior points:

$$x_i = \frac{\pi(i + 1/2)}{N_g}, \quad i = 0, \ldots, N_g - 1. \tag{28}$$

Functions with even parity are represented with cosine series as

$$f(x) = \sum_{k=0}^{N_c-1} f_k \phi_k^c(x), \tag{29}$$

while functions with odd parity are represented with sine series as

$$g(x) = \sum_{k=1}^{N_c-1} g_k \phi_k^s(x). \tag{30}$$

The Nyquist mode $k_N = N_c$ is dropped from the sine series, since the corresponding cosine mode vanishes on the grid when $N_g = N_c$.

The expansion coefficients are given explicitly by

$$f_k = \frac{2 - \delta_{k,0}}{\pi} \int_0^\pi f(x)\phi_k^c(x)dx \tag{31}$$

$$= \frac{2 - \delta_{k,0}}{N_g} \sum_{i=0}^{N_g-1} f(x_i)\phi_k^c(x_i), \tag{32}$$

$$g_k = \frac{2}{\pi} \int_0^\pi g(x)\phi_k^s(x)dx \tag{33}$$

$$= \frac{2}{N_g} \sum_{i=0}^{N_g-1} g(x_i)\phi_k^s(x_i) \tag{34}$$

and are computed using the fast discrete cosine transform (DCT) and discrete sine transform (DST). The same grid is used for both series, corresponding to type-II DCT/DSTs for the forward transforms, and type-III DCT/DSTs for the backward transforms. We implement transforms from both the SCIPY and FFTW libraries, and rescale the results to match the above normalizations, i.e., the coefficients directly represent mode amplitudes.

These transforms are defined to act on real arrays, but since they preserve the data-type of their inputs, they can be applied simultaneously to the real and imaginary parts of a complex array. The spectral coefficients for complex functions are therefore also complex, with their real and imaginary parts representing the coefficients of the real and imaginary parts of the function.

C. Chebyshev basis

For finite nonperiodic dimensions, we implement a Chebyshev basis consisting of the Chebyshev-T polynomials on the native interval $[-1, 1]$:

$$T_n(x) = \cos(n \cos^{-1}(x)). \tag{35}$$

The native Chebyshev grid uses the Gauss-Chebyshev quadrature nodes (also known as the *roots* or *interior* grid):

$$x_i = -\cos\left(\frac{\pi(i + 1/2)}{N_g}\right), \quad i = 0, \ldots, N_g - 1. \tag{36}$$

Near the center of the interval, the grid approaches an even distribution where $\Delta x \approx \pi/N_g$. Near the ends of the interval, the grid clusters quadratically and allows very small structures to be resolved (Fig. 4).

A function is represented as

$$f(x) = \sum_{n=0}^{N_c-1} f_n T_n(x). \tag{37}$$

The expansion coefficients are given explicitly by

$$f_n = \frac{2 - \delta_{n,0}}{\pi} \int_{-1}^1 \frac{f(x)T_n(x)}{\sqrt{1 - x^2}} dx \tag{38}$$

$$= \frac{2 - \delta_{n,0}}{N_g} \sum_{i=0}^{N_g-1} f(x_i)T_n(x_i) \tag{39}$$

and are computed using the fast discrete cosine transform (DCT) via the change of variables $x = \cos(\theta)$. The Chebyshev basis uses the sameSCIPY and FFTW DCT functions as the cosine basis, wrapped to handle the sign difference in the change-of-variables and preserve the ordering of the Chebyshev grid points. It also behaves similarly for complex functions, preserving the data type and producing complex coefficients for complex functions.

Chebyshev rational functions can also be used to discretize the half line and the entire real line [12]. These functions are not implemented explicitly in Dedalus, but can be utilized with the Chebyshev basis by manually including changes of variables in the equations.

D. Legendre basis

For finite nonperiodic dimensions, we also implement a Legendre basis consisting of the Legendre polynomials $P_n(x)$ on the native interval $[-1, 1]$. The Legendre polynomials are orthogonal on this interval as

$$\int_1^1 P_n(x)P_m(x)dx = \delta_{n,m}N_n^2, \tag{40}$$

where $N_n^2 = 2/(2n + 1)$.

The native grid points x_i are the Gauss-Legendre quadrature nodes, calculated using scipy.special.roots_legendre.

A function is represented as
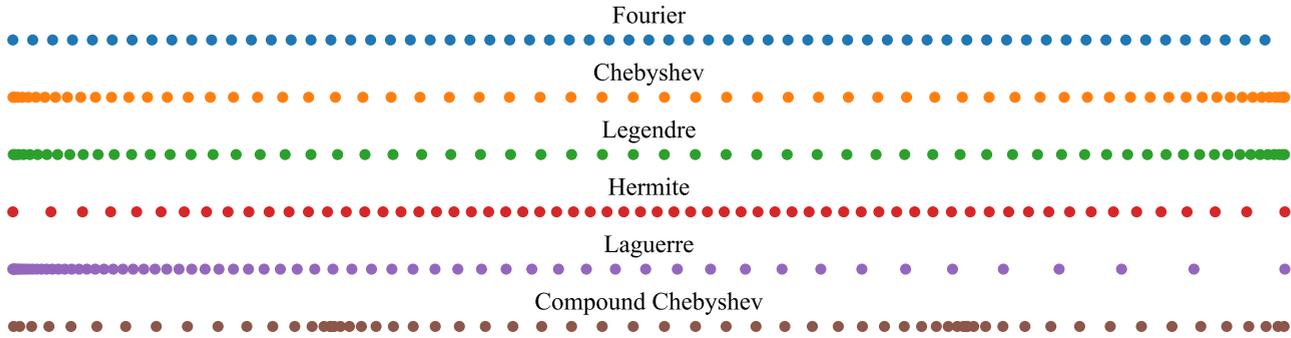
$$f(x) = \sum_{n=0}^{N_c-1} f_n P_n(x). \tag{41}$$

FIG. 4. Example collocation grids for the implemented bases. The `Fourier` basis grid is evenly spaced on a periodic interval. The `Chebyshev` and `Legendre` basis grids cluster near the ends of a finite interval. The `Hermite` basis grid spreads out over the real line. The `Laguerre` basis grid spreads one way over the half line. The `Compound` basis concatenates other bases on adjacent segments; e.g., three Chebyshev segments.

The expansion coefficients are given by

$$f_n = \frac{1}{N_n^2} \int_{-\infty}^{\infty} f(x)P_n(x)dx \qquad (42)$$

$$= \frac{1}{N_n^2} \sum_{i=0}^{N_g-1} w_i f(x_i)P_n(x_i), \qquad (43)$$

where w_i are the Gauss-Legendre quadrature weights, also computed using `scipy.special.roots_legendre`. We synthesize the basis functions with the standard recursion relations in extended precision, which prevents underflows or overflows in problems with large numbers of modes. Quadrature-based matrix-multiply transforms (MMTs) convert between grid values and coefficients.

E. Hermite basis

For problems on the whole real line $(-\infty, \infty)$, we implement a `Hermite` basis consisting of the physicists' Hermite polynomials $H_n(x)$ and the normalized Hermite functions

$$\phi_n^H(x) = e^{-x^2/2}H_n(x)/N_n, \qquad (44)$$

where $N_n^2 = \pi^{1/2}2^n n!$. The Hermite polynomials are orthogonal under the Gaussian weight as

$$\int_{-\infty}^{\infty} H_n(x)H_m(x)e^{-x^2}dx = \delta_{n,m}N_n^2. \qquad (45)$$

The "enveloped" Hermite functions incorporate the weight and normalizations so that

$$\int_{-\infty}^{\infty} \phi_n^H(x)\phi_m^H(x)dx = \delta_{n,m}. \qquad (46)$$

Since the Hermite functions exist over the entire real line, the affine map from native to problem coordinates is fixed by specifying `center` and `stretch` parameters, rather than specifying a problem interval. The native grid points x_i are the Gauss-Hermite quadrature nodes, calculated using `scipy.special.roots_hermite`.

Polynomial functions are represented in the standard basis as

$$f(x) = \sum_{n=0}^{N_c-1} f_n H_n(x), \qquad (47)$$

while functions that decay towards infinity are represented in the enveloped basis as

$$g(x) = \sum_{n=0}^{N_c-1} g_n\phi_n^H(x). \qquad (48)$$

The expansion coefficients are

$$f_n = \frac{1}{N_n^2} \int_{-\infty}^{\infty} f(x)H_n(x)e^{-x^2}dx \qquad (49)$$

$$= \frac{1}{N_n^2} \sum_{i=0}^{N_g-1} w_i f(x_i)H_n(x_i), \qquad (50)$$

$$g_n = \int_{-\infty}^{\infty} g(x)\phi_n^H(x)dx \qquad (51)$$

$$= \sum_{i=0}^{N_g-1} w_i e^{x_i^2} g(x_i)\phi_n^H(x_i), \qquad (52)$$

where w_i are the Gauss-Hermite quadrature weights, also computed using `scipy.special.roots_hermite`. We synthesize the basis functions with the standard recursion relations in extended precision, which prevents underflows or overflows in problems with hundreds of modes. Quadrature-based matrix-multiply transforms (MMTs) convert between grid values and coefficients.

F. Laguerre basis

For problems on the half real line $(0, \infty)$, we implement a `Laguerre` basis consisting of the standard Laguerre polynomials $L_n(x)$ and the normalized Laguerre functions

$$\phi_n^L(x) = e^{-x/2}L_n(x). \qquad (53)$$

The Laguerre polynomials are orthonormal under the exponential weight:

$$\int_0^{\infty} L_n(x)L_m(x)e^{-x}dx = \delta_{n,m}. \qquad (54)$$

The enveloped functions incorporate the weight so that

$$\int_0^{\infty} \phi_n^L(x)\phi_m^L(x)dx = \delta_{n,m}. \qquad (55)$$

Since the Laguerre functions exist over the positive half line, the affine map from native to problem coordinates is fixed by specifying edge and stretch parameters, rather than specifying a problem interval. A negative value of the stretch parameters can be used to create a basis spanning the negative half line. The native grid points x_i are the Gauss-Laguerre quadrature nodes, calculated using scipy.special.roots_laguerre.

Polynomial functions are represented in the standard basis as

$$f(x) = \sum_{n=0}^{N_c-1} f_n L_n(x), \qquad (56)$$

while functions that decay towards infinity are represented in the enveloped basis as

$$g(x) = \sum_{n=0}^{N_c-1} g_n \phi_n^L(x). \qquad (57)$$

The expansion coefficients are

$$f_n = \int_0^\infty f(x) L_n(x) e^{-x} dx \qquad (58)$$

$$= \sum_{i=0}^{N_g-1} w_i f(x_i) L_n(x_i), \qquad (59)$$

$$g_n = \int_0^\infty g(x) \phi_n^L(x) dx \qquad (60)$$

$$= \sum_{i=0}^{N_g-1} w_i e^{x_i} g(x_i) \phi_n^L(x_i), \qquad (61)$$

where w_i are the Gauss-Laguerre quadrature weights, also computed using scipy.special.roots_laguerre. We synthesize the basis functions with the standard recursion relations in extended precision, which prevents underflows or overflows in problems with hundreds of modes. MMTs convert between grid values and coefficients.

G. Compound bases

An arbitrary number of adjacent polynomial segments can be connected to form a Compound basis. The spectral coefficients on each subinterval are concatenated to form the compound coefficient vector, and the standard transforms operate on each subinterval. The compound basis grid is similarly the concatenation of the subinterval grids. There are no overlapping gridpoints at the interfaces since the polynomial bases use interior grids. Continuity is not required a priori at the interfaces, but is imposed on the solutions when solving equations (see Sec. IX A).

The subintervals making up a compound basis may have different resolutions and different lengths, but must be adjacent. Compound bases are useful for placing higher resolution (from clustering near the endpoints of polynomial grids) at fixed interior locations. Compound expansions can also substantially reduce the number of modes needed to resolve a function that is not smooth if the positions where the function becomes nondifferentiable are known. Figure 4 shows the grid of a compound basis composed of three Chebyshev segments.

H. Scaled transforms and dealiasing

Each basis implements transforms between N_c coefficients and scaled grids of size $N_g = sN_c$, where s is the *transform scale* . When $s < 1$, the coefficients are truncated after the first N_g modes before transforming. Such transforms are useful for viewing compressed (i.e., filtered) versions of a field in grid space. When $s > 1$, the coefficients are padded with $N_g - N_c$ zeros above the highest modes before transforming. Padding is useful for spectral interpolation, i.e., to view low resolution data on a fine grid.

Transforms with $s > 1$ are necessary to avoid aliasing errors when calculating nonlinear terms, such as products of fields, in grid space. For each basis, the dealias scale is set at instantiation and defines the transform scale that is used when evaluating mathematical operations on fields. The well-known "3/2 rule" states that properly dealiasing quadratic nonlinearities calculated on the grid requires a transform scale of $s \geqslant 3/2$. In general, an orthogonal polynomial of degree $N_g + n$ will alias down to degree $N_g - n$ when evaluated on the collocation grid of size N_g. A nonlinearity of order P involving expansions up to degree $N_c - 1$ will have power up to degree $P(N_c - 1)$. For this maximum degree to not alias down into degree $N_c - 1$ of the product, we must have $2N_g > (P + 1)(N_c - 1)$. Picking a dealias scale of $s = (P + 1)/2$ is therefore sufficient to evaluate the nonlinearity without aliasing errors in the first N_c coefficients. Nonpolynomial nonlinearities, such as negative powers of fields, cannot be fully dealiased using this method, but the aliasing error can be reduced by increasing s.

I. Transform plans

To minimize code duplication and maximize extensibility, our algorithms require that each transform routine can be applied along an arbitrary axis of a multidimensional array. SCIPY transforms include this functionality, and we built Cython wrappers around the FFTW Guru interface to achieve the same. The wrappers produce plans for FFTs along one dimension of an arbitrary dimensional array by collapsing the axes before and after the transform axis, and creating an FFTW plan for a two-dimensional loop of rank-1 transforms. For example, to transform along the third axis of a five-dimensional array of shape $(N_1, N_2, N_3, N_4, N_5)$, the array would be viewed as a three-dimensional array of shape $(N_1 N_2, N_3, N_4 N_5)$ and a loop of $N_1 N_2 \times N_4 N_5$ transforms of size N_3 would occur. This approach allows for the unified planning and evaluation of transforms along any dimension of an array of arbitrary dimension, reducing the risk of coding errors that might accompany treating different dimensions of data as separate cases.

The plans produced by FFTW are cached by the corresponding basis objects and executed using the FFTW new-array interface. This centralized caching of transform plans reduces both precomputation time and the memory footprint necessary to plan FFTW transforms for many data fields. The FFTW planning rigor, which determines how much precomputation should be performed to find the optimal transform algorithm, is also wrapped through the Dedalus configuration interface.

V. DOMAINS

`Domain` objects represent physical domains, discretized by the direct product of one-dimensional spectral bases. A Dedalus simulation will typically contain a single domain object, which functions as the overall context for fields and problems in that simulation. A domain is instantiated with a list of basis objects forming this direct product, the data type of the variables on the domain [double precision real (64-bit) or complex (128-bit) floating point numbers], and the process mesh for distributing the domain when running Dedalus in parallel.

A. Parallel data distribution

Computations in Dedalus are parallelized by subdividing and distributing the data of each field over the available processes in a distributed-memory MPI environment. The domain class internally constructs a `Distributor` object that directs the decomposition and communication necessary to transform the distributed fields between grid space and coefficient space. Specifically, a domain can be distributed over any lower-dimensional array of processes, referred to as the process mesh. The process mesh must be of lower dimension than the domain so that at least one dimension is local at all times. Spectral transforms are performed along local dimensions and parallel data transpositions change the data locality to enable transforms across all dimensions.

To coordinate this process, the distributor constructs a series of `Layout` objects describing the necessary transform and distribution states of the data between coefficient space and grid space. Consider a domain of dimension D and shape $(N_1, N_2, \ldots, N_D)$ distributed over a process mesh of dimension $P < D$ and shape $(M_1, M_2, \ldots, M_P)$.

(1) The first layout is full coefficient space, where the first P array dimensions are block-distributed over the corresponding mesh axes, and the last $D - P$ dimensions are local. That is, the ith dimension is split in adjacent blocks of size $B_{i,i} = \text{ceil}(N_i/M_i)$, and the process with index $(m_1, m_2, \ldots, m_P)$ in the mesh will contain the data block from $m_i B_{i,i} : (m_i + 1)B_{i,i}$ in the ith dimension.

(2) The subsequent $D - P$ layouts sequentially transform each dimension to grid space starting from the last dimension and moving backwards.

(3) After $D - P$ transforms, the first P dimensions are distributed and in coefficient space, and the last $D - P$ dimensions are local and in grid space. A global data transposition makes the Pth dimension local in the next layout. This transposition occurs along the Pth mesh axis, gathering the distributed data along the Pth array dimension and redistributing it along the $(P + 1)$th array dimension. This is an all-to-all communication within each one-dimensional subset of processes in the mesh defined by fixed $(m_1, \ldots, m_{P-1})$.

(4) The next layout results from transforming the Pth dimension (now local) to grid space.

(5) The transposition step then repeats to reach the next layout: all-to-all communication transposes the $(P - 1)$ and Pth array dimensions over the $(P - 1)$th mesh axis.

(6) The next layout results from transforming the $(P - 1)$th dimension (now local) to grid space.

(7) This process repeats, reaching new layouts by alternately gathering and transforming sequentially lower dimensions until the first dimension becomes local and is transformed to grid space.

The final layout is full grid space. The first dimension is local, the next P dimensions are distributed in blocks of size $B_{n,n-1} = \text{ceil}(N_n/M_{n-1})$, and the final $D - P - 1$ dimensions are local. Moving from full coefficient space to full grid space thus requires D local spectral transforms and P distributed array transpositions. This sequence defines a total of $D + P + 1$ data layouts.

Figure 5 shows the data distribution in each layout for 3D data distributed over a process mesh of shape $(4, 2)$. The layout system provides a simple, well-ordered sequence of transform/distribution states that can be systematically constructed for domains and process meshes of any dimension and shape. Conceptually, the system propagates the first local dimension down in order for each spectral transform to be performed locally. Care must be taken to consider edge cases resulting in empty processes for certain domain and process shapes. In particular, if $(M_i - k)B_{i,i} > N_i$ or $(M_i - k)B_{i+1,i} > N_{i+1}$ for any mesh axis i, then the last k hyperplanes along the ith axis of the mesh will be empty. For instance, if $M_1 = 4$ and $N_1 = 9$, then the initial block size along the lowest dimension will be $B_{1,1} = 3$, and therefore processes with $m_1 = 4$ will be empty. These cases are typically avoidable by choosing a different process mesh shape for a fixed number of processes.

For simplicity, we discussed fixed-shape global data throughout the transform process. The implementation also handles arbitrary transform scales along each dimension, meaning $N_i = N_{c,i}$ in coefficient space, and $N_i = N_{g,i} = s_i N_{c,i}$ in grid space. The default process mesh is one-dimensional and contains all available MPI processes.

B. Transpose routines

Consider the first transposition when moving from coefficient space to grid space, i.e., transposing the P and $(P + 1)$th array dimensions over the Pth mesh axis. This transposition does not change the data distribution over the lower mesh axes; it consists of separate all-to-all calls within each one-dimensional subset of processes defined by fixed $(m_1, \ldots, m_{P-1})$.

The transposition is planned by first creating separate subgroup MPI communicators consisting of each group of M_p processes with the same $(m_1, \ldots, m_{P-1})$. Each communicator plans for the transposition of an array with the *global subgroup shape* $(B_{1,1}, \ldots, B_{P-1,P-1}, N_P, N_{P+1}, \ldots, N_D)$, i.e., the subspace of the global data spanned by its subgroup processes. This array is viewed as a four-dimensional array with the *reduced global subgroup shape* $(B_{1,1} \times \cdots \times B_{P-1,P-1}, N_P, N_{P+1}, N_{P+2} \times \cdots \times N_D)$, constructed by collapsing the pre- and post-transposition dimensions. In this way, the general case of transposing a D-dimensional array distributed over a P-dimensional process mesh along an arbitrary mesh axis is reduced to the problem of transposing a four-dimensional array across its middle two dimensions.

When transposing the distribution along the pth mesh axis between the pth and $(p + 1)$th array dimensions, the global
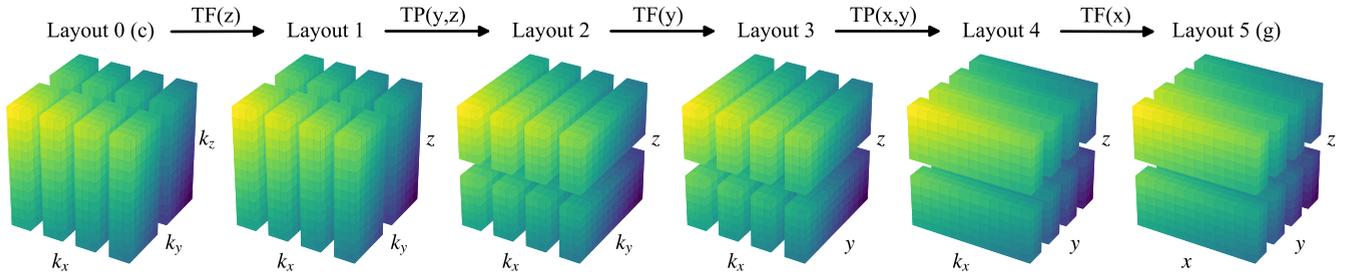
FIG. 5. The parallel data distribution for 3D data over a process mesh of shape $(4, 2)$. The global data is depicted as being split into the portions that are local to each process. The dimensions are labeled e.g., k_x when the corresponding dimension is in coefficient space, and, e.g., x in grid space. The transforms (TF) and transpositions (TP) stepping between layouts are indicated.

subgroup shape is given by $(n_1, \ldots, n_d)$ with

$$n_i = \begin{cases} B_{i,i}, & i < p \\ N_i, & i = p,\, p+1 \\ B_{i,i-1}, & p+1 < i \leqslant P+1 \\ N_i, & i > P+1 \end{cases}, \quad (62)$$

where the first p array dimensions are distributed over the corresponding mesh axes, the pth and $(p+1)$th array dimensions are alternating between being local and distributed over the pth mesh axis, the following $P - p$ array dimensions have already undergone a transposition and are distributed over the corresponding mesh axes less one, and the remaining array dimensions are local. This global shape is collapsed to the reduced global subgroup shape (G_1, G_2, G_3, G_4) where

$$G_1 = \prod_{i=1}^{p-1} n_i, \quad G_2 = n_p, \quad G_3 = n_{p+1}, \quad G_4 = \prod_{i=p+2}^{D} n_i. \quad (63)$$

Routines using either MPI or FFTW are available for performing the reduced data transpositions. The MPI version begins with the local subgroup data of shape $(G_1, B_{p,p}, N_{p+1}, G_4)$ and splits this data into the blocks of shape $(G_1, B_{p,p}, B_{p+1,p}, G_4)$ to be distributed to the other processes. These blocks are then sequentially copied into a new memory buffer so that the data for each process is contiguous. A MPI all-to-all call is then used to redistribute the blocks from being row-local to column-local, in reference to the second and third axes of the reduced array. Finally, the blocks are extracted from the MPI buffer to form the local subgroup data of shape $(G_1, N_p, B_{p+1,p}, G_4)$ in the subsequent layout. The FFTW version performs a hard (memory-reordering) local transposition to rearrange the data into shape (G_2, G_3, G_1, G_4), and uses FFTW's advanced distributed-transpose interface to build a plan for transposing a matrix of shape (G_2, G_3) with an itemsize of $G_1 \times G_4 \times Q$, where Q is the actual data itemsize.

Figure 6 shows the conceptual domain redistribution strategy for the transposition between layouts 1 and 2 of the example shown in Fig. 5. Both the MPI and FFTW implementations require reordering the local data in memory before communicating. However, they provide simple and robust implementations encompassing the general transpositions required by the layout structure. The MPI implementation serves as a low-dependency baseline, while the FFTW

routines leverage FFTW's internal transpose optimization to improve performance when an MPI-linked FFTW build is available. The FFTW planning rigor and in-place directives for the transpositions are wrapped through the Dedalus configuration interface.

These routines can also be used to group transpositions of multiple arrays simultaneously. Transposing S-many arrays concatenates their local subgroup data and the reduced global subgroup shape is expanded to $(S \times G_1, G_2, G_3, G_4)$. A plan is constructed and executed for the expanded shape. This concatenation allows for the simultaneous transposition of multiple arrays while reducing the latency associated with initiating the transpositions. The option to group multiple transpositions in this manner is controlled through the Dedalus configuration interface.

C. Distributed data interaction

For arbitrary transform scalings in each dimension, the layout objects contain methods providing: the global data shape, local data shape, block sizes, local data coordinates, and local data slices for `Field` objects. These methods provide the user with the tools necessary to understand the data distribution at any stage in the transformation process. This is useful for both analyzing distributed data and initializing distributed fields using stored global data.

The domain class contains methods for retrieving each process's local portion of the N-dimensional coordinate grid and spectral coefficients. These local arrays are useful for initializing field values in either grid space or coefficient space. Code that initializes field data using these local arrays is
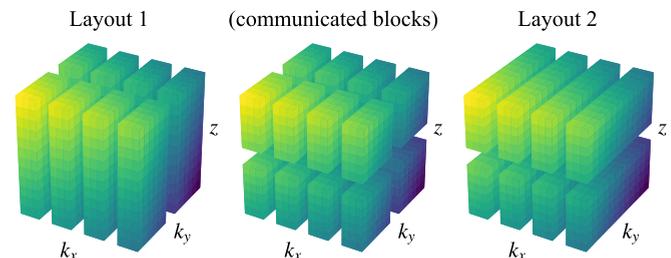

FIG. 6. The effective data redistribution that occurs during the distributed transposition between layouts 1 and 2 of the example shown in Fig. 5. This transposition is switching the second mesh axis with $M_2 = 2$ from distributing k_y to distributing z.

robust to changing parallelization scenarios, allowing scripts to be tested serially on local machines and then executed on large systems without modification.

VI. FIELDS

`Field` objects represent scalar-valued fields defined over a domain. Each field object contains a metadata dictionary specifying whether that field is constant along any axis, the scales along each axis, and any other metadata associated with specific bases (such as 'parity' for the sine/cosine bases or 'envelope' for Hermite and Laguerre bases). When the transform scales are specified or changed, the field object internally allocates a buffer large enough to hold the local data in any layout for the given scales. Each field also contains a reference to its current layout, and a data attribute viewing its memory buffer using the local data shape and type.

A. Data manipulation

The `Field` class defines a number of methods for transforming individual fields between layouts. The most basic methods move the field towards grid or coefficient space by calling the transforms or transpositions to increment or decrement the layout by a single step. Other methods direct the transformation to a specific layout by taking sequential steps. These methods allow users to interact with the distributed grid data and the distributed coefficient data without needing to know the details of the distributing transform mechanism and intermediate layouts.

The `__getitem__` and `__setitem__` methods of the field class allow retrieving or setting the local field data in any layout. Shortcuts 'c' and 'g' allow fast access to the full coefficient and grid data, respectively. To complete a fully parallelized distributed transform:

```
f = Field(domain=domain)
f['g'] = ...  # Set local grid data
f['c']  # Returns local coefficients
```

The `set_scales` method modifies the transform scales:

```
f.set_scales(10)  # Set transform scales
f['g']  # Returns 10x spectral interpolant
```

B. Field systems

The `FieldSystem` class groups together a set of fields. The class provides an interface for accessing the coefficients corresponding to the same transverse mode, or pencil, of a group of fields. A *transverse mode* is a specific product of basis functions for the first $D - 1$ dimensions of a domain, indexed by a multi-index of size $D - 1$. Each transverse mode has a corresponding 1D *pencil* of coefficients along the last axis of a field's coefficient data. The linear portion of a PDE that is uncoupled across tranverse dimensions splits into separate matrix systems for each transverse mode.

A `FieldSystem` of S fields will build an internal buffer of size

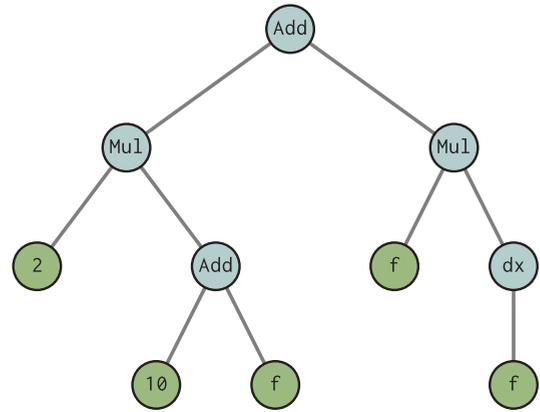$$(B_{1,1}, \ldots, B_{P,P}, N_{P+1}, \ldots, N_{D-1}, N_D \times S). \quad (64)$$

FIG. 7. An operator tree representing the expression `2 * (10 + f) + f * dx(f)`.

That is, the local coefficient shape with the last axis size multiplied by the number of fields. The system methods `gather` and `scatter` copy the separate field coefficients into and out of this buffer. Each size-$N_D \times S$ system pencil contains the corresponding field pencils, grouped along the last axis, in a contiguous block of memory for efficient access.

A `CoeffSystem` allocates and controls just the unified buffer rather than also instantiating field objects. Coefficient systems are used as temporary arrays for all pencils, avoiding the memory overhead associated with instantiating new field objects.

VII. OPERATORS

The `Operator` classes represent mathematical operations on fields, such as arithmetic, differentiation, integration, and interpolation. An operator instance represents a specific mathematical operation on a field or set of fields. Operators can be composed to build complex expressions. The operator system serves two simultaneous purposes: (1) It allows the deferred and repeated evaluation of arbitrary operations and (2) it can produce matrix forms of linear operations. Together, these features allow the implicit and explicit evaluation of arbitrary expressions, which is the foundation of Dedalus' ability to solve general PDEs.

A. Operator classes and evaluation

Operators accept operands (fields or operators) from the same domain, as well as other arguments such as numerical constants or strings. Each operator class implements methods determining the metadata of the output based on the inputs; e.g., the output parity of a `SinCos` basis. Operator classes also have a `check_conditions` method that checks if the operation can be executed in a given layout. For example, spectral differentiation along some dimensions requires that dimension to be in coefficient space as well as local for derivatives that couple modes. Finally, operators have an `operate` method which performs the operation on the local data of the inputs once they have been placed in a suitable layout.

Operators can be combined to build complex expressions. An arbitrary expression belongs to the root operator class, with operands that belong to other operator classes, eventually with fields or input parameters forming the leaves at the end of the expression tree (see Fig. 7). The `evaluate` method

computes compound operators by recursively evaluating all operands, setting the operands' transform scales to the dealias scales, transforming the operands to the proper layout, and calling the `operate` method. Arbitrary expression trees are evaluated in a depth-first traversal. The `evaluate` method can optionally cache its output if it may be called multiple times before the values of the leaves change. The `attempt` method tries to evaluate a field, but will not make any layout changes while evaluating subtrees. It therefore evaluates an expression as much as possible given the current layouts of the involved fields. Finally, operators also implement a number of methods allowing for algebraic manipulation of expressions, described in Sec. VII F.

B. Arithmetic operators

The `Add`, `Multiply`, and `Power` classes implement addition, multiplication, and exponentiation respectively. Different subclasses of these operators are invoked depending on the types of input. A PYTHON metaclass implements this multiple dispatch system, which examines the arguments before instantiating an operator of the proper subclass. For example, the `AddFieldField` subclass adds two fields by adding the local data of each field. The operation can be evaluated as long as both fields are in the same layout. Addition between fields requires compatible metadata, e.g., the same `parity` or `envelope`, settings. The `AddScalarField` class likewise adds a constant to a field. Multiplication and exponentiation of fields must occur in grid space, but otherwise have similar implementations.

The overloaded `__add__`, `__mul__`, and `__pow__` methods allow for easy arithmetic on fields and operators using PYTHON infix operators. For example, with a Dedalus field, `f`, the expression `f + 5` will produce an `AddScalarField` instance. We also override the `__neg__`, `__sub__`, and `__truediv__` methods for negation, subtraction, and division:

```
-f      # Becomes Multiply(-1, f)
f - g   # Becomes Add(f, Multiply(-1, g))
f / g   # Becomes Multiply(f, Power(g, -1))
```

C. Unary grid operators

The `UnaryGridFunction` class implements common nonlinear unary functions: `np.absolute`, `np.sign`, `np.conj`, `np.exp`, `np.exp2`, `np.log`, `np.log2`, `np.log10`, `np.sqrt`, `np.square`, `np.sin`, `np.cos`, `np.tan`, `np.arcsin`, `np.arccos`, `np.arctan`, `np.sinh`, `np.cosh`, `np.tanh`, `np.arcsinh`, `np.arccosh`, and `np.arctanh`. The operation proceeds by applying the function to the local grid space data of the operand. The overloaded `__getattr__` method intercepts Numpy universal function calls on fields and operators and instantiates the corresponding `UnaryGridFunction`. This allows the direct use of Numpy ufuncs to create operators on fields. For example `np.sin(f)` on a Dedalus field `f` will return `UnaryGridFunction(np.sin, f)`.

D. Linear spectral operators

Linear operators acting on spectral coefficients are derived from the `LinearOperator` base class and the `Coupled` or `Separable` base classes if they do or do not couple different spectral modes, respectively. These operators are instantiated by specifying the axis along which the operator is to be applied, which is used to dispatch the instantiation to a subclass implementing the operator for the corresponding basis. These operators implement a `matrix_form` method which produces the matrix defining the action of the operator on the spectral basis functions. For a basis ϕ_n and an operator A, the matrix form of A is

$$A_{ij} = \langle \phi_i | A \phi_j \rangle. \tag{65}$$

For separable operators, this matrix is diagonal by definition, and represented with a one-dimensional array. For coupled operators, this matrix is returned as a SCIPY sparse matrix.

In general, linear operators require their corresponding axis to be in coefficient space to be evaluated. Coupled operators further require that the corresponding axis is local. The local data of the operand is contracted with the matrix form of the operator along this axis to produce the local output data. Operators may override this process by implementing an `explicit_form` method if a more efficient or stable algorithm exists for forward-applying the operator. For example, forward Chebyshev differentiation uses a recursion rather than a matrix multiplication.

1. Differentiation

`Differentiate` classes are implemented for each basis. Differentiation of the `Fourier` and `SinCos` bases is a separable operator and therefore a diagonal matrix. Differentiation of the polynomial bases is a coupled operator. Differentiation of the Hermite polynomials and enveloped functions are both naturally banded and therefore require no conversion into a different test basis to retain sparsity. Differentiation of the Chebyshev, Legendre, and Laguerre bases have dense upper triangular matrices, but these are never used when solving equations. Instead, we always convert the differential equations into a test basis with banded differentiation matrices (see Sec. II B 1). These conversions are applied as left preconditioners for the equation matrices. Appendix 1 shows the full differentiation and conversion matrices. For the Chebyshev, Legendre, and Laguerre bases, forward differentiation uses $\mathcal{O}(N_c)$ recurrence relations rather than dense matrices. Template matrices are rescaled according to the affine map between the native and problem coordinates. The differentiation matrix for the compound basis is the block-diagonal combination of the subbasis differentiation matrices.

For each basis type, a differentiation subclass is referenced from the basis-class `Differentiate` method. These methods are aliased as, e.g., `dx` for a basis with name 'x' during equation parsing. Additionally, a factory function called `differentiate` (aliased as `d`) provides an easy interface for constructing higher-order and mixed derivatives using the basis names, by composing the appropriate differentiation methods:

```
dx(f)  #  Becomes xbasis.Differentiate(f)
d(f, x=2, y=2)  # Becomes dx(dx(dy(dy(f))))
```

The differentiation subclasses also examine the 'constant' metadata of their operand, and return 0 instead

of instantiating an operator if the operand is constant along the direction of differentiation.

2. Integration

Integration is a functional that returns a constant for any input basis series. Integration operators therefore set the 'constant' metadata of their corresponding axes to `True`. The operator matrices are nonzero only in the first row (called the operator vector).

The native integration vectors for each basis are rescaled by the stretching of the affine map between the native and problem coordinates. Integration for the compound basis concatenates each of the sub-basis integration vectors and places the result in the rows corresponding to the constant terms in each sub-basis.

For each basis type, an integration subclass is referenced from the basis-class `Integrate` method. Additionally, a factory function called `integrate` (aliased as `integ`) provides an easy interface for integrating along multiple axes, listed by name, by composing the appropriate integration methods:

```
integ(f, 'x', 'y')
# xbasis.Integrate(ybasis.Integrate(f))
```

If no bases are listed, the field will be integrated over all of its bases. If the operand's metadata indicates that it is constant along the integration axis, the product of the constant and the interval length will be returned.

The `antidifferentiate` method of the `Field` class implements indefinite integration. This method internally constructs and solves a simple linear boundary value problem and returns a new `Field` satisfying a user-specified boundary condition, fixing the constant of integration.

3. Interpolation

Interpolation operators are instantiated with an operand and the interpolation position in problem coordinates. The operator matrices are again nonzero except in the first row (called the operator vector) and depend on the interpolation position. The specified interpolation positions are converted to the native basis coordinates via the basis affine map. The strings 'left', 'center', and 'right' are also acceptable inputs indicating the left endpoint, center point, and right endpoint of the problem interval. Specifying positions in this manner avoids potential floating-point errors when evaluating the affine map at the endpoints.

The interpolation classes construct interpolation vectors consisting of the pointwise evaluation of the respective basis functions. Interpolation for the compound basis takes the interpolation vector of the sub-basis containing the interpolation position and places the result in the rows corresponding to the constant terms in each sub-basis. If the interpolation position is at the interface between two sub-basis, the first sub-basis is used to break the degeneracy.

For each basis type, an interpolation subclass is referenced from the `Interpolate` basis-class method. Additionally, a factory function called `interpolate` (aliased as `interp`) provides an easy interface for interpolating along multiple axes, specified using keyword arguments, by composing the

appropriate integration methods:

```
interp(f, x=0.5, y=1)
# xbasis.Interpolate(
#     ybasis.Interpolate(f, 1), 0.5)
```

If the operand's metadata indicates that it is constant along the interpolation axis, instantiation will be skipped and the operand itself will be returned.

4. Hilbert transforms

The Hilbert transform of a function $f(x)$ is the principal-value convolution with $(\pi x)^{-1}$:

$$H(f)(x) = \frac{1}{\pi} \text{p.v.} \int_{-\infty}^{\infty} \frac{f(x')}{x - x'} dx'. \qquad (66)$$

The Hilbert transform has a particularly simple action on sinusoids,

$$H(\exp(ikx))(x) = -i\text{sgn}(k)\exp(ikx). \qquad (67)$$

The actions on cosine/sine functions result from taking the real/imaginary parts. The Hilbert transform is implemented as a separable operator for the `Fourier` and `SinCos` bases and referenced from the `HilbertTransform` basis-class methods. These methods are aliased as e.g., `Hx` for a basis with name 'x'. Additionally, a factory function called `hilberttransform` (aliased as `H`) provides an easy interface for constructing higher-order and mixed Hilbert transforms using the basis names, similar to the `differentiate` factory function. The Hilbert transform subclasses also examine the 'constant' metadata of their operand, and return `0` instead of instantiating an operator if the operand is constant along axis to be transformed.

E. User-specified functions

The `GeneralFunction` class wraps and applies general PYTHON functions to field data in any specified layout. For example, a user-defined PYTHON function `func(A, B)` which accepts and operates on grid-space data arrays, can be wrapped into a Dedalus `Operator` for deferred evaluation using Dedalus fields `fA` and `fB` as

```
GeneralFunction(func, args=(fA, fB), layout='g')
```

Section XI F gives a detailed example.

F. Manipulating expressions

The operator classes also implement a number of methods allowing the algebraic manipulation of operator expressions, i.e., a simple computer-algebra system. Section VIII describes how this enables the construction of solvers for general partial differential equations.

These methods include the following:

(1) `atoms`: recursively constructs the set of leaves of an expression matching a specified type.

(2) `has`: recursively determines whether an expression contains any specified operand or operator type.

(3) `expand`: recursively distributes multiplication and linear operators over sums of operands containing any specified

operand or operator type. It also distributes derivatives of products containing any specified operand or operator type using the product rule.

(4) `canonical_linear_form`: first determines if all the terms in an expression are linear functions of a specified set of operands, and raises an error otherwise. In the case of nested multiplications, it rearranges the terms so that the highest level multiplication directly contains the operand from the specified set.

(5) `split`: additively splits an expression into a set of terms containing specified operands and operator types, and a set of terms not containing any of them.

(6) `replace`: performs a depth-first search of an expression, replacing any instances of a specified operand or operator type with a specified replacement.

(7) `order`: recursively determines the compositional order of a specified operator type within an expression.

(8) `sym_diff`: produces a new expression containing the symbolic derivative with respect to a specified variable. The derivative is computed recursively via the chain rule.

(9) `as_ncc_operator`: constructs the NCC multiplication matrix associated with multiplication by the expression. It requires that the corresponding domain only have a single polynomial basis, and that this basis forms the last axis of the domain. It further requires that the expression is constant along all other ("transverse") axes, so that multiplication by the operand does not couple the transverse modes. The method first evaluates the expression, then builds the NCC matrix with the resulting coefficients. The method allows the NCC expansion to be truncated at a maximum number of modes ($N_m < N_c$) and for terms to be excluded when the coefficient amplitudes are below some threshold ($|f_n| < \delta$) so that the matrix is sparse for well-resolved functions (see Sec. II B 3).

(10) `operator_dict`: constructs a dictionary representing an expression as a set of matrices acting on the specified pencils of a specified set of variable. This method requires that the expression be linear in the specified variables and contains no operators coupling any dimensions besides the last.

The dictionary is constructed recursively, with each coupled linear operator applying its matrix form to its operand matrices, and each transverse linear operator multiplying its operand matrices by the proper element of its vector form. Addition operators sum the matrices produced by their operands. Multiplication operators build the matrices for the operand containing the specified variables. They then multiply these matrices by the NCC matrix form of the other operand.

G. Evaluators

An `Evaluator` object attempts to simultaneously evaluate multiple operator expressions, or tasks, as efficiently as possible, i.e., with the least number of spectral transforms and distributed transpositions. Tasks are organized into `Handler` objects, each with a criterion for when to evaluate the handler. Handlers can be evaluated on a specified cadence in terms of simulation iterations, simulation time, or real-world time (wall time) since the start of the simulation. Handlers from the `SystemHandler` class organize their outputs into a `FieldSystem`, while handlers from the `FileHandler` class save their outputs to disk in HDF5 files via the h5py package

(Sec. X). The `add_task` method adds tasks to a handler and accepts operator expressions or strings (which are parsed into operator expressions using a specified namespace).

When triggered, the evaluator examines the attached handlers and builds a list of the tasks from each handler scheduled for evaluation. The evaluator uses the `attempt` methods to evaluate the tasks as far as possible without triggering any transforms or transpositions. If the tasks have not all completed, the evaluator merges the remaining atoms from the remaining tasks, and moves them all to full coefficient space, and reattempts evaluation. If the tasks are still incomplete, the evaluator again merges the remaining atoms from the remaining tasks, moves them forward one layout, and reattempts evaluation. This process repeats, with the evaluator simultaneously stepping the remaining atoms back and forth through all the layouts until all of the tasks have been fully evaluated. Finally, the `process` method on each of the scheduled handlers is executed.

This process is more efficient than sequentially evaluating each expression. By attempting all tasks before changing layouts, it makes sure that no transforms or transpositions are triggered when any operators are able to be evaluated. Additionally, it groups together all the fields that need to be moved between layouts so that grouped transforms and transpositions can be performed to minimize overhead and latency.

VIII. PROBLEMS

`Problem` classes construct and represent systems of PDEs. Separate classes manage linear boundary value problems (LBVP), nonlinear boundary value problems (NLBVP), eigenvalue problems (EVP), and initial value problems (IVP). After creating a problem, the equations and boundary conditions are entered in plain text, with linear terms on the left-hand side (LHS) and nonlinear terms on the right-hand side (RHS). The LHS is parsed into a sparse matrix formulation, while the RHS is parsed into an operator tree to be evaluated explicitly.

A. Problem creation

Each problem class is instantiated with a domain and a list of variable names. Domains may have a maximum of one polynomial basis, which must correspond to the last axis. The linear portion of the equations must be no higher than first order in time and coupled derivatives. Auxiliary variables must be added to render the system first order. Optionally, an amplitude threshold and a cutoff mode number can be specified for truncating the spectral expansion of nonconstant coefficients on the LHS. For eigenvalue problems, the eigenvalue name must also be specified; it cannot be 'lambda' since this is a PYTHON reserved word. For initial value problems, the temporal variable name can optionally be specified, but defaults to 't'.

For example, to create an initial value problem for an equation involving the variables u and v, we would write

```
problem = de.IVP(domain, variables=['u', 'v'])
```

B. Variable metadata

Metadata for the problem variables is specified by indexing the `problem.meta` attribute by variable name, axis, and then property.

The most common metadata to set is the 'constant' flag for any dimension, the 'parity' of all variables for each `SinCos` basis, the 'envelope' flag for the `Hermite` and `Laguerre` bases, and the 'dirichlet' flag for recombining the `Chebyshev`, `Legendre`, and `Laguerre` bases (enabled by default). Default metadata values are specified in the basis definitions.

For example, we can set the parity of variables in our problem along the *x* axis with

```
problem.meta['u']['x']['parity'] = 1  # cosine
problem.meta['v']['x']['parity'] = -1 # sine
```

C. Parameters and nonconstant coefficients

Before adding the equations, any parameters (fields or scalars used in the equations besides the problem variables) must be added to the parsing namespace through the `problem.parameters` dictionary. Scalar parameters are entered by value. Nonconstant coefficients (NCCs) are entered as fields with the desired data. NCCs on the LHS can only couple polynomial dimensions; an error will be raised if the 'constant' metadata is not set to `True` for all separable axes.

For example, we would enter scalar and NCC parameters for a 3D problem on a double-Fourier (*x*, *y*) and Chebyshev (*z*) domain as

```
# Scalar parameter
problem.parameters['A'] = 1e-4

# NCC parameter
z = domain.grid(2)
ncc = domain.new_field()
ncc.meta['x', 'y']['constant'] = True
ncc['g'] = z**2
problem.parameters['B'] = ncc
```

D. Substitutions

One of the most powerful features of Dedalus is the ability to define substitutions which act as string-replacement rules to be applied during the equation parsing process. Substitutions can be used to provide short aliases to quantities computed from the problem variables and to define functions similar to PYTHON lambda functions, but with normal mathematical-function syntax.

For example, several substitutions that might be useful in a hydrodynamical simulation are

```
# Substitution defining the kinetic energy
# density for a 3D fluid simulation with
# density rho and velocity (u,v,w).
problem.substitutions['KE_density'] = \
    "rho * (u*u + v*v + w*w) / 2"

# Substitution defining the Cartesian Laplacian
# of a field. Here A and Az are dummy variables
# that would be replaced by simulation variables
# in the equations. Note the system is written
# in first-order form along the z dimension.
problem.substitutions['L(A,Az)'] = \
    "dx(dx(A)) + dy(dy(A)) + dz(Az)"
```

Substitutions of the first type are created by parsing their definitions in the problem namespace, and aliasing the result to the substitution name. Substitutions of the second type are turned into PYTHON lambda functions producing their specified form in the problem namespace. Substitutions are composable and form a powerful tool for simplifying the entry of complex equation sets.

E. Equation parsing

Equations and boundary conditions are entered in plain text using the `add_equation` and `add_bc` methods. Optionally, these methods accept a `condition` keyword, which is a string specifying which transverse modes that equation applies to. This is necessary to close certain equation sets where, for example, the equations become degenerate for the transverse-mean mode and/or certain variables require gauge conditions.

First, the string-form equations are split into LHS and RHS strings which are evaluated over the problem namespace to build LHS and RHS operator expressions. The problem namespace consists of the following:

(1) The variables, parameters, and substitutions defined in the problem.

(2) The axis names representing the individual basis grids.

(3) The derivative, integration, and interpolation operators for each basis.

(4) Time and temporal derivatives for the IVP (defaulting to 't' and 'dt').

(5) The eigenvalue name for the EVP.

(6) The universal functions wrapped through the `UnaryGridFunction` class.

A number of conditions confirming the validity of the LHS and RHS expressions are then checked. For all problem types, the LHS expression and RHS must have compatible metadata (e.g., parities). The LHS expression must be nonzero and linear in the problem variables. The LHS must also be first order in coupled derivatives. The expressions entered as boundary conditions must be constant along the last axis.

For the individual problem classes, the following additional restrictions and manipulations are applied to the LHS and RHS expressions.

1. Linear boundary value problems

The linear boundary value problem additionally requires that the RHS is independent of the problem variables. This allows for linear problems with inhomogeneous terms on the RHS. Since the LHS terms are linear in the problem variables, this symbolically corresponds to systems of equations of the form

$$\mathcal{L} \cdot \mathcal{X} = \mathcal{F}, \tag{68}$$

where $\mathcal{X}$ is the state vector of variable fields, and $\mathcal{L}$ is a matrix of operators. The LHS expressions are expanded and transformed into canonical linear form before being stored by the problem instance.

2. Nonlinear boundary value problems

Nonlinear boundary value problems are systems of the form

$$\mathcal{L} \cdot \mathcal{X} = \mathcal{F}(\mathcal{X}) \tag{69}$$

The RHS can be any nonlinear function of the problem variables. In addition to the $\mathcal{L} \cdot \mathcal{X}$ and $\mathcal{F}$ expressions, the problem constructs the Frechet differential of the RHS with respect to the problem variables:

$$\mathcal{F}_{\mathcal{X}} \cdot \Delta \mathcal{X} = \partial_\epsilon \mathcal{F}(\mathcal{X} + \epsilon \Delta \mathcal{X})|_{\epsilon=0}. \tag{70}$$

This linear operator indicates the sensitivity, or directional functional derivative, of $\mathcal{F}$ with respect to changes in $\mathcal{X}$ along $\Delta \mathcal{X}$. It is constructed symbolically using the operator methods described in Sec. VII F roughly as

```
dF = 0
for var, pert in zip(vars, perts):
    dFi = F.replace(var, var + ep*pert)
    dFi = dFi.sym_diff(ep)
    dFi = dFi.replace(ep, 0)
    dF += dFi
```

In general, the Frechet derivative of an expression will contain nonconstant coefficients involving the problem variables $\mathcal{X}$, which would generally couple horizontal modes. Therefore, Dedalus only supports 1D NLBVPs. The LHS and Frechet differential expressions are expanded and transformed into canonical linear form before being stored by the problem instance.

3. Eigenvalue problems

The eigenvalue problem requires that the RHS is zero, and that the LHS terms must be linear in or independent of the eigenvalue, which we refer to as σ. This corresponds to systems of equations of the form

$$\sigma \mathcal{M} \cdot \mathcal{X} + \mathcal{L} \cdot \mathcal{X} = 0, \tag{71}$$

which are generalized linear eigenvalue problems. The $\mathcal{M} \cdot \mathcal{X}$ and $\mathcal{L} \cdot \mathcal{X}$ expressions are extracted by splitting the LHS expression on the presence of the eigenvalue variable, before replacing it with 1. These expressions are expanded and transformed into canonical linear form before being stored by the problem instance.

4. Initial value problems

The initial value problem requires that the LHS coefficients are time independent, the LHS is first order in time derivatives, and the RHS has no time derivatives. This corresponds to systems of the form

$$\mathcal{M} \cdot \partial_t \mathcal{X} + \mathcal{L} \cdot \mathcal{X} = \mathcal{F}(\mathcal{X}, t). \tag{72}$$

The $\mathcal{M} \cdot \mathcal{X}$ and $\mathcal{L} \cdot \mathcal{X}$ expressions are extracted by splitting the LHS expression on the presence of the time derivative dummy operator, before replacing it with the identity operator. These expressions are expanded and transformed into canonical linear form before being stored by the problem instance.

IX. SOLVERS

Each problem type has a corresponding solver type which builds the spectral matrices for the problem equations, implements methods for computing the solution to the equations, and stores the solution state as a `FieldSystem`.

A. Matrix construction

The problem classes begin by building the operator matrices for the LHS expression groups ($\mathcal{M} \cdot \mathcal{X}$, $\mathcal{L} \cdot \mathcal{X}$, and/or $\mathcal{F}_{\mathcal{X}} \cdot \Delta \mathcal{X}$). The matrices are constructed by first taking the set of equations and boundary conditions that apply to each pencil and calling the `operator_dict` method on each expression to build the matrices acting on the corresponding coefficients of the problem variables. For each pencil's matrix to be solvable, the number of applicable equations must equal the number of variables in the problem, S. For a given pencil, we refer to the operator matrix from the ith equation acting on the jth variable as, e.g., $L^{i,j}$. Each of these matrices is processed as follows:

(1) If the ith equation is constant along the coupled direction, then all rows except the first of each $L^{i,j}$ are dropped.

(2) If the ith equation contains a coupled derivative on the LHS, each $L^{i,j}$ is left-multiplied by the basis preconditioner matrix, which renders the derivative matrix banded (e.g., the T-to-U conversion for Chebyshev bases; see Sec. II B 1).

(3) By default, if the ith equation contains a coupled derivative on the LHS, the last row of each $L^{i,j}$ is dropped and replaced with one of the boundary conditions. The keyword option `tau` in the `problem.add_equation` method overrides this default behavior and explicitly forces or prevents the solver from replacing the last row in that equation. This implements boundary conditions using the tau method (see Sec. II A 2). For the system to be solvable, it typically requires the same number of boundary conditions as coupled differential equations. The `tau` keyword allows for nonstandard cases, such as problems with singular end points.

(4) If the last basis is compound, the rows corresponding to the final coefficient of each sub-basis, except for the last, are dropped and replaced with internal boundary conditions matching the sub-basis values at each internal interface for each variable. This enforces continuity of all variables across the subsegments.

(5) If the jth variable has been marked as constant along the coupled direction, then all columns except the first of each $L^{i,j}$ are dropped.

(6) If the jth variable has been marked for Dirichlet preconditioning (enabled by default), then each $L^{i,j}$ is right-multiplied by the Dirichlet conversion matrix. This has the effect of rearranging the columns so that the matrix acts on the coefficients of the Dirichlet expansion of the corresponding variable, rendering all Dirichlet boundary conditions banded (see Sec. II B 2).

Finally, the processed operator matrices are joined to produce the full preconditioned pencil matrix $\tilde{L}$. The matrices are interleaved so that the columns and rows are grouped by mode rather than by field (see Sec. II B 4). The bandwidth of the pencil matrix then becomes S times the maximum bandwidth of any of the individual subblocks, which is roughly set

by the bandwidth of the nonconstant coefficient expansions. Interleaved-block matrices P_p^L and P^R combining all the left and right preconditioners, projections, and reorderings are created and stored, since they will need to be applied to the RHS and solution vectors, respectively, when solving the matrix system.

The pencil matrices are stored as SCIPY sparse matrices. The matrices produced for each of the LHS expression groups ($\mathcal{M} \cdot \mathcal{X}$, $\mathcal{L} \cdot \mathcal{X}$, and/or $\mathcal{F}_{\mathcal{X}} \cdot \Delta \mathcal{X}$) are expanded to occupy the union of their sparsity patterns so that they can be added efficiently.

B. Linear boundary value solver

The linear boundary value solver is instantiated from a linear boundary value problem. It first constructs the matrices $\tilde{L}_p = P_p^L L_p P^R$ for each local pencil p from the stored LHS expression group $\mathcal{L} \cdot \mathcal{X}$. Here P_p^L and P^R explicitly indicate that the expression matrices have been preconditioned from the left and right, and the p subscripts indicate that the left preconditioner and expression matrices vary by pencil. The solver then constructs a system handler for evaluating the RHS equation and boundary condition expressions ($\mathcal{F}$).

The solver class contains a `solve` method, which first evaluates the RHS handler for $\mathcal{F}$. At this point, the linear boundary value problem is fully discretized, and conceptually consists of solving an independent matrix problem for each pencil given by

$$L_p X_p = F_p. \tag{73}$$

The equivalent preconditioned system is given by

$$\underbrace{P_p^L L_p P^R}_{\tilde{L}_p} \underbrace{(P^R)^{-1} X_p}_{\tilde{X}_p} = \underbrace{P_p^L F_p}_{\tilde{F}_p}. \tag{74}$$

For each pencil, this system is solved in the following manner:

(1) The RHS vector F_p is constructed by taking the pencil data from the RHS handler.

(2) The pencil's left preconditioner is applied to F_P to produce $\tilde{F}_p$.

(3) Since $\tilde{L}_p$ is sparse and banded, it can be efficiently solved against $\tilde{F}_p$ to produce $\tilde{X}_p$.

(4) The state-vector pencil is recovered from $\tilde{X}_p$ by reapplying the right preconditioner as

$$P^R \tilde{X}_p = P^R (P^R)^{-1} X_p = X_p \tag{75}$$

and the result is assigned to the state-vector `FieldSystem`.

After the RHS is evaluated, this process is trivially parallelized over pencils, with each process performing a series of local sparse matrix solves for its local pencils. The sparsity and bandedness of the matrix $\tilde{L}_p$ makes the linear solve an efficient process, executing in $\mathcal{O}(N_D^c)$ time. Finally, although $(P^R)^{-1}$ is a dense matrix, it never needs to be constructed, as reapplying the sparse P^R matrix to the output of the linear solve reverses the implicit preconditioning of the unknowns.

C. Nonlinear boundary value solver

The nonlinear boundary value solver is instantiated from the nonlinear boundary value problem. It constructs the matrices $\tilde{L}_p$ for each pencil and handlers for evaluating the expressions $\mathcal{F}$ and $\mathcal{L} \cdot \mathcal{X}$.

The solver class contains a `newton_iteration` method, which performs a single iteration of Newton's method to move the state vector towards the nonlinear solution. Conceptually, the Newton step iteratively approaches the solution of the nonlinear problem

$$L_p X_p = F(X)_p \tag{76}$$

by solving for the update δX^n to the state vector that will cause the future state vector $X^{n+1} = X^n + \delta X^n$ to solve the NLBVP when linearized around the current iteration X^n:

$$L_p X_p^{n+1} = F(X^{n+1})_p, \tag{77}$$

$$L_p(X_p^n + \delta X_p^n) = F(X^n + \delta X^n)_p \tag{78}$$

$$\approx F(X)_p + F_X \delta X_p \tag{79}$$

$$\Rightarrow \underbrace{(L_p - F_X)}_{A_p} \delta X_p \approx \underbrace{F(X)_p - L_p X_p}_{B_p}. \tag{80}$$

The Newton iteration begins by evaluating the RHS handlers for $\mathcal{F}$ and $\mathcal{L} \cdot \mathcal{X}$ and building the matrices $\tilde{F}_p$, which discretize the Frechet derivative of $\mathcal{F}$ using the current state vector. For each pencil, the update is then determined in the following manner:

(1) The RHS vector B_p is constructed by combining the pencil data from the RHS handlers.

(2) The left preconditioner is applied to produce $\tilde{B}_p$.

(3) The LHS matrices are combined to produce $\tilde{A}_p$, which is solved against $\tilde{B}_p$ to produce $\delta \tilde{X}_p$.

(4) The right preconditioner is applied to recover δX_p.

(5) The state vector is updated as $X_p \to X_p + \delta X_p$.

We note that the sparse matrix being solved changes at each iteration, since it depends on the evaluation of the Frechet derivative at the current state vector. The magnitude of the perturbations can be monitored to determine when the solver has converged. Convergence can depend sensitively on the initial values of the state vector, but the iterations converge rapidly (quadratically) for sufficiently good starting positions. The initial conditions are set by modifying the fields in the `solver.stateFieldSystem`.

D. Eigenvalue solver

The eigenvalue solver is instantiated from the eigenvalue problem. It constructs the $\tilde{M}_p$ and $\tilde{L}_p$ matrices and solves the eigenvalue problem for a single pencil at a time, storing the resulting eigenvalues and eigenvectors. The class contains a `set_state` method which will set the solver's state vector to the specified eigenmode for visualization or further computation.

The solver class contains two methods for solving the generalized eigenvalue problem for a specified pencil, which conceptually takes the form

$$\sigma M_p X_p + L_p X_p = 0. \tag{81}$$

1. Dense solver

The first is the `solve_dense` method, which converts the LHS matrices to dense arrays and uses the `scipy.linalg.eig` routine to directly solve the full generalized eigenvalue problem. This has the advantage of solving for all of the SN_D^c eigenmodes of the discretized system. However, the computational cost scales as $\mathcal{O}((SN_D^c)^3)$, which becomes prohibitive at large resolutions.

2. Sparse solver

The second is the `solve_sparse` method, which solves for a subset of the eigenmodes near a specified target eigenvalue σ_T. The generalized problem for the preconditioned matrices is first rearranged as a regular eigenvalue problem using a shift and inversion:

$$\underbrace{(\tilde{L}_p + \sigma_T \tilde{M}_p)^{-1} \tilde{M}_p}_{\tilde{A}_p} \tilde{X}_p = -\frac{1}{(\sigma - \sigma_T)} \tilde{X}_p = \lambda_p \tilde{X}_p. \quad (82)$$

A SCIPY sparse linear operator is constructed to represent the left-side operator $\tilde{A}_p$. This is applied to a vector by first applying $\tilde{M}_p$, and then solving the result against $(\tilde{L}_p + \sigma_T \tilde{M}_p)$. This generalized linear operator is then passed to the `scipy.sparse.linalg.eig` routine, which uses the implicitly restarted Arnoldi method in ARPACK to iteratively compute a specified number of eigenmodes with the largest magnitude λ. The right preconditioner is applied to the resulting eigenmodes to recover X_p, and the computed values of λ are inverted and shifted to recover the corresponding σ.

This shift-and-invert formulation allows using sparse regular eigenvalue solvers for the generalized eigenvalue problem, with the requirement that $(\tilde{L}_p + \sigma_T \tilde{M}_p)$ is full rank.

E. Initial value solver

The initial value solver is instantiated from the initial value problem with one of the time-stepping classes as an argument, defining the integrator to be used to step the problem forward in time. The solver constructs the $\tilde{M}_p$ and $\tilde{L}_p$ matrices for each pencil and a handler for evaluating the RHS expressions $\mathcal{F}$.

Conceptually, the discretized problem takes the form

$$M_p \partial_t X_p + L_p X_p = F(X, t)_p \quad (83)$$

where the systems for different pencils are only coupled through the RHS terms. In general, M_p may not be a full-rank matrix, due to the presence of constraint equations and boundary conditions. This system is integrated using mixed implicit-explicit schemes, where the LHS terms are integrated implicitly, and the RHS terms are integrated explicitly. The time-stepping loop is written by the user, allowing for detailed control of and interaction with the model as time-stepping progresses.

1. Initial conditions

The solver's initial state must be set before beginning a simulation. The solver state is stored in the `solver.state` field system, and initial conditions are set by directly modifying the variables in this system before beginning integration,

e.g.,

```
# Set the initial u field
x = domain.grid(0)
u = solver.state['u']
u['g'] = np.cos(x)
```

When possible, it is best to begin a simulation with *consistent* initial conditions that satisfy the constraint equations and boundary conditions; see Sec. XI E. Initial conditions that are inconsistent may introduce persistent errors or stability problems with some time-stepping schemes.

Initial conditions can also be loaded from the analysis files produced by Dedalus (Sec. X) via the `solver.load_state` method. This is particularly useful for restarting simulations from a checkpoint saved by a previous simulation.

2. Time evolution

The `step` method of the initial value solver advances the state by one time step, producing X^{n+1} from X^n, where the superscripts denote the temporal iteration of the state vector. The method accepts the time step `dt` as an argument. The method then gathers the state system, calls the specified integration routine to update the system, and scatters the updated state system back to the field objects. In general, the integration step will evaluate the RHS handler to perform the temporal integration and will simultaneously evaluate any other scheduled handlers (e.g., analysis tasks) attached to the evaluator.

3. Time step determination

The implemented time-stepping schemes accommodate changing the time step between iterations during a simulation. While the user can implement any desired algorithm for determining the time step, the CFL class in the `dedalus.extras.flow_tools` module can help determine what time step might adequately resolve physical timescales in the evolving solution. The `add_frequencies` and `add_velocities` methods allow users to enter expressions corresponding to state-dependent frequencies and velocities of processes in their simulation, using the same string-based parsing system that is used to enter equations. CFL frequencies are derived from the entered velocities by dividing their values by the grid spacing. Internally, the CFL class builds an auxiliary handler to evaluate these frequencies at a specified cadence, and as the simulation runs the suggested time step is determined via the `compute_dt` method as follows:

(1) At each point on the grid, all of the specified frequencies are added.

(2) The maximum total frequency from the entire grid is taken and inverted to determine the CFL time step.

(3) This time step is then multiplied by a *safety factor* (typically 0.1–0.5), specified at the CFL instantiation with the `compute_dtsafety` keyword.

(4) The resulting time step is then bounded to lie with absolute levels set by the `min_dt` and `max_dt` keywords, and a within relative factors of the previous time step set by the `min_change` and `max_change` keywords.

(5) If the fractional change from the previous time step to the newly determined time step is smaller than the `threshold` parameter, the previous time step is returned. Otherwise, the newly determined time step is returned.

The absolute limits can be useful to prevent the time step from vastly overstepping relevant dynamics or grinding to a halt due to a spurious feature of the solution. The relative limits help prevent ill-conditioning that may occur for some schemes when the time step varies too suddenly. The thresholding option allows the time step to be frequently reevaluated but avoids modifying it by inconsequential amounts. This can have significant performance advantages since factorizations of the IVP matrices are stored and reused when the time step remains the same between iterations.

4. Termination

To help determine when a simulations should terminate, the initial value solver implements the `proceed` property, which determines whether any of the following three criteria apply:

(1) The simulation time has exceeded the value assigned to the `solver.stop_sim_time` attribute.

(2) The wall time (in seconds) since the solver was instantiated has exceeded the value assigned to the `solver.stop_wall_time` attribute.

(3) The iteration count has exceeded the value assigned to the `solver.stop_iteration` attribute.

The wall-time stop is particularly useful for stopping simulations before hard time-limits on cluster job submissions have been reached, allowing for clean termination and potential post-processing of the data before a job is terminated by the system.

A simple time-stepping loop in an IVP can take the form

```
while solver.proceed:
    dt = CFL.compute_dt()
    solver.step(dt)
```

This will continue time stepping until any of the specified stopping criteria have been reached, adjusting the time step along the way via the CFL handler.

F. Time steppers

Rather than implementing a single specific time-stepping scheme, Dedalus implements general algorithms for applying mixed implicit-explicit (IMEX) multistep and Runge-Kutta integrators along with a range of specific integrators of each type. These IMEX schemes implicitly integrate the LHS terms and explicitly integrate the RHS terms. This provides temporal stability for linearly stiff equations without requiring iterative algorithms for integrating the nonlinear terms.

1. Multistep IMEX integrators

A general multistep IMEX scheme with s steps temporally discretizes preconditioned systems of the form of Eq. (83) into the general form

$$\sum_{j=0}^{s} a_j \tilde{M}_p \tilde{X}_p^{n-j} + \sum_{j=0}^{s} b_j \tilde{L}_p \tilde{X}_p^{n-j} = \sum_{j=1}^{s} c_j \tilde{F}_p^{n-j}, \qquad (84)$$

where in general the coefficients a_j, b_j, and c_j depend on the time steps separating the steps, $dt^{n-1} = t^n - t^{n-1}$. This expansion is rearranged to solve for the new state X_p^n as

$$\underbrace{(a_0 \tilde{M}_p + b_0 \tilde{L}_p)}_{\tilde{A}_p^n} \tilde{X}_p^n = \underbrace{\sum_{j=1}^{s} c_j \tilde{F}_p^{n-j} - a_j \tilde{M}_p \tilde{X}_p^{n-j} - b_j \tilde{L}_p \tilde{X}_p^{n-j}}_{\tilde{B}_p^n}.$$

$$(85)$$

The `MultistepIMEX` class implements this structure using double-ended queues to store `CoeffSystems` containing $\tilde{M}\tilde{X}$, $\tilde{L}\tilde{X}$, $\tilde{F}$, and dt for the s most recent steps. The class implements a `step` method, called with the latest time step dt^{n-1}, which produces X^n as follows:

(1) The time-step queue is rotated with the newest value replacing the oldest and the scheme coefficients a_j, b_j, c_j are evaluated using the time-step history.

(2) $\tilde{M}\tilde{X}^{n-1}$ and $\tilde{L}\tilde{X}^{n-1}$ are evaluated for all local pencils without building the dense inverse of the right preconditioner as, e.g.,

$$\tilde{L}_p \tilde{X}_p = P_p^L L_p P^R (P^R)^{-1} X_p = P_p^L L_p X_p. \qquad (86)$$

(3) The RHS handler is evaluated and the data for each pencil is left preconditioned and stored in the $\tilde{F}^{n-1}$ coefficient system.

(4) For each pencil, $\tilde{A}_p^n$ is solved against $\tilde{B}_p^n$ to produce $\tilde{X}_p^n$. A matrix solver which stores and reuses factorizations of each $\tilde{A}_p^n$ can reduce the solve time if the coefficients a_0 and b_0 remain unchanged from the previous iteration.

(5) Applying the right preconditioner recovers the state vector X_p^n.

Specific multistep schemes are implemented as subclasses of the `MultistepIMEX` and define the scheme coefficients a_j, b_j, and c_j via the `compute_coefficients` method. Dedalus currently implements a number of Crank-Nicolson leap-frog, Crank-Nicolson Adams-Bashforth, and semi-implicit BDF methods from Wang and Ruuth [98], ranging from first to fourth order schemes. The multistep methods only require a single evaluation of the RHS per iteration. However, since they depend on previous iterations, they cannot run full-order when beginning a simulation. Instead, each scheme falls back on lower-order schemes for the first s iterations of a simulation. Multistep schemes may also become ill-conditioned if the time step is varied abruptly.

2. Runge-Kutta IMEX integrators

A general Runge-Kutta IMEX scheme temporally discretizes preconditioned systems of the form of Eq. (83) by constructing stages indexed by $i = 1, \ldots, s$ as

$$\tilde{M}_p \tilde{X}_p^{n,i} - \tilde{M}_p \tilde{X}_p^{n,0} + dt \sum_{j=0}^{i} H_{i,j} \tilde{L}_p \tilde{X}_p^{n,j} = dt \sum_{j=0}^{i-1} A_{i,j} \tilde{F}_p^{n,j},$$

$$(87)$$

where $\tilde{F}^{n,j}$ is evaluated at time $t^{n,j} = t^{n,0} + dt c_j$, $X^{n,0} = X^n$, and $t^{n,0} = t^n$. The H, A, and c tableaus define a specific scheme. This expansion is rearranged to sequentially solve for

the stages as

$$\underbrace{(\tilde{M}_p + dt H_{i,i} \tilde{L}_p)}_{\tilde{A}_p^i} \tilde{X}_p^{n,i}$$

$$= \tilde{M}_p \tilde{X}_p^{n,0} + dt \underbrace{\sum_{j=0}^{i-1} A_{i,j} \tilde{F}_p^{n,j} - dt \sum_{j=0}^{i-1} H_{i,j} \tilde{L}_p \tilde{X}_p^{n,j}}_{\tilde{B}_p^i}. \quad (88)$$

We implement "globally stiffly accurate" methods where the final stage is the advanced solution, i.e., $X^{n+1} = X^{n,s}$ and $t^{n+1} = t^{n,s} = t^n + dt$. These schemes do not require $\tilde{M}_p$ to be full rank, which it generally is not for Dedalus problems with algebraic constraints and/or boundary conditions.

The `RungeKuttaIMEX` class implements this structure using `CoeffSystems` to store $\tilde{M}\tilde{X}^{n,0}$ as well as $\tilde{L}\tilde{X}^{n,i}$ and $\tilde{F}^{n,i}$ for all of the stages. The class implements a `step` method, called with the time step dt, which produces X^{n+1} as follows:

(1) $\tilde{M}_p \tilde{X}^{n,0}$ is evaluated for all local pencils.

(2) Then for each stage $i = 1, \ldots, s$:

(a) the RHS handler is evaluated and the data for each pencil is left preconditioned and stored in the $\tilde{F}^{n,i-1}$ coefficient system. For each pencil, $\tilde{L}_p \tilde{X}^{n,i-1}$ is also evaluated.

(b) For each pencil, $\tilde{A}_p^i$ is solved against $\tilde{B}_p^i$ to produce $\tilde{X}_p^{n,i}$. A matrix solver which stores and reuses the factorizations of each $\tilde{A}_p^i$ can be used to reduce the solve time if the time step dt has remained unchanged from the previous iteration.

(c) The right preconditioner is applied to recover $X_p^{n,i}$, which is assigned to the state-vector. The solver simulation time is set to $t^{n,0} + dt c_i$.

Specific multistep schemes are implemented as subclasses of the `RungeKuttaIMEX` base class and define the H, A, and c tableaus. Dedalus currently implements a number of first, second, and third-order methods from Ascher *et al.* [99] and Sprague *et al.* [100]. A particular advantage of the Runge-Kutta methods is that they do not depend on any previous iterations of the state variables, so they can take full-order time steps at the beginning of a simulation and trivially accommodate adaptive time stepping. The cost is that the higher-order schemes perform multiple evaluations of the RHS per iteration, but they tend to run stably with larger CFL safety factors than the multistep schemes.

The ease of switching integrators allows users to easily test a variety of schemes to find the best option for their particular problem. In addition, the multistep and Runge-Kutta base classes make it straightforward to implement new time-stepping schemes.

G. Matrix solvers

Dedalus implements a generic interface for matrix solvers in the `dedalus.libraries.matsolvers` module which simplifies the implementation and comparison of different routines. The primary routines are direct sparse matrix solvers from the SUPERLU and UMFPACK libraries, wrapped through `scipy.sparse` package. For each library, we implement fast single-solve routines as well as routines that store and

apply the factorized form of a matrix. These routines enable fast solves against multiple right-hand sides at the expense of initially computing the factorization. This is particularly useful for initial value problems where the time step is fixed or varying slowly.

Additional routines implement a variety of algorithms specialized to banded matrices and block-diagonal matrices, which result from full-Fourier problems and can be efficiently inverted. The solver interface is designed to be easily extensible, allowing users to simply wrap and test new routines for specific problems.

The matrix solver routine can be specified when instantiating a `Solver` object, and the default can be set using the Dedalus configuration interface.

X. ANALYSIS AND POST-PROCESSING

The Dedalus handler system enables saving arbitrary analysis tasks while an initial value problem is running. This system utilizes the same symbolic parsing system as is used to specify equations and efficiently evaluates the analysis tasks alongside the RHS terms on a specified cadence. Post-processing tools simplify merging and interacting with the resulting analysis files.

A. File handlers

After building a initial value solver, instances of the `FileHandler` class can be attached to the solver's evaluator object to coordinate the periodic output of some simulation data to HDF5 files using the h5py library. Each file handler is instantiated with an output directory path and the cadence at which handler's tasks will be evaluated. This cadence can be in terms of any combination of simulation time (specified with `sim_dt`), wall time (specified with `wall_dt`), and iteration (specified with `iter`). Simulation time cadences are often useful for data analysis; wall time cadences are often useful for checkpointing, e.g., saving the full state of a simulation every hour. To limit the file sizes produced by the handler, the outputs are split up into different *sets* over time, each containing some number of *writes* that can be limited with the `max_writes` keyword. For example, to setup a file handler to be evaluated every few iterations:

```
output = solver.evaluator.add_file_handler(
    'output', iter=5, max_writes=100)
```

Multiple file handlers can compute and save different sets of tasks at different cadences. For example, you may want to occasionally save full copies of the state variables for checkpointing, more frequently save snapshots of some variables for visualization, and very frequently save scalar quantities such as the total energy in the simulation.

B. Analysis tasks

Tasks, or expressions to be computed and saved by the file handler, are added to a given handler using the `add_task` method. Tasks are entered in plain text and parsed using the same namespace that is used for equation entry. For each task the output layout, scaling factors, and a name can also be specified. For example, creating a task to evaluate the kinetic

energy density of a flow might look like:

```
output.add_task("0.5*rho*(u**2+v**2+w**2)",
    layout='g', name='KE')
```

For checkpointing, you can also simply specify that all of the state variables should be saved:

```
output.add_system(solver.state, layout='g')
```

C. Post-processing

By default, the output files for each file handler are arranged hierarchically as follows:

(1) At the top level is output directory that was specified when the handler was constructed, e.g., '`./output/`'.

(2) Within this directory are subdirectories for each set of outputs, with the same name plus a set number, e.g., '`output_s1/`'.

(3) Within each subdirectory are HDF5 files containing the local data for each process, with the same name plus a process number, e.g., `output_s1_p0.h5`.

Often it is preferable to deal with the global dataset when performing analysis or visualization in post-processing. The distributed process files can be merged into global files for each set using the `merge_process_files` function from the `dedalus.tools.post` module. For some analyses, it is additionally convenient to merge the output sets together into a single file that is global in space and time, which can be done with the `merge_sets` function. However, this can generate very large files, and is not usually necessary for analyses that are local in time, e.g., individually plotting each output of a task. To assist with performing such tasks in parallel, the `visit_writes` function will coordinate all available processes to apply a given function to each output across all sets from a handler.

Together, the symbolic specification of analysis tasks and helper functions for merging and interacting with the output files can dramatically simplify user interactions with simulation products. High-level plotting functions for plotting slices of fields and tasks are implemented in the `dedalus.extras.plot_tools` module, and example scripts utilizing these tools to construct visualizations in parallel are available. The HDF5 output file format was chosen because it is widely used in the scientific community, and allows users to easily examine and visualize simulation outputs using a wide variety of tools and languages.

XI. BENCHMARKS AND EXAMPLES

We demonstrate the features and performance of the Dedalus codebase with a variety of examples involving different types of PDEs from a variety of fields. The examples and some of the unique features that they demonstrate are the following:

(1) *Parallel scaling*: strong scaling test of an incompressible hydrodynamics IVP across many nodes.

(2) *Kelvin-Helmholtz*: accuracy benchmark of compressible hydrodynamics with a finite-volume code.

(3) *Nonlinear Schrödinger network*: complex-valued PDE on a network of 1D segments, programatic extension of Dedalus interface to form spectral element method.

(4) *Orszag-Tang vortex*: moderate Mach-number vortices in compressible magnetohydrodynamics, regularized and resolved shocks.

(5) *Quasigeostrophic flow*: asymptotically reduced equations for rotating incompressible flow, LBVP to balance initial conditions.

(6) *Cylindrical Stokes flow*: low Reynolds-number flow in an annulus using polar coordinates and nonconstant coefficients.

(7) *Atmospheric waves*: NLBVP to solve for the structure of an atmosphere with radiative diffusion, EVP to examine normal modes.

(8) *Diamagnetic levitation*: electrodynamics and rigid body mechanics, immersed boundaries, ODE integration, nonlocal boundary conditions.

Example scripts for these problems are available online [101]. The project gallery [102] is updated with user contributed examples on an ongoing basis.

A. Parallel scaling

A parallel scaling suite [103] for Dedalus is publicly available. Figure 8 shows performance and parallel scaling results from 32 to 2048 cores for 3D Boussinesq hydrodynamics and magnetohydrodynamics simulations in Fourier-Fourier-Chebyshev domains. The tests were run on the Flatiron Institute's Popeye cluster [104] using 32 Intel Skylake cores per node.

The code's speed, as measured in mode-iterations per core-second, is plotted against the number of cores, as measured in pencils per core. The plateaus for each resolution indicate the regions of ideal strong scaling; large 3D problems are able to scale efficiently to thousands of cores. The roll off at high core counts indicates the end of efficient strong scaling: the parallel efficiency of Dedalus typically remains above 50% down to 8 pencils per core. The ratios of the plateau values for different resolutions indicate that the weak scaling efficiency is proportional to $1/\log_2 N$, as expected for FFT-based computations. Boussinesq hydro with eight variables is twice as fast as Boussinesq MHD with 16 variables, indicating that execution timescales linearly with the number of problem variables.

B. Kelvin-Helmholtz accuracy benchmark

Lecoanet *et al.* [24] performed an accuracy benchmark comparing the finite-volume code Athena [105] and Dedalus. Both codes simulated the Kelvin-Helmholtz instability in a moderate Mach-number compressible flow. At low-to-moderate resolution, numerical errors from the finite-volume method can cause unphysical secondary instabilities to develop within the rolls created by the flow. By directly comparing the nonlinear evolution of the flows at late times, the authors found that the finite-volume method requires a resolution of 16384^2 cells to avoid these spurious instabilities and achieve the same accuracy as Dedalus at a resolution of 2048^2 modes (Fig. 9).
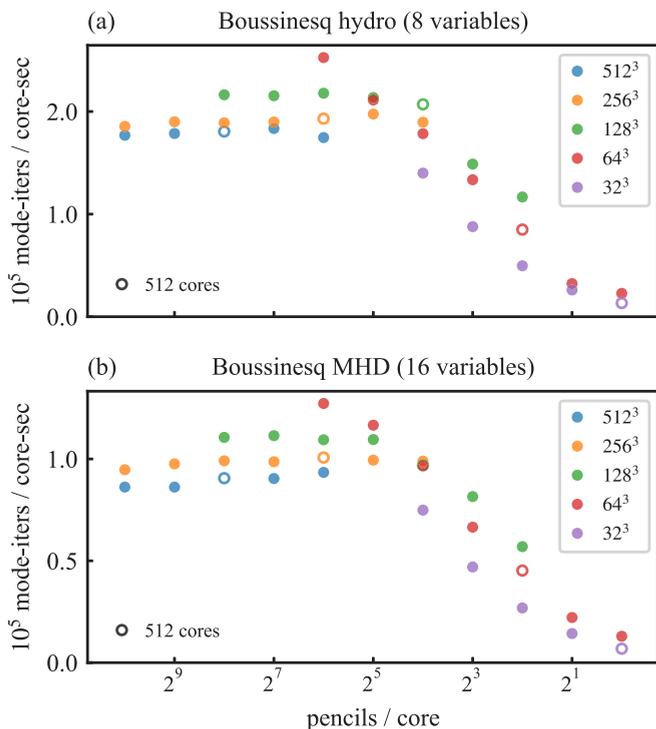
FIG. 8. Dedalus performance (mode-iterations per core-second) vs parallel scaling (pencils per core) from 32 to 2048 cores for 3D Boussinesq (a) hydrodynamics and (b) magnetohydrodynamics. Colors correspond to different 3D (Fourier-Fourier-Chebyshev) resolutions. For each resolution, the open circle indicates the run on 512 cores, with the core count increasing by factors of two to the right. Efficient strong scaling is seen down to roughly 8 pencils per core. Weak scaling shows the expected $\propto 1/\log_2 N$ efficiency for FFT-based computations. Execution timescales linearly with the number of problem variables.

This test demonstrates the power of high-order methods for solving PDEs with smooth solutions. At low-to-moderate Mach numbers with finite dissipation, the flow solution lacks strong shocks and its spectral expansion converges rapidly. Generally, for incompressible and low-Mach-number flows in simple geometries, the rapid convergence of spectral methods outweighs their larger per-iteration computation cost, making them the ideal method for simulating a broad range of astrophysical and geophysical flows.

C. Nonlinear Schrödinger network

The nonlinear Schrödinger equation (NLS) is classical field equation describing the dispersive behavior of wave packets in a weakly nonlinear medium [106]. The *focusing* NLS is a PDE for the complex-valued field ψ given by

$$i\frac{\partial \psi}{\partial t} + \frac{1}{2}\frac{\partial^2 \psi}{\partial x^2} = -\psi |\psi|^2. \tag{89}$$

In this example, we simulate the 1D NSE on a quantum graph, i.e., a network of connected segments with differential equations (see Berkolaiko and Kuchment [107] and Noja [108] for applications). This is achieved using a Dedalus domain with a single Chebyshev segment but separate variables for
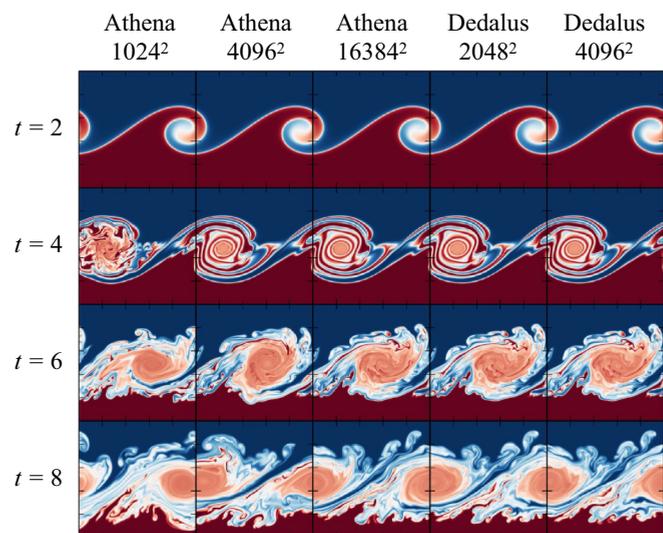
FIG. 9. Snapshots of a moderate Mach-number Kelvin-Helmholtz instability test problem simulated at various resolutions with a finite-volume code (Athena) and Dedalus. The finite-volume method introduces small errors which trigger unphysical secondary instabilities in the vortex rolls. These spurious instabilities disappear as the simulation resolution is increased. Quantitative comparisons show comparable accuracy between the finite-volume method with 16384^2 degrees of freedom and Dedalus with 2048^2 degrees of freedom. Figure adapted from Lecoanet *et al.* [24].

the solution on each segment, each governed by the NSE and coupled through their boundary conditions to mimic the network. This demonstrates how the parsing system in Dedalus can be combined with a simple PYTHON workflow to mimic a spectral element method by "connecting" distinct domains via boundary conditions.

We begin by loading any user-defined network defined by the planar positions of N_v-many vertices and a list of N_e-many directed edges connecting two vertices. We then build a domain with a single 1D, 64-mode Chebyshev basis, which serves as the underlying elemental basis for the spectral element method. We define an IVP by constructing variables and adding equations for each edge. This is done by looping over edges to create variable names encoding the edge index and using PYTHON string substitution to insert these names into the equations. This programatically adds N_e copies of the NLS to the problem, one for each edge, with the derivative operators weighted by the inverse of the corresponding edge length (L_e) to account for the different true lengths of the graph segments:

$$i\frac{\partial \psi_e}{\partial t} + \frac{1}{2L_e^2}\frac{\partial^2 \psi_e}{\partial x^2} = -\psi_e |\psi_e|^2, \quad e = 1, \ldots, N_e. \tag{90}$$

We encode edge-neighbor information using the graph incidence matrix. Continuity of the solution at each vertex is imposed by iteratively matching the solution at each incident edge:

$$\psi_e(v_n) = \psi_{e'}(v_n), \quad \forall\, e,\, e' \text{ incident to vertex } v_n. \tag{91}$$

Finally, Kirchoff's law for conservation of flux is imposed at each vertex by requiring the sum of the gradients of the
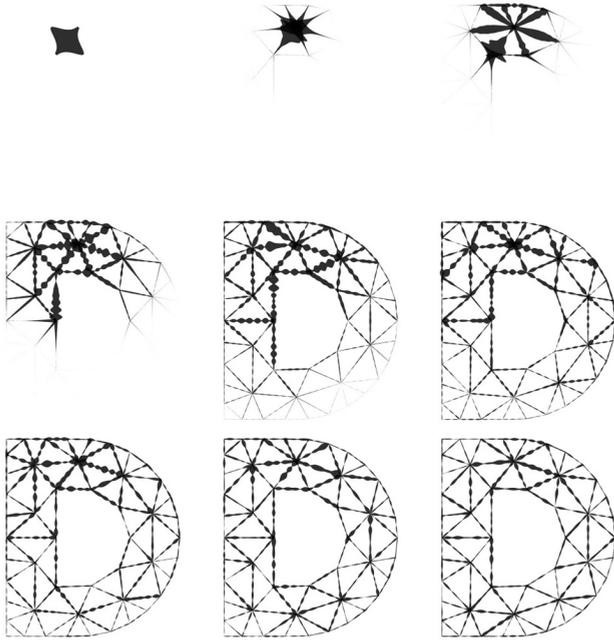
FIG. 10. Evolution of the nonlinear Schrödinger equation on a network, simulated by coupling the boundaries of different fields on a 1D Chebyshev segment. A soliton initially isolated to one segment scatters at the vertices and fills the network over time.

solution on each incident edge, weighted by the edge lengths and signed by whether the edge is incoming or outgoing to that vertex ($\sigma_e = \pm 1$), to be equal to zero:

$$\sum_{e \in v_n} \frac{\sigma_e}{L_e} \frac{\partial \psi_e}{\partial x}\bigg|_{v_n} = 0. \qquad (92)$$

A bright soliton is placed on a single edge as the initial condition, and the problem is integrated forward in time using the SBDF2 time stepper. The soliton translates to the end of the segment then scatters and disperses into the other segments and eventually fills the graph. Snapshots of the evolution are shown in Fig. 10. This example demonstrates the composability of the Dedalus API and the advantage of working within a high-level PYTHON environment.

D. Orszag-Tang vortex

Spectral methods can also correctly solve for high-Mach-number flows which develop shocks, provided that diffusion is introduced in the simulation to regularize the shocks. The Orszag-Tang vortex problem [109] is a standard compressible magnetohydrodynamics (MHD) test problem in astrophysics (e.g., for the fixed-grid Godunov code ATHENA [110,111], the finite difference code FLASH [112], smoothed-particle hydrodynamic codes [113], moving mesh codes [114], etc.). The problem can be simulated with shock-capturing algorithms (e.g., Riemann solvers) or those with numerical diffusivity which can regularize the shocks (e.g., SPH). Here, we explicitly add diffusion of momentum, heat, and magnetic fields to the model to regularize the shocks.

We note that a diffusive shock creates entropy at a rate that is independent of (but mediated by) the microphysical diffusion. For example, in Burger's equation, $\partial_t u + u \partial_x u = \partial_x(\nu \partial_x u)$, the dissipation rate across a shock is

$$\frac{d}{dt}\int \frac{u^2}{2}dx = -\int \nu|\partial_x u|^2 dx \approx -\frac{\Delta u^3}{12}. \qquad (93)$$

The leading order dissipation is *independent of the viscosity*, and the correction is on the order of the inverse Reynolds number with logarithmic corrections. Numerical simulations of shocks should therefore give results that are mostly independent of the viscosity, provided the Reynolds number is large enough. This principle underlies shock-capturing algorithms, but can also be leveraged in spectral simulations. Because of the weak dependence on the Reynolds number, diffusivities much larger than the natural values can be used to regularize shocks while still respecting important global balances. As long as the resulting diffusive-shock length scale $\Delta \ell \sim \nu/\Delta u$ is resolved, a spectral computation will be free of Gibbs ringing and produce accurate results.

We simulate the Orszag-Tang vortex in a 2D domain which is periodic in both x and y directions, and comprises $[0, 1]^2$. A vortex with velocity $\mathbf{u} = (-\sin(2\pi y), \sin(2\pi x))$ is initialized in an ideal gas with constant density and pressure, $\rho = 25/(36\pi)$ and $p = 5/(12\pi)$, and ratio of specific heats $\gamma = 5/3$. The initial magnetic field is specified by a vector potential $A_z = B_0(\cos(4\pi x)/(4\pi) + \cos(2\pi y)/(2\pi))$, with $\mathbf{B} = \nabla \times \mathbf{A}$, and $B_0 = 1/\sqrt{4\pi}$. The adiabatic sound speed $\gamma p/\rho = 1$, so the flow is supersonic in parts of the domain, which leads to the formation of MHD shocks. We solve the equations

$$\partial_t \mathbf{u} + \nabla T' + T_0 \nabla \Upsilon - \nu(\nabla^2 \mathbf{u} + (1/3)\nabla\nabla\cdot\mathbf{u})$$
$$= -\mathbf{u}\cdot\nabla\mathbf{u} - T'\nabla\Upsilon + \nu\nabla\Upsilon\cdot\mathbf{S}$$
$$+ (\mathbf{B}\cdot\nabla\mathbf{B} - (1/2)\nabla|\mathbf{B}|^2)e^{-\Upsilon}, \qquad (94)$$

$$\partial_t \Upsilon + \nabla\cdot\mathbf{u} = -\mathbf{u}\cdot\nabla\Upsilon, \qquad (95)$$

$$\partial_t T' + (\gamma - 1)T_0\nabla\cdot\mathbf{u} - (\chi/c_v)\nabla^2 T'$$
$$= \mathbf{u}\cdot\nabla T' - (\gamma - 1)T'\nabla\cdot\mathbf{u} + (\chi/c_v)\nabla T'\cdot\nabla\Upsilon$$
$$+ (\nu/2c_v)(\mathrm{Tr}(\mathbf{S}^2) + \mathrm{Tr}(\mathbf{S})^2) + (\eta/c_v)|\nabla\times\mathbf{B}|^2 e^{-\Upsilon}, \qquad (96)$$

$$\partial_t A_z - \eta\nabla^2 A_z = \mathbf{e}_z\cdot(\mathbf{u}\times\mathbf{B}). \qquad (97)$$

Here, ν, χ, and η are the viscosity, thermal diffusivity, and magnetic diffusivity, $T_0 = 1/\gamma$ is the background temperature, T' is the temperature perturbation, and $\Upsilon = \ln\rho$. The heat capacity at constant volume $c_v = (\gamma - 1)^{-1} = 3/2$ normalizes the conduction and heating in the energy equation for T'. In the nonlinear viscous terms, the symmetric stress tensor $\mathbf{S} = \nabla\mathbf{u} + (\nabla\mathbf{u})^T - (2/3)\nabla\cdot\mathbf{u}\,\mathbf{I}$. This is an MHD-analog of the equations introduced in Lecoanet *et al.* [23].

We run the simulation at rather high resolution, using 4096 modes in the x and y directions. We use 3/2 dealiasing in each direction, although this does not eliminate aliasing errors from converting from Υ to ρ. We set all the diffusivities equal to 10^{-4}. For time stepping, we use a third-order, four-stage DIRK/ERK method (RK443 of Ascher *et al.* [99]), with an initial time step size of 2.5×10^{-5}. We use adaptive time stepping based on the CFL condition associated with both
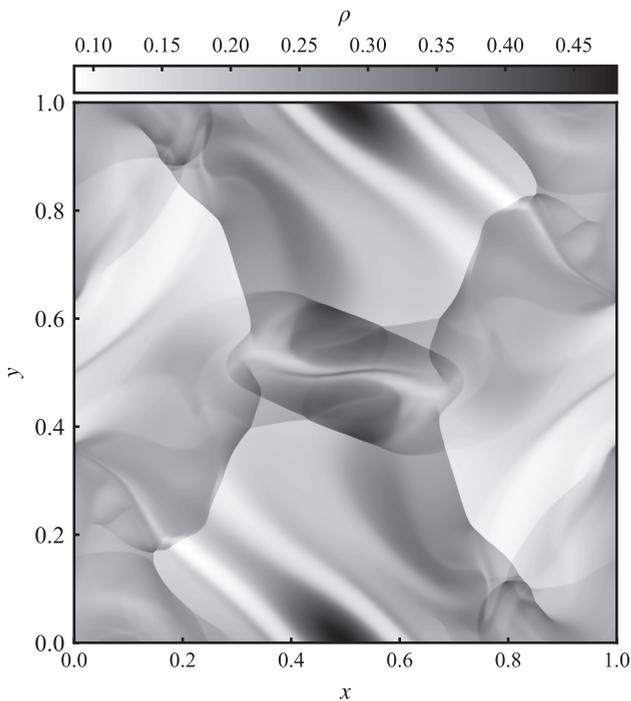
FIG. 11. Density at $t = 0.5$ in the Orszag-Tang vortex test. The sharp density jumps visualize the MHD shocks. The structures in the upper left and lower right corners are due to shock-shock interactions, which are correctly treated in Dedalus.

flow and Alfvén speeds (but not the sound speed), with a safety factor of 0.6. The simulation is run to $t = 1$.

Figure 11 shows the density at $t = 0.5$. There are very sharp density gradients due to the formation of shocks. Even more impressive, Dedalus is able to correctly follow the interaction between multiple shocks, which produces the features in the upper left and lower right corners of the figure. These are similar to what is found in high resolution simulations with shock-capturing codes, e.g., Ref. [111].

To better visualize the shocks, we plot a horizontal profile of the density at $y = 0.3125$ in Fig. 12. At this height, there are shocks at $x \approx 0.3$ and ≈ 0.7. The inset shows the solution very close to $x = 0.7$. The shock is well-resolved by 4–5 grid
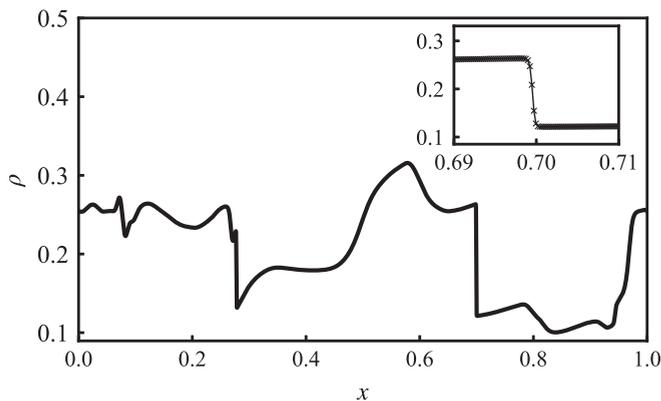
FIG. 12. A profile of the density at $t = 0.5$ and $y = 0.3125$ in the Orszag-Tang vortex test. The inset shows the density profile around $x = 0.7$, with crosses denoting grid points (using the nondealiased grid). The shock is resolved by several grid points.

points. Note that the density stays smooth despite no diffusion in the density equation. Rather, it appears that the combination of viscosity and thermal diffusivity cause the temperature and pressure to regularize, which in turn regularizes the density via the ideal gas equation of state, $p = \rho T$.

E. Quasigeostrophic flow

One of the original motivations behind Dedalus was to allow the straightforward computation of nonstandard equation sets. It is a common practice for modelers to start with something universal, such as Navier-Stokes, or Maxwell equations, make a series of asymptotic reductions, and arrive at a new set of equations that can capture an interesting physical regime. It is often not clear how useful these equations are until they are simulated. But it is risky to write "off-the-shelf" solvers for new equations before there is good evidence that they are useful to at least a few people.

The quasigeostrophic (QG) model is a classic approximation used in geophysical and astrophysical fluid dynamics. Initially, the QG model was intended to reduce the overall cost of simulations. The idea is to filter sub-dominant fast-timescale waves, while retaining essential nonlinear dynamics. Current state-of-the art simulations no longer require the QG assumption on the basis of cost. However the equations are still used widely because of their relative simplicity and explanatory power. QG is a nonstandard model that became widely adopted. Since the advent of QG, other strongly nonlinear reduced models appear in different fields occasionally, but computational results can lag by several years. For example, in stratified fluids [23,115,116], in magnetized fluids [117,118], in Langmuir turbulence [119,120], in near-inertial wave dynamics [121–123], and in rapidly rotating fluids [124–126] (theory) [100,127,128] (simulation).

The solution of the QG equation has a long history starting with numerical weather prediction in the 1950s [129,130]. Under certain restrictive assumptions, the QG problem can be reduced to 2D equations on the surface, e.g., Refs. [131,132], or three-dimensional equations in a triply periodic domain, e.g., Refs. [133,134]. However, the most general case requires a three-dimensional layer with boundaries e.g., Ref. [135], and nonconstant coefficients. In our current work, we pick QG as an interesting example problem because it highlights many features unique to Dedalus. We choose the most difficult and general case of a finite 3D layer because it illustrates the most design features within a single model. Like previous work, it would be straightforward to simplify our example script to model a triply periodic domain, or a two-dimensional layer, if one wanted.

The traditional QG equations collapse to a single, second-order equation for the potential vorticity (PV). This is equivalent to a first-order formulation with two dynamical variables and two first-order equations. A first-order formulation does not increase the computational cost because the second-derivative matrix has double the bandwidth of the first-derivative matrix. Dedalus can use the traditional PV formulation in terms of the streamfunction and its vertical derivative. Here, we describe an alternative formulation choosing more physically meaningful variables, which makes the problem

more straightforward to pose and generalize; at no additional cost.

We solve for two variables: w (upwelling) and p (pressure). All other physical quantities (e.g., buoyancy and velocity) are diagnostic in terms of w and p. The upwelling field is not part of the traditional formulation. Although we solve for two variables instead of a single variable (PV) in the traditional formulation, using the vertical velocity makes the boundary conditions much simpler; we do not need to solve a nonlinear buoyancy advection equation on the boundary.

We use Dedalus substitutions to define several of the important physical variables in terms of the pressure p,

$$u = -\partial_y p, \tag{98}$$

$$v = \partial_x p, \tag{99}$$

$$\theta = \partial_z p, \tag{100}$$

$$\zeta = \partial_x v - \partial_y u = (\partial_x^2 + \partial_y^2)p. \tag{101}$$

Dedalus substitutions can be recursive, e.g., to define ζ. The substitutions reflect the well-known geostrophic and hydrostatic diagnostic balances inherent in the theory. Pressure assumes the role of the stream function, ψ. We subsequently define the advection substitution with an argument,

$$D(q) \equiv u\,\partial_x q + v\,\partial_y q. \tag{102}$$

We also use substitutions to prescribe horizontal fourth-order hyperdiffusion,

$$L(q) \equiv \partial_x^2 q + \partial_y^2 q, \quad \Delta_4(q) = L^4(q). \tag{103}$$

The problem parameters are the Coriolis variation β, the Ekman friction γ, the stratification profile $\Gamma(z)$, the thermal-wind profile $U(z)$, and the hyperdiffusivities ν_4 and κ_4. We solve the coupled prognostic equations,

$$\partial_t \zeta + U\,\partial_x \zeta + \beta v - \partial_z w + \nu_4 \Delta_4(\zeta) = -D(\zeta), \tag{104}$$

$$\partial_t \theta + U\,\partial_x \theta - U'v + \Gamma w + \kappa_4 \Delta_4(\theta) = -D(\theta). \tag{105}$$

The traditional formulation solves the entire buoyancy evolution equation on the boundary. This is a complicated way to impose no vertical flow. Because the upwelling is part of our formulation, we impose the boundary condition directly. More precisely, we impose Ekman flux boundary conditions.

$$w \mp \gamma_\pm \zeta = 0 \quad \text{at} \quad z = 0, 1. \tag{106}$$

Ekman pumping conditions result from a closure model of an asymptotically thin viscous boundary layer [136]. We could easily include boundary stress or topography on the right-hand side.

The $-U'$ shear term in the bulk equation comes from a large-scale background thermal-wind profile

$$\partial_z U = -\partial_y \Theta, \tag{107}$$

where $\Theta(y, z) = y\,S(z)$ is a linear north-south-varying buoyancy profile. Unstable modes extract energy from this thermal variation. The evolution equations are a coupled pair of first-order equations in ∂_z for p and w. There is only one independent time derivative as ζ and θ are both related to p,

as is PV

$$\text{PV} = \zeta + \partial_z(\Gamma^{-1}\theta) = (\partial_x^2 + \partial_y^2 + \partial_z \Gamma^{-1}\partial_z)p \tag{108}$$

Looking at the diagnostic balances, we can see that

$$\partial_z \zeta - (\partial_x^2 + \partial_y^2)\theta = 0. \tag{109}$$

This is the "thermal-wind" balance that follows from geostrophy and hydrostatic balance. The vertical velocity field, w, satisfies its own diagnostic balance at each time step, which removes nonbalanced forcing terms. *The vertical velocity here is exactly analogous to the pressure field for incompressible hydrodynamics*; it is a Lagrange multiplier enforcing the consistent evolution of ζ and θ, which are both related to the pressure.

Balanced initial conditions. We initialize the pressure with filtered random noise on the grid; vorticity and buoyancy follow directly. In practice, this is likely to be sufficient to initialize the simulation. Some time steppers may correct for unbalanced initial conditions. For completeness, we illustrate how to solve a linear boundary-value problem (LBVP) for the upwelling given an initial pressure field.

Knowing $p(t = 0, x, y, z)$, we set up the coupled problem for $w(t = 0, x, y, z)$:

$$L(\varpi_t) - \partial_z w = -D(\zeta) - U\,\partial_x \zeta - \beta v - \nu_4 \Delta_4(\zeta), \tag{110}$$

$$\partial_z \varpi_t + \Gamma w = -D(\theta) - U\,\partial_x \theta + U'v - \kappa_4 \Delta_4(\theta), \tag{111}$$

$$w = \pm\gamma_\pm \zeta \quad \text{at } z = 0, 1. \tag{112}$$

We could collapse these to a single system for w. However, we can mostly reuse the original system if we introduce a *slack variable*, ϖ_t. The slack variable takes the place of the time evolution terms in the full system. Solving this LBVP gives the vertical velocity at $t = 0$ and $\varpi_t = \partial_t p|_{t=0}$. This general approach can be used to derive balanced initial conditions for other equations with time-independent constraints, e.g., incompressible hydrodynamics.

The simulation. We nondimensionalize z with the domain depth, H. We nondimensionalize x, y with the Rossby radius of deformation at the top of the box, $L = NH/f$, where N is the characteristic background buoyancy frequency at $z = H$, and f is twice the background planetary rotation rate. We solve for the relative nonlinear fluctuations around a linearly unstable thermal wind profile; $U(z) = z$, $U'(z) = 1$. We nondimensionalize time using the background shear rate. Here we instead use a depth-dependent nondimensional stratification parameter, which necessitates a fully three-dimensional approach.

$$\Gamma(z) = e^{2z - 2} \tag{113}$$

Dedalus expands this profile in a Chebyshev series up to a cutoff tolerance of 10^{-8}.

The presence of the shear implies a large-scale background buoyancy gradient. We include a background planetary vorticity gradient (β effect) with $\beta = 0.1$. To saturate the simulation we use hyper-diffusivities $\nu_4 = \kappa_4 = 10^{-6}$, bottom friction $\gamma_- = 0.16$, but no top friction. The horizontal domain size is 40×20 Rossby radii in the x, y directions respectively. We use 256×128 Fourier modes in the x, y directions, and 32 Chebyshev modes in the z direction, all using 3/2 dealiasing.
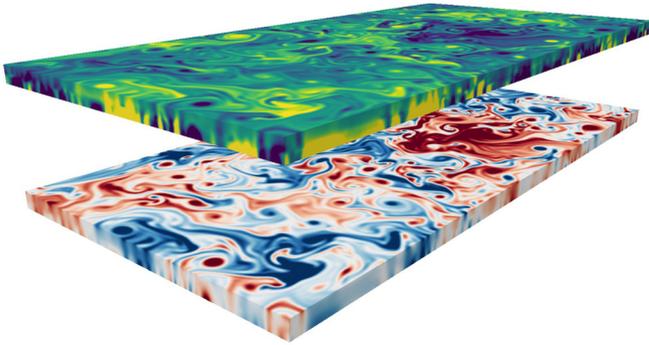
FIG. 13. Surface and lateral slices of the PV (top) and buoyancy perturbation θ (bottom) in a 3D quasigeostrophic flow. Both images are at $t = 200$ in the statistically saturated state.

The Chebyshev tau method balances energy to exponentially high accuracy; exact energy conservation could be still be imposed, however, by formulating a self-adjoint system with an alternative orthogonal polynomial basis [65,137,138].

The whole configuration is baroclinically unstable. The motion is a kind of side-ways convective heat transport. For background, we highly recommend the book "Atmospheric and Oceanic Fluid Dynamics" by Geoffrey K. Vallis [139]. To summarize the phenomenology: "It is the instability that gives rise to the large- and mesoscale motion in the atmosphere and ocean—it produces atmospheric weather systems, for example—and so is, perhaps, the form of hydrodynamic instability that most affects the human condition."

We compute the nonlinear evolution until the system reaches a statistically stationary state, which takes roughly 75 dynamical time units. Figure 13 shows the PV and vertical vorticity ζ at $t = 200$. The solution contains a sea of compact eddies that form, merge, and breakup over several dynamical times. For these parameters, the system is mostly two-dimensional. There are, however, nontrivial variations in the depth-dependence of PV.

F. Stokes flow

Many important biophysical and industrial fluid problems occur in a high-drag (or low Reynolds number) limit $(u_0 \ell_0 / \nu = \text{Re} \ll 1)$ where the momentum equation reduces to Stokes flow. Here, we present and solve a simple test problem demonstrating boundary-driven Stokes flow in curvilinear geometry with time-dependent tracer fields.

We simulate the classic "unmixing" demonstration e.g., Ref. [140], where a Taylor-Couette device with an inner-cylinder radius, R_{in}, turns n times and then reverses back to the start. For low Re, the flow is reversible, and m-many dye tracers will appear to mix and then unmix. We nondimensionalize lengths with the gap width, $\ell_0 = R_{\text{out}} - R_{\text{in}}$, and velocities with the maximum innercylinder speed, u_0. The Peclet number $u_0 \ell_0 / \kappa = \text{Pe}$ controls the tracer mass diffusion. Taking $\text{Re} \to 0$,

$$\nabla^2 \mathbf{u} = \nabla p, \tag{114}$$

$$\nabla \cdot \mathbf{u} = 0, \tag{115}$$

$$\partial_t c_m + \mathbf{u} \cdot \nabla c_m = \frac{\nabla^2 c_m}{\text{Pe}}, \quad m = 1, 2, 3. \tag{116}$$

The domain is a noslip two-dimensional cylindrical annulus with radius $1 \leqslant r \leqslant 2$ and angle $0 \leqslant \theta < 2\pi$. The diffusive dye flux $\mathcal{F}_m = -\nabla c_m / \text{Pe} = 0$ at the boundaries. The flow vanishes at the outer wall; $\mathbf{u}(t, R_{\text{out}}, \theta) = 0$. The tangential velocity at the inner wall is a time-dependent regularized square wave

$$u_\theta(t, R_{\text{in}}, \theta) = \frac{\arctan(50 \sin(t/n_{\text{rot}}))}{\arctan(50)}. \tag{117}$$

The time-dependent boundary condition uses the `GeneralFunction` mechanism described in Sec. VII E. The initial dye fields are circular Gaussians with widths $\delta = 1/4$ centered at $r_{\text{dye}} = 3/2$, $\theta_m = 2\pi m/3$.

Since the domain is a cylindrical annulus, which excludes the origin, it can be accurately discretized using a direct product of Fourier and Chebyshev bases in θ and r, respectively. For simulating the full disk, nondirect product bases imposing certain regularity conditions at the origin are necessary [64] (these will be implemented in future versions of Dedalus).

In polar coordinates, the differential operators ∇, ∇^2 contain nonconstant coefficients proportional to $1/r$ and $1/r^2$ multiplying the radial and azimuthal derivatives, which can simply be added when entering the equations. Naively (for example) one might implement the angular part of the Laplacian as 'd(u, th = 2)/r2' where th is the θ spatial variable. However, the Chebyshev expansion r^{-2} converges very slowly, leading to dense nonconstant coefficient matrices. To avoid this, we multiply all equations by r or r^2 (depending on the order of derivatives). The resulting nonconstant coefficient matrices only contain bandwidth one or two.

We solve this system with 512×512 modes using the Runga-Kutta 443 time stepper and a fixed $dt = 0.005$. The simulation runs approximately four rotations forwards and backwards; $n_{\text{rot}} = 8$. Figure 14 shows snapshots of the flow at five times, symmetric about the middle of the simulation. The dye spreads out into thin sheets as the inner cylinder rotates in the counter-clockwise direction; the sheets congregate back, differing from the original Gaussian blobs due to the action of the dye. The final shapes are not Gaussian. Shear dispersion leads to thin structure (sharp gradients) in the radial direction, and hence the diffusion is not isotropic.

This simulation uses high $\text{Pe} = 10^7$, causing the sheets to become very thin. The dye-patches area stays constant in the absence of diffusion. Equating the initial and final area $A_0 \sim \pi \delta^2/4 \approx A_{\text{f}} \sim n_{\text{rot}} 2\pi r_{\text{dye}} \delta r$, implies the sheet width $\delta r \sim 1/200$. For the diffusion time across the sheets to be greater than the total simulation time, $\tau_D \gg 2 n_{\text{rot}} \simeq 10$, we require $\text{Pe} \gtrsim 10/\delta r^2 \sim 4 \times 10^5$. The example code also works with $\text{Pe} = \infty$, disabling the $\nabla^2 c_m/\text{Pe}$ terms and flux boundary conditions. In this case, the well-resolved sheets return to the original Gaussian with only slight time-stepping errors.

G. Atmospheric waves

Stratified (nonrotating, neutral) atmospheres support two classes of linear waves: acoustic waves and gravity waves. The properties of these waves are easy to compute in constant coefficient atmospheres (like isothermal atmospheres), but require numerical solutions when the atmosphere structure introduces nonconstant coefficients. Solving for the nonlinear
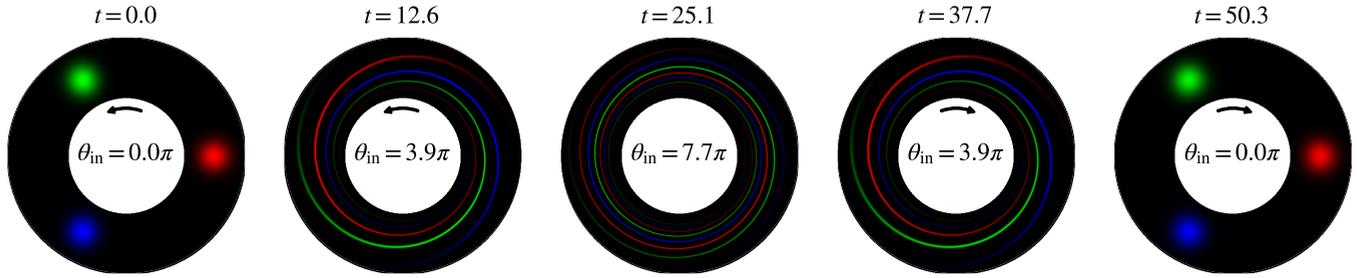
FIG. 14. Evolution of the tracer fields in a reversible Taylor-Couette Stokes flow. The three separate tracer fields are visualized as red, green, and blue in a single RGB image. As time progresses, we rotate the inner cylinder first counter clockwise and then clockwise. By the end of the simulation, the flow has returned to its initial conditions, modified only by mass diffusion of the dye.

atmospheric background structure is challenging. Here we use a nonlinear boundary value problem (NLBVP) to solve for an atmosphere profile, and then solve an eigenvalue problem (EVP) for the wave modes.

We study an optically thin plane-parallel atmosphere coupled to an underlying, optically thick adiabatic layer. This models the Sun's photosphere and lower atmosphere, or Jupiter's atmosphere spanning the radiative-convective boundary and the lower stratosphere. For the background structure, we solve hydrostatic and thermal equilibrium, including radiation transport under the Eddington tensor approximation, e.g., Refs. [141–144]:

$$\frac{\partial P_g}{\partial z} + \frac{\partial P_r}{\partial z} = -\rho g, \tag{118}$$

$$\frac{\partial P_r}{\partial z} = -\frac{\rho \kappa F_r}{c}, \tag{119}$$

with P_g the gas pressure, P_r the radiation pressure, ρ the density, g the constant gravity, κ the opacity, F_r the constant radiative flux, and c the speed of light. We use an ideal gas equation of state $P_g = \rho R T$ with R the gas constant and T the temperature, the Eddington tensor closure $P_r = f a T^4$ with a the radiation constant and $f = 1/3$, and a Kramer-like opacity law for κ:

$$\kappa(\rho, T) = \kappa_0 \left(\frac{\rho}{\rho_0}\right)^a \left(\frac{T}{T_0}\right)^b, \tag{120}$$

With these substitutions, and nondimensionalizing by ρ_0, T_0, and a length scale L, the equations become

$$\frac{1}{\rho}\frac{\partial \rho}{\partial z} + \frac{1}{T}\frac{\partial T}{\partial z} = -\frac{g^*}{T}\left[1 - \left(\frac{F}{F_{Edd}}\right)\rho^a T^b\right], \tag{121}$$

$$\frac{1}{T}\frac{\partial T}{\partial z} = -Q\rho^{a+1}T^{b-4}, \tag{122}$$

where $g^* = gL/RT_0$, $F_{Edd} = gc/\kappa_0$, and $Q = \rho_0 \kappa_0 F_r L / 4 f a c T_0^4$.

For $F \ll F_{Edd}$, the optically thick deep solution will be nearly polytropic with $\rho \propto T^m$ where $m = (3 - b)/(1 + a)$. We fix $a = 1$, $b = 0$ and take $g^* = m + 1$ to define the lengthscale L. Q is taken to be $1 - 1.68 \times 10^{-4}$, corresponding to $\ln(T_{eff}/T_0) = -2$ in a gray atmosphere. We solve this nonlinear boundary value problem in Dedalus using $\ln \rho$ and $\ln T$ as dynamical variables with 128 Legendre modes. A dealiasing factor of 2 is used, which reduces but cannot eliminate aliasing errors since exponential nonlinearities (formally including

infinite polynomial orders) are present in the formulation. Solutions converge to a relative tolerance of 10^{-8} in $\mathcal{O}(10)$ Newton iterations. When $F = 0$ there is an analytic solution due to Brandenburg [145] which our numerical solution matches to 10 digits in the L_2 norm, verifying the correctness of the implementation. We proceed here with $F/F_{Edd} = 10^{-5}$, for which there is no analytic solution.

We plot the temperature, density, and pressure in Fig. 15. The bottom, optically thick, part of the atmosphere is nearly polytropic with $\rho \propto T^{(3/2)}$, as expected for this choice of a and b [146]. The top, optically thin, part of the atmosphere is nearly isothermal while the density and pressure drop exponentially. This region has a constant Brunt-Väisälä (buoyancy) frequency

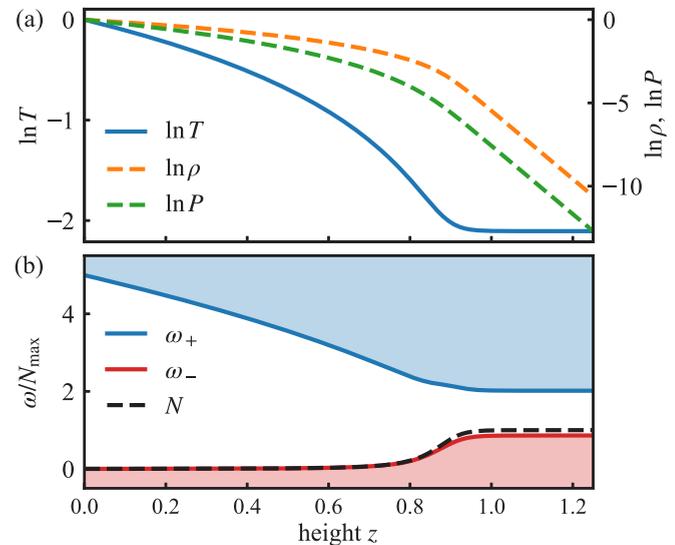$$N^2 = -g \partial_z s / C_P. \tag{123}$$

FIG. 15. (a) Structure of a balanced atmosphere with an Eddington closure for radiation transport and a Kramer-like opacity law with $a = 1$, $b = 0$. At depth, the atmosphere is nearly an adiabatic polytrope with constant s/C_P and $N^2 \approx 0$. The upper atmosphere is nearly isothermal with exponentially decaying ρ and P. (b) The upper atmosphere is a resonant cavity for internal gravity waves with $N^2 > 0$. Frequencies are normalized by the Brunt-Väisälä frequency N in the isothermal layer.

Next we consider the oscillation modes of this atmosphere. The character of waves modes depends on their frequency relative to $\omega_\pm$, which are defined as

$$\omega_+^2 = \omega_L^2 + \omega_{ac}^2, \tag{124}$$

$$\omega_-^2 = \frac{\omega_L^2}{\omega_L^2 + \omega_{ac}^2} N^2, \tag{125}$$

where the Lamb frequency ω_L and acoustic frequency ω_{ac} are related to the properties of the background atmosphere e.g., Ref. [147]. Acoustic waves have frequencies greater than ω_+, while gravity waves have frequencies less than ω_-. We plot $\omega_\pm$ for our atmosphere in Fig. 15. High-frequency sound waves can propagate all the way to the bottom of the atmosphere, whereas gravity waves mostly stay in the optically thin isothermal layer.

We write the ideal, linearized fully compressible equations as

$$\partial_t w + \partial_z T_1 + T_0 \partial_z \ln \rho_1 + T_1 \partial_z \ln \rho_0 = 0, \tag{126}$$

$$\partial_t u + \partial_x T_1 + T_0 \partial_x \ln \rho_1 = 0, \tag{127}$$

$$\partial_t \ln \rho_1 + w \partial_z \ln \rho_0 + (\partial_x u + \partial_z w) = 0, \tag{128}$$

$$\partial_t T_1 + w \partial_z T_0 + (\gamma - 1) T_0 (\partial_x u + \partial_z w) = 0, \tag{129}$$

for the perturbation velocities w and u, and thermal and log density perturbations T_1 and $\ln \rho_1$. The nonconstant coefficients of the atmosphere are $\partial_z \ln \rho_0$, T_0 and $\partial_z T_0$ and we take $\gamma = 5/3$. For simplicity, we impose impenetrable top and bottom boundaries ($w = 0$).

We formulate this as an eigenvalue problem in Dedalus by replacing $\partial_t = i\omega$ and $\partial_x = -ik_x$ and solving for complex eigenvalue ω given specified horizontal wavenumbers k_x. We use the eigentools [148] package to automatically test whether eigenvalues are numerically converged using the techniques described in Boyd *et al.* ([83], Ch. 7.5). As expected, we find about 50% of the eigenvalues are converged for sufficiently high resolution. Here we solve with 256 Legendre modes for twenty distinct k_x, themselves logrithmically spaced. Legendre expansions produce optimal polynomial approximations in the L^2 norm and may be preferable to Chebyshev expansions in problems where the lack of a fast transform is unimportant, such as dense eigenvalue problems.

Figure 16 shows the frequencies (or equivalently periods) of the wave modes calculated with the Dedalus eigenvalue solver. Because sound waves have frequencies greater than ω_+ and gravity waves have frequencies less than ω_-, we also plot the minimum of ω_- and maximum of ω_+ over the vertical extent of the atmosphere (both of which occur in the isothermal layer). This allows us to easily distinguish sound waves from gravity waves. We also find that, at fixed horizontal wave number, the frequency spacing between sound waves is about constant, whereas the period spacing between gravity waves is about constant. This well-known property follows from the dispersion relation in the large-vertical-wave-number limit.

Figure 17 shows vertical velocity eigenfunctions w, scaled by $\sqrt{\rho}$. These modes correspond to the marked modes in Fig. 16. All eigenfunctions have been normalized by the mode
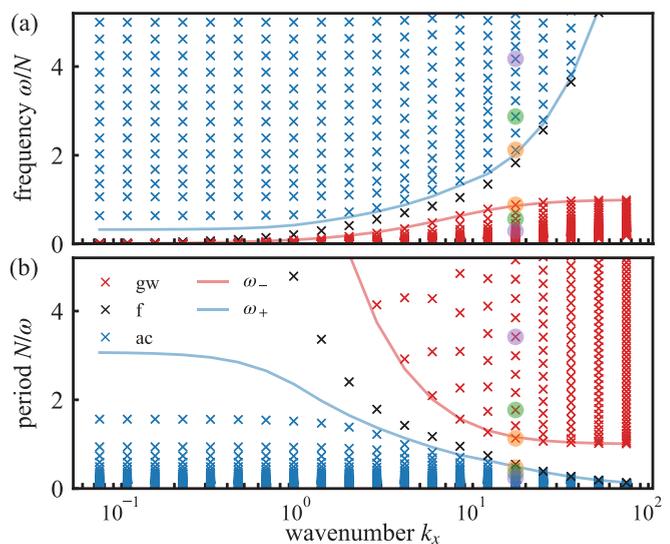

FIG. 16. Eigenmode (a) frequencies and (b) periods for a radiative atmosphere. The mode frequencies ω are normalized by the Brunt-Väisälä frequency N in the isothermal layer. Acoustic modes ("ac", blue), gravity modes ("gw", red), and f-modes ("f", black) are identified by their frequencies. Acoustic waves are expected to have nearly equal frequency spacing, while gravity waves have nearly equal period spacing, as is observed. Colored circles highlight modes shown in Fig. 17.

kinetic energy:

$$w = \frac{w_{EVP}}{W}, \quad W = \sqrt{\frac{\int \rho (u_{EVP}^2 + w_{EVP}^2) dz}{\int \rho dz}}. \tag{130}$$

We find that the eigenfunctions follow the intuition of the propagation diagram in Fig. 15. Gravity modes are trapped in the upper atmosphere, while acoustic modes span a larger
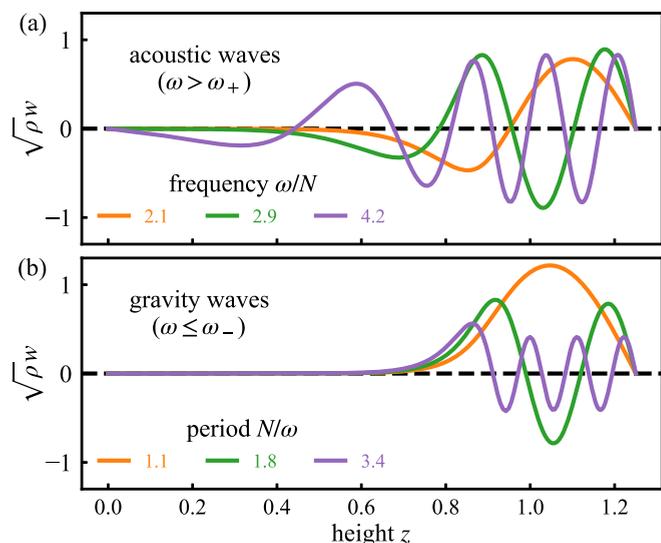

FIG. 17. Vertical velocity eigenfunctions for a radiative atmosphere, scaled by $\sqrt{\rho}$, including (a) acoustic modes and (b) gravity modes. Gravity modes are almost entirely confined to the isothermal portion of the atmosphere and are evanescent in the adiabatically stratified deep interior. Acoustic modes propagate in both regions, with high frequency waves propagating deeper.

region of the full atmosphere with the highest frequency modes reaching the bottom of the domain. The effects of the nonconstant coefficients are visible in the acoustic modes which have a varying vertical wavelength in the deep adiabatic interior and a nearly constant vertical wavelength in the upper isothermal region.

This example of wave eigenfunctions and eigenvalues demonstrates the ability to link a complex atmosphere with detailed solves. This example considered ideal waves in a bounded atmosphere, but can be easily extended to nonadiabatic waves with thermal damping, to magnetohydrodynamic waves propagating through a background magnetic field, and to systems with open boundary conditions.

H. Diamagnetic levitation

This example computes the motion of a levitating rigid body under the competing influence of gravity and an imposed diffuse background magnetic field. There are three general classes of magnetic levitation: (i) *AC electromagnetic suspension* of a conductor relying on an alternating background field; (ii) *motion-induced suspension* such as Maglev trains and spinning tops; and (iii) *diamagnetic levitation*, which results from magnetically induced eddy currents from a dc field within special types of macroscopic media. Diamagnetic levitation is usually the weakest form, but superconductivity provides the important exception. Past experimental work on diamagnetic levitation used an approximately 16 Tesla solenoid to suspend a frog quasi-stably (and mostly safe for the frog) [149].

For context, magnetic levitation of any kind is actually a rigid-body version of magnetohydrodynamic buoyancy (or Parker instability), which are well-know in astrophysics [150]. Maglev train physics is very similar to differential-rotation induced magnetic buoyancy; which is important in stellar and planetary interiors [151]. All types of levitation would be amenable simulation in Dedalus; including magnetohydrodynamic buoyancy. Here we focus exclusively on solid-body diamagnetic levitation.

Electrodynamics. We use the (nonrelativistic) Maxwell's equations for the magnetic flux density B, electric field E, magnetic field H, and current density J:

$$\partial_t B + \nabla \times E = 0, \tag{131}$$

$$\nabla \cdot B = 0, \tag{132}$$

$$\nabla \times H = J. \tag{133}$$

Closing the system requires a constitutive relation between H and B, and Ohm's Law between E and J (in a frame moving with velocity v),

$$H = \frac{B}{\mu}, \quad E = \rho J - v \times B. \tag{134}$$

The constitutive parameters are the resistivity ρ (inverse electrical conductivity), the magnetic permeability μ, and the diffusivity $\eta = \rho/\mu$.

We use a magnetic potential $B = \nabla \times A$ to enforce the magnetic Gauss's law $\nabla \cdot B = 0$. In two dimensions, we can use a scalar magnetic potential (three dimensions would

require a gauge choice). Then Faraday's law of induction is

$$\partial_t A = v_x B_y - v_y B_x + \rho \left[\partial_y \frac{B_x}{\mu} - \partial_x \frac{B_y}{\mu} \right], \tag{135}$$

where $B_x = \partial_y A$ and $B_y = -\partial_x A$.

Eulerian velocity. We embed a freely movable (yet coupled) rigid body within the neutral background. First, the rigid-body translation and rotation produce the local Eulerian "fluid" (continuum) velocity

$$\text{if} \quad (x, y) \in \mathcal{M}(x_0, y_0, \theta), \quad \text{then}$$

$$v_x(t, x, y) = \dot{x}_0(t) - (y - y_0(t)) \dot{\theta}(t), \tag{136}$$

$$v_y(t, x, y) = \dot{y}_0(t) + (x - x_0(t)) \dot{\theta}(t). \tag{137}$$

The set $\mathcal{M}(x_0, y_0, \theta)$ represents the points inside the body with center of mass $x_0(t)$, $y_0(t)$ and orientation angle $\theta(t)$. The Eulerian velocity vanishes outside the body.

Mask function. We use a *smooth* mask function (rather than strictly binary) to indicate the solid-object region. We model our solid as an ellipse with 2:1 semimajor/semiminor axis ratio. We use an adjustable error function profile for the mask function

$$M_\varepsilon(x, y) = \frac{1}{2} \left\{ 1 - \text{Erf} \left[\frac{\sqrt{\pi}}{2\varepsilon} \left(\frac{x^2}{a^2} + \frac{x^2}{b^2} - 1 \right) \right] \right\}. \tag{138}$$

The smoothed-mask function approach is called the "smoothed-volume-penalty method" (SPV). Recent detailed analysis shows this method is quite competitive with all other techniques for treating moving solid objects. Also, the smooth transition region aids in controlling error compared to an abrupt transition [67].

Coupled ODEs. Translation and rotation about the center of mass follow a set of ordinary differential equations describing Newton's laws of conservation of momentum and angular momentum.

$$\frac{d}{dt} \begin{bmatrix} x_0(t) \\ y_0(t) \end{bmatrix} = \begin{bmatrix} \bar{v}_x(t) \\ \bar{v}_y(t) \end{bmatrix}, \tag{139}$$

$$\frac{d}{dt} \begin{bmatrix} \bar{v}_x(t) \\ \bar{v}_y(t) \end{bmatrix} = \frac{1}{m} \begin{bmatrix} \bar{F}_x(t) \\ \bar{F}_y(t) \end{bmatrix} - \begin{bmatrix} 0 \\ g \end{bmatrix}, \tag{140}$$

$$\frac{d}{dt} \begin{bmatrix} \theta(t) \\ I\,\omega(t) \end{bmatrix} = \begin{bmatrix} \omega(t) \\ \bar{\tau}_z(t) \end{bmatrix}. \tag{141}$$

The parameters m and I are the object's mass and moment of inertial; g represents gravitational acceleration. For the ellipse $I/m = \pi\,a\,b\,(a^2 + b^2)/4$. Solving this system of 6 ODEs requires computing the total force, $\bar{F}_x$, $\bar{F}_y$, and torque, $\bar{\tau}_z$. Each depends on the background magnetic field and material properties of the rigid body.

Lorentz force. We compute the total force on the object by integrating the Lorentz force density

$$\mathcal{L} = J \times B = \nabla \cdot \left[H\,B - \frac{H \cdot B}{2} I \right] + \frac{H \cdot B}{2} \nabla \ln \mu. \tag{142}$$

The force on the rigid body is

$$\bar{F} = -\int \mathcal{L} \, dx = -\int \frac{H \cdot B}{2} \nabla \ln \mu \, dx. \quad (143)$$

The gradient of the permeability makes sense because the material is highly localised, yet infinitely differential via Eq. (138).

Magnetic susceptibility. For most everyday substances (e.g., frogs), the magnetic permeability is extremely close to that of free space, μ_0. Defining

$$\chi = \frac{\mu}{\mu_0} - 1. \quad (144)$$

we assume $|\chi| \ll 1$ (e.g., graphite has $\chi \approx -2 \times 10^{-5}$),

$$\frac{1}{\mu} \approx \frac{1 - \chi M_\varepsilon}{\mu_0}, \quad \nabla \ln \mu \approx \chi \nabla M_\varepsilon. \quad (145)$$

This simplifies our equations considerably. This assumption would not be justified for a partial superconductor where $\chi \approx -1$. In the case of the force and torque,

$$\bar{F} \approx -\chi \int \frac{|B|^2}{2\mu_0} \nabla M_\varepsilon \, dx, \quad (146)$$

$$\bar{\tau} \approx -\chi \int \frac{|B|^2}{2\mu_0} (r - r_0) \times \nabla M_\varepsilon \, dx. \quad (147)$$

Considering a negative upward magnetic pressure gradient, it follows that levitation requires $\chi < 1$. The proportionality of the force to χ shows the need for very strong magnetic fields.

Simulation. We solve the magnetic induction equation in nonmoving electrically neutral background. Most of the domain is filled with a harmonic magnetic field to a good approximation; like the air around us. We nondimensionalize the kinematics of the simulation based gravity and the semi-minor axis of our elliptical rigid body. Therefore $g = 1$, $a = 2$, and $b = 1$. We choose a mask smoothing parameter $\varepsilon = 0.025$. We solve the magnetic induction equation under the above approximations in a rectangular 16×8 domain. We use a Fourier series with 1024 modes for the x direction and a Chebyshev series with 512 modes for the y direction. In both directions, we use the $3/2$ dealiasing rule. We fix the vertical magnetic field at the bottom of the domain, and allow a free vacuum field at the top. We enforce the top boundary condition via the Hilbert transform in the x direction

$$A = -\frac{B_0}{k} \sin(kx) \quad \text{at} \quad y = y_0, \quad (148)$$

$$B_x - \mathcal{H}_x(B_y) = 0 \quad \text{at} \quad y = y_1. \quad (149)$$

The simulation starts with a harmonic field satisfying the boundary conditions.

We release the solid ellipse from rest with an initial $30°$ tilt. Figure 18 shows the ellipse and magnetic field at three representative times in its evolution. The object free falls until it encounters a strong enough magnetic pressure gradient to rebound. Along the way, some of the background field diffuses into the object. As the object bounces upward, the captured field resists and eventually halts the rebound. The process continues until the object is mostly sliding and rotating along the top of the magnetic arches.
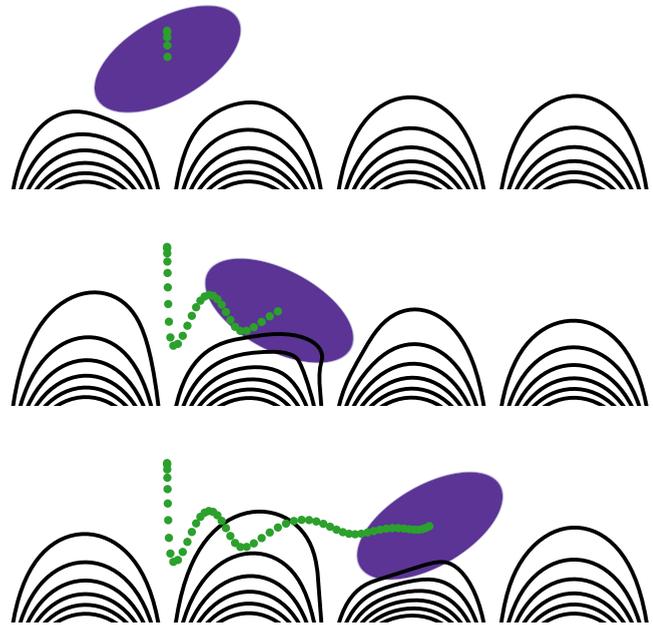

FIG. 18. Evolution of a diamagnetic object levitating in a magnetic field. The purple ellipse shows the solid-body region. The black lines show magnetic field lines the same way iron filings trace the magnetic field from a strong bar magnet. Green dots show the time-dependent trajectory of the ellipse's center of mass at equally spaced time intervals.

XII. CONCLUSION AND OUTLOOK

The integrity and reproducibility of computational science relies critically on robust, open-source, and well-supported code. We have introduced Dedalus, a public PYTHON framework for solving PDEs using spectral methods with an interdisciplinary community of users and developers.

Dedalus enables users to construct custom domains using the direct product of spectral series, to symbolically specify systems of equations and boundary conditions, and to perform custom data analysis. Dedalus supports initial value, eigenvalue, and linear and nonlinear boundary value problems. The solution of these problems is automatically parallelized using MPI, allowing for seamless scaling from individual laptops to supercomputers with tens-of-thousands of cores. The Dedalus distribution includes simple example scripts to help users become familiar with the code's features. In this paper, we have included a diverse range of example problems demonstrating more advanced features and the code's adaptability to many different physical models.

We are committed to continually enhancing and optimizing Dedalus and supporting its users. Substantial extensions to the codebase are currently underway. These include support for multiple coupled dimensions, coordinate-free equation entry, enhanced stand-alone data analysis, and non-direct-product bases for tensorial quantities in curvilinear coordinates (particularly full disks, spheres, and balls [64–66]). We actively help users to troubleshoot problems and formulate Dedalus-compatible models on our public, searchable mailing list. Our goal for the future is to continue growing through community

development and to provide robust tools for a wide range of scientific applications.

The Dedalus code and the example problems are available online at [95].

ACKNOWLEDGMENTS

APPENDIX: SPECTRAL OPERATOR MATRICES

1. Differentiation and conversion matrices

a. Fourier

Fourier differentiation is a separable operator with a matrix form

$$\partial_x \phi_k^F(x) = ik\phi_k^F(x) \Rightarrow D_{k,k'}^F = ik\delta_{k,k'}, \qquad \text{(A1)}$$

where the k in the matrix entry expression is the signed wave number of the corresponding mode.

b. Sine/cosine

Sine/cosine differentiation is a separable operator which flips the parity of its operand and has matrix forms

$$\partial_x \phi_k^c(x) = -k\phi_k^s(x) \Rightarrow D_{k,k'}^C = -k\delta_{k,k'}, \qquad \text{(A2)}$$

$$\partial_x \phi_k^s(x) = k\phi_k^c(x) \Rightarrow D_{k,k'}^S = k\delta_{k,k'}. \qquad \text{(A3)}$$

c. Chebyshev

The derivatives of the Chebyshev polynomials satisfy the recurrence relation

$$\frac{\partial_x T_n(x)}{n} = 2T_{n-1}(x) + \frac{\partial_x T_{n-2}(x)}{n-2}. \qquad \text{(A4)}$$

Therefore

$$D_{i,j} = \langle T_i | \partial_x T_j \rangle \qquad \text{(A5)}$$

$$= \frac{2j((j-i)\bmod 2)}{1+\delta_{i,0}}[i<j]. \qquad \text{(A6)}$$

The derivatives of the Chebyshev-T polynomials are sparse in the Chebyshev-U polynomials:

$$\partial_x T_n(x) = nU_{n-1}(x). \qquad \text{(A7)}$$

Furthermore, the polynomials satisfy the identity

$$2T_n(x) = U_n(x) - U_{n-2}(x), \qquad \text{(A8)}$$

which provides a sparse T-to-U conversion operator for left preconditioning Chebyshev differential equations:

$$P_{i,j}^L = \langle U_i | T_j \rangle \qquad \text{(A9)}$$

$$= \frac{\delta_{i,j} - \delta_{i,j-2}}{2 - \delta_{j,0}}. \qquad \text{(A10)}$$

d. Legendre

The derivatives of the Legendre polynomials satisfy the recurrence relation

$$\partial_x P_n(x) = (2n-1)P_{n-1}(x) + \partial_x P_{n-2}(x). \qquad \text{(A11)}$$

Therefore

$$D_{i,j} = \langle P_i | \partial_x P_j \rangle \qquad \text{(A12)}$$

$$= (2i+1)((j-i)\bmod 2)[i<j]. \qquad \text{(A13)}$$

The Legendre polynomials are equal to the Jacobi polynomials $J^{\alpha,\beta}$ with $\alpha = \beta = 0$. Their derivatives are therefore

sparse in the Jacobi polynomials with $\alpha = \beta = 1$:

$$\partial_x P_n(x) = \partial_x J_n^{0,0}(x) = \frac{n+1}{2} J_{n-1}^{1,1}(x). \tag{A14}$$

Furthermore, the Jacobi polynomials satisfy the identity

$$2(2n+1)J_n^{0,0}(x) = (n+2)J_n^{1,1}(x) - nJ_{n-2}^{1,1}(x), \tag{A15}$$

which provides a sparse conversion operator for left preconditioning Legendre differential equations:

$$P_{i,j}^L = \langle J_i^{1,1} | P_j \rangle \tag{A16}$$

$$= \frac{j+2}{2(2j+1)}\delta_{i,j} - \frac{j}{2(2j+1)}\delta_{i,j-2}. \tag{A17}$$

e. Hermite

The derivatives of Hermite polynomials are naturally sparse in the Hermite polynomials themselves:

$$\partial_x H_n(x) = 2nH_{n-1}(x). \tag{A18}$$

The Hermite polynomial differentiation matrix is therefore banded and is given by

$$D_{i,j} = \langle H_i | \partial_x H_j \rangle = 2j\delta_{i,j-1}. \tag{A19}$$

The enveloped Hermite functions satisfy

$$\partial_x \phi_n^H(x) = \sqrt{\frac{n}{2}}\phi_{n-1}^H(x) - \sqrt{\frac{n+1}{2}}\phi_{n+1}^H(x). \tag{A20}$$

The differentiation matrix for the enveloped Hermite functions is therefore also banded and given by

$$D_{i,j} = \langle \phi_i^H | \partial_x \phi_j^H \rangle = \sqrt{\frac{j}{2}}\delta_{i,j-1} - \sqrt{\frac{j+1}{2}}\delta_{i,j+1}. \tag{A21}$$

Since these matrices are both banded, no conversion operators / left preconditioners are necessary for the `Hermite` basis.

f. Laguerre

The derivatives of Laguerre polynomials are sparse in the generalized Laguerre polynomials as

$$\partial_x L_n(x) = -L_{n-1}^{(1)}(x), \tag{A22}$$

where

$$L_n^{(1)}(x) = \sum_{i=0}^{n} L_i(x). \tag{A23}$$

The Laguerre polynomial differentiation matrix is therefore

$$D_{i,j} = \langle L_i | \partial_x L_j \rangle = -1[i < j]. \tag{A24}$$

The enveloped Laguerre functions satisfy

$$\partial_x \phi_n^L(x) = -\frac{1}{2}\phi_n^L(x) - \sum_{i=0}^{n-1} \phi_i^L(x). \tag{A25}$$

The differentiation matrix for enveloped Laguerre functions is therefore

$$D_{i,j} = \langle \phi_i^L | \partial_x \phi_j^L \rangle = -\frac{1}{1+\delta_{i,j}}[i \leqslant j]. \tag{A26}$$

Both differentiation matrices are rendered sparse by conversion from Laguerre polynomials to the first generalized Laguerre polynomials. This is applied to left precondition all Laguerre differential equations and is given by

$$P_{i,j}^L = \langle L_i^{(1)} | L_j \rangle = \delta_{i,j} - \delta_{i,j-1}. \tag{A27}$$

2. Dirichlet recombination matrices

a. Chebyshev and Legendre

The Chebyshev and Legendre polynomials satisfy the endpoint conditions $T_n(\pm 1) = P_n(\pm 1) = (\pm 1)^n$. These polynomials recombine to isolate the boundary support to the first two modes as

$$D_n^T(x) = \begin{cases} T_n(x), & n = 0, 1 \\ T_n(x) - T_{n-2}(x), & n \geqslant 2 \end{cases}, \tag{A28}$$

and likewise for the Legendre polynomials. These Dirichlet polynomials therefore satisfy

$$D_n^T(\pm 1) = D_n^P(\pm 1) = \begin{cases} (\pm 1)^n, & n = 0, 1 \\ 0, & n \geqslant 2 \end{cases}. \tag{A29}$$

The sparse conversion matrix from the Dirichlet polynomials to the Chebyshev/Legendre polynomials is used as a right preconditioner to compress the boundary rows corresponding to Dirichlet boundary conditions while maintaining sparsity of the equation matrices. This matrix is given by

$$P_{i,j}^R = \langle T_i | D_j^T \rangle = \langle P_i | D_j^P \rangle = \delta_{i,j} - \delta_{i,j-2}[j > 1]. \tag{A30}$$

b. Laguerre

The Laguerre polynomials satisfy the endpoint condition $L_n(0) = 1$. These polynomials recombine to isolate the boundary support to the first mode as

$$D_n(x) = \begin{cases} L_0(x), & n = 0 \\ L_n(x) - L_{n-1}(x), & n \geqslant 1 \end{cases}. \tag{A31}$$

These Dirichlet polynomials therefore satisfy

$$D_n(0) = \begin{cases} 1, & n = 0 \\ 0, & n \geqslant 1 \end{cases}. \tag{A32}$$

The sparse conversion matrix from the Dirichlet polynomials to the Laguerre polynomials is used as a right preconditioner to compress the boundary rows corresponding to Dirichlet boundary conditions while maintaining sparsity of the equation matrices. This matrix is given by

$$P_{i,j}^R = \langle L_i | D_j \rangle = \delta_{i,j} - \delta_{i,j-1}[j \geqslant 1]. \tag{A33}$$

This matrix is used to right precondition problems using both the Laguerre polynomials and the enveloped Laguerre functions.

3. NCC multiplication matrices

a. Chebyshev

The Chebyshev polynomials satisfy the multiplicative identity

$$T_n(x)T_j(x) = \frac{T_{j+n}(x) + T_{|j-n|}(x)}{2}. \tag{A34}$$

The single-mode Chebyshev multiplication matrices are therefore

$$\langle T_i | T_n T_j \rangle = \frac{\delta_{i,j+n} + \delta_{i,|j-n|}}{2}. \tag{A35}$$

b. Legendre

The Legendre polynomials satisfy the three-term recurrence relation

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x), \tag{A36}$$

which is encoded in the Legendre Jacobi matrix J such that

$$xP_i(x) = J_{i,j}P_j(x). \tag{A37}$$

Since the Legendre Jacobi matrix encodes the action of multiplication by x, the Legendre polynomial multiplication matrices are given by the corresponding Legendre polynomials of the Jacobi matrix:

$$\langle P_i | P_n P_j \rangle = (P_n(J))_{i,j} \equiv \left(M_n^P \right)_{i,j}. \tag{A38}$$

The Legendre polynomial multiplication matrices can therefore be constructed by applying the Legendre recurrence relation to the Legendre Jacobi matrix:

$$M_0^P = I, \tag{A39}$$

$$M_1^P = J, \tag{A40}$$

$$(n+1)M_{n+1}^P = (2n+1)JM_n^P - nM_{n-1}^P. \tag{A41}$$

c. Hermite

The Hermite polynomials satisfy the three-term recurrence relation

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x). \tag{A42}$$

The Hermite polynomial multiplication matrices can therefore be constructed by applying this recurrence relation to the

corresponding Hermite Jacobi matrix, as detailed above for Legendre polynomials.

Matrices for multiplication between Hermite polynomials and enveloped Hermite functions are implemented by reweighting the Hermite multiplication matrices as

$$\left\langle \phi_i^H \middle| H_n \phi_j^H \right\rangle = \frac{N_i}{N_j} \langle H_i | H_n H_j \rangle. \tag{A43}$$

Multiplication between enveloped Hermite functions is not currently implemented as it is not band-limited when expanded in the enveloped functions. Including such terms would be possible by expanding the basis to include different powers of the Gaussian envelope.

d. Laguerre

The Laguerre polynomials satisfy the three-term recurrence relation

$$(n+1)L_{n+1}(x) = (2n+1-x)L_n(x) - nL_{n-1}(x). \tag{A44}$$

The Laguerre polynomial multiplication matrices can therefore be constructed by applying this recurrence relation to the corresponding Laguerre Jacobi matrix, as detailed above for Legendre polynomials.

Matrices for multiplication between Laguerre polynomials and enveloped Laguerre functions utilizes the same matrices since

$$\left\langle \phi_i^L \middle| L_n \phi_j^L \right\rangle = \langle L_i | L_n L_j \rangle. \tag{A45}$$

Multiplication between enveloped Laguerre functions is not currently implemented as it is not band-limited when expanded in the enveloped functions. Including such terms would be possible by expanding the basis to include different powers of the exponential envelope, i.e., the generalized Laguerre polynomials.

[1] B. Tóth, Y. Lemperiere, C. Deremble, J. de Lataillade, J. Kockelkoren, and J. P. Bouchaud, Anomalous Price Impact and the Critical Nature of Liquidity in Financial Markets, Phys. Rev. X **1**, 021006 (2011).

[2] J. Burridge, Spatial Evolution of Human Dialects, Phys. Rev. X **7**, 031008 (2017).

[3] B. L. Keyfitz and N. Keyfitz, The McKendrick partial differential equation and its uses in epidemiology and population study, Math. Comput. Modelling **26**, 1 (1997).

[4] P. J. S. Franks, NPZ models of plankton dynamics: Their construction, coupling to physics, and application, J. Oceanogr. **58**, 379 (2002).

[5] B. Lemmerer and S. Unger, Modeling and pricing of space weather derivatives, Risk Manag. **21**, 265 (2019).

[6] L. Greengard, Spectral integration and two-point boundary value problems, SIAM J. Numer. Anal. **28**, 1071 (1991).

[7] K. Julien and M. Watson, Efficient multi-dimensional solution of PDEs using Chebyshev spectral methods, J. Comput. Phys. **228**, 1480 (2009).

[8] B. K. Muite, A numerical comparison of Chebyshev methods for solving fourth order semilinear initial boundary value problems, J. Comput. Appl. Math. **234**, 317 (2010).

[9] S. Olver, S. Olver, A. Townsend, and A. Townsend, A fast and well-conditioned spectral method, SIAM Rev. **55**, 462 (2013).

[10] J. F. Gibson, Channelflow: A spectral Navier-Stokes simulator in C++ Tech. Rep. (U. New Hampshire, 2014); Channelflow.org.

[11] D. Viswanath, Spectral integration of linear boundary value problems, J. Comput. Appl. Math. **290**, 159 (2015).

[12] B. Miquel and K. Julien, Hybrid Chebyshev function bases for sparse spectral methods in parity-mixed PDEs on an infinite domain, J. Comput. Phys. **349**, 474 (2017).

[13] https://fenicsproject.org.

[14] https://www.firedrakeproject.org.

[15] http://channelflow.org.

[16] http://www.chebfun.org.

[17] https://github.com/JuliaApproximation/ApproxFun.jl.

[18] D. Lecoanet and R. R. Kerswell, Connection between nonlinear energy optimization and instantons, Phys. Rev. E **97**, 012212 (2018).

[19] S. M. Tobias and J. B. Marston, Three-dimensional rotating Couette flow via the generalised quasilinear approximation, J. Fluid Mech. **810**, 412 (2016).

[20] S. M. Tobias, J. S. Oishi, and J. B. Marston, Generalized quasilinear approximation of the interaction of convection and mean flows in a thermal annulus, Proc. R. Soc. A **474**, 20180422 (2018).

[21] G. Michel and G. P. Chini, Multiple scales analysis of slow–fast quasi-linear systems, Proc. R. Soc. A **475**, 20180630 (2019).

[22] C. D. Marcotte and V. N. Biktashev, Predicting critical ignition in slow-fast excitable models, Phys. Rev. E **101**, 042201 (2020).

[23] D. Lecoanet, B. P. Brown, E. G. Zweibel, K. J. Burns, J. S. Oishi, and G. M. Vasil, Conduction in low mach number flows. I. Linear and weakly nonlinear regimes, Astrophys. J. **797**, 94 (2014).

[24] D. Lecoanet, M. McCourt, E. Quataert, K. J. Burns, G. M. Vasil, J. S. Oishi, B. P. Brown, J. M. Stone, and R. M. O'Leary, A validated non-linear Kelvin–Helmholtz benchmark for numerical hydrodynamics, Mon. Not. R. Astron. Soc. **455**, 4274 (2015).

[25] G. M. Vasil, On the magnetorotational instability and elastic buckling, Proc. R. Soc. A **471**, 20140699 (2015).

[26] D. Lecoanet, J. Schwab, E. Quataert, L. Bildsten, F. X. Timmes, K. J. Burns, G. M. Vasil, J. S. Oishi, and B. P. Brown, Turbulent chemical diffusion in convectively bounded carbon flames, Astrophys. J. **832**, 71 (2016).

[27] D. Lecoanet, G. M. Vasil, J. Fuller, M. Cantiello, and K. J. Burns, Conversion of internal gravity waves into magnetic waves, Mon. Not. R. Astron. Soc. **466**, 2181 (2017).

[28] E. H. Anders and B. P. Brown, Convective heat transport in stratified atmospheres at low and high Mach number, Phys. Rev. Fluids **2**, 083501 (2017).

[29] S. E. Clark and J. S. Oishi, The Weakly Nonlinear magnetorotational instability in a local geometry, Astrophys. J. **841**, 1 (2017).

[30] S. E. Clark and J. S. Oishi, The weakly nonlinear magnetorotational instability in a global, cylindrical taylor–couette Flow, Astrophys. J. **841**, 2 (2017).

[31] L. K. Currie and M. K. Browning, The magnitude of viscous dissipation in strongly stratified two-dimensional convection, Astrophys. J. Lett. **845**, L17 (2017).

[32] D. Seligman and G. Laughlin, A vorticity-preserving hydrodynamical scheme for modeling accretion disk flows, Astrophys. J. **848**, 54 (2017).

[33] L. K. Currie and S. M. Tobias, Convection-driven kinematic dynamos with a self-consistent shear flow, Geophys. Astrophys. Fluid Dyn. **113**, 131 (2019).

[34] E. Quataert, D. Lecoanet, and E. R. Coughlin, Black hole accretion discs and luminous transients in failed supernovae from non-rotating supergiants, Mon. Not. R. Astron. Soc.: Lett. **485**, L83 (2019).

[35] E. H. Anders, C. M. Manduca, B. P. Brown, J. S. Oishi, and G. M. Vasil, Predicting the rossby number in convective experiments, Astrophys. J. **872**, 138 (2019).

[36] A. T. Clarke, C. J. Davies, D. Ruprecht, and S. M. Tobias, Parallel-in-time integration of kinematic dynamos, Journal of Computational Physics: X **7**, 100057 (2020).

[37] E. H. Anders, D. Lecoanet, and B. P. Brown, Entropy rain: Dilution and compression of thermals in stratified domains, Astrophys. J. **884**, 65 (2019).

[38] D. Lecoanet, M. Cantiello, E. Quataert, L.-A. Couston, K. J. Burns, B. J. S. Pope, A. S. Jermyn, B. Favier, and M. Le Bars, Low-frequency variability in massive stars: Core generation or surface phenomenon? Astrophys. J. Lett. **886**, L15 (2019).

[39] N. Tarshish, N. Jeevanjee, and D. Lecoanet, Buoyant motion of a turbulent thermal, J. Atmos. Sci. **75**, 3233 (2018).

[40] L.-A. Couston, D. Lecoanet, B. Favier, and M. Le Bars, Order Out of Chaos: Slowly Reversing Mean Flows Emerge from Turbulently Generated Internal Waves, Phys. Rev. Lett. **120**, 244505 (2018).

[41] G. K. Vallis, D. J. Parker, and S. M. Tobias, A simple system for moist convection: the Rainy–Bénard model, J. Fluid Mech. **862**, 162 (2019).

[42] M. Perrot, P. Delplace, and A. Venaille, Topological transition in stratified fluids, Nat. Phys. **15**, 781 (2019).

[43] D. Lecoanet and N. Jeevanjee, Entrainment in resolved, dry thermals, J. Atmos. Sci. **76**, 3785 (2019).

[44] B. McKim, N. Jeevanjee, and D. Lecoanet, Buoyancy-driven entrainment in dry thermals, Q. J. Royal Meteorol. Soc. **146**, 415 (2020).

[45] O. Mickelin, J. Słomka, K. J. Burns, D. Lecoanet, G. M. Vasil, L. M. Faria, and J. Dunkel, Anomalous Chained Turbulence in Actively Driven Flows on Spheres, Phys. Rev. Lett. **120**, 164503 (2018).

[46] M. Mussel and M. F. Schneider, Similarities between action potentials and acoustic pulses in a van der Waals fluid, Sci. Rep. **9**, 73 (2019).

[47] M. Mussel and M. F. Schneider, It sounds like an action potential: unification of electrical, chemical and mechanical aspects of acoustic pulses in lipids, J. R. Soc. Interface **16**, 20180743 (2019).

[48] M. Marciani and P. Delplace, Chiral Maxwell waves in continuous media from Berry monopoles, Phys. Rev. A **101**, 023827 (2020).

[49] V. Heinonen, K. J. Burns, and J. Dunkel, Quantum hydrodynamics for supersolid crystals and quasicrystals, Phys. Rev. A **99**, 063621 (2019).

[50] D. Lecoanet, M. Le Bars, K. J. Burns, G. M. Vasil, B. P. Brown, E. Quataert, and J. S. Oishi, Numerical simulations of internal wave generation by convection in water, Phys. Rev. E **91**, 063016 (2015).

[51] L. A. Couston, D. Lecoanet, B. Favier, and M. Le Bars, Dynamics of mixed convective–stably-stratified fluids, Phys. Rev. Fluids **2**, 094804 (2017).

[52] E. H. Anders, B. P. Brown, and J. S. Oishi, Accelerated evolution of convective simulations, Phys. Rev. Fluids **3**, 083502 (2018).

[53] L.-A. Couston, D. Lecoanet, B. Favier, and M. Le Bars, The energy flux spectrum of internal waves generated by turbulent convection, J. Fluid Mech. **854**, 1103 (2018).

[54] N. Balci, A. M. Isenberg, and M. S. Jolly, Turbulence in vertically averaged convection, Physica D: Nonlinear Phenomena **376-377**, 216 (2018).

[55] S. Lepot, S. Aumaître, and B. Gallet, Radiative heating achieves the ultimate regime of thermal convection, Proc. Natl. Acad. Sci. **115**, 8937 (2018).

[56] G. Michel and G. P. Chini, Strong wave–mean-flow coupling in baroclinic acoustic streaming, J. Fluid Mech. **858**, 536 (2019).

[57] K. J. Burns, D. Lecoanet, G. M. Vasil, J. S. Oishi, and B. P. Brown, The "Sphered Cube": A new method for the solution of partial differential equations in cubical geometry, arXiv:1903.12642.

[58] J. Földes, N. Glatt-Holtz, G. Richards, and J. Whitehead, Hydrodynamic stability in the presence of a stochastic forcing:a case study in convection, arXiv:1704.03840.

[59] J. Olsthoorn, E. W. Tedford, and G. A. Lawrence, Diffused-interface Rayleigh-Taylor instability with a nonlinear equation of state, Phys. Rev. Fluids **4**, 094501 (2019).

[60] G. Saranraj and A. Guha, Energy transfer in resonant and near-resonant internal wave triads for weakly non-uniform stratifications. Part I: Unbounded domain, arXiv:1808.05591.

[61] C. B. Rocha, T. Bossy, S. G. Llewellyn Smith, and W. R. Young, Improved bounds on horizontal convection, J. Fluid Mech. **883**, 5957 (2020).

[62] J.-H. Kim, W. Moon, A. J. Wells, J. P. Wilkinson, T. Langton, B. Hwang, M. A. Granskog, and D. W. Rees Jones, Salinity control of thermal evolution of late summer melt ponds on arctic sea ice, Geophys. Res. Lett. **45**, 8304 (2018).

[63] J. Olsthoorn, C. E. Bluteau, and G. A. Lawrence, Under-ice salinity transport in low-salinity waterbodies, Limnol. Oceanogr. **65**, 247 (2020).

[64] G. M. Vasil, K. J. Burns, D. Lecoanet, S. Olver, B. P. Brown, and J. S. Oishi, Tensor calculus in polar coordinates using Jacobi polynomials, J. Comput. Phys. **325**, 53 (2016).

[65] G. M. Vasil, D. Lecoanet, K. J. Burns, J. S. Oishi, and B. P. Brown, Tensor calculus in spherical coordinates using Jacobi polynomials. Part-I: Mathematical analysis and derivations, J. Comput. Phys.: X **3**, 100013 (2019).

[66] D. Lecoanet, G. M. Vasil, K. J. Burns, B. P. Brown, and J. S. Oishi, Tensor calculus in spherical coordinates using Jacobi polynomials. Part-II: Implementation and examples, J. Comput. Phys.: X **3**, 100012 (2019).

[67] E. W. Hester, G. M. Vasil, and K. J. Burns, Improving convergence of volume penalized fluid-solid interactions, arXiv:1903.11914.

[68] J. O. Wenegrat, J. Callies, and L. N. Thomas, Submesoscale baroclinic instability in the bottom boundary layer, J. Phys. Oceanogr. **48**, 2571 (2018).

[69] J. Callies, Restratification of abyssal mixing layers by submesoscale baroclinic eddies, J. Phys. Oceanogr. **48**, 1995 (2018).

[70] C. Tauber, P. Delplace, and A. Venaille, A bulk-interface correspondence for equatorial waves, J. Fluid Mech. **868**, 123 (2019).

[71] S. Kar and A. Guha, An inverse technique for reconstructing ocean's density stratification from surface data, Ocean Model. **147**, 101561 (2020).

[72] R. M. Holmes, C. de Lavergne, and T. J. McDougall, Tracer transport within abyssal mixing layers, J. Phys. Oceanogr. **49**, 2669 (2019).

[73] B. Bordwell, B. P. Brown, and J. S. Oishi, Convective dynamics and disequilibrium chemistry in the atmospheres of giant planets and brown dwarfs, Astrophys. J. **854**, 8 (2018).

[74] J. B. Parker and N. C. Constantinou, Magnetic eddy viscosity of mean shear flows in two-dimensional magnetohydrodynamics, Phys. Rev. Fluids **4**, 083701 (2019).

[75] S. Davidovits and N. J. Fisch, Sudden Viscous Dissipation of Compressing Turbulence, Phys. Rev. Lett. **116**, 105004 (2016).

[76] S. Davidovits and N. J. Fisch, Compressing turbulence and sudden viscous dissipation with compression-dependent ionization state, Phys. Rev. E **94**, 053206 (2016).

[77] A. E. Fraser, Role of stable modes in driven shear-flow turbulence, Phys. Plasmas **25**, 122303 (2018).

[78] S. Davidovits and N. J. Fisch, Viscous dissipation in two-dimensional compression of turbulence, Phys. Plasmas **26**, 082702 (2019).

[79] H. Zhu, Y. Zhou, and I. Y. Dodin, Theory of the Tertiary Instability and the Dimits Shift from Reduced Drift-Wave Models, Phys. Rev. Lett. **124**, 055002 (2020).

[80] J. B. Parker, J. W. Burby, J. B. Marston, and S. M. Tobias, Nontrivial topology of the Alfvén continuum and topological character of reversed-shear Alfvén eigenmodes, arXiv:1909.07910.

[81] Y. Zhou, H. Zhu, and I. Y. Dodin, Solitary zonal structures in subcritical drift waves: A minimum model, Plasma Phys. Control. Fusion **62**, 045021 (2020).

[82] J. B. Parker, J. B. Marston, S. M. Tobias, and Z. Zhu, Topological gaseous plasmon polariton in realistic plasma, arXiv:1911.01069.

[83] J. P. Boyd, *Chebyshev and Fourier Spectral Methods* (Dover Publications, Mineola, 2001).

[84] T. A. Driscoll and N. Hale, Rectangular spectral collocation, IMA J. Numer. Anal. **38**, dru062 (2015).

[85] C. Lanczos, Trigonometric interpolation of empirical and analytical functions, J. Math. Phys. **17**, 123 (1938).

[86] C. W. Clenshaw, The numerical solution of linear differential equations in Chebyshev series, Math. Proc. Cambridge Philos. Soc. **53**, 134 (1957).

[87] R. A. Fisher, The wave of advance of advantageous genes, Ann. Eugenics **7**, 355 (1937).

[88] A. Kolmogorov, I. Petrovskii, and N. Piscunov, A study of the equation of diffusion with increase in the quantity of matter, and its application to a biological problem, Byul. Moskovskogo Gos. Univ. 1, 1 (1937).

[89] P. Virtanen, R. Gommers, T. E. Oliphant *et al.*, SciPy 1.0–fundamental algorithms for scientific computing in PYTHON, Nature Methods **17**, 261 (2020).

[90] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, Cython: The best of both worlds, Comput. Sci. Eng. **13**, 31 (2011).

[91] M. Frigo and S. G. Johnson, The design and implementation of FFTW3, Proc. IEEE **93**, 216 (2005).

[92] L. Dalcin, R. Paz, M. Storti, and J. D'Elía, MPI for PYTHON: Performance improvements and MPI-2 extensions, J. Parallel Distributed Comp. **68**, 655 (2008).

[93] The HDF Group, Hierarchical Data Format, version 5 (1997), http://www.hdfgroup.org/HDF5/.

[94] A. Collette, *PYTHON and HDF5* (O'Reilly Media, Inc, Sebastopol, 2013).

[95] http://dedalus-project.org/

[96] J. S. Oishi, B. P. Brown, K. J. Burns, D. Lecoanet, and G. M. Vasil, Perspectives on reproducibility and sustainability

of open-source scientific software from seven years of the dedalus project, arXiv:1801.08200.

[97] http://dedalus-project.org/citations/.

[98] D. Wang and S. J. Ruuth, Variable step-size implicit-explicit linear multistep methods for time-dependent partial differential equations, J. Comput. Math. **26**, 838 (2008).

[99] U. M. Ascher, S. J. Ruuth, and R. J. Spiteri, Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations, Appl. Num. Math. **25**, 151 (1997).

[100] M. Sprague, K. Julien, E. Knobloch, and J. Werne, Numerical simulation of an asymptotically reduced system for rotationally constrained convection, J. Fluid Mech. **551**, 141 (2006).

[101] https://github.com/DedalusProject/methods_paper_examples.

[102] http://dedalus-project.org/gallery/.

[103] https://github.com/DedalusProject/dedalus_scaling.

[104] https://www.sdsc.edu/support/user_guides/popeye-simons.html.

[105] https://github.com/PrincetonUniversity/Athena-Cversion.

[106] M. Ablowitz and B. Prinari, Nonlinear Schrödinger systems: Continuous and discrete, Scholarpedia **3**, 5561 (2008).

[107] G. Berkolaiko and P. Kuchment, *Introduction to Quantum Graphs* (American Mathematical Society, Providence, 2013).

[108] D. Noja, Nonlinear Schrödinger equation on graphs: Recent results and open problems, Philos. Trans. R. Soc., A **372**, 20130002 (2013).

[109] S. A. Orszag and C. M. Tang, Small-scale structure of two-dimensional magnetohydrodynamic turbulence, J. Fluid Mech. **90**, 129 (1979).

[110] J. M. Stone, T. A. Gardiner, P. Teuben, J. F. Hawley, and J. B. Simon, Athena: A new code for astrophysical MHD, Astrophys. J. Suppl. Ser. **178**, 137 (2008).

[111] K. G. Felker and J. M. Stone, A fourth-order accurate finite volume method for ideal MHD via upwind constrained transport, J. Comput. Phys. **375**, 1365 (2018).

[112] D. Lee and A. E. Deane, An unsplit staggered mesh scheme for multidimensional magnetohydrodynamics, J. Comput. Phys. **228**, 952 (2009).

[113] T. S. Tricco, D. J. Price, and M. R. Bate, Constrained hyperbolic divergence cleaning in smoothed particle magnetohydrodynamics with variable cleaning speeds, J. Comput. Phys. **322**, 326 (2016).

[114] P. Mocz, Correspondence between constrained transport and vector potential methods for magnetohydrodynamics, J. Comput. Phys. **328**, 221 (2017).

[115] D. R. Durran, Improving the anelastic approximation, J. Atmos. Sci. **46**, 1453 (1989).

[116] G. M. Vasil, D. Lecoanet, B. P. Brown, T. S. Wood, and E. G. Zweibel, Energy conservation and gravity waves in sound-proof treatments of stellar interiors. II. Lagrangian constrained analysis, Astrophys. J. **773**, 169 (2013).

[117] G. P. Zank and W. H. Matthaeus, The equations of reduced magnetohydrodynamics, J. Plasma Phys. **48**, 85 (1992).

[118] C. S. Ng, L. Lin, and A. Bhattacharjee, High-lundquist number scaling in three-dimensional simulations of parker's model of coronal heating, Astrophys. J. **747**, 109 (2012).

[119] G. P. Chini, K. Julien, and E. Knobloch, An asymptotically reduced model of turbulent Langmuir circulation, Geophys. Astrophys. Fluid Dyn. **103**, 179 (2009).

[120] Z. Malecha, G. Chini, and K. Julien, A multiscale algorithm for simulating spatially-extended Langmuir circulation dynamics, J. Comput. Phys. **271**, 131 (2014).

[121] G. L. Wagner, G. Ferrando, and W. R. Young, An asymptotic model for the propagation of oceanic internal tides through quasi-geostrophic flow, J. Fluid Mech. **828**, 779 (2017).

[122] G. L. Wagner and W. R. Young, A three-component model for the coupled evolution of near-inertial waves, quasi-geostrophic flow and the near-inertial second harmonic, J. Fluid Mech. **802**, 806 (2016).

[123] G. L. Wagner and W. R. Young, Available potential vorticity and wave-averaged quasi-geostrophic flow, J. Fluid Mech. **785**, 401 (2015).

[124] K. Julien, E. Knobloch, and J. Werne, A new class of equations for rotationally constrained flows, Theor. Comput. Fluid Dyn. **11**, 251 (1998).

[125] K. Julien, E. Knobloch, R. Milliff, and J. Werne, Generalized quasi-geostrophy for spatially anisotropic rotationally constrained flows, J. Fluid Mech. **555**, 233 (2006).

[126] K. Julien, J. M. Aurnou, M. A. Calkins, E. Knobloch, P. Marti, S. Stellmach, and G. M. Vasil, A nonlinear model for rotationally constrained convection with Ekman pumping, J. Fluid Mech. **798**, 50 (2016).

[127] K. Julien, E. Knobloch, A. M. Rubio, and G. M. Vasil, Heat Transport in Low-Rossby-Number Rayleigh-Bénard Convection, Phys. Rev. Lett. **109**, 254503 (2012).

[128] M. Plumley, K. Julien, P. Marti, and S. Stellmach, The effects of Ekman pumping on quasi-geostrophic Rayleigh-Bénard convection, J. Fluid Mech. **803**, 51 (2016).

[129] J. G. Charney and N. A. Phillips, Numerical integration of the quasi-geostrophic equations for barotropic and simple baroclinic flows, J. Atmos. Sci. **10**, 71 (1953).

[130] N. A. Phillips, The general circulation of the atmosphere: A numerical experiment, Q. J. R. Meteorol. Soc. **82**, 123 (1956).

[131] W. R. Holland and P. B. Rhines, An example of eddy-induced ocean circulation, J. Phys. Oceanogr. **10**, 1010 (1980).

[132] R. Salmon, Baroclinic instability and geostrophic turbulence, Geophys. Astrophys. Fluid Dyn. **15**, 167 (2006).

[133] J. C. McWilliams, J. B. Weiss, and I. Yavneh, Anisotropy and coherent vortex structures in planetary turbulence, Science **264**, 410 (1994).

[134] A. Vallgren and E. Lindborg, Charney isotropy and equipartition in quasi-geostrophic turbulence, J. Fluid Mech. **656**, 448 (2010).

[135] R. Tulloch and K. S. Smith, A note on the numerical representation of surface dynamics in quasigeostrophic turbulence: Application to the nonlinear eady model, J. Atmos. Sci. **66**, 1063 (2009).

[136] S. Stellmach, M. Lischper, K. Julien, G. Vasil, J. S. Cheng, A. Ribeiro, E. M. King, and J. M. Aurnou, Approaching the Asymptotic Regime of Rapidly Rotating Convection: Boundary Layers versus Interior Dynamics, Phys. Rev. Lett. **113**, 254501 (2014).

[137] M. Watwood, I. Grooms, K. A. Julien, and K. S. Smith, Energy-conserving Galerkin approximations for quasi-geostrophic dynamics, J. Comput. Phys. **388**, 23 (2019).

[138] J. L. Aurentz and R. Mikael Slevinsky, On symmetrizing the ultraspherical spectral method for self-adjoint problems, J. Comput. Phys. **410**, 109383 (2020).

[139] G. K. Vallis, *Atmospheric and Oceanic Fluid Dynamics* (Cambridge University Press, Cambridge, 2006).

[140] E. Fonda and K. R. Sreenivasan, Unmixing demonstration with a twist: A photochromic Taylor-Couette device, Am. J. Phys. **85**, 796 (2017).

[141] D. Mihalas and B. Weibel Mihalas, *Foundations of Radiation Hydrodynamics* (Oxford University Press, Oxford, 1984).

[142] J. I. Castor, *Radiation Hydrodynamics* (Cambridge University Press, Cambridge, 2007).

[143] Y.-F. Jiang, J. M. Stone, and S. W. Davis, A godunov method for multidimensional radiation magnetohydrodynamics based on a variable eddington tensor, Astrophys. J. Suppl. **199**, 14 (2012).

[144] Y.-F. Jiang, M. Cantiello, L. Bildsten, E. Quataert, and O. Blaes, Local radiation hydrodynamic simulations of massive star envelopes at the iron opacity peak, Astrophys. J. **813**, 74 (2015).

[145] A. Brandenburg, Stellar mixing length theory with entropy rain, Astrophys. J. **832**, 6 (2016).

[146] A. Barekat and A. Brandenburg, Near-polytropic stellar simulations with a radiative surface, Astron. Astrophys. **571**, A68 (2014).

[147] B. W. Hindman and E. G. Zweibel, The effects of a hot outer atmosphere on acoustic-gravity waves, Astrophys. J. **436**, 929 (1994).

[148] https://github.com/DedalusProject/eigentools.

[149] M. D. Simon and A. K. Geim, Diamagnetic levitation: Flying frogs and floating magnets (invited), J. Appl. Phys. **87**, 6200 (2000).

[150] D. J. Acheson, Instability by magnetic buoyancy, Sol. Phys. **62**, 23 (1979).

[151] G. M. Vasil and N. H. Brummell, Magnetic buoyancy instabilities of a shear-generated magnetic layer, Astrophys. J. **686**, 709 (2008).